Agent Tcl: A flexible and secure mobile-agent system

A Thesis

Submitted to the Faculty

in partial fulfillment of the requirements for the

degree of

Doctor of Philosophy

in

Computer Science

by

Robert S. Gray

DARTMOUTH COLLEGE

Hanover, New Hampshire

30 June 1997

Examining Committee:

George Cybenko (chairman)

David Kotz

Daniela Rus

Robert Sproull

Edward Berger
Dean of Graduate Studies

For my Grandpa Wilcox

Abstract

A mobile agent is an autonomous 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 itself to another machine, and then resume execution from the point of suspension. Mobile agents have the potential to provide a *single, general framework* in which a wide range of distributed applications can be implemented efficiently and easily. Several challenges must be faced, however, most notably reducing migration overhead, protecting a machine from malicious agents (and an agent from malicious machines), and insulating the agent against network and machine failures. Agent Tcl is a mobile-agent system under development at Dartmouth College that has evolved from a Tcl-only system into a multiple-language system that currently supports Tcl, Java, and Scheme. In this thesis, we examine the motivation behind mobile agents, describe the base Agent Tcl system and its security mechanisms for protecting a machine against malicious agents, and analyze the system's current performance. Finally, we discuss the security, fault-tolerance and performance enhancements that will be necessary for Agent Tcl and mobile agents in general to realize their full potential.

Acknowledgments

Many thanks to my three advisors, Professors George Cybenko, Daniela Rus and David Kotz, for their guidance, constructive criticism, and gentle natures; to Bob Sproull, who graciously agreed to serve as the fourth member of my thesis committee and provided invaluable feedback; to Professor Fillia Makedon, who guided me through my first large research project and first published paper; to the entire computer-science faculty, particularly Professor Donald Johnson, who welcomed me into the department despite the unusual nature of my arrival; to the faculty at the University of Vermont, particularly Professors Bob Dawson, Jeanne Douglas and Margaret Epstein, who prepared me for my time here; and to Bob Smith and Erald Medlar, two of the most extraordinary high-school teachers ever, who prepared me for college.

Many thanks to Keith Kotay for showing the way with Dartmouth's first mobile-agent system; to Melissa Hirschl for implementing the Agent Tcl debugger; to Sumit Chawla and Saurab Nog for implementing the RPC mechanism; to Josh Mills and Bill Bleier for implementing most of Agent Java; to Ahsan Kabir and Dave Gondek for implementing most of Agent Scheme; and to the legion of other graduate and undergraduate students who have worked on Agent Tcl over the past two years.

Many thanks to all the people who downloaded the public releases of the Agent Tcl system and provided a stream of bug reports and suggestions.

Many thanks to Jonathan Briggs, Monica Holboke and Robin Wooding, who were my first friends at Thayer, and Sumit Chawla, Keith Kotay and Louisa Tripp, who were my first friends in the computer science department; to the Room 220 crowd, Yunxin Wu, Aditya Bhasin, Kurt Cohen, Hiro Moizumi, Brian and Flora Brewington, Rohit Sharma, Rahul Vaid, Aditya Bhasin and Vlad Ristanovic, who made me focus when I needed to focus and made me take a break when I needed a break (and, in the case of Brian and Flora, made sure that I ate at least one decent meal a month); to all my fellow students, who made the journey far easier; to Eric Hanson and Eric Leonard, who were my constant companions as an undergraduate; to Jamie Weinstock and Kiff Shelton, who were my constant companions in high school; to Michelle Decker, who stuck up for me one day in English class and cannot imagine how much it still means to me, even twelve years later; and to Amy Strang, who was my first friend ever (at least the first that I can remember).

Finally, many thanks to my parents, grandparents, aunts, uncles and cousins, who have been there for me at every moment and have guided me from my first steps to my first (and last) thesis. No one could have asked for a more loving or supportive family. I can only hope that I have at least partially repaid my immense debt.

I have left out many people that I do remember and probably some that I have forgotten. If you are not here even though I remember you, it is because I was trying to hold the acknowledgment section under two pages. If you are not here because I have forgotten you, I offer my heartfelt apologies. One day I will run into you again, hit myself in the forehead, and wonder where my mind was. In any event, if you know me at all, then know that you should be listed here too, for I realize as I am writing this that I have been truly blessed and that virtually everyone I have ever met has been both a good person and a good friend.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

A *mobile agent* is an autonomous 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 itself to another machine, and resume execution on the new machine from the point at which it left off. On each machine, it interacts with service agents and other resources to accomplish its task. In Figure 1.1, for example, an agent has migrated to interact with a search engine and will migrate again to interact with additional search engines. Once the agent has the desired information, it will migrate one last time to return to its home site so that it can present the information to its owner.

Mobile agents have several advantages. By migrating to the location of a needed resource, such as the search engine in Figure 1.1, an agent can interact with the resource without transmitting any intermediate data across the network, significantly reducing bandwidth consumption in many applications. Similarly, by migrating to the location of a user, an agent can respond to user actions rapidly. In either case, the agent can continue its interaction with the resource or user even if the network connection goes down, making mobile agents particularly attractive in mobile-computing applications. Mobile agents also allow traditional clients and servers to offload work

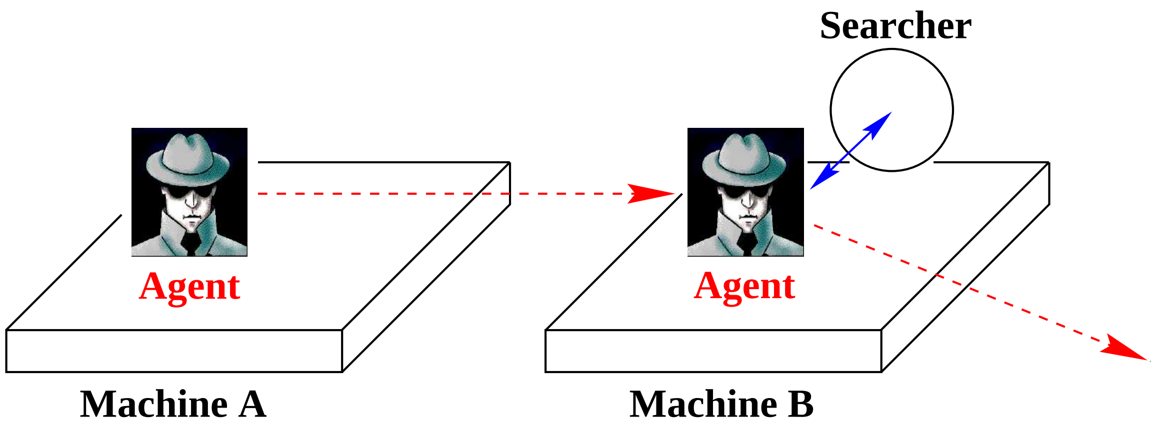


Figure 1.1: The basic idea of migration. Here an agent has migrated so that it can interact *locally* with a search engine. It will migrate again to find additional information and bring the results back to its owner.

to each other, and to *change* who offloads to whom according to machine capabilities and current loads. Similarly, mobile agents allow an application to dynamically deploy its components to arbitrary network sites, and to *redeploy* those components in response to changing network conditions. Finally, most distributed applications fit naturally into the mobile-agent model, since mobile agents can migrate sequentially through a set of machines, send out a wave of child agents that will visit multiple machines in parallel, remain stationary and interact with resources remotely, or any combination of these three extremes. Complex, efficient and robust behaviors can be realized with surprisingly little code, and, in fact, our own experience with undergraduate programmers at Dartmouth suggests that mobile agents are easier to understand than other distributed computing paradigms.

Although each of these advantages is a reasonable argument for mobile agents, any specific application can be implemented just as efficiently and robustly with more traditional techniques, such as queued RPC [JTK97], higher-level server operations, application-specific query languages, application-specific proxies within the permanent network, automated installation facilities, and active web pages that contain Java applets. Mobile agents eliminate the need for these other techniques, however, combining their strengths into a single, general, convenient framework. Distributed applications can be implemented efficiently and easily even if they must exhibit extremely flexible behavior in the face of changing network conditions. For example, a search application can migrate to a dynamically selected proxy site and do its merging and filtering there, while a server can continually migrate to new machines to minimize the average latency between itself and its clients [RASS97].

In short, the true strength of mobile agents is that they are a uniform paradigm for distributed applications, allowing both data and code to move from machine to machine. Several key research problems must be solved, however, before a mobile-agent *system* can realize the full potential of the mobile-agent paradigm. Such problems include reducing migration and communication overhead, protecting a machine from malicious agents (and an agent from malicious machines), limiting an agent's total resource consumption, insulating an agent against network and machine failures, and developing the resource discovery, network sensing, navigation, and planning services that will allow an agent to identify and reach the desired services.

Agent Tcl [Gra97, GKCR97, Gra96, Gra95] is a mobile-agent system that is under development at Dartmouth College to address some of these research problems. Agent Tcl focuses on five research areas: (1) performance, (2) support for multiple languages, (3) cryptographic authentication and restricted execution environments to protect a machine from malicious agents, (4) economic-based models to limit an agent's total resource consumption across multiple machines, and (5) networking sensing, navigation and planning services so that an agent can determine the best path through the network according to its task and current network conditions. In this thesis, we are concerned with the base Agent Tcl system, specifically its support for multiple languages, its mechanisms for protecting an agent from malicious machines, and its performance relative to traditional distributed systems.

Agent Tcl has two main components: (1) a server that runs on each machine, and (2) an execution environment for each supported agent language. The server accepts incoming agents, authenticates the identity of the owner, and passes the authenticated agent to the appropriate execution environment. The server also keeps tracks of the agents running on its machines and answers queries about their status, allows an authorized user to suspend, resume and terminate a running agent, and allows agents to communicate with each other through message passing and direct connections. In a future version of Agent Tcl, the server will also provide a nonvolatile store for agents so that an agent can be restarted after a machine failure. As in the Tacoma system [JvRS95], all other services are provided by *agents*. Such services include resource directories, network-sensing tools, higher-level communication protocols such as RPC, and resource managers. Resource managers guard access to critical system resources such as the screen, network and disk; specifically, the resource managers decide which actions an agent can perform based on the authenticated identity of the agent's owner.

Each execution environment includes the interpreter that actually executes the agent, a state-capture module that captures the complete state of the agent when the agent decides to migrate to a new machine, and a security-enforcement module that *enforces* the security policy from the resource managers. In addition, each execution environment includes a package of stub routines that the agent uses to interact with the servers and obtain the available agent services, such as migration, communication, and status queries.

Agent Tcl is similar to other mobile-agent systems, such as Tacoma [JvRS95], Ara [PS97], and Telescript [Whi94], but distinguishes itself with (1) its combination of multiple languages, a simple migration mechanism, and both low- and high-level communication protocols, (2) its simple but effective security model, and (3) its extensive support services and tools.

- **Multiple languages** (Chapter 5). Agent Tcl supports multiple, off-the-shelf languages, Tcl, Java and Scheme, and allows the straightforward addition of new languages. The agent programmer can select the language that is most appropriate for her task.

- **Migration** (Chapter 5). Agent Tcl reduces migration to a single instruction, `jump`, which automatically capture the complete state of the agent and sends the state image to the new machine. The agent continues from the point of the `jump` on the new machine. Although the system programmer must implement the `jump` instruction for each supported language, once the instruction is available, the agent programmers do not need to explicitly collect state information before migration.

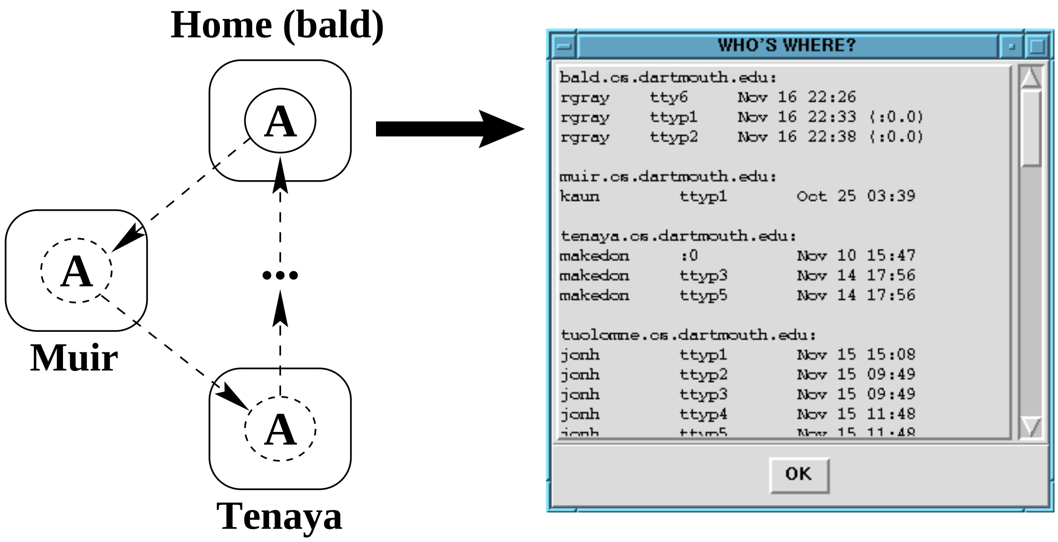
- **Communication** (Chapter 5). The base Agent Tcl system provides two low-level communication mechanisms, messaging passing and direct connections (for bulk data transfer), which work the same regardless of whether the communicating agents are on the same or different machines. Higher-level communication mechanisms, such as a Remote Procedure Call (RPC) mechanism [NCK96], are implemented at the agent level on top of the two low-level services. With this approach, the agent programmer can choose from a range of communication mechanisms, but the base system remains lightweight.

- **Security** (Chapter 6). Agent Tcl protects an individual machine against malicious agents with a simple but powerful security model that cleanly separates policy and enforcement. Agents are digitally signed during migration so that their owner can be identified. Resource manager agents use the identity of the agent's owner to decide which screen, network, disk, etc., accesses are allowed for that agent. The resource managers use traditional access-control lists to make their decisions. These lists are read from a configuration file on startup; then the machine administrator can change the lists interactively at any time through a graphical management tool. A lightweight enforcement module for each supported language enforces the decisions of the resource managers. These enforcement modules ask the resource manager to make a decision when the agent attempts a resource access for the first time, and then cache the decision so that they do not have to keep asking the resource manager. In addition to these existing mechanisms for protecting an individual machine from malicious agents, another student is working on economic-based models for limiting an agent's total resource consumption across multiple machines.

- **Support services** (Chapter 9). Agent Tcl provides numerous support services, most notably (1) a debugger that tracks an agent as it moves through the network, monitors its communication with other agents, and provides traditional debugger features such as breakpoints, watch conditions and line-at-a-time execution [HK97], (2) a docking system that allows an agent to transparently migrate to or from a mobile computer, even if the mobile computer is not currently connected to the network [GKN+97], (3) hierarchical yellow pages that provide a keyword-indexed directory of available services [GKN+97], (4) several network sensing and planning modules that allow an agent to examine the current state of the network and construct an optimal route [GKN+97, Car97], and (5) a simple Mobile Agent Construct Environment (MACE) that allows a non-programmer to graphically construct an agent [Sha97]. These services represent the research of several other students. Although they are discussed further in Chapter 9, they are not an integral part of this thesis.

A sample Agent Tcl agent is shown in Figure 1.2. This agent, which is written in Tcl, migrates through a sequence of machines and uses the Unix `who` command to find out which users are logged onto each one. Once it has migrated through all the machines, it migrates one more time to return to its home machine and present the user list to its owner. The agent first uses the `agent_begin` command to register with the Agent Tcl server on its current machine. The agent then steps through the list of machines and uses the `agent_jump` command to migrate to each one in turn. On each machine, the agent's execution continues from the point of the `agent_jump`, and the agent invokes the Unix `who` command to get the user list. Once the agent has migrated through all of the machines, it invokes `agent_jump` one last time to migrate back to its home machine where it presents the user list to its owner (the code to display the user list on the screen is not shown in the figure). Finally, the agent invokes `agent_end` to tell the Agent Tcl server that it has finished. Although the "who" agent is not the most useful agent, it illustrates the general form of any agent that migrates sequentially through a set of machines. The `exec who` can be replaced with any desired processing. The most important thing to note about the agent is that it was extremely easy to write; it is just a traditional single-machine program with an `agent_jump` command at two key points.

Agent Tcl has been used primarily in distributed information-retrieval applications, including retrieval of technical reports [RGK96], product descriptions and




```
agent_begin          # register with the local agent server

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

foreach machine $machineList {
    agent_jump $machine              # jump to each machine
    append output [exec who]         # any local processing
}

agent_jump $agent(home)              # jump back home

# display output window

agent_end                            # unregister
```

Figure 1.2: The "who" agent migrates through a set of machines and figures out who is logged onto each one. Although the "who" agent is not the most useful agent, it illustrates the general form of any agent that migrates sequentially through a set of machines. The `exec who` can be replaced with any desired processing.

prices [RGK96], medical records [Wu95], and three-dimensional drawings of mechanical parts [CBC96] (Chapter 8). Mobile agents have two advantages in these applications: (1) the search agent can migrate to the location of the relevant database, eliminating all intermediate data transfer; (2) the search agent can migrate to a dynamically selected proxy site and do its searches, merging and filtering from that proxy site, allowing it to continue even if the network link to the home machine goes down and minimizing the amount of data that it brings back to the home machine. The second advantage is particularly critical if the search is launched from a mobile computer, which typically has an unreliable, low-bandwidth connection into the network. Agent Tcl has also been used to track purchase orders in a workflow application [CGN96] and in several network management and information-retrieval applications at non-Dartmouth sites.

Agent Tcl agents allow these applications to minimize their bandwidth consumption and to proceed with their task even if network links go down. On the other hand, when compared with traditional client/server implementations, the total *task completion time* is longer except in low-performance networks. The primary cause is the high migration overhead in the current Agent Tcl system. This migration overhead can be reduced significantly, however, and in combination with several other straightforward optimizations, this reduction should allow Agent Tcl agents to perform just as well as client/server solutions in high-performance networks and much better in low-performance networks (Chapter 7). Of course, there will always be network conditions under which it is better for an agent to remain stationary and act like a traditional client, so we have begun to develop some simple formulas that will help an agent decide when and where to migrate (Chapter 7).

In rest of this thesis, we present the Agent Tcl system in detail. Chapter 2 discusses the motivation for mobile agents, Chapters 3 and 4 examine other mobile-agent systems, and Chapters 5 and 6 describe the base Agent Tcl system and its security mechanisms for protecting a machine from malicious agents. Chapter 6 also includes possible approaches for protecting an agent from malicious machines. Then, Chapter 7 analyzes the system's current performance and develops some simple formulas that an agent can use when deciding whether to migrate or remain stationary. Finally, Chapter 8 looks at several existing and potential applications for Agent Tcl, Chapter 9 presents the Agent Tcl components that other students are developing as part of their research, and Chapter 10 considers the security, fault-tolerance and performance enhancements that will be necessary for Agent Tcl and other mobile-agent systems to realize their full potential.

Chapter 2

Motivation

Mobile agents have several performance advantages where performance can be a matter of bandwidth, latency, robustness, or simply ease of development.

Bandwidth. In the traditional client/server model [Lew95], the server provides a fixed set of operations that a client invokes from a remote machine. If the server does not provide an operation that matches the client task exactly, either the client must make a series of cross-network calls to lower-level operations, or the server developer must add a new operation to the server. The first option brings intermediate data across the network on every call, potentially wasting a significant amount of network bandwidth, especially if the intermediate data is not useful beyond the end of the client task. The second option is an intractable programming task as the number of distinct clients increases. In addition, it discourages modern software engineering since the server becomes a collection of complex, specialized routines rather than simple, general primitives. Mobile agents, on the other hand, do not waste bandwidth even if the server provides only low-level operations, simply because an agent can migrate *to the server* where it performs any desired processing before returning just the final result to the client [Whi94]. Agents that do more work avoid more intermediate messages and conserve more bandwidth.

Despite this advantage, there are several tradeoffs that must be considered. First, although a mobile agent eliminates the transmission of intermediate data, the agent itself must first be sent to the server. If the agent is larger than the intermediate data, it will obviously consume more bandwidth than a traditional client/server implementation. Second, if the network between the client and server has high bandwidth and low latency, the amount of intermediate data must be quite large for the mobile agent to have a shorter completion time, especially when one considers the time needed for the server to start up the necessary execution environment for the incoming agent. Finally, the mobile agent uses less processing time at the client but more time at the server, since the agent performs all intermediate computations at the server *and* is typically written in an interpreted language. If the server is low-powered or heavily loaded, a CPU-intensive agent can easily take longer than the corresponding cross-network calls. Taken together, these tradeoffs mean that mobile agents display a much greater performance advantage as network bandwidth decreases, network load increases, the amount of intermediate data increases, and server power increases. Conversely, mobile agents can display a severe performance *disadvantage* as conditions move in the other direction. Any mobile-agent system must allow an agent to examine current network and machine conditions and then decide whether to migrate to a remote resource *or* remain at the current site and access the resource using the equivalent of message passing [SS94] or remote procedure call (RPC) [BN84].

Latency. By migrating to the location of a resource, an agent can interact with the resource much faster than from across the network. Such faster interaction is one of the main motivations for Java-enabled web browsers. Latency between the user (i.e., the screen, keyboard and mouse) and a web-based application is much lower if part or all of the application is downloaded to the client's machine in the form of a Java applet, allowing the application to respond much more rapidly to user actions [CH97]. Mobile agents can be used for the same purpose. Of course, whether latency is actually important depends entirely on the application, and whether mobile agents actually reduce the latency depends on the current server load, the network latency, and the time needed for agent migration relative to the number of operations that the agent invokes. Clearly a migrating agent that invokes only a single operation will always have a larger end-to-end latency than the corresponding cross-network call. Thus, as with bandwidth, the mobile-agent system must allow the agent to examine current network and machine conditions so that it can decide whether to migrate or remain stationary.

Mobile computing. A migrating agent does not require a permanent connection with its home site and can proceed with its task even if the home site is unreachable. Thus mobile agents are ideally suited for mobile computing in which computers can be disconnected from the network for long periods of time [Whi94]. For example, a laptop or other mobile device can send an agent out into the network. The agent will continue with its work even if the laptop disconnects and will be ready with a result when the laptop reconnects. Similarly, a service can send an agent onto a laptop to continue interacting with the user [TLKC95]. In addition to periods of disconnection, the networks involved in mobile computing (or more precisely the links at the edge of these networks) are often characterized by low bandwidth and high latency, making mobile agents even more attractive. By migrating to or from the laptop, the agent can minimize use of the low-bandwidth, high-latency link.

Offloading. Mobile agents allow a low-powered client to offload work to a high-powered proxy or an overloaded server to offload work to clients. In the latter case, a server would send back both data and an agent that performs additional processing on the data. One example is a server that sends back a data set along with an agent that can present the data in various ways; the user can view and manipulate the data without any further contact with the server, reducing server load and allowing much faster response times to user actions [Kna96]. In such an application, the agent serves the same purpose as a Java applet, which is downloaded into a Java-enabled browser so that a Web-based application can present a complex graphical interface without annoying delays [CH97]. Of course, the same effect can be achieved if the user installs special client software on their machine. The need for a separate installation step, however, makes it much more unlikely that a user will try a new Internet service or use a service that she only needs once [Kna96]. Both mobile agents and Java applets eliminate the installation step and allow a user to try a service with minimal effort.

Dynamic deployment. Mobile agents allow rapid development, testing and installation of distributed applications since application components can be deployed "on-the-fly" to arbitrary network sites. In addition, even after it starts execution, the application can *redeploy* its components in response to changing network conditions. Such dynamic redeployment is used in [RASS97] to ensure that an Internet "chat" server is always located at the network position that minimizes the average latency between it and its current clients.

Intelligent data [Kna96]. Intelligent (or active) data can be viewed as a special form of dynamic deployment. Here the messages in some arbitrary messaging system contain both data and the code that is needed to handle that data on the destination machine. Such code might be a viewer for a multimedia data type or a decompression routine to uncompress compressed data [Kna96]. Including code in the messages allows the destination machine to handle new data types without user effort and makes it trivial to introduce better viewers and decompression algorithms. Two existing systems with such *active messages* are (1) Safe-Tcl/MIME in which Tcl scripts are embedded inside MIME-enabled mail messages and executed (inside a safe execution environment) when the message is received or read [LO95] and (2) active networks in which packets contain small code fragments that are executed on each router [TSS+97].

Convenient paradigm. Mobile agents are a convenient paradigm for distributed applications. First, application components can dynamically deploy themselves throughout the network as described above. Second, mobile agents hide the communication channels but not the location of the computation [Whi94]. This makes mobile agents easier to use than low-level facilities in which the programmer must explicitly handle communication details, but more flexible and powerful than schemes such as process migration in which the system decides when to move a program based on a small set of fixed criteria. Third, a migrating agent needs to worry about network failures only during migration; all resource operations are invoked locally to the resource. Fourth, many tasks, such as network management, information retrieval and workflow, fit naturally into the *jump-act-jump* model of mobile agents. The agent migrates to a machine, performs a task, migrates to a second machine, performs a task that might be dependent on the outcome of the first task, and so on. Finally, our own experience with undergraduate programmers at Dartmouth suggests that mobile agents are easier to understand than many other distributed computing paradigms. It has also been suggested that mobile agents move the programmer away from the rigid client-server model to the more flexible peer-peer model in which programs communicate as peers and act as either clients or servers depending on their current needs [Coe94]. Although this is perhaps true conceptually, it does not appear to be an inherent characteristic of mobile agents, but rather a matter of whether the programmer *chooses* to have her agents act as peers. Currently, in most of the mobile-agent applications of which we are aware, there are clearly identifiable client and server agents, with the client agents migrating to obtain the services of the server agents. At the same time, most mobile-agent systems provide flexible, high-level communication primitives, making it relatively straightforward to implement peer-peer agents if desired.

Summary. Although each of the advantages above is a reasonable argument for mobile agents, it is essential to realize that for any specific application, the same performance can be realized with other techniques [HCK95]. Such techniques include queued RPC [JTK97], higher-level server operations, application-specific query languages, application-specific proxies within the permanent network, automated installation facilities, and active web pages that contain Java applets. Mobile agents eliminate the need for these other techniques, however, and allow a wide range of distributed applications to be implemented easily within the same, general framework *and* to exhibit extremely flexible behavior in the face of changing network conditions. For example, consider the technical report agent shown in Figure 2.1. The agent's task is to search a distributed collection of technical reports for information relevant to the user's query. The agent firsts asks the user to enter a free-text query. Then, if the connection between the home machine and the network is reliable and of high-

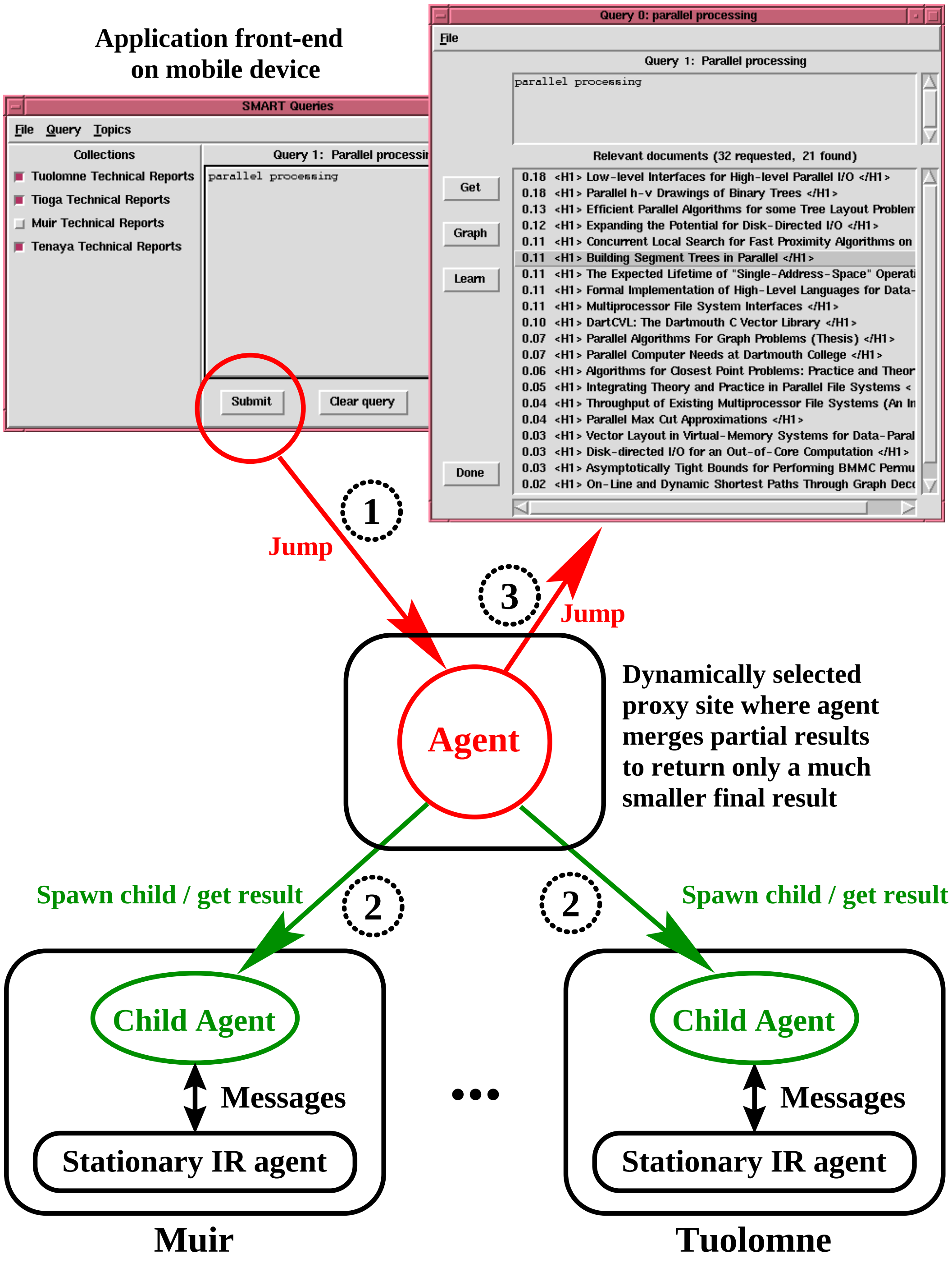


Figure 2.1: An example application. Here a mobile agent is searching a distributed collection of technical reports. The agent first decides whether to move to a dynamically selected proxy site. Then it decides whether to spawn child agents or simply interact with the individual document collections from across the network.

bandwidth, the agent will stay on the home machine. If the connection is unreliable or of low bandwidth, such as if the home machine is a mobile device, the agent will jump to a *proxy site* within the network. This initial jump reduces the use of the poor-quality link to just the transmission of the agent and the transmission of the complete result, allowing the agent to proceed with its task even if the link goes down. The proxy site is dynamically selected according to the current location of the home machine and the document collections.

Once the agent has migrated to a proxy site if desired, it must interact with the stationary agents that serve as an interface to the technical report collections. If these stationary agents provide high-level operations, the agent simply makes RPC-style calls across the network (using the agent-communication mechanisms). If the stationary agents provide only low-level operations, the agent sends out child agents that travel to the document collections and perform the query there, avoiding the transfer of large amounts of intermediate data. Information about the available search operations is obtained from the same directory services that provide the location of the document collections.[1] Once the agent has the results from each child agent, it merges and filters these results, returns to the home machine if necessary, and presents the results to the user. Although the behavior exhibited by this agent is complex, it is actually quite easy to implement; the decisions whether to jump and create children involve little more than two `if` statements that check the information returned from the network monitor on the home machine and the directory services. It is hard to imagine any other technique that would allow us to provide an equally flexible implementation with the same small amount of work. More importantly, as long as migration and the agent language are both fast enough, the agent's performance

[1] Chapter 9 discusses the directory services that are used in Agent Tcl.

should be comparable to or better than that of any other technique, regardless of the network conditions and without any application-specific support from the search engines or proxy sites.

It is the "as long as" in the previous sentence that brings us to one of the key challenges facing a mobile-agent system. Although migration does not need to be as fast as a single RPC call, it must be a low-latency operation. In addition, to support compute-intensive agents, the mobile-agent system must include languages that are nearly as fast as compiled C. As we will see in the performance analysis section, although Agent Tcl does not meet these performance goals yet, there is reason to hope that it can. In other words, there is reason to hope that Agent Tcl and mobile agents in general can realize their full potential and serve as an efficient, general framework for most distributed applications.

Chapter 3

Related work

The popular definition of an agent is an intelligent software servant that either (1) relieves the user of a routine, burdensome task such as appointment scheduling or (2) filters the overwhelming amount of online information so that the user sees only the information that is relevant to her current needs [Haf95, Rog95]. This definition—due to its broadness and its ability to capture the imagination—has made "agent" a buzzword within both the academic and commercial worlds. Applications are often described as "agent-based" solely to draw attention or increase sales. For example, *No Hands Software* once described its Magnet^{TM} program as the "first intelligent agent for the Macintosh" even though it was essentially a file-finder [Fon93]. This inappropriate use of the term makes it more difficult to separate hype from actual research, but there appear to be five legitimate research areas in which the term "agent" is used—*artificial intelligence and robotics*, *personal assistants*, *distributed information retrieval*, *software interoperation* and *mobile agents.*

First, we briefly consider the other kinds of agents to underscore the differences between non-mobile and mobile agents and to illustrate potential applications for mobile agents.

3.1 Other kinds of agents

3.1.1 Artificial intelligence and robotics

Here an *agent* is an entity that perceives its environment with sensors and acts on its environment with effectors [RN95]. Such an agent can be either hardware with physical sensors and effectors or software with simulated sensors and effectors. This definition of an agent is used to provide a unified framework for artificial intelligence, to discuss software artifacts from a robotics viewpoint, and of course to discuss physical robots. This definition is not considered further except to note that it subsumes the definitions below.

3.1.2 Personal assistants

Here an *agent* is a program that relieves the user of a routine, burdensome task such as appointment scheduling or e-mail disposition. These agents are distinguished from traditional utilities by (1) their use of machine learning so that they can adapt to user habits and preferences [Mae94] or (2) their use of automated reasoning so that they can make complex inferences about the work environment [Rie94].

Maes presents a series of agents that start with a minimum amount of domain knowledge and learn how to perform the task by observing and interacting with the user and other agents [Mae94]. The mail agent, for example, uses memory-based reasoning to filter electronic mail. It remembers every situation-action pair that occurs when the user filters her mail manually. When a new situation occurs, the agent predicts what action should be taken by comparing the new situation with the nearest memorized neighbors. Depending on its confidence in the prediction, the agent will perform the action itself, suggest the action to the user, or do nothing.

In addition to learning from example, the agent accepts directives from the user and solicits suggestions from other mail agents.

Riecken has developed a more complex system called M that uses five inference engines[1] to automatically group the documents that are presented during the course of a virtual multimedia conference [Rie94]. The inference engines infer relationships among the documents based on the actions that the users apply to them. For example, if two documents are placed close together within the virtual conference room, the spatial engine might conclude that the documents deal with the same subject. The engines post their conclusions to a dynamic set of blackboards and use the conclusions of the other engines to continue the reasoning process. Eventually one theory about document relationships emerges as the most likely.

A mobile-agent system should be able to support these applications in a distributed setting. At a minimum this means that mobile agents should be able to easily access external resources that provide learning and reasoning capabilities.

3.1.3 Distributed information retrieval

Here an *agent* is a program that searches multiple information resources for the answer to a user query. Typically the resources contain large volumes of data and are distributed across a network of heterogeneous machines. In addition, the agents are characterized by (1) the use of knowledge-intensive techniques to avoid manual intervention or brute-force search and, in some cases, (2) the concurrent execution of multiple subsearches and *communication of partial results from one subsearch to the others*. The partial results from one subsearch are used to narrow the scope of other subsearches.

[1] functional, structural, causal, spatial and temporal

Etzioni and Weld present a softbot that accepts a request and then develops a plan that satisfies the request using available Internet resources [EW94]. Softbot stands for "software robot" and arises from the *artificial intelligence* definition above. In this case, the sensors are Internet resources such as archie, gopher and netfind, and the effectors are resources such as ftp, telnet and mail. The softbot encodes its knowledge of these resources as declarative logic. It accepts requests expressed in a subset of first-order logic and performs a standard backtracking search to develop an appropriate plan. The example softbot in [EW94] uses a combination of Internet resources to resolve underspecified e-mail addresses when sending messages.

Vesser has written a succession of papers that develop a model for distributed searching. One of the more recent is [OPL94], which recasts the model in terms of agents. In the model a search involving multiple distributed resources is performed by a collection of cooperating agents. Each agent is responsible for searching one resource and communicating partial search results to the other agents so that the other agents can refocus their search. The standard example is a vacation planner that searches multiple databases— weather, car rental, hotel and "places of interest"—to plan an appropriate vacation for the user. Each agent searches its assigned resource independently but uses partial results from the other agents to adjust its search criteria when needed. For example, the place agent might assume good weather initially, but then redo portions of its search when the weather agent tells it that bad weather is forecast for a particular area. Eventually the agents arrive at a consistent vacation plan.

These examples do not demand a particular implementation. Indeed the search could run entirely at a single site and simply invoke the necessary remote services. Mobile agents allow a straightforward and efficient implementation, however; mobile agents could be dispatched to each resource site and then could communicate partial results and redo their searches without the involvement of the home site (and the corresponding network traffic).

3.1.4 Software interoperation

Here an *agent* is a program that communicates correctly in a *universal* communication language. Since all agents use the same communication language, an agent can interoperate with any other agent, regardless of their underlying implementation. This definition of an agent is closely related to agent-based software engineering in which applications are implemented as a collection of autonomous, cooperating peers. There are two approaches to agent-based interoperation—*procedural* and *declarative*.

In the *procedural* approach, agents exchange *procedural directives*. The recipient agent executes the directives to perform some task on behalf of the sender. Most existing systems that use the procedural approach are based around high-level scripting languages. An application is sent a script that guides the application through the desired task. Notable examples of the script-based approach include Tcl, AppleScript, Hewlett-Packard's NewWave environment and the Autonomous Knowledge Agents (AKA) project [GK94, Joh93].

Genesereth [GK94] points out several disadvantages of the procedural approach. Writing procedures might require information about the recipient that is not available to the sender; procedures only compute in one direction; and procedures are difficult to merge. Instead Genesereth argues for the *declarative* approach in which agents exchange declarative statements. The recipient performs an inference process to derive results from the sender's declarative statements. These declarative statements are written in the Agent Communication Language (ACL). ACL has three components—

a vocabulary, an inner language called the Knowledge Interchange Format (KIF), and an outer language called the Knowledge Query and Manipulation Language (KQML) [GK94]. The vocabulary is a dictionary of words specific to the application area. Each word has an English definition and a set of formal annotations written in KIF. KIF is a prefix version of first-order predicate calculus that can express data, procedures, and relationships among the data. The atoms of KIF are the words from the vocabulary. A KQML expression consists of a directive followed by one or more KIF expressions. Directives include *telling* an agent that a KIF expression is true, *asking* an agent if a KIF expression is true, and so on.

Agents that use KQML can communicate with each other directly, but this places the burden of interoperation squarely on the programmer. Instead Genesereth proposes a federated architecture in which *facilitators* handle interoperation [GSS94]. Essentially each agent is assigned a facilitator. An agent communicates only with its facilitator, but facilitators communicate with each other. Each agent posts its *capabilities* and application-specific facts to its facilitator. When an agent needs information, it sends a request to its facilitator. The facilitator uses backward inference to find an answer to the request; typically the facilitator will invoke the services of other agents during this process. The advantage of the facilitator approach is that each agent communicates with single system agent that *appears* to handle all requests itself. The main concern with the facilitator approach is scalability, i.e. the size of the shared vocabulary, the cost of the inference process, and the size of the facilitator's knowledge base. The first problem is addressed by allowing agents to use different vocabularies and providing translation features; the second and third problems are addressed by limiting the amount and kind of information that each facilitator stores internally.

The federated approach is similar to directory assistance, distributed object managers and automatic brokers. Directory assistance allows a program to find a desired service. Distributed object managers provide transparent access to a distributed collection of objects; messages are automatically routed to the destination object even if the sender does not know the object's network location. Automatic brokers provide both functions by first identifying an appropriate recipient for a message and then forwarding the message. An example of directory assistance is the X.500 protocol; distributed object managers include CORBA, DSOM, OLE and OpenDoc; automatic brokers include ToolTalk and the Publish and Subscribe Service on the Macintosh [GK94]. The federated approach is distinguished by the amount of processing done in the facilitator; each facilitator performs backward inference rather than simple pattern matching. [GK94].

It is unclear whether the procedural or declarative approach is better. The procedural approach is suggested when the sender is requesting a task that the recipient does not know how to perform in its entirety. The declarative approach is suggested for knowledge-intensive applications in which planning and inference are required. Mobile agents represent a hybrid approach. The mobile agents are procedures that migrate to a remote machine so that they can execute at the location of the data, but they do not have to communicate with procedural directives. The communication facilities in most mobile-agent systems are flexible enough to support any communication protocol, including the exchange of declarative statements. In addition, agents can make use of external services that provide planning and inference capabilities. There is no need to build declarative logic into the agent language.

One of these services—directory assistance, distributed object managers, automatic brokers or federated inference engines—will be essential in a mobile agent system so that an agent can (1) find agents that perform a needed task and (2) communicate with agents without knowing their current network location. Since mobile agents are *mobile*, keeping track of recipient locations without system support would be a nearly intractable programming challenge even if all the agents came from the same application and communicated only among themselves. Whichever solution is adopted, it must be extended so that it performs effectively in a highly dynamic environment. Mobile agents come into existence, change network location and terminate continuously. These changes must be visible to other agents.

3.2 Mobile agents

A mobile agent is an autonomous program that can migrate under its own control from machine to machine in a heterogeneous network. The history of mobile agents is complex. Here we examine the clear forerunners to mobile agents and discuss existing mobile-agent systems.

3.2.1 Message passing

The message-passing model provides two communication primitives: `send`, which sends a message to a destination process, and `receive`, which receives the message. In client/server computing, the client sends a request message to the server; the server receives the message, handles the request and sends back a response. The `send` and `receive` primitives can be *blocking* or *nonblocking* and *synchronous* or *asynchronous*. *Blocking* means that the primitives do not return control to the caller until the message has been successfully sent or received; *nonblocking* means that the primitives return immediately. *Synchronous* means that the send primitive does not return until the recipient issues a corresponding receive; *asynchronous* means that the send primitive returns as soon as the message arrives on the destination machine. Message passing is powerful and flexible, but requires the programmer to handle low-level details, such as keeping track of which response goes with which request, converting data between client and server formats, determining the address of the server, and handling communication and system errors [SS94].

3.2.2 Remote procedure call (RPC)

Remote procedure call (RPC) [BN84] relieves the programmer of these details. RPC allows a program on the client to invoke a procedure on the server *using the standard procedure call mechanism*. Most RPC implementations use *stub* procedures [BN84, SS94]. A client that makes a remote procedure call is actually calling a local stub. This client stub puts the procedure name and parameters into a message and sends the message to the remote machine. A server stub on the remote machine receives the message, extracts the procedure name and parameters, and invokes the appropriate procedure. The server stub waits for the procedure to finish and then sends a message containing the result to the client stub. The client stub returns the result to the client. The original RPC implementations blocked the client until the server returned the result. Current RPC extensions either allow concurrent invocation of procedures on multiple servers or make RPC asynchronous [SS94]. These variations are more flexible but make programming more difficult.

Other disadvantages of traditional RPC were described in [GG88]. It is difficult to send incremental results from the server to the client; implementations are commonly optimized for short results rather than bulk data transfer; and there is no way to pass pointers or procedure references to the server. This last limitation obviates any protocol that requires the server to invoke a client-specified procedure *on the* *client machine*. [GG88] addresses these problems by allowing procedure references to be passed as arguments and introducing the *pipe* abstraction. A *pipe* is a connection between the client and the remote procedure that exists for the duration of the remote procedure call; incremental results and bulk data are transferred along the pipe.

3.2.3 Remote evaluation

The main problem with RPC is that the client is limited to the operations provided at the server. Since the server often does not provide an operation that meets the client's needs exactly, the client must make several remote procedure calls, bringing intermediate data across the network on every call. If the intermediate data is not useful beyond the end of the client's task, a significant amount of network bandwidth has been wasted. To address this problem, researchers have turned to remote evaluation in which a *subprogram* is sent from the client to the server. The subprogram executes on the server and returns its result to the client.

Falcone [Fal87] describes a system in which clients and servers program each other using a variant of Lisp called the Network Command Language (NCL). Each server provides a library of NCL functions. A client that requires a service sends an NCL expression to the appropriate server. The expression can use any functions provided at the server or sent as part of the expression. The server evaluates the expression and returns the result to the client. The result is an NCL expression itself and can perform arbitrarily complex processing on the client. The NCL expression can invoke multiple functions on either the client or server and thus avoid the overhead of multiple remote procedure calls.

Remote evaluation (REV) is similar to NCL in that a procedure can be sent to a remote server for evaluation [SG90]. REV, however, uses client and server stubs in much the same way as RPC and can be used with any language. All that is needed for a new language is stub generators and linking facilities so that the procedure can invoke the operations provided at the server. The procedure can be transmitted as source, intermediate or compiled code. The choice of transmission format depends on the language, the desired level of security and the heterogeneity of the network. Stamos and Gifford identify four security concerns—authentication, availability, secrecy and integrity. Authentication and availability consist of verifying the identity of the client and preventing denial of service attacks and can be addressed with standard techniques [SG90]. Secrecy and integrity consist of preventing unauthorized access to and destruction of server information and require more complex solutions. Stamos and Gifford present three solutions—separate address spaces for each procedure, careful interpretation in a single address space, and digital signatures with a single address space. The first and third solutions support compiled code while the second and third avoid the overhead of multiple address spaces. Digital signatures are an open research area in which a program is compiled by a trusted third party. The third party checks the program for security violations and applies a cryptographic signature to the compiled code if no security violations are present. The server knows that certain security checks have been performed already if it receives a digitally signed procedure. The advantage of REV over NCL is that REV can be incorporated into any programming language, allowing the programmer to use the most appropriate language for the application. On the other hand, NCL is symmetric since procedures can be sent from the client to the server *and* from the server to the client.

The procedures in NCL and REV must be self-contained. All functions and variables referenced in the procedure must be provided at the server or included in the procedure, making the semantics of a passed procedure different than that of a local procedure. Specifically, the passed procedure cannot access functions and global variables defined in the caller. The developers of REV and NCL were primarily concerned with moving computation to a remote machine and imposed this limitation to simplify the implementation. SUPRA-RPC (SUbprogram PaRAmeters in Remote Procedure Calls), on the other hand, seeks to allow normal procedure call semantics for both local and passed procedures [Sto94]. Essentially SUPRA-RPC extends REV with additional stubs that are invoked whenever the procedure references an out-of-scope variable or function. The server makes a callback to the client to handle the out-of-scope reference. SUPRA-RPC implementations exist for C, C++ and Lisp, but the C and C++ implementations work only in a homogeneous environment, since compiled code is passed from machine to machine.

There are several schemes that can be viewed as a domain-specific form of remote evaluation. Postscript programs are often sent to remote printers and displays. Scripting systems such as Apple Script allow scripts to be sent from one application to another [Joh93]. MIME/Safe-Tcl allows Tcl scripts to be embedded in e-mail messages; the scripts are executed automatically when the message is received or viewed [LO95]. The IBM Intelligent Communications Network uses Intelligent Objects that contain both data and procedures and can be sent from one application to another; the recipient application can execute the embedded procedures [Rei94]. The Decode-Encode language (DEL) allowed an emulator for one terminal type to be transparently downloaded into a different terminal type [Rul69]. The Wit and Wit 2 systems send interface code from a Unix server onto a palmtop device to minimize use of a poor-quality wireless link [Wat95]. Some database systems allow a user to define complex SQL commands and store these commands on a server; the stored commands are executed at the server end during a user transaction [BP88]. The

Bayou filesystem allows a mobile computer to cache files and then continue accessing these cached files while disconnected; when the laptop reconnects, it sends a code fragment to the filesystem server to reconcile its file changes with the permanent copy of the file [TTP+95].

3.2.4 Java applets, Java servlets, Java RMI and JavaOS

Java applets are Java programs that are associated with a World Wide Web (WWW) page [CW97]. When a user views the page with a Java-enabled browser, the program is downloaded automatically and executed on the user's machine. By executing on the browser's machine, the program can present a complex, graphical interface and react rapidly to user actions, since there are no network delays. Untrusted applets are executed in a secure Java interpreter so that they can not access or destroy sensitive information. The *Metis* thin-client framework can be viewed as a generalization of Java applets; here an arbitrary client downloads an application front-end written in Java, which then finds and interacts with one or more network services to realize the complete application [ZPMD97]. Other systems support applets that are written in other languages. The Grail web browser[2] from CNRI, for example, executes applets written in Python, while the Tcl plugin[3] from Sun Microsystems executes applets written in Tcl/Tk.

Java servlets are Java programs that can be dynamically loaded into an executing Java-enabled web server, such as Sun Microsystem's Web Java Server 1.0 [Cha96]. Java servlets allow rapid introduction of new server functionality and are more efficient than CGI scripts, since there is no startup overhead on each access. Like untrusted applets, untrusted servlets are executed in a secure Java interpreter so that they can

[2] http://grail.cnri.reston.va.us/grail/
[3] http://www.sunscript.com/products

not access or destroy sensitive information.

The Java Remote Method Invocation (RMI) subsystem allows Java objects on different machines to invoke each other's methods [WRW96, RWWB96]. It also provides state serialization and unserialization facilities for transferring a Java object from one machine to another. Most Java-based distributed systems, including mobile-agent systems in which the agents are written in Java, use RMI for communication.

Finally, JavaOS is a lightweight operating system that executes Java programs directly [Mad96]. It is targeted towards mass-market devices such as set-top boxes, telephones and network computers, which have limited storage facilities and need to download applications from the network as needed.

3.2.5 Inferno

Inferno is a lightweight, networked operating system from Lucent Technologies that is also targeted towards mass-market devices such as set-top boxes, telephones and network computers [Luc96]. Inferno is Lucent's answer to Java and JavaOS.[4] An Inferno application consists of a set of modules written in a C-like language called Limbo; the modules are compiled into the bytecodes for a RISC-like virtual machine and then downloaded on demand to an Inferno platform. Once downloaded, the modules are compiled "on-the-fly" into native code; execution speed is within a factor of two of natively compiled code. Limbo is not a safe language so modules are digitally signed by trusted authorities. Like most operating systems, however, Inferno itself prevents unauthorized access to system resources.

[4] The Inferno development group would disagree with this statement since they view Inferno as a more complete product. Inferno and JavaOS are roughly equivalent, however, and compete in the same markets.

3.2.6 Mobile agents

Mobile agents extend REV and applets by allowing a program to (1) move through a sequence of machines, carrying its current state along with it, and (2) communicate easily with other such moving programs. The recent popularity of mobile code has led to an explosion in the number of mobile-agent systems. Here we describe a few representative systems in detail and mention other systems briefly.

Kali Scheme [CJK95]. Kali Scheme is an extension of Scheme 48 [KR95], which is an efficient, multi-threaded Scheme implementation based around a bytecode interpreter. Kali Scheme provides a distributed set of address spaces in which the threads execute. New threads can be spawned in either local or remote address spaces, and an existing thread can be migrated from one address space to another, continuing in the new address space from the point at which it left off. When spawning a new thread, Kali Scheme transmits the *closure* that contains the desired computation; when migrating an existing thread, Kali Scheme transmits the current *continuation* of that thread. Since continuations can be extremely large, Kali Scheme divides the continuation into frames and initially transmits only the first few frames to the target address space. Other frames are fetched from the source machine if and when needed. In addition to migration, Kali Scheme allows two threads to exchange arbitrary data objects, and allows a thread to refer to a data object in a different address space via a proxy object. Kali Scheme does not address security issues, since these issues are orthogonal to the author's main goal of providing distributed-computing abstractions within a higher-order language such as Scheme. Kali Scheme's main weakness is that the implementation is Scheme specific (and, in fact, is contained entirely within the Scheme 48 virtual machine itself), preventing any straightforward extension to additional languages.

Messengers [TDiMMH94, DiMMTH95, Tsc94]. A *messenger* is a message that contains both data and code. A server on each host accepts incoming messengers and executes each messenger's code within its own thread. Colocated messengers communicate with each other through a shared dictionary of key/value pairs and synchronize their actions through process queues.[5] A messenger can spawn new messengers and can move through the network by transmitting itself from one machine to another. When moving to a new machine, the messenger restarts execution at a specified entry point in its code, and must decide what to do next based on its current state. A messenger's code can be written in either $M\phi$ or Scheme; $M\phi$ is similar to Postscript except that it has migration and communication commands rather than graphics commands. Messengers are a lightweight mobile-code mechanism and are intended for use in communication protocols and distributed operating systems; in both cases, the protocol and operating system components are dynamically deployed, rather than pre-installed. The Messengers system does not address most security issues, but the Messengers group is working on market-based approaches for fairly allocating resources among competing messengers [Tsc97].

Obliq [Car95, BC95, BN97]. Obliq is an interpreted, lexically scoped, object-oriented language. An Obliq object is a collection of named fields that contain methods, aliases and values. An object can be created at a remote site, cloned onto a remote site, or migrated with a combination of cloning and redirection. Implementing mobile agents on top of these mobile objects is straightforward. An agent consists of a user-defined procedure that takes a *briefcase* as its argument; the briefcase contains the objects that the procedure needs to perform its task. The agent migrates

[5]Only the messenger at the head of the queue is allowed to execute. All other messengers in the queue are blocked until the first messenger puts itself onto the end of the queue.

by moving its procedure and current briefcase to the target machine; the target machine invokes the procedure, which examines the briefcase to decide what to do next. Obliq includes an interface toolkit called Visual Obliq that a migrating agent uses to interact with a user. When the agent migrates, the current state of its displayed interface is captured and recreated exactly on the target machine; although this is an interesting feature, its usefulness is unclear, since it is difficult to imagine an application in which the agent could not easily recreate the display itself. Obliq does not address security issues.

Omniware [LSW95, ATLLW96]. Omniware code is written in C++ (or any other language for which an appropriate compiler exists), compiled for a RISC-like virtual machine, and later sent to a destination machine where it is converted into native code. Software fault isolation, which essentially adds a range check to every memory access, prevents the native code from corrupting the execution environment [WLAG93]. With this arrangement, Omniware provides portable, secure code that is only 25 percent slower than natively compiled C/C++ on average [LSW95]. Therefore, although Omniware is not a general mobile-agent system itself, it or a similar execution environment is likely to find its way into most mobile-agent systems so that these systems can support compute-intensive applications. Omniware could even be the *only* execution environment, since the interpreters for agents written in other languages could be compiled for the Omniware virtual machine and sent as needed to the destination machines [LSW95].

Sumatra [RASS97, RAS96]. Sumatra is an extension to Java [CW97] that supports both distributed objects and mobile code. One or more instances of the Sumatra execution engine run on each machine.[6] Each engine hosts one or more

[6] The Sumatra execution engine is just the extended Java interpreter.

threads and their objects. Each object can be either independent or part of a particular object *group*. A program can move an object group (along with the associated Java bytecodes) from one engine to another, invoke the methods of an object in a different engine[7], create a new thread in a different engine, and migrate an executing thread from one engine to another. When migrating an object group, every local reference to a group object is converted into a proxy reference that will transparently redirect method invocations to the actual object. When migrating a thread, Sumatra captures and transfers the Java stack along will all non-group objects that are reachable from the stack; references to group objects are turned into proxy references on the target machine.[8] Sumatra's most notable feature, and its main research focus, is a set of distributed resource monitors that measure network latency, network bandwidth and machine load; an agent queries these resource monitors and combines current network conditions with its own knowledge of its task to decide if and when to migrate objects and threads. Sumatra does not provide any security mechanisms aside from those already present in Java. Similar to Kali Scheme, its main weakness is that the implementation is Java-specific (and in fact is contained entirely within the Java virtual machine), once again preventing any straightforward extension to additional languages.

Tacoma [JvRS95, JvRS96, MvRSS96, Knu95], Tacoma Too [Sch97a], and security automata [Sch97b] . Tacoma (Tromsø and COrnell Moving Agents) is a mobile-agent system that supports numerous agent languages, including Tcl, Scheme, Perl, Python, Java and C. Each agent and machine has a *briefcase* or *file*

[7]Sumatra was developed before Java's Remote Method Invocation (RMI) package became available. Thus Sumatra has its own remote invocation facility.

[8]The idea is that an object group might be providing a service to multiple client threads; the service should remain stationary even though its client threads are moving from machine to machine.

cabinet of *folders* that contain both data and procedures. Aside from folders and their containers, the single abstraction in Tacoma is the *meet* operation, which an agent uses to request a service from another agent. During the *meet* operation, the requesting agent passes a *briefcase* to the target agent; this briefcase contains the arguments for the request. The target agent returns a folder of results to the requesting agent if necessary. All other services in Tacoma are provided by other agents. For example, an agent migrates to a remote machine by passing a briefcase containing its code and state to the `tac_firewall` agent on the remote machine. The agent does not continue execution from the point at which it left off; instead the `tac_firewall` agent restarts the incoming agent by calling some specified entry point. Although this migration mechanism requires more programmer effort, both to capture the desired state information and to ensure that the appropriate task is performed on each machine, there is no need to have state-capture routines built into the agent languages themselves. In combination with the simplicity of the *meet* abstraction, the ability to use unmodified interpreters allows the rapid integration of new languages into Tacoma. Important features of Tacoma are "rear-guard" agents, electronic cash, brokers, and its use of the Horus toolkit for reliable group communication. A rear-guard agent is left behind whenever an agent migrates to a new machine; this rear guard restarts the agent if the agent "vanishes" due to a machine failure [JvRS95]. Electronic cash is used to pay for services and to prevent runaway agents.[9] In addition, agents can be digitally signed and encrypted, and the `tac_firewall` agent can be instructed to reject agents that came from unauthorized users. Broker agents provide directory services. Most of these features are not available in the public release.

[9]Runaway agents are impossible since an agent cannot continue once its financial reserves are exhausted.

Tacoma Too is a version of Tacoma that is based around the ML language [Sch97a]. Tacoma Too has the same *meet* abstraction as Tacoma and is being used in a prototype active network. An offshoot of the Tacoma Too project is concerned with securely executing Java agents using software fault isolation and security automata [Sch97b]. A security automata enforces a security policy and is similar to a finite-state automata. Transitions between states correspond to agent actions. A security exception is raised if an agent attempts an action for which the current state has no outgoing transition. For example, suppose that an agent is allowed to communicate with its home machine only if it has not read from a certain file. Then the initial state in the security automata would have two transitions: (1) communicating with the home machine, which simply loops back to the initial state, and (2) reading from the file, which leads to a second state. The second state would have one transition: (1) reading from the file, which simply loops back to the second state. Thus, if an agent reads from the file and then *attempts* to communicate with the home machine, the security automata will raise a security exception, since the second state has no outgoing transition for communicating with the home machine. Tacoma Too is also exploring various forms of proof-carrying code [Sch97b].

Telescript [Whi94, Whi95b, Whi95a, Whi96] and Odyssey [Gen97]. Telescript, later marketed as part of the Tabriz web-server package, was the first commercial mobile-agent system. It was developed at General Magic, Inc., and was primarily used in the AT&T PersonaLink network. In Telescript, each network site is divided into one or more virtual *places*, much like the address spaces of Kali Scheme and the multiple execution engines of Sumatra. Telescript agents are written in an imperative, object-oriented language that is similar to both Java and C++; this language is compiled into virtual-machine bytecodes. A Telescript agent uses the *go* command to migrate from one place to another, continuing execution from the point at which it left off. An agent can interact with other agents in two ways. The agent can *meet* with an agent that is in the same place; the two agents receive references to each other's objects and communicate by invoking each other's methods. In addition, an agent can *connect* to a remote agent; the two agents pass objects along the connection.

Each network site runs a Telescript server that maintains the places at the site and executes incoming agents. The engine continuously writes the internal state of executing agents to nonvolatile store so that the agents can be restored after a node failure. The engine also provides two security mechanisms. First, each agent carries cryptographic credentials that the place uses to authenticate the identity of the agent's owner. Second, each agent carries a set of permits that give it the right to use certain Telescript instructions and certain amounts of available resources. One permit, for example, might specify a maximum agent lifetime or a maximum amount of disk space. Each engine and place impose their own permits on incoming agents to prevent these agents from taking malicious action. Agents that attempt to violate the conditions of their permits are terminated immediately [Whi94]. Despite the fact that until recently Telescript was one of the most secure, fault-tolerant, and efficient mobile-agent systems, is has been withdrawn from the market, mainly because it was overwhelmed by the rapid spread of Java. The AT&T PersonaLink network is also defunct.

Odyssey is General Magic's replacement for Telescript. It is essentially the same system except that it is implemented entirely in Java. One notable exception is that Odyssey does not have the *go* instruction, since Java does not provide facilities for capturing an executing program's complete state, making it impossible to implement the *go* instruction without modifying the Java virtual machine. General Magic decided that a modified virtual machine would prevent the widespread acceptance and use of Odyssey. Thus, although an Odyssey agent does carry all of its objects along with it, it must either restart execution on the destination machine or follow an itinerary in which specific methods are executed at specific destinations. In the first case, the agent examines the current state of its objects to decide what to do next. In the latter case, the agent system automatically invokes the correct method; the agent specifies its itinerary before its first migration and can modify the itinerary at any time. Like Sumatra, Odyssey is intimately tied to the Java language with no clear way to integrate additional languages. General Magic is not selling Odyssey; instead they are using it as a key *internal* component of their new personal messaging service.

Other Java-based systems. The *IBM Aglets* system [LO97] is one of the more complete commercial offerings. It provides authentication and access control, a globally unique namespace, whiteboards and message passing, and a simple management environment. IBM Aglets is extremely similar to the Odyssey system, and, in fact, the developer of IBM Aglets now works in the Odyssey group at General Magic.

The *Liquid Software* project [HMPP96] has two goals: (1) use mobile agents to efficiently solve large-scale information-retrieval problems and (2) develop a "gigabit" compiler that can verify and compile an intermediate code representation as fast as it arrives over the network, producing efficient, secure native code while hiding the compilation latency. To explore potential solutions, the project members are building a prototype system that runs on top of the Scout operating system and uses Java bytecodes as the intermediate code representation. The prototype will also address security issues and the granularity of the interface between the agents and the underlying operating system.

The evolving *Mobile Objects and Agents (MOA)* system is a CORBA-based system in which both static and active Java objects[10] can move from one machine to another [CMB96, MBZM96]. CORBA is a distributed-object manager that already provides object naming, object location and remote invocation [YD96]. MOA builds on CORBA to provide object persistence, a hierarchical object cache, and a location-independent name service that uses URL's. In addition, MOA conforms to the new Mobile Agent Facility (MAF) standard [MAF97]. MOA is currently used in the Distributed Client project in which an application is partitioned into client-side and network-side components; these components cooperate when the network is available but continue operation even when the network is disconnected, presenting previously retrieved data to the user and finding additional data to send back to the user as soon as the network reconnects [MBZM96]. MOA will eventually enforce access restrictions according to the mobile object's owner, its current requirements, and possibly its past migration history.

FTP Software's *CyberAgents* provided a visual editor, a visual agent manager, some debugging tools, OLE and HTML support, and extensive logging and report-generation facilities [FTP96]. FTP Software no longer sells CyberAgents, but a version of CyberAgents that uses the *TRAC* language (rather than Java) is used in some of FTP Software's network management tools [Gre97a].

Other Java-based systems include Mole [SBH96, BHR+97], Concordia [Mit97a, WPW+97], Voyager [Voy97], and Wasp [Fun97]. The Mole project is focusing on security mechanisms for protecting an agent from a malicious machine, while the Wasp project is concerned with integrating mobile agents with Web servers. Voyager and Concordia are commercial systems. Concordia is similar to Odyssey and IBM Aglets, while Voyager combined mobile code with a full-featured Object Request

[10] An *active* object is an object with its own thread of control.

Broker (ORB).

Other systems. There are many other mobile-agent systems, most of which provide minimal security. *Ara* agents are written in Tcl or C++ (which is compiled into an interpreted bytecode called MACE) and can migrate at any point during their execution; Ara allows an agent to checkpoint its state to disk and enforces limits on CPU time and memory usage, but does not yet protect resources such as the filesystem and network [PS97, Pei96]. The Ara group is currently implementing additional security mechanisms (such as digital signatures and access restrictions for all system resources) and adding support for the Java language.

Tripathi and Karnik propose a mobile-agent system that uses CORBA as its lowest layer [TK93]. The proposed *Distributed Internet Execution Environment (Dixie)* will combine the Prospero file system, the Tk toolkit, and interpreters for several languages into a virtual operating system that accepts and executes applications sent from other hosts [Gai94]; little implementation work had been done at the time of this writing, however. The *Frankfurt Mobile Agents Infrastructure (ffMAIN)* [LDD95] allows agents written in Tcl to migrate and communicate using the standard HTTP protocol; their agent server is a modified HTTP server.

Intelligent routers are written in an interpreted expression language called MPL.1 and can migrate from machine to machine with a *moveto* instruction; a version of the router system that runs on homogeneous machines uses ISIS to detect and recover from node failures and other faults [WVF89, Voo91]. *IBM Itinerant Agents* is a proposed system that focuses on knowledge-based routing of service requests and security issues [CGH$^+$95]; it is not under active development and has given way to the more recent IBM Aglets system [LO97]. *LogicWare* [Log96] supports collaborative applications through an active object space; this object space can include mobile agents called *Mubots* that move from machine to machine in response to changing network conditions and the location of the participants.

Agents in the *MESSENGERS*[11] system are written in C, compiled into machine-independent assembly code, and then interpreted at each node; the agents construct a logical network on top of the physical network as they execute [BFD96]. *Mobile Service Agents (MSA)* are written in an extended version of the functional programming language Facile; an MSA agent spawns a new agent by submitting a *closure* containing data and one or more functions [TLKC95, Kna96].

The *Rover* toolkit combines relocatable objects,[12] queued remote-procedure-calls (QRPC) and stable logging of object state to support fault recovery; although Rover is primarily used in disconnected or partially connected client-server computing, it can be used with some effort as a more general mobile-agent system [JdT+95, JTK97, JK96]. The *Smart Messages* system [HCS97] associates one or more reactive planners with each application-level message; the reactive planners are activated whenever the message is queued somewhere within the network and allow the message to control its own routing, filtering and error recovery.

SodaBot agents are written in a very high-level language called SodaBotl that provides threads, an interface toolkit, location-independent communication primitives, and both automatic code distribution and an explicit *hop* operation; SodaBot agents execute in a restricted execution environment that limits the agent's total lifetime and prevents unauthorized access to the filesystem and external programs [Coe94]. The *Tube* is similar to Kali Scheme in that agents are written in Scheme, execute inside multithreaded servers, and create new agents and migrate from machine to machine

[11] The *MESSENGERS* system is not the same as the *Messengers* system that was described in more detail above. They were developed independently at different universities.

[12] The methods in a Rover object are implemented in the Tcl scripting language for portability.

by transferring a *closure* or *continuation* respectively; in addition, the Tube uses a safe Scheme interpreter and includes noticeboards, a user-interface toolkit, a service registry, an event system, continuous media streams, and a unique form of RPC in which the server sends the client stubs to the client [HBB96].

WAVE represents the ultimate extension of the mobile-agent paradigm with every application implemented as a "wave" of extremely compact, lightweight agents that recursively spread themselves through a virtual network[13]; although WAVE is syntactically awkward and demands a highly recursive programming style, it has been used effectively in several applications, including distributed simulation, network management, and distributed database retrieval [Sap96]. [KK94] implements a simple mobile-agent system in which the agents are written in a scripting language similar to AWK and migrate from machine to machine with a *moveto* instruction; this system was the forerunner to Agent Tcl here at Dartmouth.

3.3 Safe languages

Finally, there are several interpreted languages where either the language itself or specific interpreters provide security features that are attractive in mobile-agent systems; these security features range from restricted namespaces to bytecode verification. Such languages include Java [CH97], Scheme 48 [KR95], Tcl [Ous94, OLW97], Lua [dIC96], and Python [Lut96], as well as Obliq [Car95] and Telescript [Whi94], which were specifically designed for mobile objects or agents. Except for Lua, all of these languages are used in at least one mobile-agent system.

[13]As in the MESSENGERS system, the virtual network is mapped onto a physical network with one or more virtual nodes per physical node. This mapping makes it easy to run the same application on differing numbers of physical machines.

Chapter 4

Trends and distinctions

Existing mobile-agent systems can be compared along several axes.

Research versus commercial. Aside from Telescript [Whi94], most mobile-agent systems were strictly research projects until about two years ago. Since then, the number of commercial systems has increased dramatically and now includes Odyssey [Gen97], IBM Aglets [LO97], Concordia [WPW+97], Voyager [Voy97] and Omniware [LSW95]. As the reader might expect, commercial systems provide much better administration and development tools than research systems. In addition, all commercial systems provide sufficient security mechanisms to protect a machine from malicious agents. Otherwise the system could not be used in an open network environment and would never achieve commercial success. Research systems usually ignore administration, auditing and development tools, and often ignore security if security is orthogonal to the main research interest (such as the use of a high-order language in distributed computing [CJK95]). Agent Tcl is entirely a research project, but does provide sufficient security mechanisms to protect a machine from malicious agents and also includes a full-featured debugger and a *simple* visual programming environment. Aside from the debugger and visual programming environment, however, there are no administration or development tools.

Languages. Nearly all mobile-agent systems use imperative languages, most notably C/C++ [LSW95, JvRS95, PS97, BFD96], Java [Gen97, LO97, RASS97, HMPP96, CMB96, SBH96], and various scripting languages [JvRS95, PS97, JdT$^+$95, Coe94]. Functional languages such as Scheme are used only in a few research systems [CJK95, HBB96, TLKC95]. Java is the most popular imperative language and is used in every commercial system and several research systems. There are three reasons for Java's popularity: (1) its virtual machine architecture makes programs both portable and efficient, (2) its existing security features allow the safe execution of untrusted code, and (3) it enjoys unprecedented market penetration, mainly due to its use in active web pages. Agent Tcl supports both imperative and functional languages, namely Java, Tcl and Scheme, allowing the programmer to pick the language that is most appropriate for his task.

Interpreters versus native code. Currently, for reasons of portability and security, nearly all mobile-agent systems either interpret their languages directly, or compile their languages into bytecodes for some virtual machine and then interpret the bytecodes. Agent Tcl is no exception. Tcl is interpreted directly; Java and Scheme are both compiled into bytecodes for a virtual machine.[1] Due to the widespread recognition that agents must execute at near-native speed to be competitive with traditional distributed-computing techniques in certain applications, however, several researchers are experimenting with "on-the-fly" compilation. The agent is initially compiled into some intermediate code representation but then is compiled into native code on each machine that it visits, either as soon as it arrives [LSW95, HMPP96] or while it is executing [HMPP96]. When the entire agent is compiled upon arrival, software fault isolation (SFI) is typically used to prevent the native code from corrupting the execu-

[1] Agent Tcl uses Scheme 48, an implementation of Scheme based around a virtual machine[KR95].

tion environment and violating security constraints [LSW95]. Software-fault-isolated native code runs only 25 percent slower than natively compiled code [LSW95]; moreover "gigabit" compilers will be able to compile the agent as fast as it arrives over the network, completely hiding the compilation time [HMPP96]. Although Agent Tcl will eventually use "on-the-fly" compilation, either with Java or some other language, we have no immediate implementation plans, preferring instead to do a much more extensive round of performance analysis first.

Migration. Different mobile-agent systems provide different migration mechanisms. Two distinct migration models can be identified:

- *jump*. The system provides a primitive operation called *jump* that automatically captures the executing agent's complete state and sends this state to a new machine; the new machine restores the state and the agent continues execution from the *exact point* of the *jump*. [Whi94]. Systems that use the *jump* migration model include Agent Tcl, Kali Scheme [CJK95], and Telescript [Whi94].

- *known entry point*. The system moves the variables and methods of the agent to the new machine, and then restarts agent execution at some specified method [LO97]. With an object-oriented language, the typical system automatically captures the complete state of all existing objects [LO97]; with other languages, the typical system requires the programmer to explicitly assemble a package of variables and methods [JvRS95]. To ease the burden on the programmer, many systems allow the agent to follow a pre-established *itinerary*, which specifies a list of machines and the method that should be executed on each one [LO97, Gen97]. All commercial systems use the *known entry point* model with an itinerary, simply because all commercial systems use Java, which cannot support the *jump* model without modifications to the standard virtual machine. Every system that supports the *jump* model also supports the *known entry point* model.

In addition to these two migration models, many systems that use object-oriented languages allow an agent to move an individual object to a remote machine and then invoke that object's methods; the object does not have its own thread of control [LO97, Gen97, Car95, RASS97].

Which of the two migration models is best remains unclear. The *jump* model is more convenient for the end programmer since she does not have to explicitly check the current state and figure out what to do next at each entry point. At the same time, the *jump* model requires more effort from the system developer, since the complete state of an agent must be captured. Modifying interpreters to support this complete state capture is time-consuming and unattractive in a commercial setting. The ultimate success of the *jump* model likely rests on whether the developers of popular mobile-agent languages such as Tcl and Java can be convinced to add the necessary state capture routines to their interpreters. Agent Tcl uses modified Tcl, Scheme and Java interpreters and supports both migration models.

Communication. Communication mechanisms break down in two ways: low-level versus high-level, and location-dependent versus location-independent. For example, IBM Aglets provides messages, byte streams, and remote method invocation [LO97], while Tacoma provides the single *meet* operation [JvRS95]. Tacoma requires a sender to know the current machine of the recipient [JvRS95], while IBM Aglets provides a globally unique namespace [LO97]. There has been wide disagreement over which communication mechanisms are best for mobile agents. Our viewpoint is simple:

low-level mechanisms are *too* low-level for many agents, forcing the programmer to reinvent the desired protocol, while high-level mechanisms are *too* high-level for many others, forcing the programmer to either accept unnecessary overhead or communicate outside the agent framework. Thus, Agent Tcl provides bytestreams, message passing, and location-independent addresses at its lowest level. Higher-level protocols are implemented at the agent level; currently Agent Tcl implements an RPC protocol and a simple speech-act protocol that will be replaced with KQML.

Security. Existing mobile-agent systems focus on protecting an *individual* machine from malicious agents (or a group of machines that are under single administrative control) [Whi94, Gen97, PS97]. Typically the agent's owner or sending machine digitally signs the agent; the receiving machine verifies the digital signature, accepts or rejects the agent based on its signature, assigns access restrictions to the agent based on its signature and migration history, and then executes the agent in a secure execution environment that enforces the restrictions. Aside from encrypting an agent in transit and allowing an agent to authenticate the destination machine before migrating, most existing systems do not provide any protection for the agent or for a group of machines that are *not* under single administrative control.[2] One notable exception is Tacoma which uses rear guard agents to regenerate agents that suddenly disappear, various replication and voting schemes to handle malicious machines that provide incorrect information, and electronic cash to prevent an agent from living forever [JvRS95, MvRSS96]; these mechanisms are only a fraction of a complete solution. Agent Tcl is in the same state as other mobile-agent systems, successfully

[2] A machine might insert new code into an agent, modify the agent's state, terminate the agent, or reroute the agent to a new destination; an agent might migrate forever between two machines or send one child agent to every machine on the Internet.

protecting a machine from malicious agents, but not yet protecting an agent from malicious machines or a group of machines that are not under unified control.

Fault tolerance. Most systems provide only a nonvolatile store so that agents can live past machine failure [Whi94]. Tacoma, however, provides rear guard agents that restart vanished agents [JvRS95]. In addition, the same voting and replication schemes that allow Tacoma to partially handle malicious machines also allow agents to continue with their task even if one or more copies of a desired service are unavailable. Fault tolerance was not addressed in the initial phase of the Agent Tcl project, and Agent Tcl currently provides no fault-tolerance mechanisms, although a nonvolatile store is slowly being integrated.

Chapter 5

Implementation – Base system

5.1 Overview

Agent Tcl has four main goals:

- Reduce migration to a single instruction, `jump`, and allow this instruction to occur at arbitrary points. The instruction should capture the complete state of the agent and transparently send this state to the destination machine. The programmer should not have to explicitly collect state information, and the system should hide all transmission details, even if the destination machine is a mobile computer that is temporarily disconnected or has changed its network address.

- Provide communication mechanisms that are flexible, efficient and low-level, but that hide all transmission details, including whether the agents are on the same or different machines. These mechanisms should be on the level of message queues or byte streams. Higher-level mechanisms, such as remote method invocation (RMI), whiteboards and KQML, should not be implemented in the base system itself, but rather at the *agent-level* on top of the lower-level mechanisms. This layered approach allows cooperating agents to use the communication mechanism that is most appropriate for their task, either one of the base mechanisms or a higher-level mechanism available through a communication-services agent. Providing only a higher-level mechanism in the base system would force many agents to fit their communication into an inappropriate protocol or to communicate outside of the agent framework, which impose severe efficiency and portability penalties respectively.[1] At the same time, if a higher-level mechanism such as RMI is used heavily, it can be moved into the base system alongside the lower-level mechanisms. For example, all Java-only mobile-agent systems provide RMI in the base system, since it is one of the most effective ways for object-oriented Java programs to communicate.

- Use a high-level scripting language as the main agent language and TCP/IP as the main transport mechanism, but support multiple languages and transport mechanisms, and allow the *straightforward* addition of a new language or transport mechanism. Multiple languages are particularly important since, although a high-level scripting language such as Tcl is appropriate for many agents, it is ill-suited for agents that require large amounts of code or that perform speed-critical tasks.

- Provide effective security and fault-tolerance in the uncertain world of the Internet.

[1] It is surprising how many projects not only make this mistake but go out of their way to criticize those systems that *do* provide lower-level mechanisms. The mistake and criticism seems to arise from considering only a limited set of applications and from the misconception that providing only lower-level mechanisms in the base system means that all agents must use those lower-level mechanisms directly.

The overall goal is an efficient, robust and secure mobile-agent system that will allow the programmer to select the most appropriate language for her task and rapidly develop even large-scale distributed applications.

The architecture of Agent Tcl is shown in Figure 5.1. The architecture builds on the server model of Telescript [Whi94], the multiple languages of Ara [Pei96], and the transport mechanisms of two predecessor systems at Dartmouth [Har95, KK94]. The architecture has five levels. The lowest level is an API for the available transport mechanisms. The second level is a server that runs at each network site. The server performs the following tasks:

- *Status and administration.* The server keeps track of the agents that are running on its machine and answers queries about their status. The server also allows an authorized user to suspend, resume and terminate a running agent.[2]

- *Migration.* The server accepts each incoming agent, authenticates the identity of its owner, and passes the authenticated agent to the appropriate interpreter. The server selects the best transport mechanism for each outgoing agent.

- *Communication.* The server provides a two-level namespace for agents and allows agents to send messages to each other within this namespace. The first level of the namespace is the network location of the agent; the second level is a location-unique integer that the server picks for the agent or a location-unique symbolic name that the agent picks for itself. Location-independent namespaces are provided at the *agent level.* A message is an arbitrary sequence of bytes with no predefined syntax or semantics except for two types of distinguished

[2]The suspend and resume operations have not been fully implemented. The next section discusses the current status of the Agent Tcl implementation.

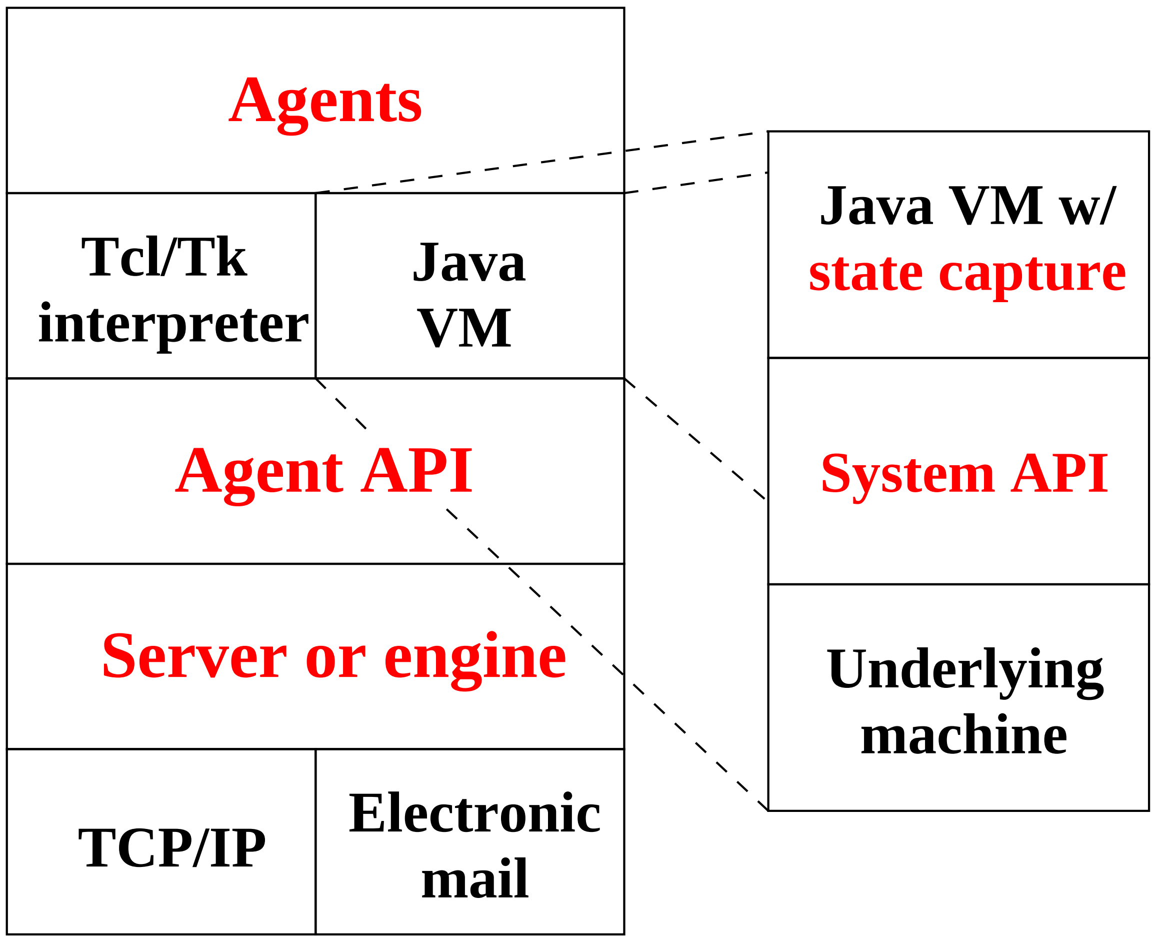


Figure 5.1: The architecture of Agent Tcl. The five levels consist of an API for the available transport mechanisms, a server that accepts incoming agents and mediates agent communication, a language-independent core, an interpreter for each supported language, and the agents themselves.

messages. An *event* message provides asynchronous notification of an important occurrence while a *connection* message requests or rejects the establishment of a *meeting*. A meeting is a named message stream between agents and is more convenient and efficient than message passing (since the programmer can watch for messages on a particular stream and the server often can hand control of the stream to the interpreter). The server buffers incoming messages, selects the best transport mechanism for outgoing messages, and creates a named message stream once a connection request has been accepted.

- *Nonvolatile store*. The server provides access to a nonvolatile store so that agents can back up their internal state as desired. The server will restore the agents from the nonvolatile store in the event of machine failure.[3]

As in Tacoma all other services are provided by *agents*. Such services include resource directories, network sensing, location-independent naming, higher-level communication, and access control. The most important service agents in the implemented system are *resource manager* agents that guard access to critical system resources such as the screen, network and disk. These resource managers are discussed in the security chapter.

The third level of the Agent Tcl architecture is a language-independent core that connects each agent to the server. The core provides the following operations (in cooperation with the server):

- `begin` and `end`. An agent calls the `begin` operation to register with the server and the `end` operation to unregister. The agent can use the other agent operations only while it is registered.

[3]The nonvolatile store has not been fully implemented. The next section discusses the current status of the Agent Tcl implementation.

- `jump`. An agent calls the `jump` operation to migrate to a new machine. The `jump` operation captures the complete state of the executing agent and sends the state image to the new machine. The server on the new machine loads the state image into the appropriate execution environment and resumes agent execution from the point of the `jump`. The copy of the agent on the original machine terminates as soon as the state image is delivered successfully to the new machine.

- `fork`. The `fork` operation is the same as the `jump` operation except that it clones the agent onto the new machine. Both copies of the agent continue execution from the point of the *fork* operation.

- `submit`. The `submit` operation creates a new agent on the local or a remote machine. The new agent is specified as a collection of language-specific objects. A new Tcl agent, for example, is specified as an initial script and a set of Tcl variables and procedures. A new Java agent is specified as a set of Java objects with the method of one object designated as the agent's entry point. In all cases, the `submit` operation sends the new agent to the destination machine where it is loaded into the appropriate interpreter and executed.

  `name`. The agent uses the `name` operation to register a unique symbolic name with the local server.

  `send` and `receive`. The `send` and `receive` operations allow agents to send and receive messages. The recipient is identified by the machine on which it is executing along with either the unique integer that its server assigned to it or the unique symbolic name that it requested for itself with the `name` operation. The `send` operation is asynchronous and simply delivers the message to the recipient's server; the server buffers the message until the recipient calls the `receive` operation. The `send` and `receive` operations have two variants: one which sends a normal message and one which sends a high-priority event message. In both cases, if an agent wants to receive messages only from certain source agents, it can set a *message mask* so that only messages from those source agents can be received. Messages from other agents are buffered until the agent resets or changes the *message mask*.

- `meet`. An agent uses the `meet` operation to request a meeting with another agent. The recipient is specified in the same manner as when sending a message, the machine plus either the unique integer or unique symbolic name. The `meet` operation itself blocks until the recipient either accepts or rejects the meeting, but the core provide several additional operations that allow an agent to establish the meeting asynchronously.

- `status`. The `status` operation returns information about the other agents that are executing on the local machine. It returns either a list of all agents or the owner, sending machine, etc., for a specific agent.

- `notify`. The `notify` operation asks the server to notify the agent when some other agent comes into existence or terminates. The notification takes the form of a high-priority event message.

- `select`. The `select` operation allows the agent to wait for an incoming message (either coming through the server or across a meeting) or for input on an arbitrary file descriptor.

- `suspend`, `resume` and `force`. An authorized agent uses these routines to suspend, resume and terminate other agents.[4]

- `checkpoint`. An agent uses the *checkpoint* operation to backup its current state to nonvolatile store.[5]

All of these operations are subject to authorization checks and resource limits. These checks and limits are discussed in the security chapter. In addition, all operations that involve a remote machine, such as `send` and `jump`, block until either an agent-specified timeout expires or the remote machine returns an acknowledgment. Nonblocking versions of the operations are certainly useful in some agents and will eventually be implemented; the nonblocking versions will use the standard technique of returning a redeemable *future*. The core also includes a cryptographically-secure random number generator that is used in the encryption subsystem and made available at the agent level for use in electronic-cash protocols, agent-specific encryption and so on. Finally, if the language supports event-driven programming, the core can be told to generate events in response to incoming messages. These events are dispatched during the language's event loop. Typically, an agent that uses an event loop would not use the `select` operation.

The fourth level of the Agent Tcl architecture consists of one interpreter for each supported language. We say *interpreter* since it is expected that most of the languages will be interpreted due to portability and security constraints (although "just-in-time" compilation is feasible for languages such as Java, and Omniware uses software

[4] The `suspend` and `resume` operations have not been fully implemented. The next section discusses the current status of the Agent Tcl implementation.

[5] The `checkpoint` operation has not been fully implemented. The next section discusses the current status of the Agent Tcl implementation.

fault isolation to securely execute native code [LSW95]). Each interpreter has four components: the interpreter itself, a security module that prevents unauthorized access to system resources such as the file system, a state module that captures and restores the internal state of an executing agent, and a set of stub routines that provide access to the generic core. Adding a new language to Agent Tcl consists of writing the security module, the state-capture module and the stub routines. The security module does not determine access restrictions but instead ensures that an agent does not bypass the resource managers or violate the restrictions imposed by the resource managers. The state-capture module must provide two functions for use in the generic core. The first, *captureState*, takes an interpreter instance and constructs a machine-independent byte sequence that represents its internal state. The second, *restoreState*, takes the byte sequence and restores the internal state.

Ara, which also supports multiple languages, additionally requires each interpreter to implement a set of scheduling operations and to allocate memory and access systems resources through functions defined in the core [PS97]. Agent Tcl does not need the scheduling operations since it executes each agent within its own process and relies on the underlying Unix system to schedule the processes. We hope to avoid these scheduling operations even when we make Agent Tcl multi-threaded. In addition, instead of providing memory allocation and system access functions in the core, Agent Tcl requires each language to implements its own security enforcement module. Although this approach requires more coding work, the enforcement modules for our chosen languages can be implemented without any modifications to the standard language interpreters.[6] Thus, the only modifications to the standard interpreters in the

[6]For example, the most recent version of Tcl includes Safe Tcl, which allows security checks to be associated with dangerous commands. Similarly, Java provides a security manager class that

current system are the addition of the state-capture routines, making it much easier to migrate from one interpreter version to the next. For this reason, especially with only three supported languages, we do not feel that it is worthwhile to take the Ara approach and route all memory allocation and system access calls through the core. As Agent Tcl matures and supports more languages, however, we will need to adopt the Ara solution so that we do not have to reimplement the enforcement module for each language, something that is quite time-consuming.

The top level of the Agent Tcl architecture consists of the agents themselves.

5.2 Current status

Agent Tcl currently supports three languages, Tcl, Java, and Scheme, but has not been completely implemented.

- The Java and Scheme security modules and the Scheme state-capture module are not complete.
- TCP/IP is the only transport mechanism.
- The nonvolatile store and the `checkpoint` operation have not been implemented. The nonvolatile store and other fault-tolerance issues are discussed in the future work chapter.
- The `suspend` and `resume` operations have not been implemented.

The rest of the architecture has been fully implemented. In addition, several service agents exist, including the resource managers, a docking system that allows

performs security checks before every system access. These existing, builtin mechanisms are sufficient to implement Agent Tcl's current security model. Advantages and limitations of the current security model are discussed further in the security chapter.

agents to transparently migrate between mobile computers, a yellow pages directory that allows an agent to find an appropriate service agent through keyword search or interface matching, a network-sensing agent that tracks current network latency, bandwidth and connection status, and an RPC agent (and library) that allows agents to communicate with each other with the equivalent of RPC calls. In the rest of this chapter, we describe the Tcl, Java and Scheme subsystems. The next chapter covers the resource managers and the other security mechanisms. Chapter 8 discusses the other service agents mentioned above.

5.3 Agent Tcl

5.3.1 Tcl

Tcl is a high-level scripting language that was developed in 1987 and has enjoyed enormous popularity [Wel95]. A sample Tcl program is shown in Figure 5.2. Tcl has several advantages as a mobile-agent language. Tcl is easy to learn and use due to its elegant simplicity and an imperative style that is immediately familiar to any programmer. Tcl is interpreted so it is highly portable and relatively easy to make secure. Tcl can be embedded in other applications, which allows these applications to implement *part* of their functionality with mobile Tcl agents. Finally, Tcl can be extended with user-defined commands, which makes it easy to tightly integrate agent functionality with the rest of the language and allows a resource to provide a package of Tcl commands that an agent uses to access the resource. A package of Tcl commands is more efficient than encapsulating the resource within an agent and is an attractive alternative in certain applications.

Tcl has several disadvantages. Since Tcl is a high-level interpreted language, it

```
proc factorial n {
  set product 1
  for {set i 2} {$i <= n} {incr i} {
    set product [expr $product * $i]
  }
  return $product
}
set n 10
set fac [factorial $n]
```

Figure 5.2: A Tcl script that computes the factorial of a given number n

is much slower than native machine code. In addition, Tcl provides no code modularization aside from procedures, which makes it difficult to write and debug large scripts. These disadvantages have not been a hindrance so far since mobile agents tend to involve high-level resource access wrapped with straightforward control logic, a situation for which Tcl is uniquely suited. A mobile Tcl agent is usually short even if it performs a complex task, and is often more than efficient enough when compared to resource and network latencies. In addition, several groups are working on structured-programming extensions to Tcl and on faster Tcl interpreters [Sah94]. Tcl is clearly not suitable for every mobile-agent application, however, such as performing search operations against large, distributed collections of numerical data. For this reason, Java was added to the system as discussed below. Java is much more structured than Tcl and has the potential to run within a small factor of native speed through "just-in-time" compilation. We expect, however, that Tcl will continue to be the main agent language and that Java (or an even faster execution environment such as Omniware) will be used only for speed-critical agents.

5.3.2 State capture

The main disadvantage of Tcl is that it provides no facilities for capturing the *complete* internal state of an executing script. Such facilities are essential for providing transparent migration at arbitrary points. Adding these facilities to Tcl was straightforward but required the modification of the Tcl core. The basic problem is that the Tcl core evaluates a script by making *recursive* calls to `Tcl_Eval`. The handler for the `while` command, for example, recursively calls `Tcl_Eval` to evaluate the body of the loop. Thus a portion of the script's state is on the *C* runtime stack and is not easily accessible. Our solution adds an explicit stack to the Tcl core. We split the command handlers into one or more *subhandlers* where there is one subhandler for each code section before or after a call to `Tcl_Eval`. Each call to `Tcl_Eval` is replaced with a push onto the stack. `Tcl_Eval` iterates until the stack is empty and always calls the current subhandler for the command at the top of the stack. The subhandlers are responsible for specifying when the command has finished and should be popped. Figure 5.3 illustrates this process for the `while` command.

It is important to note that our modified Tcl core is *fully* compatible with the standard Tcl core. A command procedure that makes a recursive call to `Tcl_Eval` will work correctly on top of the modified core; it will just be impossible to capture the script's complete state when that command procedure is on the invocation stack. This means that existing Tcl extensions will work without modification (as long as the extension does not use the `tclInt.h` header file). An extension has to be modified only if the developer wants an agent to be able to carry the extension's state from machine to machine. In this case, the developer must make the same changes as for the `while` command and must provide callback routines for state capture and restoration.

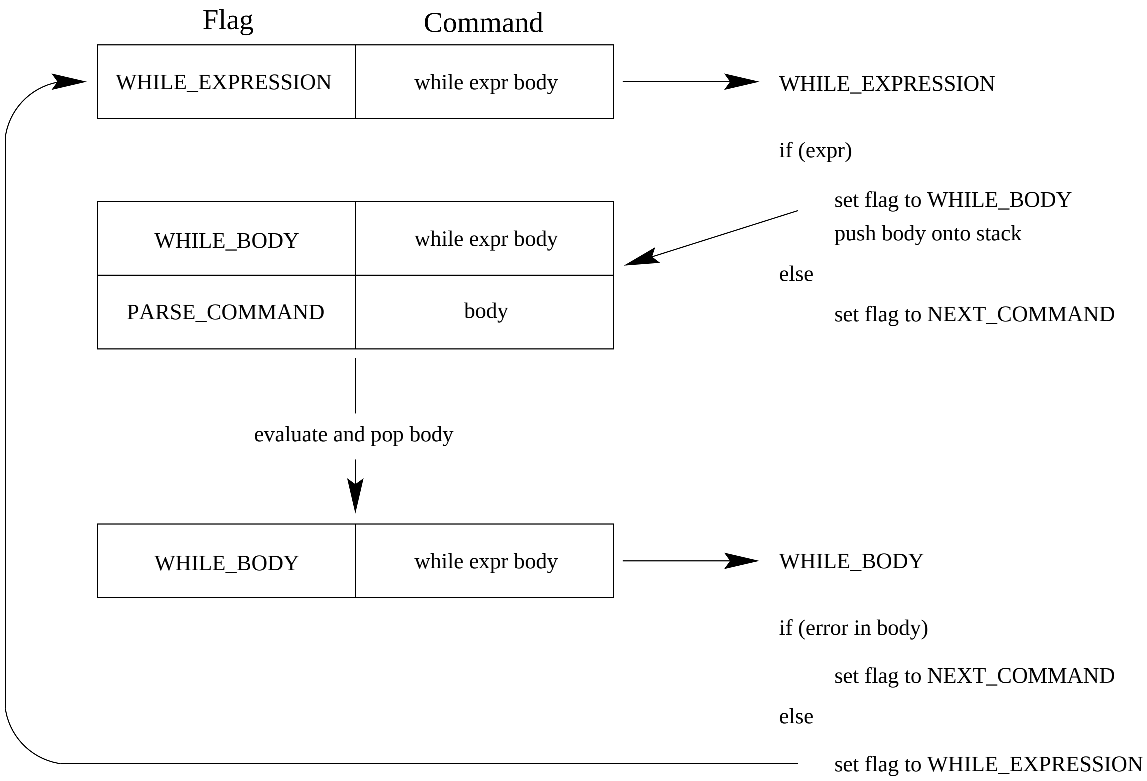


Figure 5.3: An example of how the stack works. The command stack is on the left and the two subhandlers for the `while` command are on the right. A subhandler sets the `NEXT_COMMAND` flag when the `while` command has finished and should be popped. (The actual implementation duplicates the `WHILE_EXPRESSION` code inside `WHILE_BODY` to avoid an extra iteration through the evaluation loop.)

The explicit stack is simpler and more flexible than the Ara solution, in which the C runtime stack must be captured in a portable way, and in which the *same version* of the Tcl interpreters must be present on both the source and destination machines [Pei96]. On the other hand, the explicit stack is less efficient. Our modified Tcl core runs Tcl programs approximately 10 percent slower than the standard Tcl core, whereas Ara's modified Tcl core imposes no significant overhead. It appears that our performance penalty can be reduced significantly with additional optimization, however, and it would also be possible to include both the standard and modified Tcl cores within the same interpreter so that an agent could run on top of the standard, faster core if it did not want to migrate in mid-execution.

Once the explicit stack was available, it became trivial to write procedures that save and restore the internal state of a Tcl script. These two procedures are the heart of the state-capture module for the Tcl interpreter. They capture and restore the stack, the procedure call frames, and all defined variables and procedures. Such things as open files and linked variables are currently ignored.[7]

The advantages of Tcl are strong and the disadvantages are either easily overcome or are unimportant in many agents. Thus Tcl was chosen as the main language for the Agent Tcl system. The same advantages have led to the use of Tcl in other mobile-agent systems such as Tacoma [JvRS95] and Ara [Pei96].

```
set query {mobile agents}
set machines {muir tenaya tioga tuolomne}
agent_begin          # register with the agent system
foreach machine $machines {
  agent_jump $machine
  agent_send "$machine query_engine" 0 $query
  set sender [agent_receive code result -blocking]
}
agent_jump $agent(home-server)
agent_end
```

Figure 5.4: An example Tcl agent that migrates through a sequence of machines and interacts with a search engine on each machine. A real agent would record the results from each search engine and present the results to its owner once it returned to the home machine. The *agent* array is a global array that is automatically available inside any Tcl agent; it contains information about the agent's current and home machines. The *0* in the `agent_send` call is the message code.

5.3.3 Interface to the agent system

The interface between a Tcl agent and the agent system is a modified version of Tcl 7.5 and a Tcl extension. The modified version of Tcl 7.5 provides the explicit stack and the state-capture routines. The extension provides the commands that an agent uses to migrate, communicate, and create child agents. The most important commands are `agent_begin`, `agent_submit`, `agent_jump`, `agent_send`, `agent_receive`, `agent_meet`, `agent_accept`, and `agent_end`. Internally each command is just a stub that calls the corresponding operation in Agent Tcl's language-independent core (which is written in C++ and used for all agent languages). The core operations contact an agent server, transfer an agent, message or request, and wait for a response. If an error occurs in the core operation, the stub will throw a Tcl exception. The agent can catch these exceptions and take appropriate action. The main difference between the current implementation and the planned architecture is that when migrating, creating a child agent, or sending a message, the current implementation bypasses the local server and interacts directly with the destination server over TCP/IP. This approach was adopted to simplify the initial implementation and will change as additional transport mechanisms are added.

An agent is simply a Tcl script that runs on top of the modified version of Tcl 7.5. The agent uses the `agent_begin` command to register with a server and obtain an identification within the two-level namespace. The identification consists of the IP address of the server, a unique integer, and a unique symbolic name that

[7] An error occurs if the agent opens a file or creates a linked variable, migrates to a new machine, and then tries to use the file or linked variable. This "delayed" error can lead to confusion. Instead the `jump` command should (optionally) fail if the current state image contains objects that cannot be transferred.

the agent specifies later with the agent_name command. The integer and symbolic name are unique on the agent's current machine, but are not globally unique. The agent_submit command is used to create a child agent on a particular machine. The agent_submit command accepts a Tcl script, optionally encrypts and digitally signs the script, and sends the script to the destination server. The server authenticates this agent, selects an integer identifier for the agent, and starts a Tcl interpreter in which to execute the agent. If the agent wants a symbolic name as well as an integer identifier, it can call agent_name once it starts executing. The agent_jump command migrates an agent to a particular machine. The agent_jump command captures the internal state of the agent, optionally encrypts and digitally signs the state image, and sends the state image to the destination server. The server authenticates this agent, selects a new integer identifier for the agent, and starts a Tcl interpreter. The Tcl interpreter restores the state image and resumes agent execution at the statement immediately after the agent_jump. The agent loses its symbolic name when it jumps and must reacquire the name using the agent_name command if desired.

The agent_send and agent_receive commands are used to send and receive messages. The agent_meet and agent_accept commands are used to establish a meeting between agents. Meetings are named message streams, and although they are not required for communication, they are more efficient and convenient than independent messages. The source agent uses agent_meet to send a connection request to the destination agent. The destination agent uses agent_accept to receive the connection request and send either an acceptance or rejection. An acceptance includes a TCP/IP port number to which the source agent connects. The protocol works even if both agents use agent_meet. The agent with the lower IP address and integer identifier selects the port and the other agent connects to that port. A flexible RPC mechanism has been built on top of the direct connection mechanism [NCK96]. The server will take on more of the responsibility for establishing a direct connection as additional transport mechanisms are added.

Agent Tcl also includes a (slightly) modified version of Tk 4.1 so that an agent can present a graphical interface and interact with the user of its current machine. Event handlers can be associated with incoming messages and with direct connections.

A sample Tcl agent is shown in Figure 5.4. The agent first registers with the agent system (`agent_begin`) and then migrates through a sequence of machine (`agent_jump`). On each machine, the agent sends a query to a stationary search agent (`agent_send`), and then waits for the search agent to send back the results (`agent_receive`). Once the agent has obtained results from all the search agents, it migrates one last time to return to its home machine (`agent_jump`), and then tells the agent systems that it has finished (`agent_end`). A tutorial on writing Tcl agents can be found in Appendix C.

5.4 Agent Java

Agent Java is partially complete. Agents can send and receive messages and migrate from machine to machine, but cannot change the message mask or establish meetings with other agents. Bill Bleier and Joshua Mills have done much of the implementation work.[8]

[8] Bill Bleier and Joshua Mills are both undergraduates in the computer science department and have spent two semesters each working on Agent Java.

5.4.1 Java

Java is an object-oriented language that is syntactically similar to C++ except that there are no structures or unions, no functions, no multiple implementation inheritance, no operator overloading, no automatic type casts, and no pointers [GM95, Sun97b, CH97]. A sample Java program is shown in Figure 5.5. Memory in Java is garbage-collected so there is no *delete* operator. Java is multi-threaded and includes thread synchronization primitives at the language level. Most importantly, a Java program is typically compiled into bytecodes for a stack-based virtual machine, namely the *Java Virtual Machine* [Sun97c]. The program is then executed with an efficient, low-level interpreter. The use of bytecodes and an interpreter has two powerful advantages. First, a bytecode-compiled Java program is highly portable and can run unchanged on any machine that provides the Java interpreter, reducing development costs significantly and making applications such as active Web pages much easier. Second, since the bytecodes are defined to have a statically predictable effect on the type state of the stack, it is possible to run a compiled Java program through a pre-execution verification process [Sun97a]. At the end of this verification process, which also relies on the fact that Java does not provide features such as pointers that can be used to forge access to subobjects, it is known that the program always access objects as their actual type, always calls methods with the correct number and types of arguments, and does not overflow the operator stack. In combination with a class loader that prevents the program from replacing a system class with its own definition, and a security manager that can check for and deny unauthorized access to system classes, the verification process allows a machine to safely run a Java program from an untrusted source. Without this security, a Java-enabled web browser would be impossible.

```
class Factorial {
  public static int factorial (int n) {
    int product = 1;
    for (int i = 2; i <= n; ++i) {
      product = product * i;
    }
    return (product);
  }
  public static void main (String args[]) {
    int n = 10;
    int fac = factorial (n);
  }
}
```

Figure 5.5: A Java program that computes the factorial of a given number n

Java has several advantages as a mobile-agent language. Since Java programs are compiled into low-level, *interpreted* bytecodes (which some Java interpreters then compile "on-the-fly" into native machine code), Java is much faster than Tcl but shares Tcl's high portability. Second, Java is far more suitable for large agents since it provides code modularization via classes. Third, Java has become one of the most popular languages ever, leading to a fast-growing body of Java-literate programmers who would be able to write efficient Java agents with minimal learning time. Finally, since Java is targeted towards mobile-code applications, the Java language and run-time environment already include a range of security features that make it relatively easy to securely integrate Java into a mobile-agent system. The main disadvantage of Java is that although Java is simpler than C++, it still has a long learning curve and development cycle relative to Tcl, making it unattractive for smaller mobile agents. Although the size of a "typical" mobile agent is not yet clear, it does appear that most mobile agents are small enough for Tcl to be an effective language choice from a code maintenance standpoint. Java would be used only in speed-critical applications for which Tcl is simply too slow.

5.4.2 State capture

Release 1.1 of Sun's Java Development Kit (JDK) includes an *object serialization* or *pickling* facility [RWWB96, WRW96]. This facility allows a program to create a serialized representation of the state of a Java object. This bytestream can be stored on disk or transmitted to a remote machine. The bytestream can then be used to recreate an equivalent object. For security reasons, the object serialization facility includes a cryptographic "fingerprint" of the object's complete type in the bytestream, allows a class to exclude sensitive data fields from the serialization process, and allows a class to provide its own pickling methods that either (1) pack, unpack and verify any desired subset of the object's state or (2) throw an exception if the class is sensitive enough that it should not be pickled at all.

All commercial Java-based mobile-agent systems, such as IBM Aglets [LO97, Ven97], General Magic's Odyssey [Gen97] and Mitsubishi's Concordia [WPW+97], rely on this object serialization facility. Since the current serialization facility cannot capture the state of an *executing thread*, and since modifications to the Java virtual machine would severely limit acceptance of a commercial product, none of the three systems provide a `jump` operation. Instead, when an agent wants to migrate, it specifies some subset of its objects. These objects are serialized, sent to the destination machine along with their class definitions and then unserialized. To restart execution of the agent, the agent server on the destination machine calls a known entry method within one of the transferred objects. This method checks the current state of the objects to decide what to do next. Academic systems are not concerned with commercial acceptance and so are free to modify the Java virtual machine. The Sumatra system [RASS97] modifies release 1.0.2 of Sun's Java Development Kit so that it is possible to capture the complete state of an executing Java thread. The basic modification is the addition of a type stack that is parallel to the existing stack and maintains additional type information about the objects that are on the stack. To capture and restore the state of the thread, the Sumatra serialization routines pickle and unpickle the normal stack, the type stack, every object that is reachable from the stack (i.e., every object that is reachable from the thread), and the class definitions for those objects. A `jump` method, exactly analogous to the Agent Tcl `agent_jump` command, is built on top of these two serialization routines. Since one of the subgoals of the Agent Tcl project is to compare the `jump` command with other, more programmer-intensive migration techniques, we made the same modifications to the Java virtual machine as Sumatra so that we could provide a `jump` command in Agent Java.[9] Sumatra and Agent Java do not actually use Sun's object serialization facilities since the source code for these facilities was not available until recently.[10] Instead, both systems use their own serialization facility, which is more efficient than Sun's, but uses a less compact representation for the serialized objects.

5.4.3 Interface to the agent system

The interface between a Java agent and the agent system is the *Agent* class, which is shown in Figure 5.6. Since the same C++ core is used for all three agent languages, the *Agent* class is simply an interface to pre-existing C++ code. Thus nearly all of its methods are *native* methods whose implementation is written in C++. All of these native methods are essentially stubs that convert the Java arguments into C++ arguments, invoke the corresponding operation in the C++ core, and then convert the C++ result into a Java result. The native methods throw Java runtime exceptions if an error occurs within the C++ core. The agent can catch these runtime exceptions and take appropriate action. Although it will eventually be interesting to reimplement the C++ core in Java, both to make the agent system more portable and to allow the agent system to be downloaded into a Java-enabled web browser, it is not worthwhile to undertake such a large porting effort at this time. Thus the *Agent* class and its use of native methods to interface with the C++ core will not change in the near future.

The constructor and finalizer for the *Agent* class call two native methods called

[9] We actually use some of the Sumatra code, which the Sumatra group graciously made available to us.

[10] Source code for Sun's JDK 1.1 had only been available for a few weeks at the time of this writing. Both Sumatra and Agent Java use JDK 1.0.2.

```
public class Agent {
    private int handle;
    private native int createNativeAgent ();
    private native void deleteNativeAgent ();
    public Agent() {
        handle = createNativeAgent ();
    }
    protected void finalize() throws Throwable {
        deleteNativeAgent ();
    }
    public native void begin (String machine, double seconds);
    public native void end (double seconds);
    public native void send
        (AgentId destId, Message message, double seconds);
    public native ReceivedMessage receive (double seconds);
    ...
    public native AgentId submit
        (String machine, AgentBody body, ClassList list, double seconds);
    public native void jump
        (String machine, ClassList list, double seconds);
    ...
}
```

Figure 5.6: The *Agent* class is the main interface between a Java program and the Agent Tcl system.

```
public class AgentBody {

    public void run (Agent agent) {

        throw (EmptyBody);

    }

}
```

Figure 5.7: A subclass of the *AgentBody* class is passed to the *submit* method. The subclass encapsulates the code and objects that will make up the submitted agent.

`createNativeAgent` and `deleteNativeAgent`. The `createNativeAgent` method creates an instance of the C++ agent class defined in the C++ core and returns a handle to that instance. All other native methods use this handle to identify the C++ agent instance on which to operate. The `deleteNativeAgent` method deletes the C++ instance when the Java instance is garbage-collected. All of the other methods in the *Agent* class correspond exactly to the commands provided to Tcl agents. To register with the agent system, for example, a Java agent creates an instance of the *Agent* class and then invokes the `begin` method on that instance. When the agent finishes, it invokes the `end` method on the same instance. This use of the *Agent* class is shown in the example Java agent in Figure 5.8. A *stationary* Java program, i.e., a Java program that never migrates, can create and use multiple instances of the *Agent* class, either within the same or different threads. Thus, a stationary Java program can appear as multiple agents within the agent system. Although the wisdom of this approach is still under consideration, it does appear to be useful in those Java programs that act as servers for other agents. A child agent created with the `submit` method or an agent that has migrated with the `jump` method cannot create any instances of the *Agent* class, but instead can use only the single instance that the agent system (transparently) passes to it upon its creation or arrival. This instance connects the agent with the server on its current machine.

Once the agent has registered with the agent system using the `begin` method, it can use the other methods in the *Agent* class. The `send` and `receive` methods, for example, are used to send and receive messages. The *AgentId* argument to `send` specifies the recipient agent; the *Message* argument contains the message. The *ReceivedMessage* return value from `receive` contains both a *Message* and *AgentId* instance; the *Message* instance contains the incoming message, and the *AgentId* instance identifies the source agent. The *ReceivedMessage* instance also contains a security vector that indicates whether the source agent's owner and machine could be authenticated. Most of the other methods of the *Agent* class are not shown in Figure 5.6. There is one method for each Agent Tcl command and generally the methods take the same arguments and return the same results; several methods involve additional supporting classes such as the *Meeting* class that controls a meeting endpoint (although the interface to the underlying C++ meeting routines is not yet complete as noted at the start of the section). One notable difference between Agent Tcl and Agent Java is that Agent Java does not allow an agent to impose a wall or CPU time restriction *on itself*, primarily because it not clear yet how to handle the restriction violation within Java. Agent Java does enforce the same machine-imposed restrictions, however, terminating the agent if it exceeds its maximum allowance of wall or CPU time.

The two remaining methods shown in Figure 5.6 are the `submit` and `jump` methods, which require special consideration. The `submit` method, like the `agent_submit` command in Agent Tcl, creates a child agent. The parent agent must first create an instance of a *subclass* of the *AgentBody* class that is shown in Figure 5.7. This

```
import agentjava.*;
class search {
  public static void main(String args[]) {
    String[] machines = new String[4];
    machines[0] = "muir"  ; machines[1] = "tenaya";
    machines[2] = "tenaya"; machines[3] = "tuolomne";
    String query = "mobile agent";
    Agent agent = new Agent();
    AgentId aid = agent.begin (agent.getLocalMachine(), 10.0);
    for (int i = 0; i < 4; i += 1) {
        agent.jump (machines[i], 10.0);
        AgentId engineId = new AgentId (machines[i], "query_engine");
        Message request = new Message (0, query);
        agent.send (engineId, request, 10.0);
        RecMessage response = agent.receive (10.0);
    }
    agent.jump (agent.getHomeMachine(), 10.0);
    agent.end (10.0);
  }
}
```

Figure 5.8: A simple Java agent that migrates through a sequence of machines and interacts with a search engine on each machine. A real agent would record the results from each search engine and present the results to its owner once it returned to the home machine. The *10.0* in some of the method calls specifies a timeout of 10 seconds; the *0* passed to the *Message* constructor is the message code.

subclass encapsulates the data and methods that will make up the new child agent. This subclass can contain arbitrary methods and data elements, but must override the `run` method that is defined in the *AgentBody* superclass. The agent initializes the subclass instance as desired, and then passes the instance to the `submit` method as the *AgentBody* argument. The `submit` method serializes the subclass instance and all associated class definitions, and then sends the serialized bytestream to the destination machine. The Agent Tcl system on the destination machine unserializes the bytestream and then invokes the `run` method to start up the new child agent. An *Agent* instance that connects the new agent to its local server is passed to the `run` method as an argument; the new agent is not allowed to create any additional instances of the *Agent* class.

There are two important things to note about this creation process. First, the new child agent can achieve a rough approximation to true migration by continually submitting the same *AgentBody* instance to the next machine in sequence and then immediately exiting on the current machine. Second, although it can be argued that the two classes *Agent* and *AgentBody* should be more closely related from a semantics viewpoint, their current "nonrelationship" seems to be the best alternative. The basic problem is that conceptually an *AgentBody* does not become an *Agent* until it has been submitted to the destination machine. There does not appear to be any way to adequately reflect this fact within a static class hierarchy. In any event, use of the current *AgentBody* class is straightforward and should not be a burden to the agent programmer.

The `jump` command serializes the stack of the calling thread, all objects reachable from the stack, and all associated class definitions (as discussed in the state capture section above). The serialized bytestream is transmitted to and restored on the given destination machine where the agent continues execution from the point of the `jump`. All `Agent` instances within the agent are transparently updated to refer to a single new *Agent* instance, namely an *Agent* instance that connects the agent to its new agent server. In addition, as with `submit`, the agent will not be allowed to create any new *Agent* instances. It is important to note that the the `jump` method moves only the single calling thread onto the destination machine; all other threads are terminated at the time of migration. Similarly the `fork` method clones only the calling thread. Although this choice has proven reasonable so far, it will be important to see if it remains reasonable as larger and more complex Java agents are written. A related issue is that there is currently no tight relationship between the *Agent* instance and some specific thread. Any thread that has a reference to the *Agent* instance can call the `jump` or `fork` method, and it is that thread that will end up on the destination machine.

Unlike Java-enabled web browsers, Agent Java cannot fetch class definitions from the home machine. And, unlike Sumatra, Agent Java does not provide remote objects whose methods can be invoked through local stubs.[11] Thus all objects and class definitions that the agent might use must be transmitted during the initial `submit`, `jump` or `fork`. The necessary objects can be captured easily, namely by serializing all objects reachable from the *AgentBody* instance or all objects reachable from the thread stack. Class are more complex, however, since an agent will not necessarily have any instances of a needed class at the time of the `submit`, `jump` or `fork`. Thus all three methods currently take a `ClassList` argument that specifies by name any

[11]One of the main reasons for not allowing remote objects is that it is a communication mechanism that only Java agents can use. One of our primary design decisions is to support multiple agent languages and language-independent communication mechanisms.

class definitions that should be sent along with the agent, even if the agent currently contains no instances of those classes. The *ClassList* class provides several methods for inspecting and changing its list of classes. We intend to eventually eliminate the *ClassList* argument and have the Agent Java system determine automatically which classes are used inside the agent. One way to automatically determine the class list is the dynamic linking technique of [AS97] which, given an incoming agent and the set of libraries available at the current machine, determines if there are any unbound references *before* the agent starts executing.

5.5 Agent Scheme

Agent Scheme is partially complete. Agents can currently send and receive messages, but cannot migrate from machine to machine or establish meetings with other agents. Dartmouth undergraduates Ahsan Kabir and David Gondek have done most of the implementation work so far.

5.5.1 Scheme

Scheme is a statically scoped and properly tail-recursive dialect of Lisp [Dyb87, CR91]. Although Scheme supports several programming paradigms, such as imperative and message passing, it is oriented towards and most commonly used for functional programming as illustrated in Figure 5.9. Scheme provides first-class procedures and lambda expressions, supports both closures and continuations, evaluates both the operator and operand positions of a procedure call, and expresses iteration with procedure calls only [MIT97b].

The are four reasons to select Scheme as a mobile-agent language. First, since Scheme supports functional programming and is used nearly exclusively in modern

```
(define factorial
         (lambda (n)
           (if (zero? n)
               1
               (* n (factorial (- n 1))))))
(factorial 10)
```

Figure 5.9: A Scheme program that computes the factorial of a given number n.

artificial intelligence research, it opens up mobile-agent programming to a much wider community and allows the more convenient expression of agents that plan or learn. Second, there are several existing Scheme interpreters that support the full language, but that are still lightweight and extremely efficient, most notably Scheme 48 which is based around a virtual machine [KR95]. Third, since Scheme is a type-safe language and is lexically scoped, it already provides the initial layer of security that is needed when executing untrusted agents [CJK95]. Finally, there has already been work on moving ongoing Scheme computations from one machine to another [CJK95].

5.5.2 State capture

We selected Scheme 48 as the Scheme interpreter for Agent Tcl due to its efficiency, easy portability and its ability to interface with native C/C++ libraries. Unfortunately, although Scheme 48 provides full support for both closures and continuations, which respectively contain exactly the state information that is needed when creating a child agent (*submit*) or migrating an existing agent (*jump*), it does not provide any support for serializing (and later restoring) a given closure or continuation. In the case of migration, it is possible to capture the necessary state information with the *capture-state* function shown in Figure 5.10. This function, however, actually cap-

```
( define (capture-state filename) (
    call-with-current-continuation (
      lambda (p)
      (build-image (lambda(x) (p 1)) filename)
      (p 0)
    )
  )
)
```

Figure 5.10: A naive state capture routine for Scheme 48. This routine has two problems. First, it writes the state image out to disk rather than to an arbitrary bytestream. Second, the state image contains the entire interpreter heap rather than just the continuation p. Fixing these problems requires additions to the Scheme 48 virtual machine.

tures the state of the entire heap of the Scheme 48 virtual machine, leading to a state image that is both extremely large and much larger than necessary. Even for the simplest Scheme program, the state image will typically be several hundred kilobytes, making this state-capture implementation impractical.

Instead of capturing the entire heap, we need to capture only the desired continuation or closure. This improved state capture requires additions to the Scheme 48 virtual machine. Fortunately, Kali Scheme [CJK95] has already made these additions. Kali Scheme is derived from Scheme 48 and allows either a closure or a executing thread (in the form of the thread's continuation) to be transferred from one machine to another. Kali Scheme also provides remote object proxies and cross-machine communication primitives in the lowest levels of the Scheme 48 virtual machine, neither of which are needed in the Agent Tcl system. Our current task is to extract the state-capture routines from Kali Scheme, so that we can create a version of Scheme 48 that supports the necessary state capture without also including Kali Scheme's other distributed-computing facilities. Although this task is conceptually straightforward, it is time-consuming and will likely take several more months of part-time undergraduate effort.

5.5.3 Interface to the agent system

Agent Scheme uses the same generic C++ core as Agent Tcl and Agent Java and provides a set of Scheme functions that interact with this core. In general, these Scheme functions correspond exactly to the commands in Agent Tcl, taking the same arguments and returning the same results. The exceptions are `agent_begin` and `agent_end`, which must be called as a pair (with the code for the agent in between). Since this pair of calls is somewhat inconsistent with the functional programming

```
(define (queryAgent machine query) (
    (agent_let ((agent-local-machine) 10.0 agentId)
        (agent_send (list machine "query_engine") query 0 10.0)
        (agent_receive 10.0)
    )
)
(define machine "muir")
(define query "mobile agent")
(queryAgent machine query)
```

Figure 5.11: A simple Scheme agent that interacts with a search engine on a remote machine. The *10.0* in the `agent_let`, `agent_send`, and `agent_receive` calls specifies a timeout of 10 seconds; the *agentId* in the `agent_let` call is the local variable in which the agent's identification is stored; the *0* in the `agent_send` call is the message code. Agent Scheme does not yet support migration, so the more complex Tcl and Java agents above do not have a corresponding Scheme implementation.

paradigm, we instead encapsulate `agent_begin` and `agent_end` inside an `agent_let` macro. This macro is equivalent to the standard Scheme `let`, except that it automatically calls `agent_begin` before executing the `let` body, and automatically calls `agent_end` after executing the `let` body. In other words, `agent_let` automatically turns its body into an agent, and stores the identification of the new agent in a variable specified in the `agent_let` call, making this identification information accessible in the body.

A sample Scheme agent is shown in Figure 5.11. Since the migration facilities of Agent Scheme are incomplete, the sample agent does not migrate from machine to machine like the Tcl and Java agents, but instead remains stationary and simply interacts with a local search engine. The agent uses the `agent_let` macro to register (and unregister) with the agent system and then uses the `agent_send` and `agent_receive` functions to interact with the search engine.

Chapter 6

Implementation – Security mechanisms

A mobile agent is a *program* that moves from machine to machine and executes on each. Neither the agent nor the machines are necessarily trustworthy. The agent might try to access or destroy privileged information or consume more than its share of some resource. The machines might try to pull sensitive information out of the agent or change the behavior of the agent by removing, modifying or adding to its data and code. A mobile-agent system that does not detect and prevent such malicious actions can never be used in real applications. In an open network environment, intentional attacks on both machines and agents will start as soon as the system is deployed, and even in a closed network environment with trusted users, there is still the danger of misprogrammed agents, which can do significant damage accidentally. Security is perhaps the most critical issue in a mobile-agent system and can be divided into four interrelated problems:

- *Protect the machine.* The machine should be able to authenticate the agent's owner, assign resource limits based on this authentication, and prevent any violation of the resource limits. To prevent both the theft or damage of sensitive information *and* denial-of-service attacks, the resource limits must include access rights (reading a certain file), maximum consumptions (total CPU time), and maximum consumptions per unit time (total CPU time per unit time).

- *Protect other agents.* An agent should not be able to interfere with another agent or steal that agent's resources. This problem can be viewed as a subproblem of protecting the machine, since as long as an agent cannot subvert the agent communication mechanisms and cannot consume or hold excessive system resources, it will be unable to affect another agent unless that agent chooses to communicate with it.

- *Protect the agent.* A machine should not be able to tamper with an agent or pull sensitive information out of the agent without the agent's cooperation. Unfortunately, without hardware support, it is impossible to prevent a machine from doing whatever it wants with an agent that is currently executing on that machine. Instead we must try to detect tampering as soon as the agent migrates from a malicious machine back onto an honest machine, and then terminate or fix the agent if tampering has occurred. In addition, we must ensure that (1) sensitive information never passes through an untrusted machine in an unencrypted form, (2) the information is meaningless without cooperation from a trusted site, or (3) that theft of the information is not catastrophic and can be detected via an audit trail.

- *Protect a group of machines.* An agent might consume excessive resources in the network as a whole even if it consumes few resources at each machine. Obvious examples are an agent that roams through the network forever or an agent that creates two child agents on different machines, each of which creates two child agents in turn, and so on. An agent and its children should eventually be unable to obtain any resources anywhere and be terminated. If the network machines are under single administrative control, solutions are relatively straightforward; if the machines are not, solutions are much more complex.

All of these problems have been considered in the mobile-agent literature [LO95, CGH+95, TV96, PS97], but only the first two have seen significant implementation work. These same two problems are addressed in the current implementation of Agent Tcl using PGP [KPS95], Safe Tcl [LO95, OLW97] and Java security managers [CH97]. First we present the current implementation and then potential solutions for the remaining two security problems.

6.1 Protecting the machine (and other agents)

Protecting the machine involves two tasks:

- *Authentication.* Verify the identity of an agent's owner.
- *Authorization and enforcement.* Assign resource limits to the agent based on this identity and enforce those resource limits.

Agent Tcl, like other mobile-agent systems, handles these two tasks with public-key cryptography and secure execution environments that perform authorization checks before each resource access.

6.1.1 Authentication

Each Agent Tcl server distinguishes between two kinds of agents: *owned* and *anonymous*. An *owned* agent is an agent whose owner could be authenticated and is on the server's list of authorized users. An *anonymous* agent is an agent whose owner could not be authenticated or is not on the server's list of authorized users. Each server can be configured to either accept or reject anonymous agents. If a server accepts an anonymous agent, it gives the agent an extremely restrictive set of resource limits.

RSA public-key cryptography is used to authenticate an agent's owner. Each owner and machine in Agent Tcl has a public-private key pair. The server can authenticate the owner if (1) the agent is digitally signed with the owner's public key or (2) the agent is digitally signed with the sending machine's key, the server trusts the sending machine, and the sending machine was able to authenticate the owner itself. In the second case, the sending machine would have authenticated the owner in one of the same two ways: (1) the agent was signed by the owner or (2) the agent was signed by one of the *sending machine's* trusted machines (and that trusted machine was able to authenticate the owner itself). Thus, trust is transitive, and trust relationships must be established carefully. Typically machines under single administrative control would trust each other and no one else.

Agent Tcl uses Pretty Good Privacy (PGP) for its digital signatures and encryption. PGP is a standalone program that allows the secure transmission of electronic mail and is in widespread use despite controversies over patents and export restrictions [KPS95]. PGP encrypts a file or mail message using the IDEA algorithm and a randomly chosen secret key, encrypts the secret key using the RSA public-key algorithm and the recipient's public key, and then sends the encrypted key and file to the recipient. PGP optionally adds a digital signature by computing an MD5 cryptographic hash of the file or mail message and encrypting the hash value with the sender's private key. Although PGP is oriented towards interactive use, it can be used in an agent system with minimal effort. In the current implementation, Agent Tcl runs PGP as a separate process, saves the data to be encrypted into a file, asks the PGP process to encrypt the file, and then transfers the encrypted file to the destination server. This approach is much less efficient than tightly integrating PGP with the rest of the system, but is simpler and more flexible, especially since it becomes trivial to create an Agent Tcl distribution that does not include PGP or that uses different encryption software [Way95].

An agent chooses whether to use encryption and signatures when it migrates or sends a message to another agent. If the agent is not concerned with interception during migration, it turns off encryption. If the agent is not concerned with tampering during migration and can accomplish its task as an *anonymous* agent, it turns off signatures. When sending a message, the agent makes the same decisions, except that it turns off signatures only if the recipient does not need to verify the sender's identity. Turning off either encryption or signatures is a significant performance gain due to the slowness of public-key cryptography, and thus most agents will turn off encryption and signatures whenever the needed resources and the network environment allow it. In the rest of this section, we assume that the agent does *not* want to be an anonymous agent and does *not* want to send anonymous messages, and thus has digital signatures turned on.

When an agent registers with its home server using the `begin` command (Figure 6.1), the registration request is digitally signed with the owner's private key, optionally encrypted with the destination server's public key, and sent to the server. The server verifies the digital signature, checks whether the owner is allowed to register an agent on its machine, and then accepts or rejects the request. If the agent and the server are on different machines, all further requests that the agent makes of the server must be protected to prevent tampering and masquerade attacks.[1] Ideally, the system would

[1] A masquerade attack here is another agent passing itself off as the registered agent.

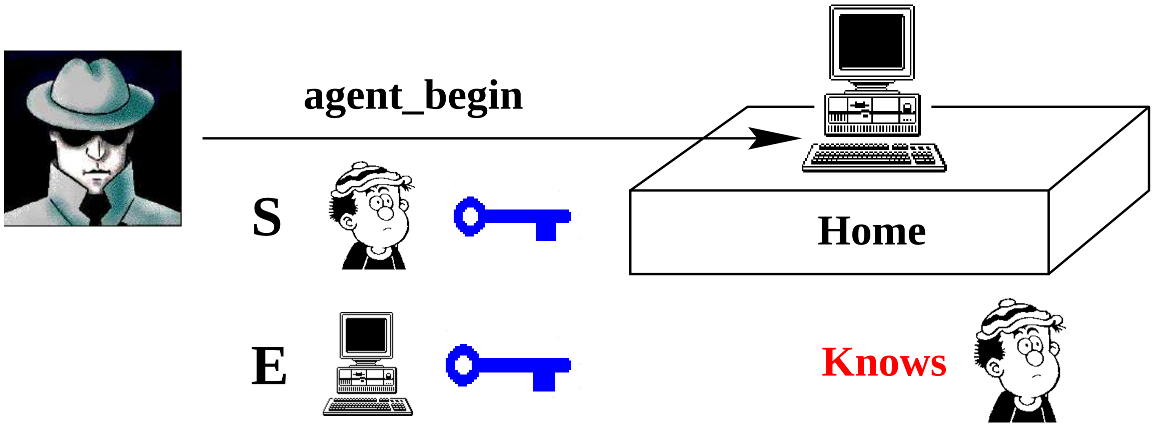


Figure 6.1: Encryption for the begin command. When an agent uses the `begin` command to register with the server on its home machine, the registration request is signed with the owner's private key (S) and optionally encrypted with the receiving machine's public key (E).

generate a secret session key, known only to the agent and the server, and then use this session key to encrypt the requests [KPS95]. PGP does not provide direct access to its internal secret-key routines, however, making it impossible to generate and use session keys without modifying PGP. Therefore, the current implementation of Agent Tcl handles the additional requests in the same manner as the initial registration request, digitally signing them with the owner's private key. Since public-key algorithms are much slower than secret-key algorithms, we will switch to secret sessions keys once we replace PGP with a more flexible encryption library. When the agent and the server are on the same machine (which is the predominant case), there is no need for a session key, since it is impossible to intercept or tamper with the additional requests or to masquerade as the registered agent.[2] Thus all additional requests are transmitted in the clear.

[2]The server uses different communication channels for local agents and can tell without cryptography whether a request came from a specific local agent.

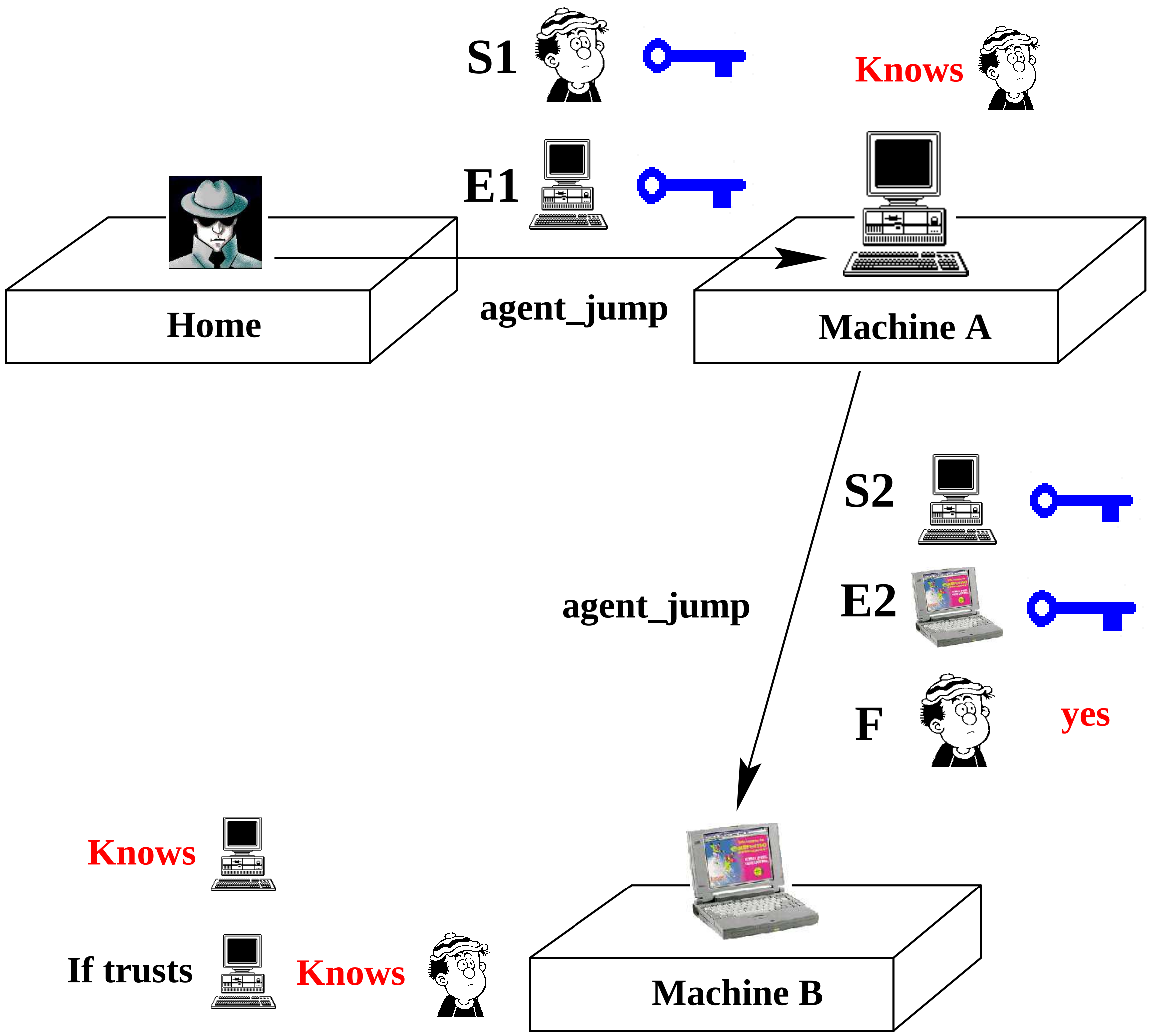


Figure 6.2: Encryption for the jump command. On the first jump, the agent is signed with the owner's private key (S1). On the second and later jumps, the agent is signed with the sending machine's private key (S2), and the sending machine sets a flag (F) to indicate whether it was able to authenticate the agent's owner itself; if the target machine trusts the sending machine, and the sending machine reports that it was able to authenticate the agent's owner, the target machine considers the owner authenticated.

When an agent migrates for the *first time* with the `jump` command, the state image is digitally signed with the owner's private key, optionally encrypted with the destination server's public key, and sent to the destination server. The server verifies the digital signature, checks whether the owner is allowed to send agents to its machine, and accepts or rejects the incoming agent. This process is shown in Figure 6.2. Of course, once the agent has migrated, the owner's private key is no longer available. Therefore, for all subsequent migrations, the agent is digitally signed with the private key of the sending server. If the destination server trusts the sending server, and the sending server was able to authenticate the owner itself, the destination server considers the owner authenticated and gives the agent the full set of resource limits for that owner. If the destination server does not trust the sending server, or the sending server could not authenticate the owner itself, the destination server considers the agent to have no owner and will either (1) accept the agent as an anonymous agent or (2) reject the agent if it is not allowed to accept anonymous agents. Typically, Agent Tcl servers are configured so that machines under single administrative control trust each other but no one else.[3] Thus, if an agent migrates from its home machine into a set of mutually trusting machines (and then stays within that set), each machine will be able to directly (or indirectly) authenticate the owner, and will give the agent the full set of access permissions for that owner. Once the agent leaves the set of machines, however, it becomes anonymous, and remains anonymous even when it comes back, since the untrusted machines might have modified the agent in a malicious way. While the agent is on a particular machine, it will make requests of that machine's server. As in the case when an agent registers with a server on

[3]For example, all the machines in the Computer Science Department at Dartmouth trust each other.

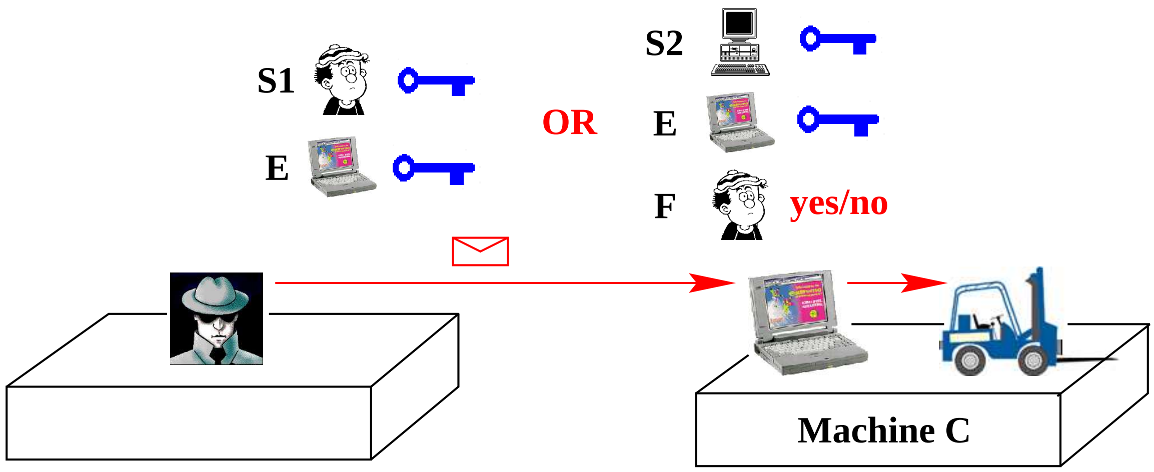


Figure 6.3: Encryption for the send command. If the agent has not left its home machine, the message is signed with the owner's private key (S1). If the agent has left its home machine, the message is signed with the sending machine's key (S2), and the sending machine sets a flag (F) to indicate whether it was able to authenticate the agent's owner itself; if the target machine trusts the sending machine, and the sending machine reports that it was able to authenticate the agent's owner, the target machine considers the owner authenticated.

the same machine, however, no encryption or digital signatures are needed for these requests.

When a new child agent is created on a different machine (with the `fork` or `submit` command), or when a message is sent to an agent on a different machine (with the `send` command), the same strategy is used as with `jump`. The message or child agent is signed with the owner's key if the sending agent is still on its home machine, and with the machine's key if the sending agent has already migrated (Figure 6.3). The recipient server will believe the owner's identity if it trusts the sending server. When receiving a message, the recipient agent gets both the message and a security vector. The security vector specifies the owner of the sending agent, whether the owner could be authenticated, the sending machine, whether the sending machine could be authenticated, whether the message was encrypted, and whether the sending agent is on the same machine. The recipient agent, which might be controlling access to some resource such as a database, bases its own security decisions on this security vector. When a new agent is created on the same machine, or a message is sent to an agent on the same machine, no encryption or digital signatures are required. The new agent inherits the security information of its parent. The recipient of the message gets the same five-element security vector.

This authentication scheme has five weaknesses. First, and most serious, once an agent leaves its home group of trusted machines, it becomes anonymous as soon as it migrates again. Having the agent become anonymous is essential in the current system since a malicious machine can modify an agent arbitrarily (or lie about the identity of its owner). Thus, when dealing with machines that do not trust each other, an application that needs the full access rights of its owner to accomplish its task cannot send out a single agent that migrates through the machines, since the agent will become anonymous on the second jump. Instead the application must send an agent to the first machine, wait for the results, send a new agent to the second machine, and so on. Although this problem does not prevent an application from accomplishing its task, it places an additional burden on the programmer, and reintroduces some of the network traffic that mobile agents are meant to avoid. At the same time, it is important to note that many applications operate entirely within a set of trusted machines, and that many others, especially in the Internet, can be accomplished with anonymous agents. Solving the multi-hop authentication problem revolves around detecting malicious modifications to an agent. Then, confident that certain kinds of malicious modifications (such as modifications to the static code) will always be detected, a machine can assign access rights that fall somewhere between those of an anonymous agent and those of the actual owner. Detecting malicious modifications is discussed below.

The remaining four problems are less serious and have clear solutions. First, PGP is extremely slow, especially since Agent Tcl executes PGP as a separate process. PGP must be replaced with a faster encryption library. Second, PGP does not provide access to its internal encryption routines, making it impossible to generate session keys for ongoing communication. The replacement library must support both public-key and secret-key cryptography. Once the system can generate session keys, it should use session keys rather than public/private keys whenever possible due to the speed advantage of secret-key cryptography, For example, two servers that are communicating extensively might generate a shared session key, even if different agents are responsible for each communication. Third, Agent Tcl does not include an automatic distribution mechanism for the public keys. Each server must already *know* the public keys of all authorized users so that it can authenticate incoming agents (agents signed with an unknown public key become anonymous). A modest key-distribution or certification mechanism must be added to Agent Tcl to reduce the burden on the system administrator. Finally, the system is vulnerable to replay attacks in which an attacker replays a migrating agent or a message sent to an agent on a different machine. Here a server could have a distinct series of sequence numbers for each server with which it is in contact.

6.1.2 Authorization and enforcement

Once the identity of an agent's owner has been determined, the system must assign access restrictions to the agent (*authorization*) and ensure that the agent does not violate these restrictions (*enforcement*). In other words, the system must guard access to all available resources. We divide resources into two types. *Indirect* resources can be accessed only through another agent. *Builtin* resources are directly accessible through language primitives (or libraries) for reasons of efficiency or convenience or simply by definition. Builtin resources include the screen, the filesystem, memory, real time, CPU time, and the agent servers themselves.[4]

For indirect resources, the agent that controls the resource enforces its own access restrictions, rejecting or allowing requests from other agents based on the security vector attached to the incoming communication. Typically, the resource agent would simply check each request against an access list, although one request could return capabilities for use in later requests. Care must be taken with capabilities, however, since a migrating agent will carry its capabilities along with it, possibly through malicious machines. One reasonable solution is to allow an agent to obtain a capability only if it is on the same machine as the resource, and include sufficient identification information in the capability so that it becomes invalid as soon as the agent leaves[5]; this solution makes it impossible for valid capabilities to exist on other machines, preventing theft *and* eliminating severe administrative problems. Agent Tcl will eventually provide both access-list and capability libraries for use in resource

[4] The agent servers are accessed through the agent commands, such as `begin`, `jump` and `send`. All agent commands use server CPU cycles; several use server memory; and several require network access.

[5] For example, the capability could include the agent's id and the time at which it arrived on the local machine. The agent will get a different timestamp (and usually id) if it leaves and returns, making it impossible to reuse the capability after a migration. In addition, since the ids are locally unique, no other agent can *ever* have the same combination of id and timestamp, making it impossible to transfer the capability to another agent.

agents; currently each resource agent must provide its own implementation.

For builtin resources, the agent servers enforce several absolute access policies. For example, an agent can *terminate* another agent only if its owner is the system administrator or if it has the same owner as the other agent. The *name* operation reserves certain symbolic names for certain agent owners, preventing an arbitrary agent from masquerading as a service agent (such as a yellow page agent that provides directory services). The *notify* operation requires the server to remember which agent asked for the notification, taking up server memory. Thus, the server has a per-agent limit on the number of outstanding notifications; the limit is small for visiting agents, but large for agents that belong to the machine's owner or administrator, since notifications are the most efficient and convenient way to implement monitoring tools that track which agents are currently on the machine.[6] There are similar access policies for the other agent operations. In particular, most operations can be configured to reject requests from remote machines. In a typical configuration, for example, the *begin* operation rejects any request from a remote machine, allowing only agents on the local machine to register with the server. The *begin* operation also imposes a limit on the total number of agents and the total number of *anonymous* agents executing on the machine at one time. The specific limits and access restrictions are specified in a server configuration file.

For all other builtin resources, security is maintained using the language-specific security (or enforcement) module and a set of *resource manager* agents. When an agent requests access to a builtin resource, either implicitly or explicitly, the security module forwards the request to the appropriate resource manager. The resource

[6]Or, more precisely, notifications will be the most convenient way once an agent can request notifications for a wider range of events.

manager, which is just a stationary agent, checks an access list, decides whether the request should be allowed, and returns the decision to the security module. The security module then enforces the decision (and also caches the decision when appropriate to minimize the load on the resource managers). This approach provides a clean separation between security policy and enforcement, with the same resource managers making security decisions for all agents, regardless of their implementation language. There are six resource managers in the current system.

- *Consumables.* The *consumables* manager handles those resources where, even though they never run out, each agent should be given only a limited amount to prevent system overload. Currently these resources are wall time, CPU time, number of child agents, maximum depth of the parent-child hierarchy, and number of migrations. Limits on these resources are enforced across *groups* of mutually trusting machine. When making its decision, the consumables manager takes into account how much resource the agent has used on the other machines within the group.[7] Since access to these resources is either implicit (CPU time) or takes place through the generic agent core (migration), enforcement actually takes place in the core, with the language-specific security module simply setting the new limits after the manager returns its decision. In addition, in contrast with the other builtin resources, the agent starts with a small allowance and must *explicitly* ask the manager for more. Notably absent from this set of consumable resources are memory and CPU time *per unit time* (as well as agent operations per unit time since an agent might be able to overwhelm the local server without using much CPU time). Thus, a visiting agent can currently

[7] Migrating agents include a vector that specifies how much of each resource they have used so far.

mount denial-of-service attacks against other agents by allocating all available virtual memory, sitting in a computationally-intensive loop, or flooding the local server with requests. Fixing the memory problem is trivial for Java and Scheme 48, which already have their own memory-allocation routines that enforce a maximum heap size, but more complex for Tcl, which calls the standard *malloc* and *free* routines directly. A solution for Tcl that works on all platforms might be impossible without changing its memory allocation routines. Fixing CPU time is more difficult since we are currently relying on the underlying Unix system for scheduling; scheduling mechanisms must be added to Agent Tcl so that it can timeslice the agents itself.

- *Filesystem.* The *filesystem* manager controls read and write access to files and directories. It also imposes a maximum size on writable files so that an agent cannot fill up the filesystem. The manager controls access only to the *entire* file. Record-based access control is simply too fine-grained for a mobile-agent system. If record-based access control is required, the file should be hidden behind a stationary service agent. The main weakness of the current *filesystem* manager is that it does not impose a limit on disk accesses per unit time, making it possible for an agent to thrash the local disk. As with CPU time per unit time, the ideal solution is a scheduler that would allocate disk access "slots" among the agents that require disk access.

- *Libraries* The *libraries* manager determines which libraries of Tcl functions, Scheme functions or Java classes each agent can load.

- *Programs* The *programs* manager determines which external programs each agent can execute. Since an external program is not subject to the same security checks as the agents themselves, execute permissions are given only to those owners whose agents can be *trusted* to use the program properly. Typically, then, an agent belonging to the system administrator is allowed to execute any program, a service agent is allowed to execute only the programs that it needs to provide the service, and all other agents are not allowed to execute any programs. If an external program provides functionality that is useful for all agents, it can be hidden behind a stationary service agent that performs the necessary security checks.

- *Network*. The *network* manager decides which agents are allowed to directly access low-level TCP/IP and UDP network services. It does not attempt to limit certain agents to certain ports or certain remote machines; it either grants complete access or no access at all. Thus, as with external programs, network access is usually only given to the system administrator and specific service agents. Eventually, all agents should be allowed to access certain network services such as Sun RPC, especially when they are on a dedicated proxy site. Then, if a resource is not on an agent-enabled machine, an agent can migrate as close as possible to that machine and interact with the resource using standard cross-network calls [MAF97]. In addition, most agent commands can generate network traffic, particularly the commands that send messages and establish meetings. To prevent an agent from flooding the network, the network manager and enforcement modules must impose a maximum transmission rate. Ideally, as with CPU time and disk access, a maximum transmission rate would be set for all agents as a group, and transmission "slots" would be scheduled among the agents that are trying to transmit packets.

- *Screen.* Like network access, screen access is controlled at the coarsest level: an agent can do either anything or nothing. Therefore, in the current system, it is unreasonable to give an anonymous agent any screen access, since the agent might immediately create a window that covered the entire screen and grab the global focus. Here we are planning a two-part solution: (1) allow an individual agent to ask the *user* for screen access through some central control panel, and (2) provide the agent with a window that mimics a full screen. In the latter case, the agent could do anything it wanted on the virtual screen, but could not move or resize that virtual screen itself.

Of course, a machine can have other hardware devices to which an agent might need access, such as a microphone, speaker, camera or printer. Many of these devices, such as a printer, can be efficiently hidden behind a stationary service agent; this stationary service agent performs any desired security checks before proceeding with a request. Other devices, such as a microphone, might need to be accessible through a library for efficiency. In this case, resource managers for the devices must be added to the system.

Each resource manager has a configuration file that specifies the access rights and limits for a particular *owner*. The manager simply loads this access list on startup and then checks the owner of each requesting agent against the list. Of course, the manager also takes into account whether the owner could be authenticated and whether the requesting agent is on the same machine. Anonymous agents are given limited access rights (mainly read access to certain libraries and initialization files), and remote agents are given no access rights.

The enforcement module is different for each language.

Tcl. The Tcl enforcement module is implemented with Safe Tcl. Safe Tcl is a Tcl extension that is designed to allow the safe execution of untrusted Tcl scripts [LO95, OLW97]. Safe Tcl provides two interpreters. One interpreter is a "trusted" interpreter that has access to the standard Tcl/Tk commands. The other interpreter is an "untrusted" interpreter in which all dangerous commands have been replaced with links to secure versions in the trusted interpreter. The untrusted script executes in the untrusted interpreter. Dangerous commands include obvious things such as opening or writing to a file, creating a network connection, and creating a toplevel window. Dangerous commands also include more subtle things such as ringing the bell, raising and lowering a window, and maximizing a window so that it covers the entire screen. Some of these subtle security risks do not actually involve damage to the machine or access to privileged information, but instead involve serious annoyance for the machine's owner.

Agent Tcl uses the generalization of Safe Tcl that appears in the Tcl 7.5 core [LO95]. Agent Tcl creates a trusted and untrusted interpreter for each incoming agent. The agent executes in the untrusted interpreter. All dangerous commands have been removed from the untrusted interpreter and replaced with links to secure versions in the trusted interpreter. The secure version contacts the appropriate resource manager and allows or rejects the operation depending on the resource manager's response. The secure version also caches the resource manager's response on an internal access list so that it does not have to contact the resource manager again when the same operation is performed later. For example, if the agent issues the Tcl command `exec ls`, the `exec` procedure in the trusted interpreter checks the internal *program* access list. If permission to execute `ls` has already been granted, the command proceeds. If permission to execute `ls` has already been denied, the command throws a security exception. Otherwise the command contacts the *program* resource manager, adds the response to the *program* access list, and then either proceeds or throws the security exception.

An agent can also explicitly ask a resource manager for access permissions with the `require` command. The `require` command takes the symbolic name of the resource manager—e.g., *filesystem*—and a list of *(name, quantity)* pairs that specify the desired access permissions—e.g., *(/home/rgray/test.dat, read)*. The `require` command is actually just a link to a procedure in the trusted interpreter. This procedure sends the list of desired access permissions to the appropriate resource manager. The procedure waits for the response and then adds each access permission to the internal access lists, indicating for each whether the request was granted or denied. Regardless of whether a request is made with the `require` command or by invoking a Tcl command, the resource manager will send back the most general access permissions possible, effectively preloading the internal access lists and eliminating future requests. For example, if an agent requests access to a particular file, but is actually allowed to access the entire filesystem, the manager's response will grant access to the entire filesystem. In addition, although an agent can contact the resource managers directly in the current implementation, such contact accomplishes nothing since the response will not go through the trusted interpreter and therefore will not have any effect on the internal access lists.

Finally, an agent can impose access restrictions on *itself* with the `restrict` command. The `restrict` command takes two arguments: a list of access restrictions and a Tcl script. The command executes the script under the given access restrictions. In the case of the *consumable* resources, these access restrictions remain in effect even when the agent migrates to a new machine. For example, the agent can restrict itself to a particular number of children, even if it is migrating and creating the children on different machines. More usefully, perhaps, the agent can restrict itself to a specific amount of CPU or wall time.

The Safe Tcl security module does not provide safe versions of all dangerous commands. For example, an agent that arrives from another machine cannot use the *Tk* `send` command, which sends a Tk event to another Tk interpreter.[8] In addition, there are no safe versions of the network and screen commands, since the resource managers either grant complete access to the screen and network or no access at all. The network and screen commands simply remain "hidden" until the resource managers grant access. Since all of the annoyance security threats, including ringing the bell, involve screen commands, only trusted owners should be given screen access in the current system. Once the system provides a virtual screen, anonymous agents can be given screen access as well, although the `bell` command will need to be handled specially. Despite the coarse-grained access to the network and screen, the simple kernel-user model of Safe Tcl protects the machine well. No direct access to system resources is possible, and there is no way for an agent to subvert the resource manager decisions, since the agent cannot modify the access lists in the trusted interpreter.

Java. The Java enforcement module is implemented as a Java security manager [CH97]. A Java security manager is a class that provides a set of access-control methods, such as `checkExec`, `checkRead`, and `checkExit`. The Java system classes call these methods to see if the corresponding operation is allowed. For example, the `System.exec` method calls `checkExec` to see if the Java program is allowed to execute

[8] It is likely that the *Tk* `send` command will never be available since it is difficult to make secure and agents should communicate within the agent framework anyways.

the specified external program.[9] Our security manager for agents is exactly equivalent to the Safe Tcl mechanism above: each `checkXXX` method contacts the appropriate *resource manager* and then throws a security exception if the resource manager denies access. Our security manager also provides the `require` and `restrict` operations. The `restrict` operation is actually split into two methods `addRestriction` and `removeRestriction`; the resource limits apply to whatever code appears between the calls to these two methods. Implementation of the Java security manager is not yet complete. Since the methods follow the same logic as the corresponding Safe Tcl procedures, however, implementation will proceed rapidly.

Scheme. Scheme 48 has a module system [KR95]. A module is a set of Scheme functions with some of those functions marked as *exported* or *public*; a program can load the module and invoke any of the exported functions. Providing the Scheme enforcement module is mainly a matter of redefining the system modules so that they no longer export dangerous functions, but instead export secure versions of those functions that perform the same security checks as in Tcl and Java. Although implementation work is just starting, it appears that the necessary module redefinitions can be accomplished without changing the Scheme 48 virtual machine.

6.1.3 Summary

The mechanisms for protecting the machine are nearly complete. There are two remaining implementation issues. First, the implementation of the Java and Scheme enforcement modules is incomplete. The remaining implementation work is not difficult, however, and involves little more than reimplementing the Safe Tcl security checks. Second, screen and network access is controlled at the coarsest possible level,

[9] The *filename* of the external program is a parameter to `checkExec`.

with an agent either getting complete access or no access at all. Instead, agents should be given a virtual screen and restricted access to *common* high-level network services such as RPC and HTTP.

There are also three remaining architectural issues. First, the current architecture requires that a new enforcement module be written for each language. This approach minimizes the changes to the standard interpreters, but is time-consuming and error-prone. Eventually we will move to the Ara model in which the core provides secure versions of all system functions [PS97]; these core functions would still contact the resource managers to determine access rights.

Second, Agent Tcl uses discretionary access control, in which each resource has an associated access list that specifies the allowed actions for each agent owner. Many other security models exist, such as (1) mandatory access control, in which programs, people and data are assigned classification levels, and information can not flow from higher to lower levels, (2) security automata [Sch97a], in which a program's current allowed actions depend on its past resource usage,[10] and (3) computer immunology [FHS97, Gre97b], in which a program is considered malicious if its current pattern of resource usage does not match its normal pattern. Although none of these models are incompatible with Agent Tcl's current architecture, architectural extensions would be needed for all three. As it becomes clearer which of the three models are useful in a mobile-agent environment, we will consider implementing one or more of them.

Finally, an agent can still mount several denial-of-service attacks: (1) it can sit in a tight loop and consume CPU time as fast as possible; (2) it can flood the local agent server with requests; (3) it can flood the local network by sending requests to remote

[10] For example, an agent might be permitted to communicate with a remote machine as long as it has not read from a sensitive file.

agent servers as fast as possible (or by using some network service such as RPC to which it has been given direct access); (4) it can allocate all available virtual memory; and (5) it can thrash the local disk by randomly reading from any file to which it has been given access (or by allocating a data structure that is too large for main memory and then accessing the data structure in such a way as to cause page fault after page fault). Preventing these denial-of-service attacks is not difficult; preventing them without artificially reducing performance is difficult. Ideally the system would specify how much CPU time, memory, network bandwidth and disk bandwidth should be made available for *all* agents, and then allocate or schedule the available capacity among the agents.[11] Doing this with reasonable overhead will probably require tighter system integration, with agents running in threads rather than in their own processes. For this reason, we plan to start with fixed, per-agent limits for each of these resources, and then move on to the more flexible resource "scheduling" at a later time. Once the remaining denial-of-service attacks are eliminated, Agent Tcl will successfully protect machines from malicious agents and agents from each other.

6.2 Protecting a group of machines

Protecting a group of machines divides two distinct subcases: (1) all the machines are under single administrative control, such as in a departmental LAN, or (2) all the machines are *not* under single administrative control, such as in the Internet.

When the machines *are* under single administrative control, protecting the machines is straightforward. An agent is assigned a maximum resource allowance when

[11] The exact scheduling algorithm is an open question. There probably should be at least two classes of agents, stationary service agents and visiting agents, with service agents having higher priority.

it first enters the machine group. The allowance and the amount that the agent has used so far is propagated along with the agent as it migrates. If the agent exceeds its group allowance, it is terminated.[12] Agent Tcl already provides this kind of group protection. The *consumables* resource manager imposes a limit on how many times an agent can jump, how much CPU time it can use, etc., within an administrator-specified group of mutually trusting machines; the limits and the amounts used so far are included in the migrating agent.[13] Although Agent Tcl enforces this group limit on consumables, it does not yet enforce a separate per-machine limit. Per-machine

[12]Alternatively, the agent could be sent back to its home machine or to a designated proxy site, although the current Agent Tcl system does not provide such functionality. An agent can inspect its group allowance, however, and can migrate out of the machine group if it sees that it is about to run out of some resource.

[13](1) In network environments where there is no threat of packet tampering, the group allowance can be enforced correctly based solely on the sending machine's address that is included in the migration header, i.e., it can be enforced correctly without encryption or digital signatures. This is simply because each machine within the group knows about the maximum allowance for each owner and will reset an incoming agent's *reported* maximum allowance if it is above the actual maximum. Thus, although a malicious agent that is entering the group for the first time might have a group machine's address in its migration header, it will end up with the same maximum allowance that it would have gotten without lying. Then, once the agent is inside the group, it is impossible for the address in the migration header to be wrong since the servers on the group machines are trustworthy and no packet tampering can take place. (2) On the other hand, in network environments where packet tampering is possible, each machine must digitally sign the agent so that a group allowance can not be increased during transit. Of course, if packet tampering is possible, the agent must be digitally signed for many other reasons as well, such as preventing the insertion of code, modifications to variables, etc. In this case, the Agent Tcl server on each machine can be configured to accept only signed agents (the signer can be the owner or the sending machine).

limits could be provided easily within the existing framework, however.

When the machines are not under single administrative control, matters become much more complex. One solution would be to standardize on a maximum number of migrations and children per agent. Each agent carries along a count of migrations and children so far, and each agent server increments these counts when appropriate. When the agent reaches the limit, its current server denies permission to create a new child or to migrate again. Unfortunately, this solution has three major drawbacks. It does not prevent someone from flooding the network with multiple agents, it is vulnerable to a malicious machine that provides the "service" of resetting the counts of any agent that passes through it to zero, and it imposes an absolute resource limit on all agents, rather than allowing an agent to "pay" for as many network resources as it needs.

A more attractive solution is to use a market-based approach in which agents pay for their resource usage with cryptographically-protected electronic cash [CB97, SD95]. When an agent is created, it is given a finite currency supply from its owner's own finite currency supply. The currency does not need to be tied to legal currency, but it must be impossible to spend a currency unit more than once, and it must be impossible for a user to quickly accumulate an arbitrarily large supply. The agent pays for its resource usage with its currency and shares its currency with any child agents that it creates. Eventually the agent and all its children run out of currency and are sent back to the home machine, which either provides more currency or terminates the agent. This market-based approach raises two important issues.

- Payment granularity. An agent could pay to migrate onto a machine, but then be subject to absolute resource limits while on that machine. Alternatively, an agent could pay for every resource that it uses, such as CPU time, memory, disk space, and network bandwidth, obtaining as much capacity as it wants as long as its financial resources hold out.[14] Both models have their advantages. The first model is more efficient since it involves only a single payment per machine. In addition, since an agent knows the size of this single payment before it migrates, it also knows exactly how much currency it should have left when it leaves each machine, making it relatively easy to detect when a malicious machine has stolen currency and to identify the exact machine responsible.[15] The second model is more attractive from an economics standpoint since it allows a machine to charge for exactly the resources that the agent uses. On the other hand, it is less efficient and much harder to audit, After all, who is to say aside from the machine itself whether an agent uses 0.5 CPU seconds or 0.6.

- Legal versus nonlegal currency. Using legal currency allows an agent owner to send out as many agents as she can afford, and allows a service provider to make *real* money, which can be given to its own agents, used to maintain the service, or simply viewed as profit. On the other hand, a malicious machine that steals electronic cash is stealing *real* money. Although this theft is not necessarily harder to detect, an audit will be much more involved, since the auditor must decide whether to return *real* money to an aggrieved agent (and *must* impose a corresponding fine on the malicious service provider).

  Using nonlegal (but universal) currency makes the audit process simpler. In

[14] There will still be absolute resource limits, of course, but they might be set much higher.

[15] Machines would be required to log all incoming agents and their current currency reserves and to provide this information whenever an aggrieved agent requested an audit from an authorized third-party. There would be a time limit on audit requests so that a machine could safely truncate its logs.

a borderline case, for example, the auditor could return a limited amount of *fake* money to an aggrieved agent without imposing a corresponding fine on the service provider, simply because the auditor would not be losing real money. On the other hand, a machine can do nothing with collected currency aside from distributing it to its own agents, since the currency has no meaning outside of the agent system. In addition, fake currency opens up the issue of how to generate and distribute the currency, such that each owner has sufficient currency for their legitimate tasks, but the global currency supply does not grow without bound. Injecting a fixed amount of currency into the system on startup and relying on normal economic processes seems insufficient, since some owners will mainly have client agents that spend the currency for network services; the currency supply of these owners will steadily shrink.

Since the sole purpose of electronic cash here is to prevent an owner from flooding the network with agents, it seems that a hybrid *subscription* approach is best. An owner subscribes to a trusted banking service, paying real currency to get a certain amount of fake currency *per day*. The subscription rates might be nonlinear so that a single user can obtain a modest amount of fake currency for free (or nearly free), but cannot obtain a large amount without a significant cash outlay. The agent spends this fake currency to migrate onto service machines, but the spent currency is not added to the service provider's supply. Instead, once the banking system verifies that the fake currency has not been spent before, the fake currency simply ceases to exist.[16] Otherwise, the obvious attack is to set up a fake (or real) service, collect a large amount of currency, and then flood the network with agents. For a similar

[16] If an agent spends currency on its owner's machines, the currency can be collected and reused. Alternatively, certain machines might not charge certain users.

reason, currency that is unspent after a twenty-four hour period becomes invalid, so that an owner can not hoard currency for a later flooding attack. Of course, using fake currency to control agent propagation does not prevent a service from charging real money at the application level.

Many other agent projects plan to use electronic cash to control agent propagation, including Tacoma [JvRS95], Ara [PS97], and Messengers [BFD96]. Little implementation work has been done by any of these projects. Agent Tcl does have a simple banking system that provides cryptographically-protected digital cash, but machines do not yet charge agents for migration or other services.

6.3 Protecting the agent

Protecting an agent from a malicious machine is the most difficult security problem. Unless "trusted (and tamper-resistant) hardware" is available on each agent server [CGH+95], something which is extremely unlikely in the near future, there is no way to prevent a malicious machine from examining or modifying any part of the agents that visit it. Thus, the real problem is not to prevent theft and tampering, but instead to prevent the machine from using stolen information in a meaningful way and to detect tampering as soon as possible, ideally as soon as the agent migrates onto the next machine. Unfortunately, there is no single mechanism that can solve this problem, and it is unlikely that there will ever be a complete *technical* solution, due to the unimaginable variety of theft and tampering attacks that can be mounted against a visiting agent. Instead, some part of the solution will always be sociological and legal pressures [CGH+95]. There are several partial technical solutions, however. Hopefully, by picking and choosing from these partial solutions, most agents will be able to protect themselves adequately for their current task, but still move freely throughout the network. Before considering some of these partial solutions, it is worthwhile to consider two broad categories of tampering attacks.

- *Normal routing.* The malicious machine allows the agent to continue with its normal itinerary, but holds the agent longer than necessary, charges the agent extra money, or modifies the agent's code or state. Holding the agent longer than necessary prevents a time-critical agent from accomplishing its task. Modifying the agent's code or state causes the agent to perform some work on behalf of the malicious machine, take some dangerous action, or simply reach an incorrect result. These modification threats are why Agent Tcl agents currently become *anonymous* as soon as they migrate through an untrusted machine.

- *Rerouting.* The malicious machine reroutes the agent to a machine that it would not have visited under normal circumstances, or prevents the agent from migrating at all and pretends that it is the next machine on the agent's normal itinerary. The latter attack might be used against an agent that is migrating through a sequence of service providers, attempting to find the best price for some service or product. A service provider can hold the agent on its machine, masquerade as the other service providers, and report higher prices than its own price. Although such an attack requires the service provider to recognize what a particular agent is doing and then update the agent's state as if it had actually visited the other machines, many applications will involve pre-packaged agents that users purchase from the application developers. Recognizing and fooling these well-known agents will not be difficult.

Now, with both theft and these two tampering attacks in mind, we can consider the partial solutions.

- *Trusted machines and noncritical agents.* The first solution is simply to realize that many agents do not need protection at all, either because they are performing some noncritical task (e.g., an anonymous agent interacting with a free search engine), or because they operate entirely on trusted machines (e.g., an agent that is installing new software on a department's machine). Trusted machines can include not only all the machines in your own department, but also machines belonging to large, well-known corporations, such as America Online, Microsoft, Netscape, and United Airlines.

- *Partitioning.* An agent can migrate through trusted machines only, such as a set of general proxy sites under the control of a trusted Internet service provider. Then it either interacts with untrusted resources from across the network using standard RPC, or sends out child agents that contain no sensitive data and will not migrate again, instead just returning their result. More complicated partitioning schemes can be used if needed. In fact, partitioning can achieve as much client protection as in traditional distributed computing, since the sensitive portion of the agent can always be left on the home machine.

- *Replication and voting.* Tacoma uses a replication and voting scheme to handle malicious machines that either terminate an agent outright or provide the agent with incorrect information [MvRSS96]. Here, if the task requires a single agent to visit n services in sequence, the application instead sends out several agents, each of which visits distinct but supposedly equivalent copies of the n services. The agents exchange results after each stage, each agent keeping the majority result. Although this scheme prevents many kinds of attacks, it also has several drawbacks. First, there must be multiple copies of each service[17]; in addition, since the copies might be functionally equivalent but not identical, the agent must be able to handle different interfaces and different result formats. Second, if the agents are spending money to access the services, the user will spend much more money than if a single agent had migrated through a single copy. Finally, the cryptographic overhead is large. Despite these disadvantages, replication and voting schemes will be used in many agents, since they are the only way to handle services that provide incorrect information (assuming that the incorrectness cannot be easily detected). Tacoma also includes *rear-guard* agents that restart a vanished agent.

- *Components.* Perhaps the most powerful idea is to divide each agent into components [CGH+95]. Components can be added to the agent as it migrates, and each component can be encrypted and signed with different keys. The agent's static code and the variables whose values never change would make up one component, and would be signed with the owner's key before the agent left the home machine. If a malicious machine modifies the code or variables, the digital signature becomes invalid and the next machine in the migration sequence will immediately detect the modification. In addition, if an agent obtains critical information from a service, it can put this information into its own component. Then the component is signed with the machine's key to prevent tampering, and can even be encrypted with a trusted machine's key (e.g., the home machine or a proxy site) so that other machines cannot examine it. Of course, the agent must return to that trusted machine before it can use the information again

[17] And the copies cannot be under the control of a single organization. Otherwise all the copies might have the same malicious behavior.

itself. Similarly, any code or data that is not needed until the agent reaches a particular machine can be encrypted with that machine's key. For example, an agent might encrypt the bulk of its electronic cash with a proxy site's key, so that it could migrate through untrusted machines without worrying about theft. The agent would return to the proxy site when it needed to spend the cash. Depending on the migration model, this component approach also allows a machine to place greater trust in an agent that has migrated through untrusted machines. For example, if the code to be executed on the current machine is in its own component, digitally signed with the owner's key, and this code does not depend on any volatile variables, the code can be executed with the owner's permissions, rather than as *anonymous*. Finally, components make it easier for an agent to use the partitioning approach above; an agent can leave a particular component behind on a trusted machine, or can create and send out a child agent that includes only certain components.

- *Self-authentication.* In most agents, certain parts of the agent's state will change as the agent migrates from machine to machine, such as the variable values and the control information on the interpreter's stack. Although it is impossible to detect all malicious modifications to this state information, it is possible to construct an authentication routine that will examine the state information for any obvious inconsistencies or impossibilities [PS97]. The authentication routine could also examine the current set of components. Such an authentication routine would be placed in its own component and digitally signed with the owner's key. Each agent server would execute the authentication routine, terminating the agent (and notifying the home machine) if the routine finds any inconsistencies. The authentication routine would run as *anonymous* and would only have authority to examine the state image. Like the components themselves, such an authentication routine allows a machine to place greater trust in an agent that has migrated through untrusted machines.

- *Migration history.* It is possible to embed a tamper-proof migration history inside a moving agent [MvRSS96]. This movement history allows the detection of some rerouting attacks, particularly if an agent is following a fixed itinerary, and, in combination with additional digital signatures, makes it impossible for a malicious machine to drop an entire component from the agent. The movement history could also be examined inside the authentication routine above.

- *Audit logs.* Machines should keep logs of important agent events so that an aggrieved agent or owner can request an audit from an authorized third-party [CGH+95]. The auditor would seek to identify the machine responsible for a theft or modification and penalize that machine appropriately. The exact contents of the audit logs is largely an open question. It is clear that all electronic-cash transfers must be logged, however, so that a machine cannot steal electronic cash without providing the desired service. Of course, a malicious machine can construct a false log, so the auditor must look for log entries that are inconsistent with log entries from other machines, rather than just log entries that explicitly indicate a malicious action. In addition malicious machines can collude in their logging to make an honest, intervening machine *look* malicious. Thus, in some situations, the auditor can impose serious sanctions only after it has observed an apparent attack happening to *multiple* agents (that are following different migration trajectories).

- *Encrypted algorithms.* Recent work [San97, Hoh97] involves encrypting a program and its inputs in such a way that (1) the *encrypted* program is directly executable, (2) the encrypted program performs the same task as the original program, and (3) the output from the encrypted program is also encrypted and can only be decrypted by the program encrypter. Although this work is in its infancy and remains either theoretical or unproven, it has great promise for mobile-agent systems, since it would become much harder for a malicious machine to make a targeted modification, i.e., a modification with a known, useful effect, to an agent or its state.

Even taken together, these techniques cannot provide complete protection. In addition, many of the techniques involve substantial cryptographic and logging overhead, forcing an agent to trade performance for protection. Most agents should be able to realize adequate protection through some combination of these techniques, however, while still maintaining reasonable performance. The overriding issue is how to design a protection interface that allows the agent to easily use the desired combination of techniques.

Chapter 7

Performance analysis

Different applications have different performance constraints. Some applications need to continue interacting with a user or resource even if a network link goes down; other applications need to minimize network traffic; and still others need to minimize total completion time or per-operation completion time. By nature, mobile agents are an attractive choice for the first two kinds of applications, since they can move to the other side of an unreliable link or closer to the necessary information sources. Whether they are an attractive choice for the last kind of application, however, depends on their migration latency, communication latency, and execution speed. In this chapter, we examine exactly these three measures, comparing mobile agents against traditional client-server systems based around TCP/IP and remote procedure call (RPC). The current Agent Tcl system has not been turned for performance, and, as we will see, it suffers from high migration overhead and the slowness of interpreted languages. The performance numbers are good enough, however, to suggest that a combination of faster languages and additional system engineering will make Agent Tcl competitive even for compute-intensive applications in high-performance networks.

We limit the scope of the performance analysis in three ways.

- *Tcl agents.* All agents are written in Tcl. Tcl is the only language for which implementation work is sufficiently complete. Java agents cannot establish meetings, and Scheme agents cannot migrate, making it impossible to run a full set of performance experiments for Java and Scheme agents.

- *Anonymous agents.* The agents do *not* use encryption; they are neither encrypted nor digitally signed. Agent Tcl currently uses a separate PGP process for encryption and digital signatures. This approach is convenient and flexible, but also extremely slow, performing far worse than the encryption subsystems found in secure client-server systems. Thus, since we are planning to replace PGP with a faster encryption library later, it did not seem worthwhile to do a detailed performance comparison of secure agents and secure client-server computing now.

- *Low-level operations.* We measure the performance of low-level operations rather than entire applications. Low-level operations include migration, sending and receiving messages, and sending and receiving messages over a *meeting*. Since there are several obvious performance enhancements that can be made to Agent Tcl, and the low-level measurements all confirmed the need for these enhancements, it did not seem worthwhile to devote time to higher-level measurements. Instead, application measurements will wait until the two faster languages, Java and Scheme, are in place, and a few of the major performance enhancements are finished.

Despite this limited scope, the analysis highlights several areas of good performance as well as several needed performance enhancements. The analysis is divided into two sections. The first section examines the base performance of Agent Tcl, identifies the needed performance enhancements, and compares mobile agents with traditional cross-network communication. The second section develop a simple formula that, under simplifying assumptions about processor and language speeds, indicates whether an application should be written as a stationary client process or as a migrating Tcl agent.

7.1 Base performance

Two machines were used to measure base performance. The first machine, *bald.cs.dartmouth.edu*, is a 200 MHz Intel Pentium running Linux 2.0.0. It has 16 megabytes of physical memory and 128 megabytes of swap space. The second machine, *q.cs.dartmouth.edu*, is a 133 MHz Intel Pentium running FreeBSD 2.1.6.1. It also has 16 megabytes of physical memory and 128 megabytes of swap space. The two machines are connected with a 10 megabit Ethernet and one router. All single-machine experiments were performed on *bald*. In the two-machine experiments, the client or source machine was always *q*, and the server or destination machine was always *bald*.

These two machines and their experimental roles are not meant to reflect any average or generic network. They are simply one particular example of a server and lower-powered client connected with a reliable, low-latency, high-bandwidth network. Such a network environment is the most interesting for the initial round of performance measurements, since *if* a mobile agent has a performance advantage over traditional client-server systems in a mid-performance network, this advantage will usually increase as latency, bandwidth and reliability become worse. The two exceptions relate to code size and processor power.

- If the agent's code size is greater than the size of the intermediate results, and network bandwidth drops, the additional time needed to transmit the agent might become the dominant factor, increasing the total completion time.

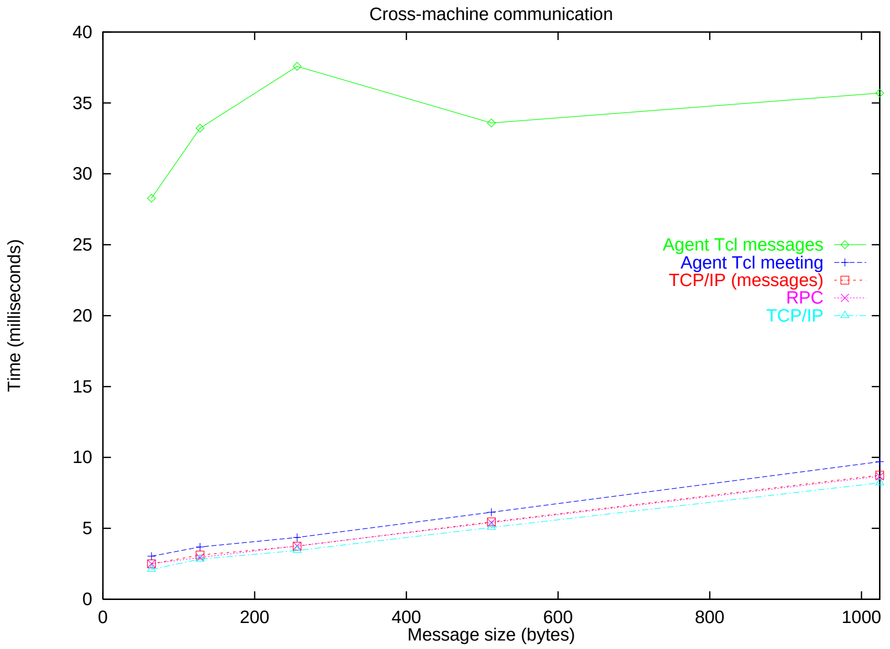
- If the processor power of the server decreases or the processor power of the client increases, an agent sent to the server might no longer outperform native code on the client, even if the agent reduces network usage.

If the agent's code size is less than the size of the intermediate results, however, and the client and server machines do not change, the relative performance of the mobile agent must increase as network performance drops, since the agent transmits less data across the network (and makes the same or fewer cross-network connections). It is this observation that led us to do the initial performance measurements in a mid-performance network.

7.1.1 Inter-agent communication

The two graphs in Figure 7.1 compare communication times for agents and traditional client-server techniques when the two communicating entities are on *different* machines.[1] The *TCP/IP* curve shows the time needed for a (non-agent) client and server process to exchange a request and response over an *already connected* TCP/IP connection. Each request begins with two four-byte integers, one that specifies the size of request and another that specifies the size of the desired response; the rest of the request is just dummy data. The server waits for a client connection and then sits in a tight loop, receiving requests and sending back dummy responses of the appropriate size. Then client connects to the server and then times how long it takes to exchange one hundred request and responses (for each request and response size). This process is repeated multiple times over a twelve-hour period [2]. Then, If we end

[1]The data for Figure 7.1 was taken from Tables A.12, A.10, A.7, A.8 and A.6.
[2]9 p.m. to 9 a.m.



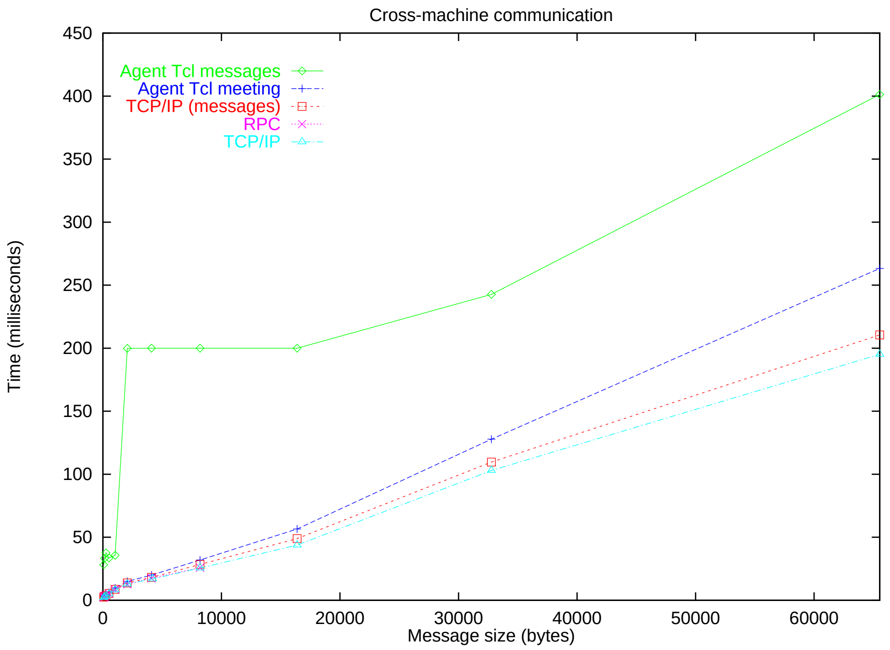


Figure 7.1: Time in milliseconds for two processes or agents on different machines to exchange a request and a response. Each data point is an average of several timings; the largest standard deviation is 5.3 percent. Note that the two graphs show the same data; the top graph just shows the data for a smaller range of message sizes, so that the details are clearly visible. Also the RPC curve ends at a message size of 8,192 bytes due to an argument-size limitation.

up with n times for each request and response size, the longest n / 2 times are thrown out.[3] The remaining times are averaged and then divided by one hundred to get the average per-request time. All of the average times are shown in Appendix A; the graphs only plot the times for which the request and response sizes are the same.

The *RPC* curve is the same as the *TCP/IP* curve except that the client and server process are using Sun RPC over UDP. Here, the request is the single argument to the remote procedure, and the response is the result value. Both the argument and result value are strings. The *RPC* curve only goes up to a request/response size of 8,192 bytes, since the RPC implementation on the client machine could not handle arguments larger than that size.

The *TCP/IP (messages)* curve is the same as the *TCP/IP* curve except that the client and server process are using the messaging subsystem from Agent Tcl. The `send` operation in this message subsystem accepts a structure that contains the message elements, serializes the message elements, and sends the resulting bytestream across the connection. Similarly, the `receive` operation receives a bytestream from the connection, unserializes the bytestream, and returns a structure that contains the message elements. In this case, the message consists of a single buffer of binary data. *TCP/IP (messages)* is about 5 percent slower than *TCP/IP*. The slowdown is from the serialization and unserialization routines and three dynamic-memory allocations; the three allocations are a buffer for the outgoing serialized message, a buffer for the incoming serialized message, and a buffer for the single message component.

The *Agent Tcl messages* curve is the time needed for two agents to exchange a request and response; the *Agent Tcl meetings* curve is the time needed for two agents

[3] The experiments were *not* run on an isolated network. Hopefully, by throwing out the longest n / 2 times, we end up keeping only those times for which there was little other network traffic.

to exchange a request and response over an *already established* meeting.[4] Although meetings use TCP/IP and the same messaging subsystem as the *TCP/IP (messages)* curve, the performance of meetings is about 20 percent worse than that of the *TCP/IP* curve, rather than just 5 percent. There are three sources of additional overhead: (1) parsing the Tcl commands that send and receive the message, (2) setting a timeout on each receive (and removing the timeout when the message arrives), and (3) using asynchronous I/O (rather than a blocking `read` or `select`) to detect when a message has arrived over the meeting. Since the Tcl version is not multi-threaded, asynchronous I/O is needed for event-driven agents, in which an incoming message cause the immediate execution of some Tcl procedure.[5] Although some optimization is possible, these three sources of overhead cannot be reduced significantly, except that a compiled agent language will avoid most of the parsing overhead.

Messages, on the other hand, range from two to sixteen times slower than the *TCP/IP* curve, even though they also use TCP/IP and the messaging subsystem. Messages suffer from the same three sources of overhead as meetings, but also have two more critical inefficiencies.

- *Connections.* Each message involves a new TCP/IP connection between the sending agent and remote server. Worse, most TCP/IP implementations do not acknowledge the first packet sent over a new connection immediately, and do not send the second packet until this acknowledgment is received [Ste94].[6] Specifically, once the first packet arrives over a connection, the receiving ma-

[4]The time needed to establish the meeting will be considered later, when we look at the total time needed for an agent to migrate onto a remote machine, interact with a service agent, and return a result to the home machine.

[5]The Java version does not use asynchronous I/O. It creates a *watcher* thread for each meeting.

[6]This scheme is known as *slow-start*.

chine waits for data going in the opposite direction along the connection. If data shows up, the machine piggybacks the acknowledgment onto the data packet. If data does not show up within a timeout interval (typically 200 milliseconds), the machine sends a separate acknowledgment packet. The sending machine sends the second packet only when it receives the acknowledgment. You can see this effect in Figure 7.1 where the *Agent Tcl messages* time jumps from 4 milliseconds to 200 milliseconds when the message size goes from 1,024 bytes to 2,048 bytes. 1,024 bytes fit in one packet, so the receiving server immediately gets the entire message and immediately sends back the application-level response indicating that the message was received and buffered; the TCP/IP acknowledgment is piggybacked on this response. 2,048 bytes require two packets, so the server has to wait for 200 milliseconds before the second packet shows up with the rest of the message. The reason that the delay is not 400 milliseconds is that Linux 2.0.0. aggressively reduces the *delayed acknowledgment* timeout based on packet inter-arrival times. In our environment, the timeout is already near zero immediately after the connection has been established. Thus, only the message sent from the Linux machine to the FreeBSD machine encounters the 200 millisecond delay.

Eliminating the delayed acknowledgment simply requires an application-level acknowledgment for the first packet (if the first packet does not contain the entire message). This acknowledgment could be a single dummy byte or (more likely) 4 zero bytes, since Agent Tcl's messaging subsystem interprets 4 zero bytes as an empty message. The messaging subsystem can simply ignore any empty messages that arrive before the real server response. Eliminating the reconnection on every message is more difficult since Agent Tcl is currently based around multiple processes. Although two servers whose agents were communicating heavily could certainly cache the open connection, the time needed to get an outgoing message onto that connection (either by transferring the message or file descriptor to another local process) could easily outweigh the reconnection time. It might reasonable, however, for each server to allow a *few* remote agents to hold open a connection, much as if the agents had established a *meeting* with the server rather than a specific agent. In addition, as more and more of the system is multi-threaded, a dedicated connection between two servers will become more attractive, since accessing that connection would not require interprocess communication.

- *Multiple processes.* There are actually five process involved in receiving a message from a remote agent: (1) a server process that buffers incoming messages, (2) a server process that watches the server's TCP/IP port, (3) a server process that is forked to handle the incoming message, (4) a "background" process that serves as the interface between the server and the recipient agent, and (5) the recipient agent itself. The process that is watching the server's TCP/IP port sees the incoming message and *forks* a process to handle the message. This process transfers the message to the main server process, which buffers the message in its internal queue. Eventually, the background process reaches an idle point and asks the main server process for any new messages. The main server process transfers the incoming message to the background process, which finally transfers it to the recipient agent. The *fork* (and the corresponding process) can be eliminated by replacing the single socket watcher with a pool of socket watchers. A more complete solution, however, is to multi-thread the entire server and have a pool of threads watching the TCP/IP port rather than a pool of processes.

The message is then transferred between processes only once, when it is transferred from the server buffers to the recipient agent. Unfortunately, eliminating this one last transfer (i.e., allowing the server to insert the message directly into the agent's internal queues) requires either multi-threading the entire system or far more flexible shared-memory facilities than most systems provide. Both approaches, however, are reasonable long-term implementation goals.

Although the effect of these two problems is reduced as message size increases (since network transmission time dominates the total time), most messages are apt to be small. Thus the two problems must be addressed. Our immediate plans are to eliminate the delayed acknowledgment (through the application-level acknowledgment), multi-thread the server, and then reexamine communication performance.

Figure 7.2 shows the communication times when the two agents or processes are on the *same* machine.[7] We ran the same experiments as before, except that we now include a Unix domain socket, both with and without the Agent Tcl messaging subsystem. In addition, Agent Tcl meetings and messages work slightly differently when the two agents are on the same machine. Meetings use a Unix domain socket rather than TCP/IP, and the agent does not need to establish a connection for each message, since each agent has a permanent connection with its local server. The message still goes through three extra processes before finally reaching the recipient, however, the background process for the sender, the main server process, and the background process for the recipient. Thus, aside from the delayed TCP/IP acknowledgment, the meetings and messages have the same sources of overhead as when the two agents were on different machines. Since the transmission time over a Unix domain socket is much lower than over TCP/IP, however, the effect of these overheads is much more

[7]The data for Figure 7.2 was taken from Tables A.11, A.9, A.4, A.2, A.5, A.3, and A.1.

pronounced, with meetings twelve times slower than a bare Unix socket and messages fifty-two times slower.

With messages, multi-threading the server will again eliminate two of the extra processes, making messages only twice as slow as meetings rather than four times as slow. In addition, even with each agent in its own process, it is possible to bypass the server entirely when sending a message. For example, each agent could have its own Unix domain socket within some designated directory. Although such a scheme would require careful access control to prevent one agent from masquerading as another[8], it is workable. The performance improvement when sending only a single message is difficult to predict and will probably vary widely from machine to machine; since the system could hold open the connection that is established for the first message, however, sending multiple messages to the same agent should be nearly as fast as sending those same messages over a meeting. In fact, if it turns out that the performance improvement is small for only a single message, we can dispense with the special Unix domain socket and simply have the agents automatically establish a meeting when sending more than one message. As before, improving performance further will require more aggressive multi-threading or more flexible shared-memory facilities.

Finally, as before, a compiled agent language would not have the parsing overhead of Tcl. For meetings, the current overhead (for parsing the Agent Tcl commands that send and receive the messages) is 25 percent (220 milliseconds) for 64-byte messages and 55 percent (33000 milliseconds) for 64-kilobyte messages. For messages, the parsing overhead is 8 percent (300 milliseconds) for 64-byte messages and 28 percent (34000 milliseconds) for 64-kilobyte messages. The parsing actually takes about the

[8]Currently it is the main server process that prevents masquerade attacks.

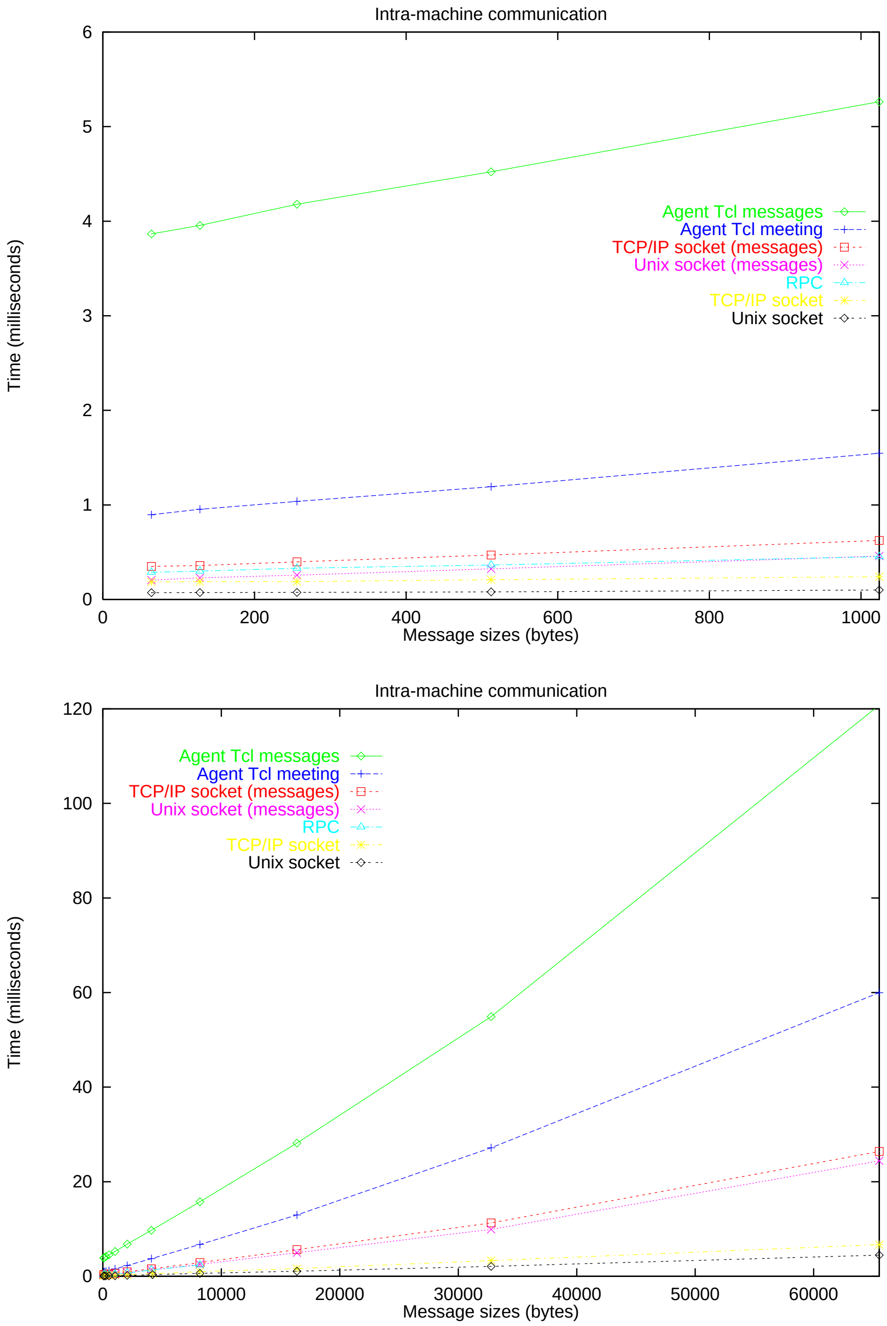


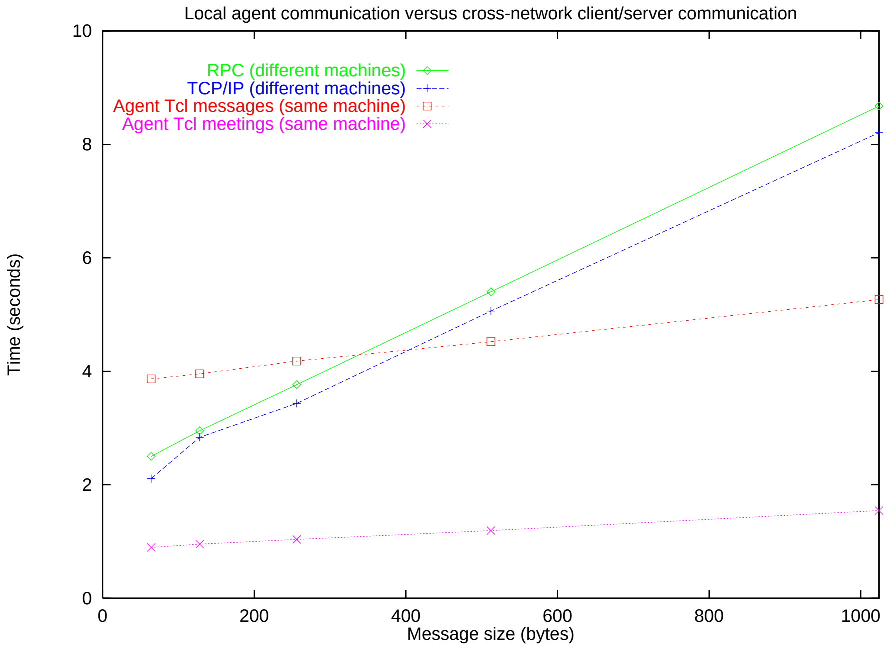
Figure 7.2: Time in milliseconds for two processes or agents on the same machine to exchange a request and a response. Each data point is an average of several timings; the largest standard deviation is 4.7 percent. Note that the two graphs show the same data; the top graph just shows the data for a smaller range of message sizes, so that the details are clearly visible. Also the RPC curve ends at a message size of 8,192 bytes due to an argument-size limitation.

same amount of time for both meetings and messages, but has a much greater relative impact on meetings, which are much faster overall.

Figure 7.3 compares communication times for two agents on the *same* machine with communication times for two client-server processes on *different* machines.[9] Local agent meetings are always faster than cross-network communication (e.g, TCP/IP), ranging from two to six times faster as message size increases, with all but the smallest message size three times faster or more. On the other hand, local agent messages are slower than cross-network communication for data sizes below 400 bytes (due to the high overhead of copying the message between server processes), but 50 percent faster than cross-network communication for data sizes above 1024 bytes. Multi-threading the server and thus eliminating two extra message copies will make local agent messages just as fast as cross-network communication for the smallest messages and more than twice as fast for the largest messages. Bypassing the server altogether will improve performance further, possibly making messages as fast as meetings if multiple messages are sent to the same recipient. Finally, it is easy to believe that nearly 75 percent of the Tcl parsing overhead can be eliminated. For example, the stationary service agent could be written in C or C++, and the migrating agent could be written in Java. Then, local agent meetings would become three to seven times faster than cross-network communication, with all but the two smallest message sizes four times faster or more. Messages would see a similar (but smaller) improvement.

Thus, even though they are written in Tcl, two agents on the same machine can communicate faster than two client/server processes on different machines (in most cases), meaning that we can reduce communication time by dispatching an agent to a remote machine. The question then is how much work the Tcl agent can

[9]The data for Figure 7.3 was taken from Tables A.8, A.6, A.11 and A.9.



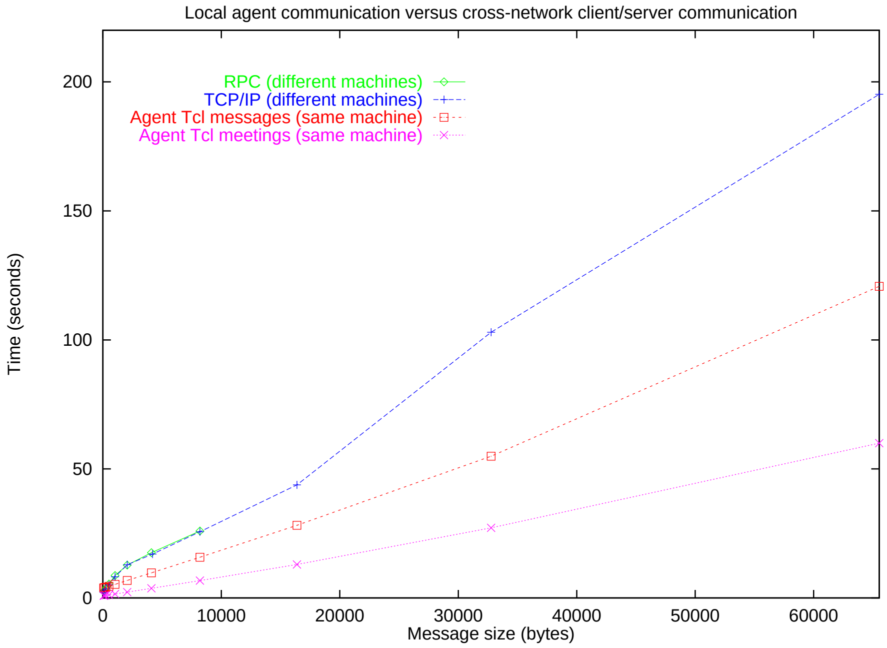


Figure 7.3: The *Agent Tcl messages* and *Agent Tcl meetings* curves are communication times for two agents on the same machine; the *TCP/IP* and *RPC* curves are for two client/server processes on different machines. As in the other figures, the two graphs show the same data, just for a different range of messages sizes, and the RPC curve ends at a message size of 8,192 bytes.

perform before its CPU time exceeds the savings in communication time. As one might expect, the answer depends entirely on the agent's task, especially since Tcl is an interesting mix of high-level string and list commands and low-level control commands such as `if`, `for` and `while`. To get a general impression, however, we will consider two tasks: (1) see if a string contains a given substring (which can be done with a built-in Tcl command called `string` that is implemented in C), and (2) find the minimum element in a list of integers (which must be done with a loop and comparison statement written in Tcl). With meetings, local agent communication is 1.2 milliseconds faster than cross-network communication for 64-byte messages, 3.9 milliseconds faster for 512-byte messages, and 140 milliseconds faster for 64-kilobyte messages. Within these time periods, the agent can determine that a 5K, 17K, or 512K string respectively does *not* contain a given 5-element substring.[10] On the other hand, it can find the minimum integer in a list of only 11, 37 or 1200 integers respectively. For comparison, within these same time periods, a compiled (but non-optimized) C++ program on the same machine can determine that an 18K, 58K, or 2,090K string respectively does not contain a given 5-element substring, and can find the minimum integer in a list of 4800, 15,600, or 560,000 integer respectively. These measurements lead to two conclusions. First, a Tcl agent is interested in end-to-end latency can do only a small amount of work between each resource access; otherwise the Tcl agent will take longer than a traditional client/server system that accesses the resource from across the network. Second, excluding the small amount of CPU time that the agent-enabled server saves by not writing intermediate data onto the network, an agent-enabled server might see a CPU load as much as 500 times higher than a server that provides the desired high-level operation directly (rather

[10]The string and the given substring have no characters in common.

then providing low-level primitives and allowing an agent to combine those primitives itself).

Of course, the actual increase in CPU load depends entirely on the resources that the server provides and the languages that it supports. For example, if a resource maintains a large database and allows high-level queries against this database, the additional CPU time for an agent that makes a few independent queries against the database might be insignificant, even if the agent is written in Tcl. Also, Java and Scheme 48, which have already been integrated into the system, are only ten to twenty times slower than natively compiled code, an overhead that more servers will be willing to accept. In addition, there are faster execution environments such as Omniware that can execute an agent only 25 percent slower than natively compiled code on average [ATLLW96]; if such an environment is incorporated into Agent Tcl, the CPU load on the server (excluding migration overhead) will be about the same regardless of whether the server provides low-level operations and accepts agents *or* provides high-level operations and does not accept agents.[11]

7.1.2 Agent migration

Figure 7.4 shows the time needed to create a child agent on either the local machine or a remote machine (and for the child agent to send a 64-byte dummy result to its parent).[12] As can be seen from the *Agent Tcl messages* curves in Figures 7.1 and 7.2, the time needed to send the result back to the home machine is relatively small.

[11] Of course, the server would need to discourage agents from using slower languages, such as imposing a harsh CPU-time limit, charging those agents more money, or disallowing those agents altogether. Agents that were then unwilling or unable to migrate onto the machine would have to interact with the resource from across the network.

[12] The data for Figure 7.4 was taken from Tables A.14 and A.13.

Thus, agent creation is currently an inefficient operation, taking over a tenth of a second for even the smallest agent. Most of the overhead comes from starting up a Tcl interpreter in which to execute the new agent. On our destination machine, it takes approximately 60,000 microseconds for the server to exec the interpreter and for the interpreter to read its initialization files. In addition, when the agent is created on a remote machine, the destination server must fork twice, once to create the request handler and once again to exec the interpreter. When the agent is created on the same machine, the local server must fork only once to exec the interpreter; the request handler already exists. In both cases, the request handler must communicate with the main server process to add the agent to the internal tables, and must copy the registration information and the child agent's state information into the new interpreter process.

Multi-threading the server will help here just as it helps with messages, eliminating one fork and the interprocess communication that registers the new agent with the main server process.[13] Clearly, however, the main point of attack must be the interpreter startup time. One intermediate approach, which would cut startup time approximately in half, is to embed the initialization files inside the Tcl interpreter's code, so that they do not have to be read from disk. A more attractive approach, however, is to maintain a pool of interpreter processes, each of which has already read the initialization files. Then, an incoming agent is simply passed to one of these available interpreters. In addition, to replenish the pool efficiently (i.e., to provide good migration times even under high load), one of the pool interpreters should always remain unused; when there are no other free interpreters left in the pool, this unused interpreter forks to create new interpreters. It seems likely that, except for

[13] The *fork* is only eliminated for child agents created on a remote machine.

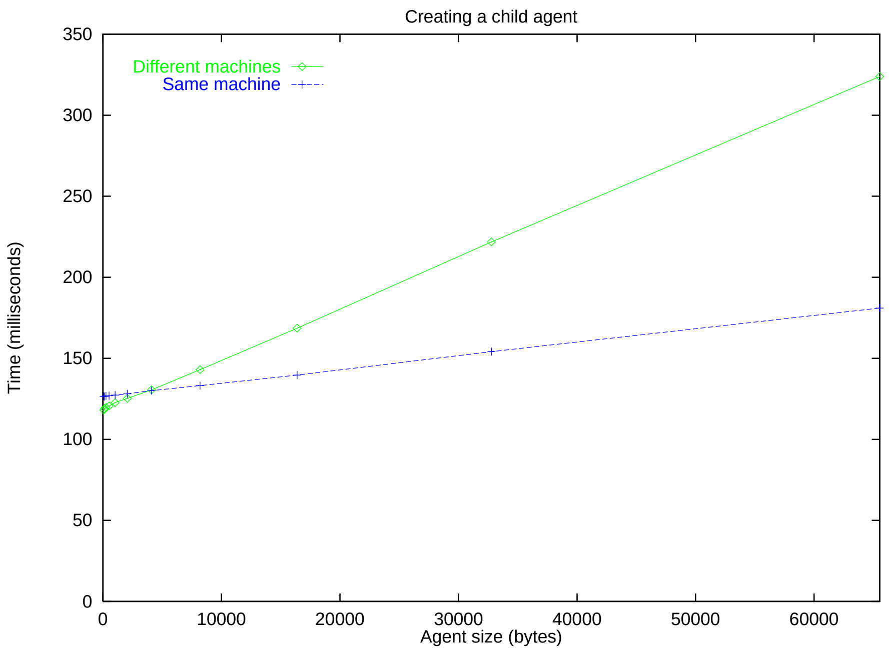


Figure 7.4: Time in milliseconds to create a child agent and for that agent to send back a 64-byte dummy result. For small agent sizes, it takes longer to create a child agent on the *local machine*, probably due to context switches and the impossibility of any computational overlap. As before, each data point is the average of multiple timings; the largest standard deviation is 10.4 percent (but the next largest deviation is only 6.6 percent percent).

the time needed to capture and restore the state information, agent creation can be made nearly as efficient as sending a message.

Figure 7.5 shows the time needed for an existing agent to migrate from the home machine to the destination machine and then back to the home machine.[14] One important note is that, in contrast with the other experiments and plots, the initial and final sizes are just the size of the dummy variable that is used to pad the agent. Every agent includes approximately 2K of additional identification and state information. Thus, every agent encounters the delayed acknowledgment problem, which accounts for 200 milliseconds of the migration overhead.[15] The rest of the overhead is from the forks and interprocess communication in the server and the time needed to start up *two* interpreters, one when the agent journeys to the remote machine and another when the agent returns to the home machine. The time needed to start the interpreter on the remote machine is 60 milliseconds as before; the time needed to start the interpreter on the slower home machine is approximately 100 milliseconds. Fixing the delayed acknowledgment, multi-threading the server, and maintaining a pool of ready interpreters should reduce migration time to just the time needed to send a message plus the time needed to capture and restore the state information. The capture and restoration time ranges from about 3 milliseconds for a 64-byte agent to about 76 milliseconds for a 64-kilobyte agent. Capture and restoration is inefficient in the current system, however, since the state information is packaged as a human-readable Tcl list for implementation and debugging convenience. Packaging the state

[14] The data for Figure 7.5 was taken from Table A.15.

[15] Note the the delayed acknowledgment problem was *not* encountered when creating child agents (Figure 7.4). The child agents were sent to the Linux machine, which does not implement the delayed acknowledgment, and the dummy result sent back to the FreeBSD machine was small enough to fit in one packet.

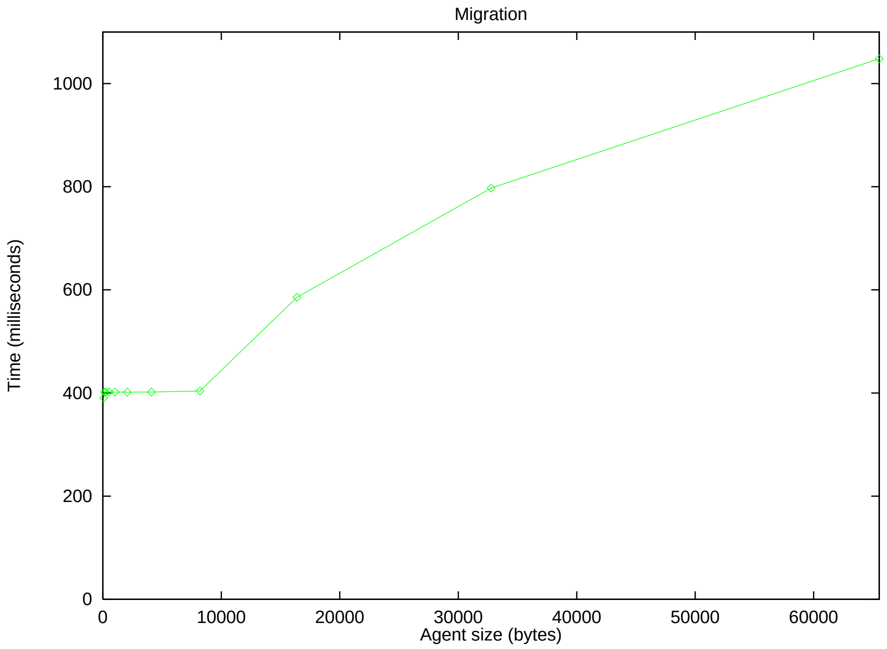


Figure 7.5: Time in seconds for an agent to jump to a remote machine and then jump back to the home machine. The "initial size" is the size of the agent when it leaves the home machine; the "final size" is the size of the agent when it returns to the home machine. As before, each data point is an average of multiple timings; the largest standard deviation is 6.8 percent.

information as a binary bytestream instead should reduce capture and restoration time by more than a factor of two.

Combining the current creation and migration times with the 11.9 milliseconds needed to *establish* a meeting, we have a somewhat negative result. When a 1-kilobyte agent is created on a remote machine and then establishes a meeting with a service agent, the agent must make at least 113 64-byte requests, 35 512-byte requests or 1 64-kilobyte request over the meeting before the time savings from the more efficient *local* communication outweighs the creation time. Similarly, when a 1-kilobyte agent migrates and establishes a meeting (and later migrates back), it must make at least 345 64-byte requests, 106 512-byte requests, or 3 64-kilobyte requests. Thus, in its current form, Agent Tcl cannot be used for applications in which end-to-end latency is the primary concern, except if (1) the network is extremely slow, (2) the intermediate results are extremely large, or (3) the downtime of an unreliable network link contributes significantly to the end-to-end latency. The performance improvements outlined above, however, should drop these thresholds to only a handful of requests. More specifically, if we eliminate interpreter startup time (60 and 100 milliseconds on the two machines used in the experiments), eliminate the delayed acknowledgment (200 milliseconds), improve the state capture and restoration routines (2 to 38 milliseconds as agent size increases), and multi-thread the server to eliminate forks and interprocess communication (tens of milliseconds for all agent sizes plus additional time as agent size increases), we will reduce the creation times in Figure 7.4 by nearly 100 milliseconds for the smallest agents; similarly, we will reduce the migration times in Figure 7.5 by nearly 400 milliseconds for the smallest agents. Conservatively assuming that this leaves us with 25 milliseconds to create a 1K child agent and 50 milliseconds to migrate a 1K agent to and from a remote machine, the child agent only needs to make 21 64-byte requests, 7 512-byte requests, or 1 64-kilobyte request, and the migrating agent only needs to make 42 64-bytes, 13 512-byte requests, or 1 64-kilobyte request.

A final performance note is that there is currently no overlapping of communication and processing. With both incoming agents and messages, the system first receives the entire bytestream, then unserializes the bytestream, and finally processes the agent or message. Overlapping these activities could lead to a significant performance improvement (when the agent or message is coming from a *different* machine), since at least some of the processing time would be hidden inside the transmission time. Unfortunately, although conceptually straightforward, such overlapping requires significant implementation work.

7.1.3 Summary

Although several optimizations can be made to the communication subsystem, the current communication times are promising and are sufficient for many applications. Instead the main problem in the current system is the migration overhead. Reducing the migration overhead and integrating faster languages should allow Agent Tcl agents to be at least competitive with traditional client-server computing in mid-performance networks such as our 10 megabit Ethernet, and significantly better in slower or less reliable networks.

7.2 When to migrate

Migration only makes sense under certain network and resource conditions. In this section, we consider an application that needs to access a network resource to perform its task and that wants to minimize its total completion time. We derive a formula that indicates whether the application should be written as (1) a stationary client process that interacts with the resource over a standard TCP/IP connection or (2) a mobile agent that migrates to the location of the resource, interacts with the resource using the Agent Tcl communication primitives, and then migrates back to the home machine. To simplify the formula, we make four assumptions.

- The stationary client and the mobile agent are written in the same language.
- All machines in the network are the same and are lightly loaded.
- Links between machines never go down.
- Machines never go down.

With these four assumptions, the formula does not have to take into account machine speeds, machine loads, the expected downtimes of machines and links, or the relative execution speed of native and mobile code.

The data used to derive the formula was obtained experimentally in an isolated network of two machines. Each machine was a 133 MHz Pentium laptop running Linux 2.027. Each machine had 16 MB of physical memory and 64 MB of swap space. Depending on the experiment, the two machines were connected with a 28.8 Kb/s modem link, a 2.0 MB/s wireless Ethernet link, or a 10.0 MB/s wired Ethernet link.

Figure 7.6 shows the time needed for two agents on the same laptop to exchange a request and response, either over an Agent Tcl meeting or with the `send` and `receive` primitives.[16] The time is plotted as a function of the total number of bytes transferred, i.e., the request size *plus* the response size. As in the previous section, the meeting data does *not* include the time needed to establish the meeting.

Computing the best-fit linear functions with discrete least-squares approximation, we see that the time t in milliseconds for the two agents to exchange an S-byte request and an R-byte response over an Agent Tcl meeting is

$$t = 0.00066(S + R) + 0.65 \tag{7.1}$$

Similarly, the time for the two agents to exchange a request and response using the `send` and `receive` primitives is

$$t = 0.0013(S + R) + 4.3 \tag{7.2}$$

[16] The data for Figure 7.6 was taken from Tables B.2 and B.1.

The top graph in Figure 7.7 shows the time needed for a client and server process on different laptops to exchange a request and response over a 28.8 Kb/s modem connection.[17] As before, the time is plotted as a function of the total number of bytes transferred. The bottom graph in Figure 7.7 shows the time needed for a Tcl agent to migrate from one laptop to the other and back, again over a 28.8 Kb/s modem connection. As in the client/server case, the time is plotted as a function of the total number of bytes transferred, i.e., the agent's initial size (its size when it leaves the home machine) *plus* the agent's final size (its size when it returns to the home machine). Similarly, the graphs in Figure 7.8 show the client/server and mobile-agent times for a 2.0 Mb/s wireless Ethernet connection, and the graphs in Figure 7.9 show the times for a 10.0 Mb/s wired Ethernet connection.[18] As in the previous section, the client/server data does *not* include the time needed to establish the connection.

Once again approximating the data with a best-fit linear function, we obtain three equations for the time that it takes the client and server processes to exchange a request and response.

$$t = 0.1785(S+R) + 272 \qquad \text{(28.8 Kb/s modem)} \tag{7.3}$$

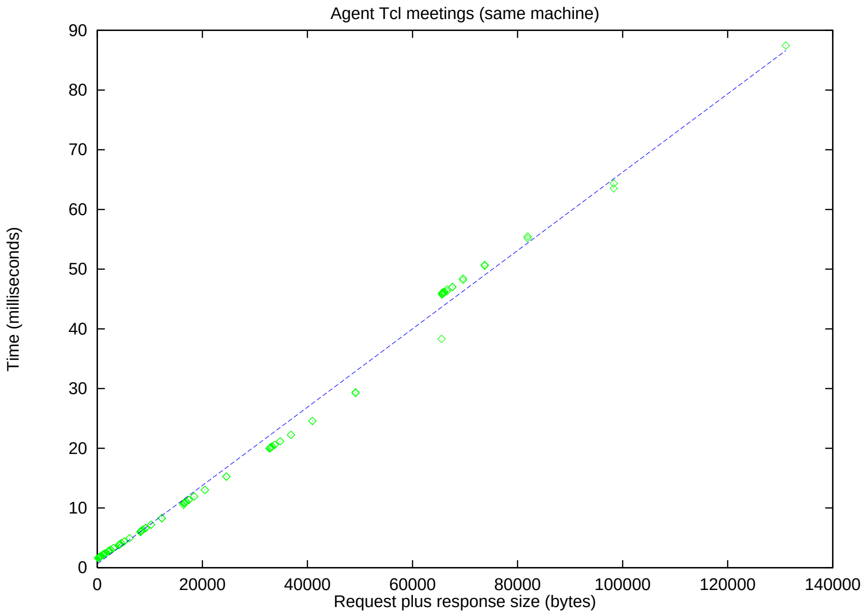
$$t = 0.0070(S+R) + 17 \qquad \text{(2.0 Mb/s wireless Ethernet)} \tag{7.4}$$

$$t = 0.0011(S+R) + 1 \qquad \text{(10 Mb/s wired Ethernet)} \tag{7.5}$$

Similarly, we obtain three equations for the time that it takes the Tcl agent to make its round-trip migration. In these three equations, I is the initial size of the

[17] The data for Figure 7.7 was taken from Tables B.3 and B.4.

[18] The data for Figure 7.8 was taken from Tables B.5 and B.6. The data for Figure 7.9 was taken from Tables B.7 and B.8.



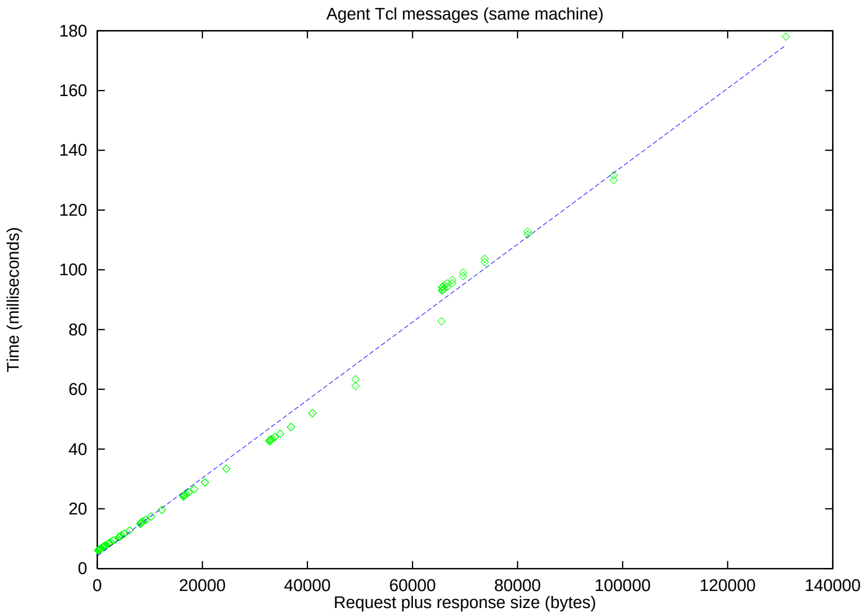
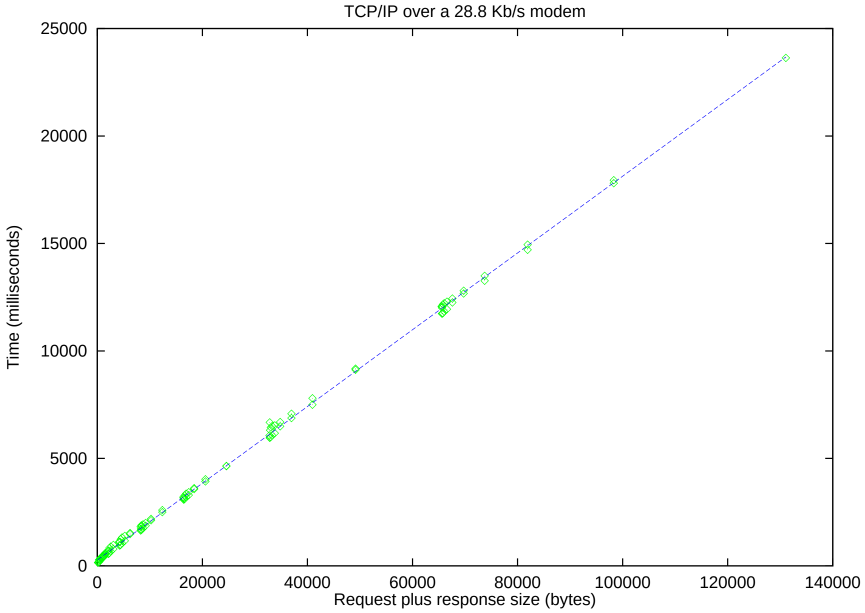


Figure 7.6: The top graph shows the time in milliseconds for two agents on the *same* laptop to exchange a request and response over an Agent Tcl meeting; the bottom graph, with Agent Tcl messages. The points are the actual data; the lines are the best-fit linear functions (least-squares approximation). Each data point is the average of multiple timings; the largest standard deviation was 3.2 percent.



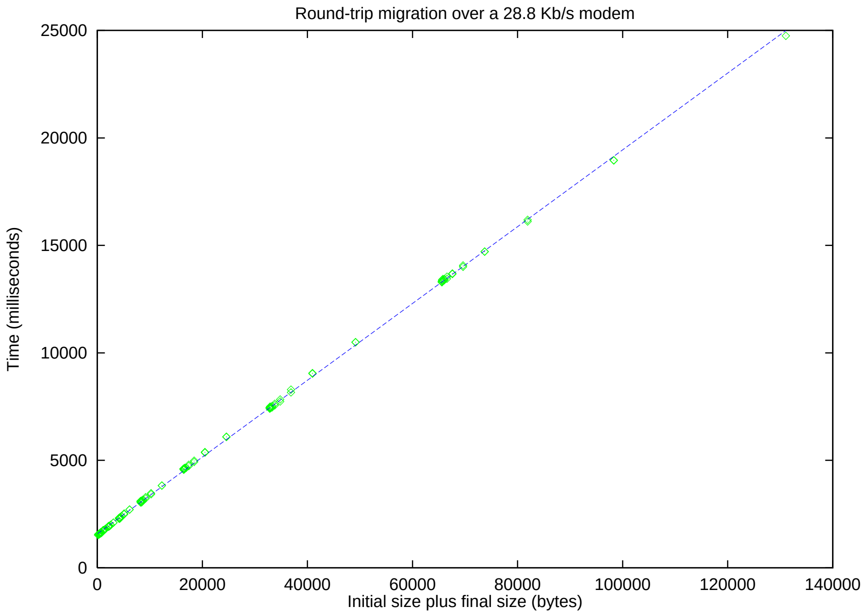


Figure 7.7: The top graph shows the time in milliseconds for two client/server processes on different laptops to exchange a request and response over a 28.8 Kb/s modem link; the bottom graph, for an agent to migrate from one laptop to the other and back. The points are the actual data; the lines are the best-fit linear functions (least-squares approximation). Each data point is the average of multiple timings; the largest standard deviation is 5.4 percent.

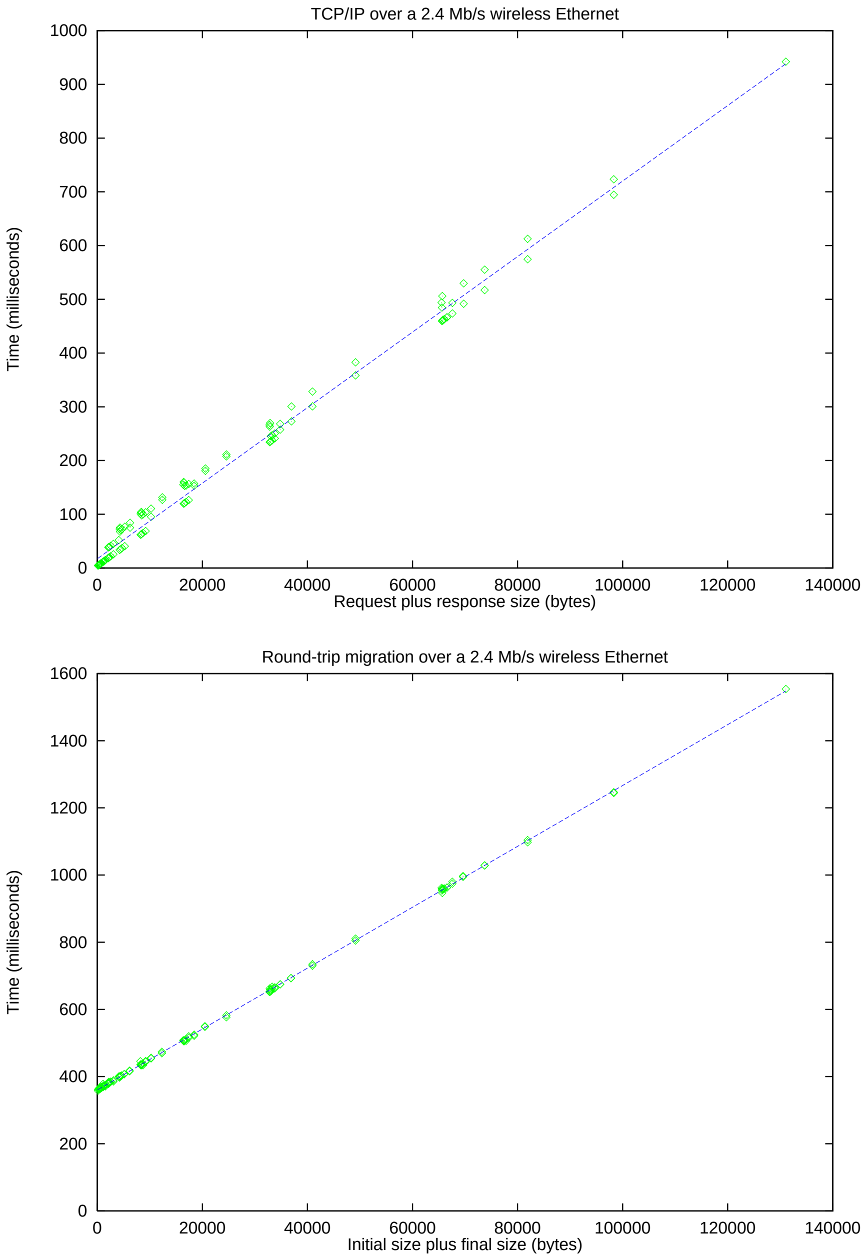


Figure 7.8: The top graph shows the time in milliseconds for two client/server processes on different laptops to exchange a request and response over a 2.0 Mb/s wireless Ethernet link; the bottom graph, for an agent to migrate from one laptop to the other and back. The points are the actual data; the lines are the best-fit linear functions (least-squares approximation). Each data point is the average of multiple timings; the largest standard deviation is 13.4 percent (but the deviation is less than 5.1 percent for all but three data points).

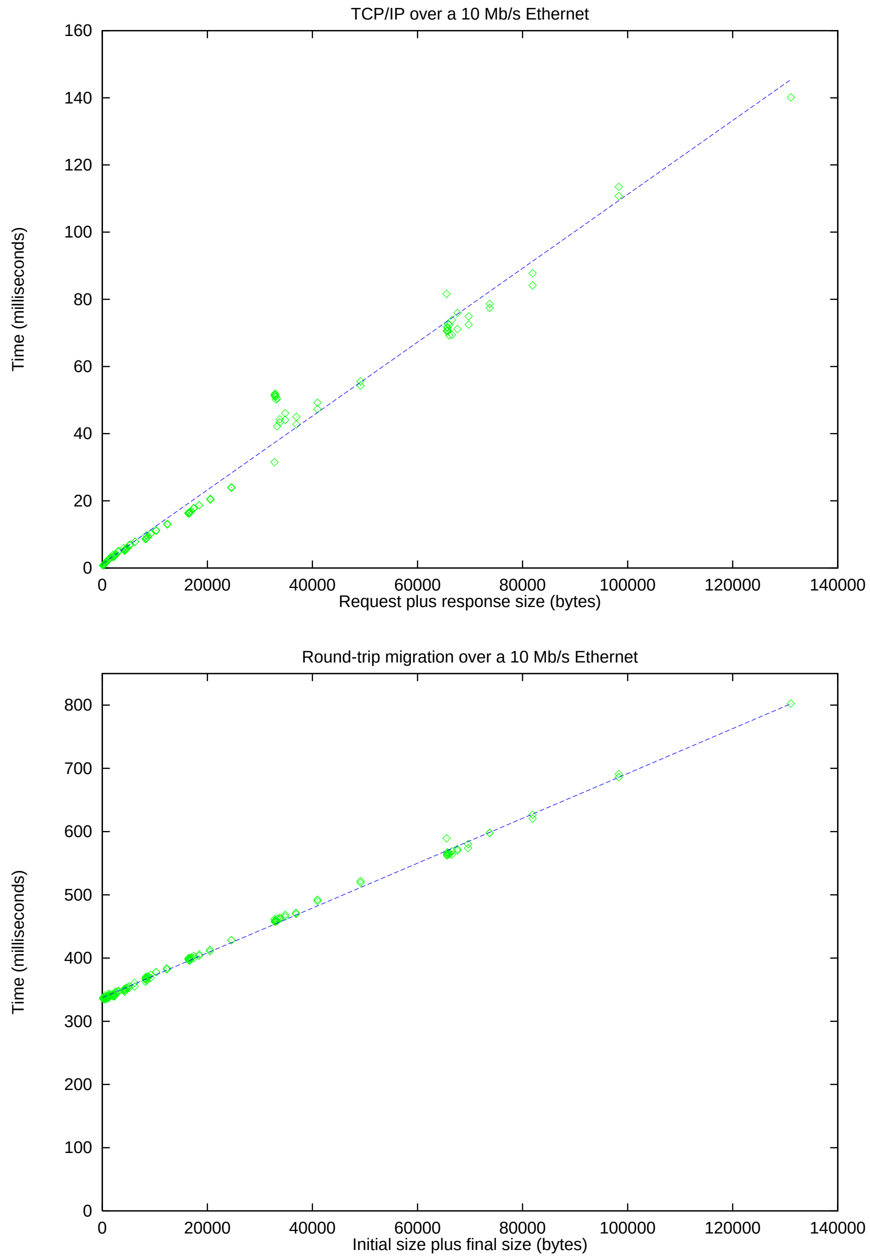


Figure 7.9: The top graph shows the time in milliseconds for two client/server processes on different laptops to exchange a request and response over a 10 Mb/s Ethernet link; the bottom graph, for an agent to migrate from one laptop to the other and back. The points are the actual data; the lines are the best-fit linear functions (least-squares approximation). Each data point is the average of multiple timings; the largest standard deviation is 24.0 percent (but the deviation is less than 5.2 percent for all but nineteen data points).

agent, and F is the final size of the agent.

$$t = 0.1786(I + F) + 1581 \qquad \text{(28.8 Kb/s modem)} \tag{7.6}$$

$$t = 0.0091(I + F) + 360 \qquad \text{(2.0 Mb/s wireless Ethernet)} \tag{7.7}$$

$$t = 0.0035(I + F) + 337 \qquad \text{(10 Mb/s wired Ethernet)} \tag{7.8}$$

In general, these equations match our expectations. The constant in each migration equation is larger than the constant in the corresponding client/server equation, since migration involves the additional overhead of establishing a TCP/IP connection between the migrating agent and the destination server, forking a request handler, and starting up the Tcl interpreter, all of which are constant across all agent sizes. Similarly, the coefficient in each migration equation is larger than the coefficient in the corresponding client/server equation, since migration involves the additional overhead of capturing and restoring a state image and copying the state image from the request handler to the new Tcl interpreter, both of which are linear in the agent size.

Now assume that an application needs to invoke n operations against some network resource; the expected request size is S bytes and the expected response size is R. Using Equation 7.3, the time for the stationary client process to invoke these n operations over the 28.8 Kb/s modem connection is

$$t_n = n(0.1785(S + R) + 272) \tag{7.9}$$

Using Equations 7.1 and 7.6, and noting that it takes 19 milliseconds to *establish* an Agent Tcl meeting on the laptops that were used in the experiments, the time for the mobile agent to invoke the same n operations is

$$t_n = 0.1786(I + F) + 1581 + n(0.00066(S + R) + 0.65) + 19 \tag{7.10}$$

This equation simply sums the time to migrate to and from the resource location, the time to invoke the n operations over an Agent Tcl meeting, and the time to establish the meeting. Assuming the agent carries only the result of the last operation back to its home machine, we have $F = I + R$ and can rewrite Equation 7.10 as

$$t_n = 0.1786(2I + R) + n(0.00066(S + R) + 0.65) + 1600 \tag{7.11}$$

Combining Equations 7.9 and 7.11, we see that when the two machines are connected with the 28.8 Kb/s modem link, the mobile agent outperforms the stationary client process whenever

$$0.1786(2I + R) + n(0.00066(S + R) + 0.65) + 1600 < n(0.1785(S + R) + 272) \tag{7.12}$$

Solving for n, we have

$$n > \frac{0.1786(2I + R) + 1600}{0.1778(S + R) + 271} \tag{7.13}$$

Following the same process, when the two machines are connected with the 2.0 Mb/s wireless Ethernet link, the mobile agent outperforms the stationary client process whenever

$$n > \frac{0.0091(2I + R) + 379}{0.0063(S + R) + 16} \tag{7.14}$$

Finally, when the two machines are connected with the 10.0 Mb/s wired Ethernet link, the mobile agent outperforms the stationary client process whenever

$$n > \frac{0.0035(2I + R) + 356}{0.0004(S + R) + 2} \tag{7.15}$$

With these three equations, we can calculate the minimum number of operations that a migrating Tcl agent must invoke against a particular resource for it to outperform stationary client/server processes. Figure 7.10 shows the minimum number of operations for a 1KB agent. If the agent is performing fewer operations, it should remain on the home machine. If it is performing more operations, it should migrate to the resource location and then back to the home machine.

We can combine our three equations into a single equation easily. The time for a client and server process to exchange n requests and responses over a TCP/IP connection is

$$t_n = n(T_B(S + R) + T_L) \tag{7.16}$$

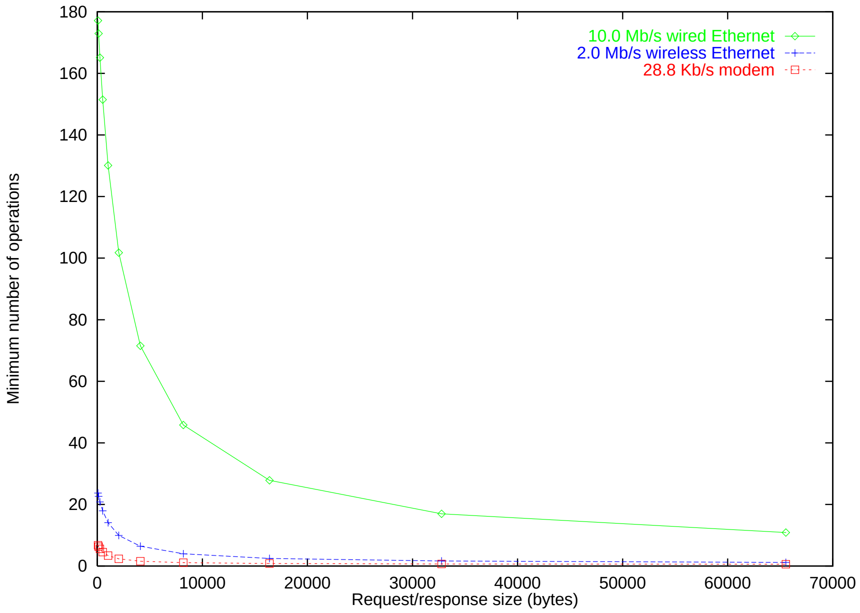


Figure 7.10: Minimum number of operations that a migrating 1KB Tcl agent must invoke against a remote resource for it to outperform stationary client/server processes. Each line corresponds to a different type of network connection between the home machine and the resource machine. The minimum number of operations was calculated from equations 7.13, 7.14 and 7.15.

where T_B and T_L are the network-specific parameters that we determined experimentally in each of the three cases (e.g, $T_B = 0.1785$ and $T_L = 272$ for the 28.8 Kb/s modem connection as seen in Equation 7.3. Similarly, the time for an agent to migrate to a remote machine and then back home is

$$t_n = A_B(2I + R) + A_L + n(0.00066(S + R) + 0.65) + 19 \tag{7.17}$$

(As before, the last two terms are the time to exchange n requests and responses over a local Agent Tcl meeting and the time to establish that meeting.) Combining these two equations, we find that the agent should migrate whenever

$$n > \frac{A_B(2I + R) + A_L + 19}{(T_B * 0.00066)(S + R) + T_L - 0.65} \tag{7.18}$$

Although this equation is a useful starting point, it must be improved in several ways. First, although the four parameters, A_B, A_L, T_B and T_L, can be calculated easily from network measurements, they are too high-level. For example, the equation should be recast in terms of the observed network bandwidth, rather than the observed round-trip migration time. Second, the equation applies only to a specific machine architecture, and only when each machine is lightly loaded. For example, the time to establish an Agent Tcl meeting and to exchange a request and response over that meeting is "hard-coded" into the equation. Although each machine might measure the meeting times experimentally (and then make the measurements available to agents), the equation should be extended to include machine loads and relative speeds. Finally, the equation applies only to a very specific agent behavior, i.e., the agent migrates to the remote machine, interacts with a resource, and migrates back to the home machine. Many other agent behaviors are possible. The equation should be turned into a set of equations that will help an agent pick the best behavior for its current environment.

7.3 Performance studies from other projects

Although few performance studies are reported for any mobile-agent system, there are a few interesting results from other projects.

Agent languages. Omniware programs are compiled into intermediate code for a RISC-based virtual machine and then into software-fault-isolated (SFI) native code; this code runs only 25 percent slower than natively compiled code on average [LSW95, ATLLW96]. Java programs are compiled into intermediate code for a stack-based virtual machine and then either interpreted or compiled on-the-fly into native code. When interpreted, Java programs run 10 to 20 times slower than natively compiled C or C++ [CH97]. When compiled on-the-fly into native code, Java programs run 2 times slower than natively compiled C or C++ *if they are compute-intensive.*[19] Since Omniware and Java both protect the local machine from malicious code, these performance numbers mean that agents can be executed nearly as fast as native code (as long as the agents are compiled for the appropriate virtual machine). In turn, this means that, excluding migration overhead, a server will see about the same CPU load regardless of whether it provides a high-level operation directly *or* provides low-level primitives and allows a migrating agent to implement the high-level operation itself.

[19] The author has not been able to find any formal performance study of just-in-time compilation for Java. The statement is based on marketing information for the various Java virtual machines as well as informal benchmarks such as the *CaffeineMarks*.

Sumatra. One application of Sumatra is an Internet chat server that positions itself so as to minimize the maximum latency between itself and its clients [RASS97]. The chat server can migrate to any machine that is participating in the chat conversation. Resource monitors continually monitor the maximum latency between each machine and others and periodically sends the latency information to the chat server. If another machine has a better maximum latency than the server's current machine, and this better maximum latency *remains stable* for some period of time, the chat server migrates to the other machine. The Sumatra developers tested their mobile chat server in a local area network, but added delays to each network link from Internet latency traces, producing a rough approximation of the Internet. In the tests, the average maximum latency between the mobile chat server and its clients was nearly four times less than between a stationary chat server and its clients.

Tacoma. Within the context of the StormCast weather-monitoring system, the Tacoma group compared two different ways of determining the maximum temperature within a particular range of days [Knu95]. In the first approach, the entire temperature record for the days of interest is downloaded to the client machine. In the second approach, a mobile agent is sent to the machine where the temperature record is stored; the agent then extracts and returns only the desired maximum temperature. The mobile agent was written in either Tcl or compiled C. Due mainly to migration overhead and the slowness of Tcl, they found that the mobile agent had better performance only when the agent was written in compiled C, *and* the temperature record for the desired period took up nearly one hundred kilobytes [Knu95]. This particular task, however, is probably the worst possible use of Tcl, as we saw above when finding a minimum integer in some set of integers. Reducing migration overhead and using a faster language such as Java might provide competitive performance without resorting to compiled code.

A second Tacoma performance study [JSvR97] found that it takes 21.7 milliseconds to start a *null* agent on a remote machine. The code for the agent was already on the remote machine, the parameters to the agent were 42 bytes of dummy data, and both the local and remote machines were 160 MHZ Hewlett-Packard C-160 workstations. In contrast, Agent Tcl takes 128 milliseconds to submit a 64-byte *null* agent to a remote machine and then receive a 64-byte result message from that agent. The code for the agent was part of the 64 bytes, the local machine was a 133 MHz Intel Pentium workstation, and the remote machine was a 200 MHz Intel Pentium workstation. The Tacoma time is much lower because it measures only how long it takes for the remote machine to acknowledge the correct startup of the new agent. In contrast, the Agent Tcl time measures how long it takes for the new agent to send back its result message, which includes not only the time needed to send the message but also the time needed for the Tcl interpreter to read and evaluate a set of initialization scripts. In addition, Agent Tcl must add the new agent to its server tables, which involves additional interprocess communication. Once these differences are taken into account, the Tacoma and Agent Tcl times are comparable.

Network management. Steward and Appleby use lightweight mobile agents to control traffic congestion in a circuit-switched telecommunications network [SA94, AS94]. There are two kind of agents: *load-management* agents, which find routes that have the highest spare capacity and adjust routing tables accordingly, and em parent agents, which randomly walk around the network and launch load-management agents when an overload is detected. None of the agents communicate directly but instead leave messages at each node that other agents can read; old messages are given less weight and eventually purged. The messages indicate whether a load agent has recently been launched on that node, whether a parent agent has recently visited, and so on. In their experiments, the mobile agents reduced maximum node utilization by between 39 and 50 percent, in addition, there were no overloaded nodes and no unused nodes [SA94, AS94]. The authors acknowledge that the same effect could be achieved with a more traditional distributed algorithm, but argue that mobile agents provide an extremely fault-tolerant solution without undue programmer effort. In particular, agents can randomly disappear without having any negative impact on the overall algorithm.

Chapter 8

Applications

Mobile agents are best viewed as a tool for implementing distributed applications, rather than as an enabling technology. In other words, their advantage lies not so much in making new distributed applications possible, but rather in providing a unified programming model and improving the performance of existing applications. Performance can be a matter of network utilization, completion time, programmer convenience, or simply availability (during a period of network disconnection). Like most mobile-agent systems, therefore, Agent Tcl is intended for use in general distributed applications. In this chapter, we first describe applications of *other* mobile-agent systems, briefly considering whether each application can be implemented effectively in the current Agent Tcl system. Then we present the existing applications of Agent Tcl.

8.1 Other systems

Kali Scheme. [CJK95] suggests several applications for Kali Scheme: *load balancing*, where executing threads are moved to a lightly loaded machine; *incremental distributed linking*, where a new procedure (such as a debugging or monitoring procedure) is dynamically inserted into a distributed computation; *parameterized client-*

server applications, where a client and server offload work to each other in the form of shipped procedures; *distributed data mining and information retrieval*, where an arbitrary search procedure is sent to a database rather than bringing large amounts of data across the network; and *executable content in messages*, where a message includes a program that performs some useful action on the local machine, such as installing new software or presenting an application front-end to the user. Using mobile agents for *load balancing* is questionable, since existing mobile-agent systems, including Agent Tcl and Kali Scheme, use interpreted languages that run at least several times slower than native code. Thus, the price of load balancing with mobile agents is a significant increase in the required CPU time, leading to longer completion times unless a large number of lightly loaded machines are available. As agent languages become faster[1], general load balancing will be a more reasonable application. At the same time, if mobile agents are chosen for other reasons, the agents can be load-balanced easily, since they can migrate at will from one machine to another. *Incremental linking* is also questionable since, as with load balancing, the price of the dynamic code insertion is interpretive overhead. In fact, such dynamic code insertion can be achieved easily in natively compiled code [HG97]. Again, if mobile agents are used for other reasons, such dynamic code insertion would be a useful capability. We do not have any immediate implementation plans, however, since code insertion introduces additional security issues and is only useful in a few applications; most applications can be organized as collections of small, cooperating agents that are replaced when needed. The last three applications are all variations on the same theme,

[1] Omniware, for example, uses on-the-fly compilation and software fault isolation to securely execute a C++ program only 25 percent slower than natively compiled code [LSW95]; similar approaches are just starting to find their way into mobile-agent systems.

either moving computation from client to server (e.g., a query against a database) or from server to client (e.g., a graphical front-end for a database), eliminating all intermediate network transmission and reducing either overall or per-operation latency. Whether migrating a query reduces overall latency depends on the number of cross-network calls eliminated, the amount of data per call, the relative speed of agents versus native code, and the relative speed of the client and server machines. On the other hand, migrating a graphical front-end almost always reduces per-operation latency, since the amount of processing per user event (e.g., a mouse click) is typically small. This client-server "customization" is one of the main applications for Agent Tcl, although migrating a Java or Scheme agent is more often a performance win than migrating a Tcl agent, due to the slowness of Tcl.

Messengers. Messengers are oriented towards communication protocols and distributed operating systems [DiMMTH95]. If a machine wants to communicate with a remote machine that does not understand the desired communication protocol, it dispatches a messenger that implements the protocol; this messenger lives on the remote machine and handles the communication channel on its behalf. Agent Tcl can be used in a similar manner. Like the Messenger system, however, Agent Tcl is more suited to application-level rather than network-level protocols due to the interpretive overhead. For example, an application can dispatch a graphical front-end to a remote machine, replacing the transmission of individual screen updates with application-specific requests and responses. In the case of a distributed operating system, each machine would have a small microkernel; the rest of the operating system would be implemented as messengers that dynamically distribute themselves as needed [TDiMMH94]. Although this is an interesting use of mobile agents, no existing mobile agent system (including Messengers) seems efficient enough to be used for arbitrary system-level components, which means that the native "microkernel" must be relatively large. Agent Tcl certainly cannot be used at the system level in its current state. It must either provide a language that can be compiled on-the-fly into native code, or accept native code directly from certain highly trusted sources. Of course, accepting native code directly makes it much more difficult to handle a heterogeneous environment.

Sumatra. [RASS97] suggests several applications for mobile programs: searching, filtering and combining "periodically generated large-volume datasets"; positioning a video or Internet chat server to minimize the average latency between the server and its clients; prefetching web pages; and more generally moving a network-intensive computation from a mobile computer to a dynamically selected proxy site within the permanent network. Agent Tcl can be used effectively in all of these applications, although most of them would need to use Java rather than Tcl to achieve sufficient speed; the chat server is a possible exception since it does relatively little processing per message.

Tacoma. The Tacoma system is used primarily in StormCast, a distributed weather-monitoring system in which the data volumes are so immense as to make data movement impractical [JvRS95]. Mobile agents allow new filtering and monitoring operations to be rapidly constructed and deployed to the data and sensor locations. The Tacoma project found that Tcl was too slow for their compute-intensive operations, but did not experiment with a more efficient interpreted language such as Java or with on-the-fly compilation. If Java is fast enough, Agent Tcl can be used "as-is" for the StormCast simulation; otherwise we must wait until Agent Tcl provides on-the-fly compilation for at least one of its languages. Tacoma is also used in active documents and to manage software installation within a networked collection of machines [JvRS96]. Agent Tcl can be used effectively in both, although active documents require additional infrastructure to inject an agent embedded inside a document into the local agent system.[2]

Telescript. Telescript was used primarily in active mail, network and platform management, and electronic commerce [Rei94]. In active mail, a Telescript agent is embedded inside an e-mail message; the agent is executed when the message is received or viewed. Typically such an agent might allow the user to sign up for some service, asking the user for needed information and then contacting the service provider. One platform-management application is automatic software updates. An agent carries the necessary files onto a machine, installs the files itself and then disappears. In electronic-commerce applications, a Telescript agent might leave a personal digital assistant (PDA), search multiple electronic catalogs for a certain product, return to the PDA with the best vendor and price, and then optionally leave the PDA again to actually purchase the product. Agent Tcl can be used for all of these applications, although active mail requires additional infrastructure, specifically a MIME type for Agent Tcl agents and a corresponding "viewer" that injects the agent into the local agent system.[3]

Mobile Service Agents (MSA). The Mobile Service Agent (MSA) system is used primarily for "follow-me" computing in which an application moves to the location of the user for more efficient interaction. The main MSA demo involves a conference proceedings [TLKC95]. When a user connects his laptop to the conference's machines, an agent containing the conference proceedings, site map and local points of interest is sent to the laptop. The user interacts with the conference proceedings via this agent and can continue interacting even when the laptop is disconnected.

[2]This infrastructure is easy to implement.
[3]This infrastructure is also easy to implement.

Other suggested applications include active e-mail, distributed information retrieval, software updates in telecommunications switches, software updates for general user applications (that are organized as a collection of cooperating agents), and automatic introduction of new components into a computer-aided manufacturing system [TLKC95, Kna96]. Except for telecommunications software, which demands code speed that existing mobile-agent systems cannot deliver, all of these applications are reasonable uses for mobile agents in general and Agent Tcl in particular.

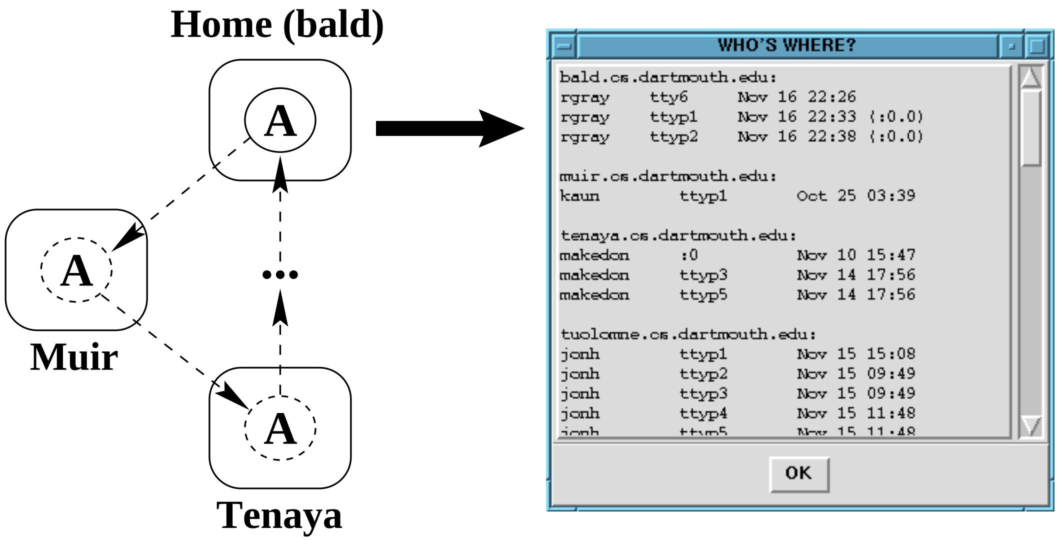
Network management. As discussed in the performance analysis section, Steward and Appleby use lightweight mobile agents to control traffic congestion in a circuit-switched telecommunications network [SA94, AS94]. *Load-management agents* find the routes that have the highest spare capacity and adjust routing tables accordingly; *parent agents* randomly walk around the network and launch load-management agents when an overload is detected. Agent Tcl is not yet suited for such an application since its migration overhead is large; however, once migration overhead is reduced, Agent Tcl's Java agents should provide sufficient performance, since the agents are performing only a small amount of computation per node. Agent Tcl also does not allow a migrating agent to leave messages behind at a node, but such a mechanism can be added easily.

8.2 Agent Tcl

Due to the research interests of the other Agent Tcl project members, Agent Tcl is used primarily in workflow, notification, and information-retrieval applications. Before describing these applications, it is useful to examine the code for a simple, complete Tcl agent that determines which users are logged onto some set of machines. This "who" agent is shown in Figure 8.1. The agent accepts a list of machines as input and then migrates from machine to machine with the *agent_jump* command, executing the Unix `who` command on each one. Once the agent has migrated through all the machines, it uses `agent_jump` one last time to return to the home machine, where it presents the list of users to its owner.

Although its task is simple and can be accomplished easily without a mobile agent, the "who" agent illustrates the general form of any agent that migrates sequentially through a set of machines. Existing Agent Tcl agents that fall into this category are a workflow agent that carries an electronic form from user to user [CGN96] and a medical agent that retrieves distributed medical records based on certain criteria [Wu95]. The workflow agent must migrate sequentially since the users need to fill out the sections of the form in order. The medical-retrieval agent chooses to migrate sequentially since the agent can discard potential candidates as it travels through the distinct databases; spawning one child agent per remote database or interacting with the databases using the traditional client/server approach increases the total network traffic even when only a single operation is being performed against each database.

In addition, the workflow and medical agents do not require continuous contact with the home machine and will continue their task even if the home machine becomes temporarily disconnected. The agents are also extremely easy to implement. The code is written as if every resource is local to the agent; the only difference is that the `agent_jump` command is used to move the agent from one machine to the next. The `agent_jump` command is not strictly necessary since we could continually resubmit a Tcl procedure that was parameterized according to the current status of the task; the procedure would use the parameters to determine what it needed to do on the current machine [JvRS95]. Such an approach, however, requires that the programmer explicitly collect the necessary state information. In the "who" agent, this state




```
agent_begin          # register with the local agent server

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

foreach machine $machineList {
    agent_jump $machine            # jump to each machine
    append output [exec who]       # any local processing
}

agent_jump $agent(home)            # jump back home

# display output window

agent_end                          # unregister
```

Figure 8.1: The "who" agent migrates through a set of machines and figures out who is logged onto each one. Although the "who" agent is not the most useful agent, it illustrates the general form of any agent that migrates sequentially through a set of machines (and is short enough to fit on one page). The `exec who` can be replaced with any desired processing.

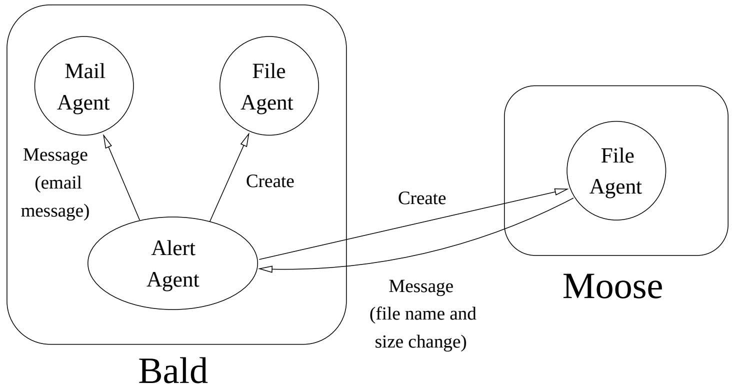
information is nothing more than an index into the machine list, but more and more state information is required as the agent becomes more complex.[4] The `agent_jump` command captures all the state information automatically.

An example notification agent is the "alert" agent that monitors a specified set of remote resources and notifies its owner of any change in resource status. Figure 8.2 shows an "alert" agent that monitors a set of files and notifies the user if the status of a file changes significantly (monitored characteristics include the Unix *rwx* bits and the file size). The agent creates one child agent for each remote filesystem using `agent_submit`. Each child agent monitors one or more files in its filesystem and sends a message to the parent when the status of a file changes significantly. The parent then contacts the owner's "mail" agent to send an e-mail message.

Since the child agents know which status changes are "significant", only the status changes that the user actually wants to see are transmitted across the network. Without mobile agents, either the remote machine would have to send back a notification of every change (which the application would filter on the home machine), or the appropriate monitoring routines would have to be pre-installed on the remote machine, limiting the application to the changes that the remote administrator considers significant. With mobile agents, the application can monitor for status changes according to any desired criteria while minimizing the ongoing network traffic.

A recent information-retrieval agent is the technical-report searcher that was discussed in the motivation section and that is shown again in Figure 8.3. The agent's task is to search a distributed collection of technical reports for information relevant to the user's query. The agent firsts asks the user to enter a free-text query. Then,

[4] The workflow agent, for example, must carry along all user-entered information. The medical-retrieval agent must carry along the results from each database.




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

  # get a name from the server

agent_begin

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

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

  # wait for one of the "file" agents to send a message saying that the
  # status of a file has changed; 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 8.2: The "alert" agent monitors a set of files and sends an e-mail message to the user when the status of a file changes significantly. A simplified version of the "alert" agent appears at bottom; procedure `file_watch`, which polls the files at regular intervals using the `file stat` command, and procedure `construct_mail`, which constructs a readable mail message, are not shown. The network location of the various agents is shown at top.

if the connection between the home machine and the network is reliable and of high bandwidth, the agent will stay on the home machine. If the connection is unreliable or of low bandwidth, such as if the home machine is a mobile device, the agent will jump to a *proxy site* within the network. This initial jump reduces the use of the poor-quality link to just the transmission of the agent and the transmission of the complete result, allowing the agent to proceed with its task even if the link goes down. The proxy site is dynamically selected according to the current location of the home machine and the document collections.

Once the agent has migrated to a proxy site if desired, it must interact with the stationary agents that serve as an interface to the technical-report collections. If these stationary agents provide high-level operations, the agent simply makes RPC-style calls across the network (using the agent-communication mechanisms). If the stationary agents provide only low-level operations, the agent sends out child agents that travel to the document collections and perform the query there, avoiding the transfer of large amounts of intermediate data. Once the agent has the results from each child agent, it merges and filters these results, returns to the home machine if necessary, and presents the results to the user. Finally, although the behavior exhibited by this agent is complex, it requires surprisingly little code; the decisions whether to jump and create children involve little more than two `if` statements that check the information returned from the network monitor on the home machine and from the directory services.

The salesman agent shown in Figure 8.4 is similar to the technical-report searcher except that it operates in two phases. First, the user enters a description of the desired product. Then the agent queries the *yellow page* directory services to identify vendors that might sell the product, and then queries each vendor to obtain product

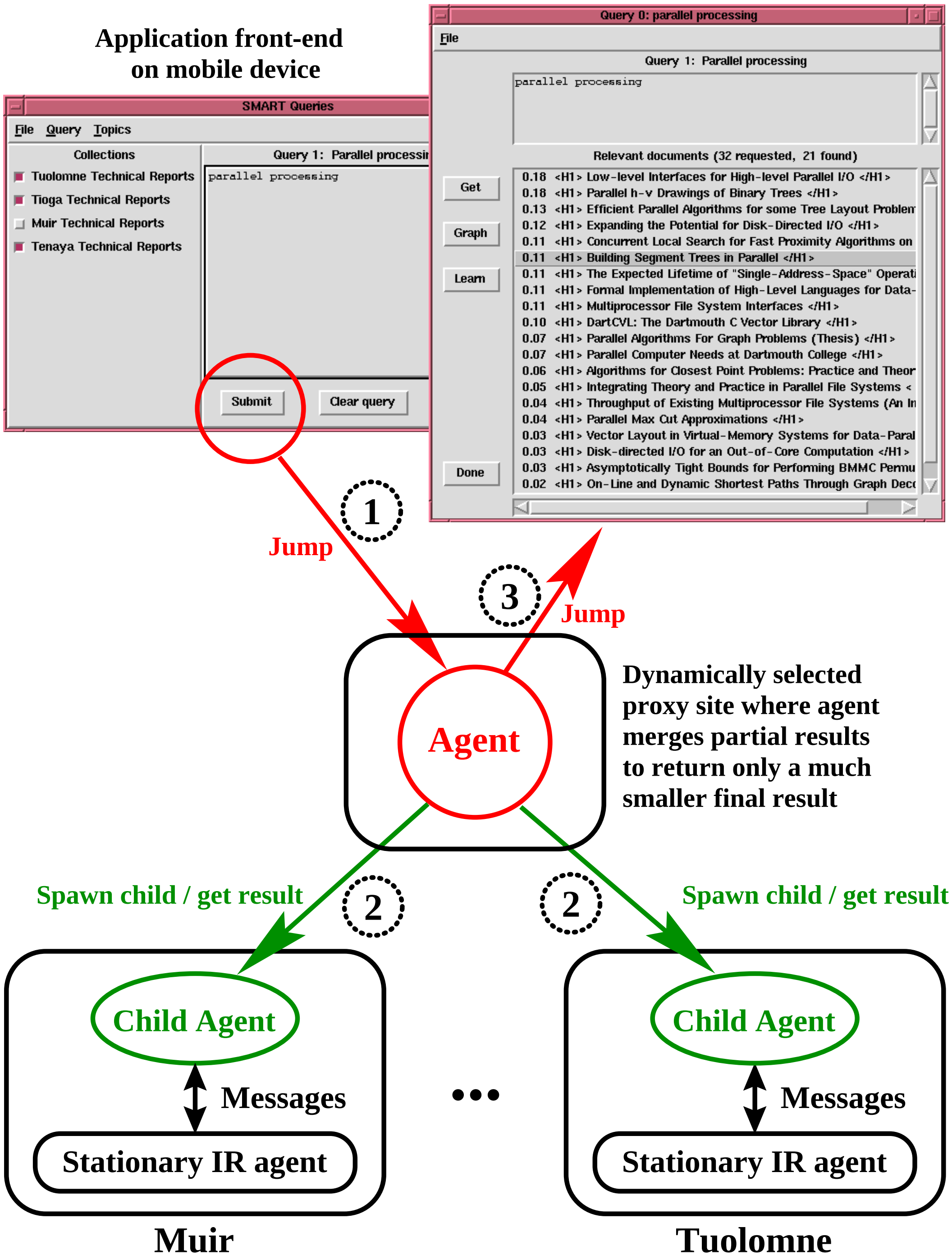


Figure 8.3: Technical report searcher. Here an agent searches a distributed collection of technical reports for information relevant to a user query. Depending on current network conditions and the granularity of the search engine interfaces, the agent might move to a proxy site within the permanent network and/or send child agents to each collection site.

availability and pricing. During this process, the agent might migrate to a proxy site, and might send child agents to each vendor, depending on the network links and vendor interfaces. Once the agent has the list of vendors and prices, it returns to the home machine and allows the user to select one or more products to purchase. If the user does select a product, she provides the agent with the appropriate amount of electronic cash, and the agent journeys into the network again to exchange the electronic cash for the product (or for some non-repudiable proof of purchase).

Agent Tcl has also been used to retrieve three-dimensional drawings of mechanical parts from distributed CAD databases [CBC96], to track purchase orders [CGN96], and in several information-retrieval applications at external sites.

8.3 Summary

None of these applications require mobile agents. Mobile agents, however, allow the applications to be implemented easily within a single, general framework, without such things as application-specify proxies and server operations, queued RPC, or automated installation facilities. We have not done any performance analysis on these applications yet, choosing instead to implement some of the more obvious performance improvements first. Since the existing agents are all written in Tcl, however, their *end-to-end* latency is worse than alternative implementation techniques, due to the migration overhead, the slowness of Tcl, and the moderate data volumes involved in our applications. On the other hand, the existing agents do reduce network utilization significantly and can continue with their task if the network link with the home machine is down. In addition, with faster languages and additional system engineering, they should be competitive in terms of end-to-end latency as well.

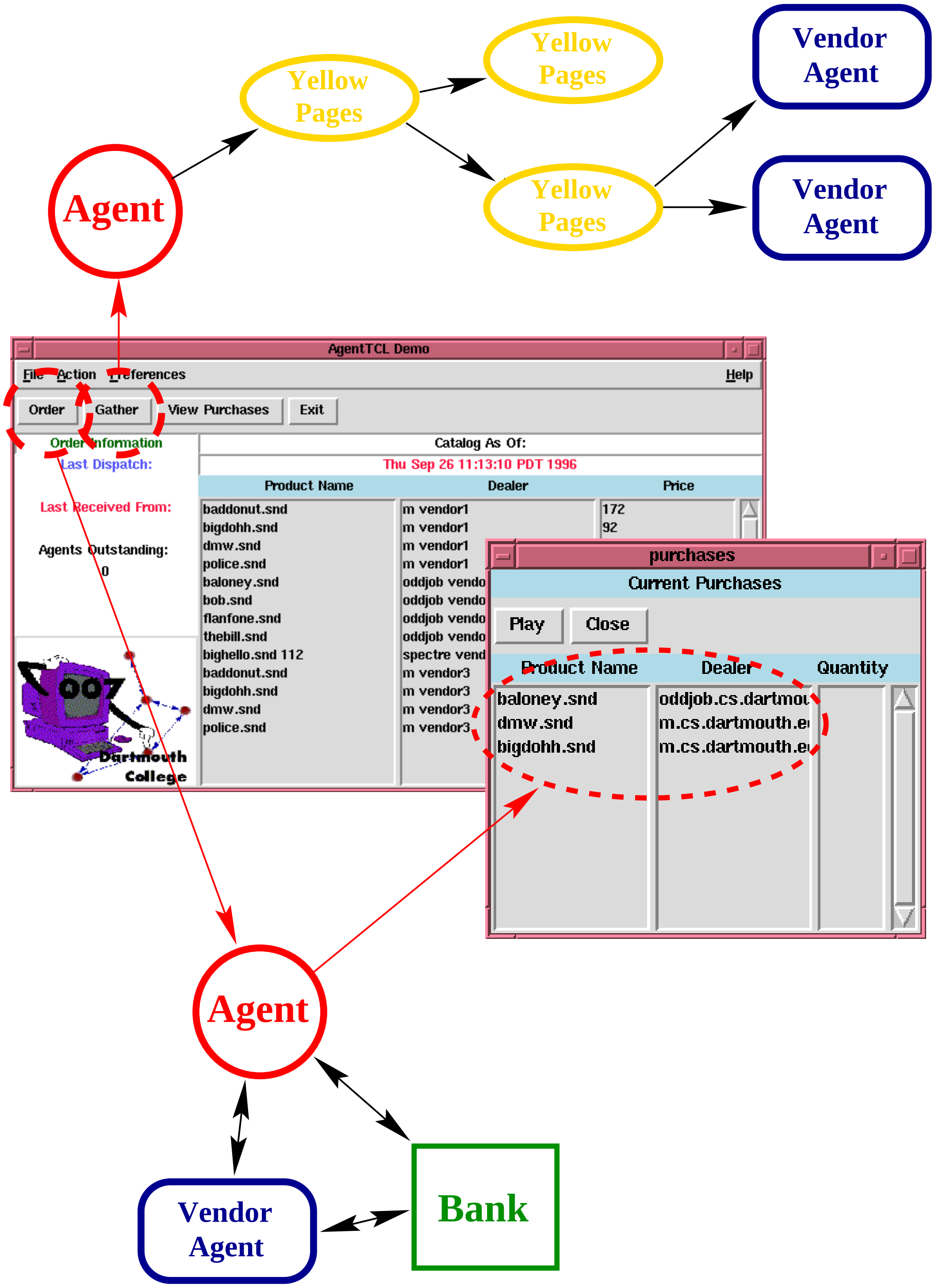


Figure 8.4: Salesman. Here an agent accepts a user-entered description of a desired product, queries the yellow page directories to identify vendors who might sell the product, queries these vendors, returns the possible purchase locations and prices to the user, and finally purchases any selected product with electronic cash.

Chapter 9

Other components

The Agent Tcl project includes many other graduate and undergraduate students from the Thayer School of Engineering and the Department of Computer Science. In this chapter, we present some of the components that these students have implemented, both to highlight their excellent work and to provide a complete picture of Agent Tcl's current capabilities.

9.1 Debugging

One of the main challenges facing an agent developer is debugging a broken agent, especially since the agent physically moves from machine to machine during its execution and does not necessarily stay in contact with the home machine. In the absence of debugging tools, the developer has to manually add debugging code that sends constant status updates to the home machine, since otherwise the first indication of a problem would be when the agent either disappeared forever[1] or returned with incorrect information. The addition and maintenance of this debugging code becomes tedious extremely quickly (as in the more common case of debugging a stationary pro-

[1]Of course, the agent's owner can never *know* that the agent has disappeared, only that it has not returned within some expected worst-case time.

gram). Therefore Melissa Hirschl[2] wrote an interactive, graphical debugger for Agent Tcl [HK97]. The debugger tracks the agent as it moves from machine to machine, monitors its communication with other agents, and provides the traditional debugger features such as breakpoints, watch conditions, and line-at-a-time execution. Figure 9.1 shows the debugger interface while debugging the "who" agent from Figure 8.1. The agent's code appears in the top half of the window, and an execution history appears in the bottom half. A breakpoint can be associated with any line or with any agent event (such as migration or incoming communication from other agents). In this case, the agent has been told to break on migration and has suspended its execution at the `agent_jump` command, right before jumping to the next machine in its list. While the agent is suspended at the breakpoint, the user can add new break points or watch conditions and can inspect all defined variables and procedures.

There are several issues that must be addressed. First, the debugger currently works for Tcl agents only. The debugging routines must be separated into language-dependent and language-independent modules and appropriate language-dependent modules must be written for Java and Scheme. Second, there is no debugging support built into the Agent Tcl system itself. Instead the debugger simply instruments the agent so that the agent sends back constant status reports and drops into an "inspect-and-evaluate" mode at each breakpoint. Although *not* requiring any debugging support from the Agent Tcl system is advantageous in its own way, it does lead to a much more complex debugger. Moving some of the debugging functionality into the Agent Tcl core would allow a cleaner and more efficient implementation. Likely work along this line is to (1) make Agent Tcl optionally call debugger-specified code

[2]Melissa Hirschl graduated in 1997 with a Master's degree in computer science. She has taken a job in the SunScript group at Sun Microsystems.

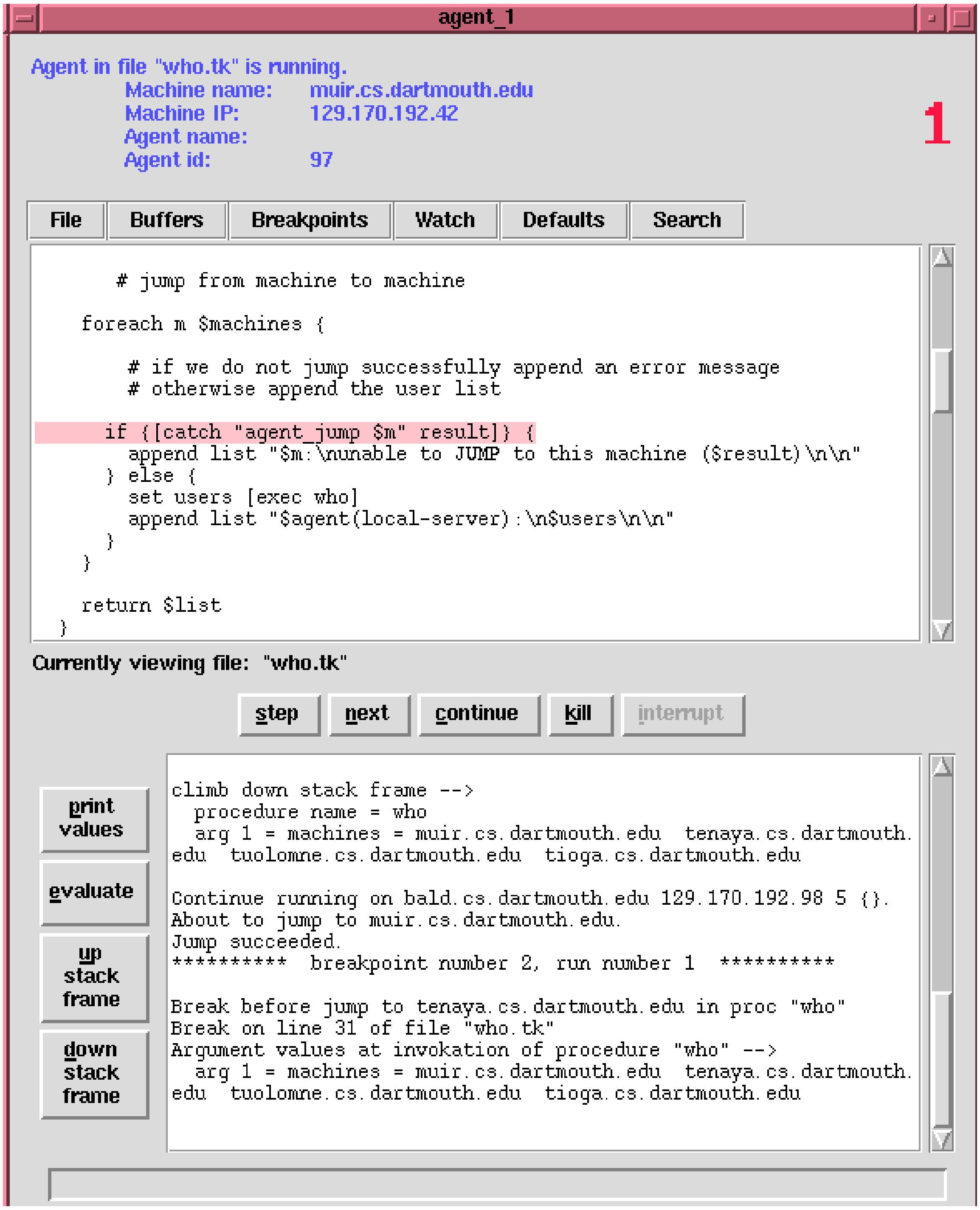


Figure 9.1: The Agent Tcl debugger. The debugger tracks the agent as it travels through the network, monitors its communication with other agents, and provides traditional features such as breakpoints, watch conditions and line-at-a-time execution. Here we are debugging the "who" agent and have told the debugger to break before each jump.

before and after agent events and (2) make the system aware of the identity and location of the debugger so that the system itself can contact the debugger if the agent dies due to some exceptional condition. Finally, the debugger interface is still heavily text-based. Current work is aimed at providing various graphical representations of an agent's path through the network and its communication with other agents.

9.2 Docking

Mobile agents are particularly useful when dealing with mobile hosts that do not have a permanent network connection. By migrating to or from a mobile host, an agent can continue interacting with a user or network resource respectively, even if the link between the host and network goes down. A basic problem, however, is what to do when an agent tries to visit or leave a mobile host that is currently disconnected from the network. The naive solution of having the agent poll the network connection is extremely inefficient. Instead Ting Cai, Saurab Nog, Vishesh Khemani and Jun Shen[3] have developed a *docking system* that allows agents to simply go to sleep until the network connection is available again [GKN$^+$97]. This docking system is shown in Figure 9.2. Each mobile host has an associated *dock*, which is some permanently connected machine within the network. A stationary agent called the *dockmaster* runs on each *dock* machine. If an agent wants to visit the mobile host, it first tries to migrate directly to the host. If the migration fails, the agent transfers itself to the

[3]Ting Cai graduated in 1996 with a Master's degree in computer science and is currently working at Bay Networks. Saurab Nog graduated in 1996 with a Master's degree in computer science and is currently working at Microsoft. Vishesh Khemani and Jun Shen are undergraduates in the computer science department. Both have spent several semesters working on the Agent Tcl project and will graduate in 1998.

dock, adds itself to the dockmaster's queue of waiting agents, and goes to sleep. The *dockmaster* waits until it makes contact with the mobile host. This contact happens in one of two ways. First, the mobile host notifies the dockmaster of its current network address whenever it reconnects to the network. Second, the dockmaster periodically polls the last known location of the mobile host (at long intervals). This polling is required since the initial migration will occasionally fail due to network congestion, even though the mobile device was actually connected to the network. In either case, once the dockmaster makes contact with the mobile host, it forwards all waiting agents onto the host. In addition to the dockmaster on the dock, there is also a dockmaster on the mobile host itself. This second dockmaster handles agents that are trying to *leave* the host. If an agent tries to leave while the network connection is down, it adds itself to the local dockmaster's queue and goes to sleep. As soon as the host reconnects to the network, the local dockmaster forwards all waiting agents to the appropriate destination.

The docking system is implemented entirely at the agent level and is completely transparent to the agents that use it. It has two notable weaknesses, however. First, it handles only migration. It must be extended to handle inter-agent communication as well, so that an agent does not have to continually resend a message if its or the recipient's network connection is down. Second, Agent Tcl itself has no knowledge of a device's temporary IP address; it knows only the device's permanent IP address; Thus agents and messages are always directed to the permanent IP address, leading to an unnecessary trip through the dock if the device is actually connected but is at a different address. To fix this problem, each migrating agent should include both the permanent and current address of its sending machine and the permanent and last-known address of its home machine. In addition, each message should include both

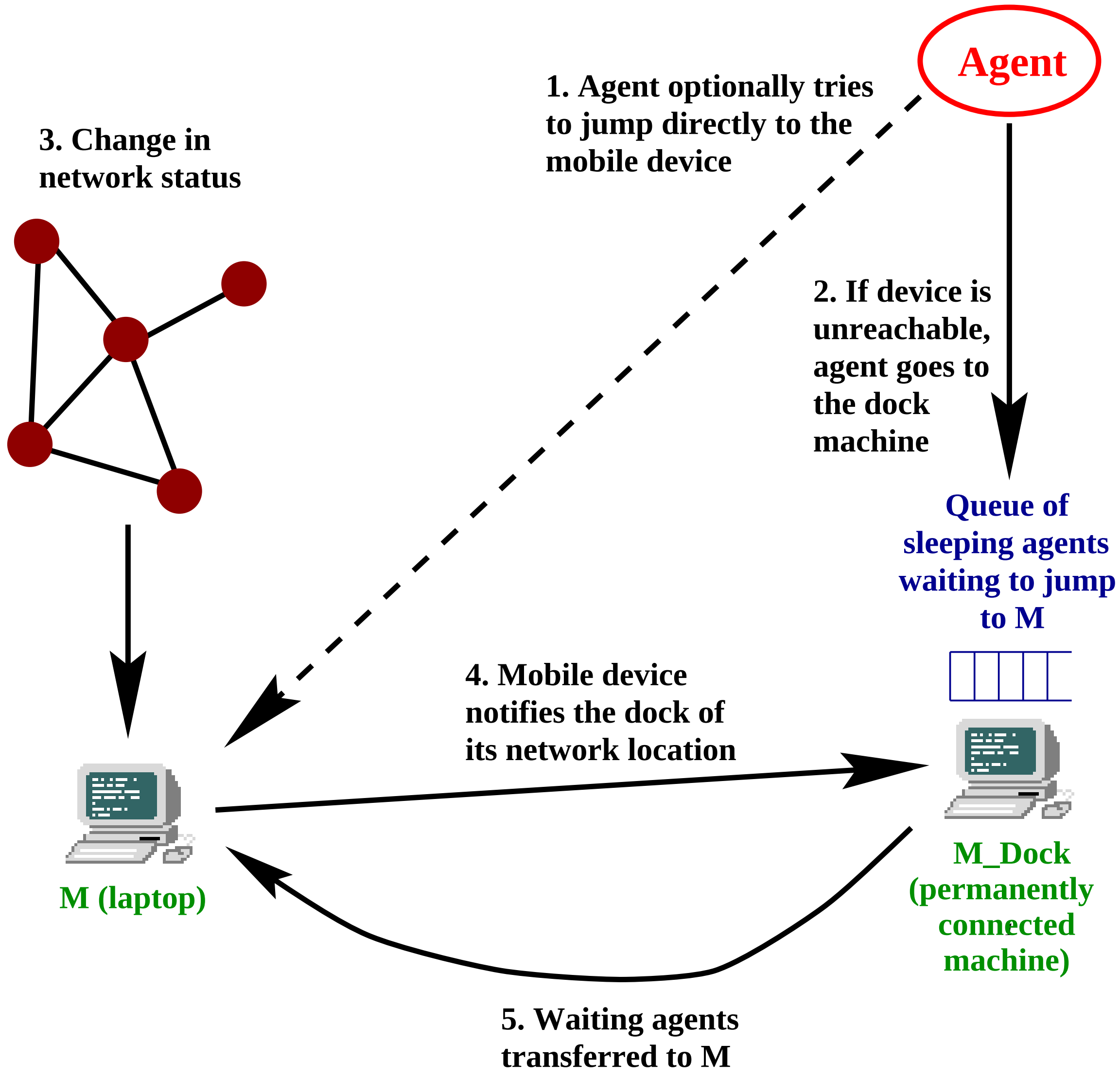


Figure 9.2: The Agent Tcl docking system. Each mobile host (*M*)has an associated *dock* (*M_Dock*). When an agent cannot reach the mobile host directly, it goes and waits at the dock. The dockmaster on the dock periodically polls the mobile host, and the mobile host notifies the dockmaster whenever it reconnects to the network. In either case, once the dockmaster makes contact with the mobile host, all waiting agents are forwarded.

the permanent and current address of the sending machine. Finally, the dockmasters should report the last-known IP address of the mobile machine, either on demand or whenever they accept an agent or message for later delivery. With the current and last-known IP addresses included in agent communication whenever possible, the servers can maintain a cache of last-known IP addresses and direct outgoing agents and messages to the last-known address of the destination machine, making the docking system much more efficient and exactly analogous to mobile IP [Joh95]. In fact, once mobile IP is widely available, the implementation of the docking system will become much simpler.[4]

9.3 Yellow pages

In a real-world environment, an agent must identify, locate and use previously unknown services. To this end, Dawn Lawrie, Mark Hoagland and Joseph Edelman[5] have developed a hierarchical service index [GKN+97]. This hierarchical index is implemented as a set of stationary *yellow page* agents, which maintain a set of entries that refer to specific service agents as well as other yellow page agents. Each entry contains a set of named fields. Each field contains either a keyword list or a definition of the interface that the server agent supports. Possible interface definitions include a set of KQML queries [GSS94] or a set of Agent RPC functions [NCK96]. To find a service, an agent contacts a yellow page agent and does an exact or ranked search on the contents of one or more fields. If this search returns other yellow page agents, the agent can optionally recurse and make the same query of these other agents. In any

[4] The docking system is still necessary so that agents do not have to poll the network connection.

[5] Dawn Lawrie graduated with a B.A. in computer science in 1997. Mark Hoagland and Joseph Edelman are both undergraduates in the computer science department and will graduate in 1998 and 1999 respectively. All three have spent several semesters working on the Agent Tcl project.

event, once the agent has a list of service agents, it will proceed to interact with one or more of those agents. To notify other agents of an available service, a service agent posts a description of its service to one or more yellow page agents. This description is simply the named fields, keyword lists, and interface definitions that should be added to the yellow page databases.

The yellow page system is used in most of our example applications, most notably the salesman application that was shown in Figure 8.4. Work is underway to (1) develop a standardized set of fields for describing agent services and (2) make the yellow pages more efficient, robust and maintainable, using techniques from other hierarchical service indices such as those in the Harvest information retrieval system [BDH+94].

9.4 Network sensing and path planning

Under certain network conditions, it is more efficient for an agent to remain stationary and interact with a resource from a remote location, rather than migrating to that resource. Unfortunately, the exact conditions depend entirely on the resource and the agent's current task. For example, in most networks, an agent that needs to invoke only a single operation against a remote resource should almost always remain stationary, so that it avoids the migration overhead.[6] On the other hand, even in high-performance networks, an agent that needs to invoke hundreds of operations should almost always migrate, so that it avoids the overhead of the cross-network calls. Of course, although making a migration decision based solely on the expected

[6]One notable exception is if the operation produces an extremely large result of which the agent needs only a small fragment. The agent can filter the result on the remote machine, rather than transmitting the entire result across the network.

number of operations is a useful "rule of thumb", it is too simplistic in many cases. Also affecting the migration decision are the machine loads, the relative speed of agent versus native code, the reliability, latency and bandwidth of the network links, the average size of each operation's result, the specific performance constraints that the agent is trying to meet, and whether the resource's machine even allows third-party agents.

Clearly, to decide if, when and where to migrate, an agent must examine machine and network conditions and then combine the current status information with knowledge of the resource and its own task. Efforts along these lines fall into three subareas: (1) provide efficient network sensors and an effective description of a resource's behavior (expected result size, expected operation latency under light load, etc.); (2) provide a library of routines that accept the current network status, the resource description, and a simple description of the agent's task and return an appropriate migration decision; and (3) make agent code independent of whether the agent migrates or remains stationary, so that the programmer does not have to write two intertwined versions. Agent Tcl addresses the third issue already, since the communication primitives are the same whether the agent is communicating with a local or remote agent. Typically, an agent would decide whether to create a child agent on the local or remote machine and then use the same code, or would decide whether to migrate and then use the same code. On the other hand, the first two issues, network sensing and a decision-making library, are much more complex and are the subject of ongoing work.

Wilmer Caripe, Katsuhiro Moizumi[7] and several undergraduates in the computer science department are working on various network-sensing and decision-making tech-

[7]Wilmer Caripe and Hiro Moizumi are both Ph.D. students in the Thayer School of Engineering.

niques [GKN$^+$97, Car97]. Both passive and active network-sensing techniques are under development. Passive techniques include piggybacking bandwidth and latency information onto existing agent traffic as well as taking round-trip timings for existing agent traffic. In the latter case, for example, the round-trip time for a request sent to an agent server gives a rough approximation of the network transit time plus the server processing time. Active techniques revolve around a set of network monitoring agents, one per site, which store the passively-collected data and actively update this data by sending out "ping" packets on request or at periodic intervals. In particular, each network monitor keeps track of the expected latency, bandwidth and uptime of the link that connects its machine with the rest of the network. When an agent is deciding whether to migrate, it asks one or more network monitors about the current conditions. Stationary agents also make use of the network monitors. The docking system, for example, relies on the local network monitor to tell it when the machine's network connection is back up. Sumatra uses a similar system of network monitors called Komodo [RAS96].

Machines have their own characteristics, most notably CPU speed and current load. Although this information can also be handled in the network monitors, we are planning to use the Simple Network Management Protocol (SNMP) instead [Car97]. The SNMP protocol associates a simple database (or MIB) with each network device. The network device fills these fields with information about its inherent and current characteristics; applications run queries against the MIB to identify the device's current status. In our case, each machine would have a MIB that included CPU speed, current load, current number of agents, and so on. The advantage of SNMP is twofold. First, it is an existing, widespread protocol, which reduces the amount of code that needs to be written for a mobile-agent system. Second, it is external to the mobile-agent system, which allows non-agent applications to query the same device information.

Once network and machine monitors are available, the remaining task is to provide the decision-making routines. There are several broad approaches, most notably mathematical models and machine-learning techniques. Our current applications use machine learning, specifically the new Q-learning algorithm [CMG95]. Here each agent reports its observed performance to a Q-learning agent on each visited site; later agents query the Q-learning agents to obtain a performance prediction based on the past observations and the current network conditions [Car97]. The current prototype considers only agent size, result sizes, observed latencies, and time of day.

There are several important research questions, but the question of immediate interest to us is how much extra network traffic the network monitors must generate to keep their performance estimates up-to-date. Initial work suggests that a sampling rate of thirty seconds to several hours generates sufficiently accurate estimates; the extra traffic at this sampling rate should be only a small fraction of the overall traffic [RASS97].

9.5 Mobile Agent Construction Environment

Although both Tcl and Java are generally regarded as easy to learn, they are still languages that fall within the realm of a capable programmer, rather than a nonprogrammer. Therefore most end users of Agent Tcl are limited to pre-packaged agents that simply ask for certain parameters (such as which document collections to search, how many results to report, and so on). The Mobile Agent Construction Environment (MACE) is a first step towards removing this limitation and allowing end users to construct their own useful agents [Sha97]. MACE, which Rohit Sharma[8] developed as part of his thesis work, is a simple visual programming environment in which a user assembles a mobile agent from a set of predefined components. The MACE environment is shown in Figure 9.3. It is oriented towards workflow applications where an agent guides a work item through a series of steps, but there is no reason that it cannot be used for other applications, provided that those applications can be expressed in the current visual programming interface. Essentially the user is provided with a set of domain-specific components that perform various operations, such as querying a parts database, filtering a list of parts according to price, or presenting the list of parts to a user. These components appear as boxes in Figure 9.3. The user selects the desired components and connects them by clicking and dragging within the main window. Each connection represents a one-way flow of data; the target component uses the result from the source component and cannot start its task until the source component finishes. Each component is annotated with either a specific machine on which the task should be performed or a routine that selects an appropriate machine based on some application criteria. Certain components also take parameters.

Once the user finishes, the graphical representation is compiled into a Tcl agent. This agent makes heavy use of Agent Tcl's migration, cloning and communication facilities to send results from one component to another and to execute each component on the appropriate machine as soon as the necessary results are available. The agent can be launched immediately or stored on the local machine for later execution. MACE needs to be extended in several ways. First, the current set of predefined components is extremely small. A more ambitious component library needs to be developed for and tested in some selected application domain. Second, MACE allows

[8]Rohit Sharma graduated in 1997 with a Master's degree in computer engineering.

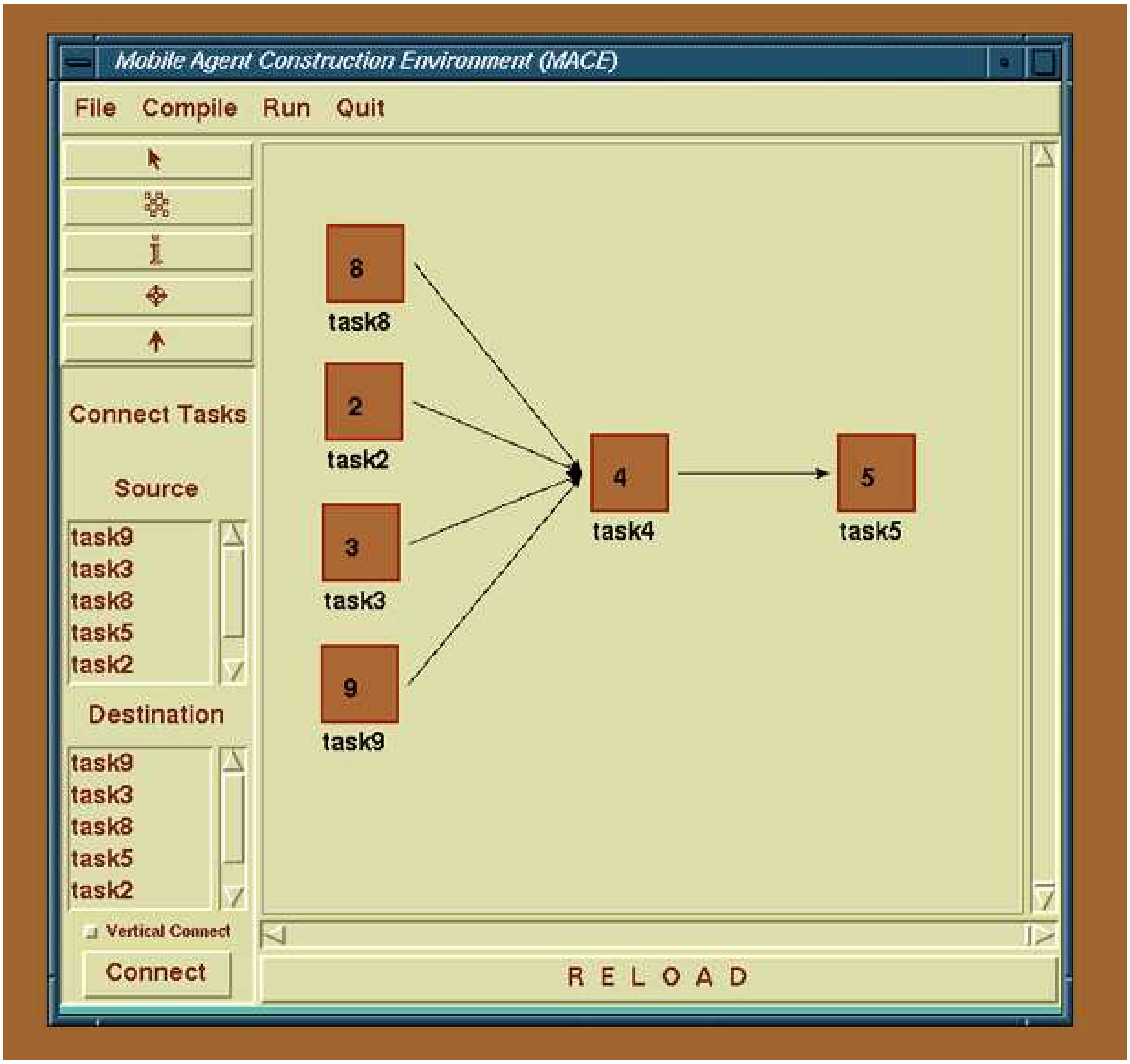


Figure 9.3: The Mobile Agent Construction Environment (MACE) allows a nonprogrammer to graphically construct an agent. This figure appears in [Sha97] and was used with permission.

limited interaction between components, namely the forwarding of a result from one component to another. A much wider range of interactions is possible, only some of which can be expressed easily in a visual programming environment. Finally, MACE generates only Tcl code. It should generate Java or Scheme code as well.

9.6 Agent RPC

Agent Tcl's builtin communication mechanisms are low-level bytestreams and message passing. The idea is to allow agents that have simple requirements to communicate with minimal overhead, while providing a base on top of which more complex communication protocols can be implemented efficiently at the agent level. Possible protocols include whiteboards, KQML [GSS94], remote procedure call (RPC) [BN84] and remote method invocation [YD96]. Agent RPC, which Saurab Nog and Sumit Chawla[9] developed as a course project, is an RPC mechanism for Agent Tcl [NCK96]. The architecture of Agent RPC is shown in Figure 9.4. Agent RPC is exactly analogous to traditional RPC and allows an agent to invoke exported operations from another agent as if those operations were local procedures. An interface definition is compiled into client and server stubs, which are included in the client and server agents. On startup, the server agent registers its location, keyword description and interface definition with one or more *nameserver* agents. To find a server agent that provides a particular service, a client agent queries the nameservers, either by name or by interface definition. In the case of interface definition, the nameservers match the desired interface against the interface of all registered server agents, returning

[9] Saurab Nog graduated in 1996 with a Master's degree in computer science and is now working at Microsoft. Sumit Chawla graduated in 1997 with a Ph.D. degree in computer science and is now working at SGI.

those agents that provide the same interface. The interface matching is quite flexible, ignoring parameter order and considering only the function name, the result type, and the number, names and types of the parameters. Once the client agent has identified an appropriate server agent, it connects to the server agent and invokes the exported server operations by calling the client stubs. Each client stub converts the procedure arguments into a message and sends this message along the connection to the server agent. The corresponding server stub unpacks the arguments, invokes the appropriate sever operation, and then sends back the result. Agent RPC has three main areas of future work. First, although Agent RPC is language-independent, the current stub compilers only generate Tcl stubs. The stub compilers should generate Java and Scheme stubs as well.[10] Second, as will be discussed further in the future work chapter, it must be possible to include arbitrary binary data in Agent Tcl messages (rather than just strings), which would make communication protocols such as Agent RPC more efficient. Finally, the separate nameservers can be eliminated by including the RPC interface definitions in the yellow pages.

[10] In addition, the stubs should work across languages so that client and server agents can be written in different languages. Since the stubs use the existing agent communication mechanisms, which work across languages already, this interoperability can be achieved without any extra work.

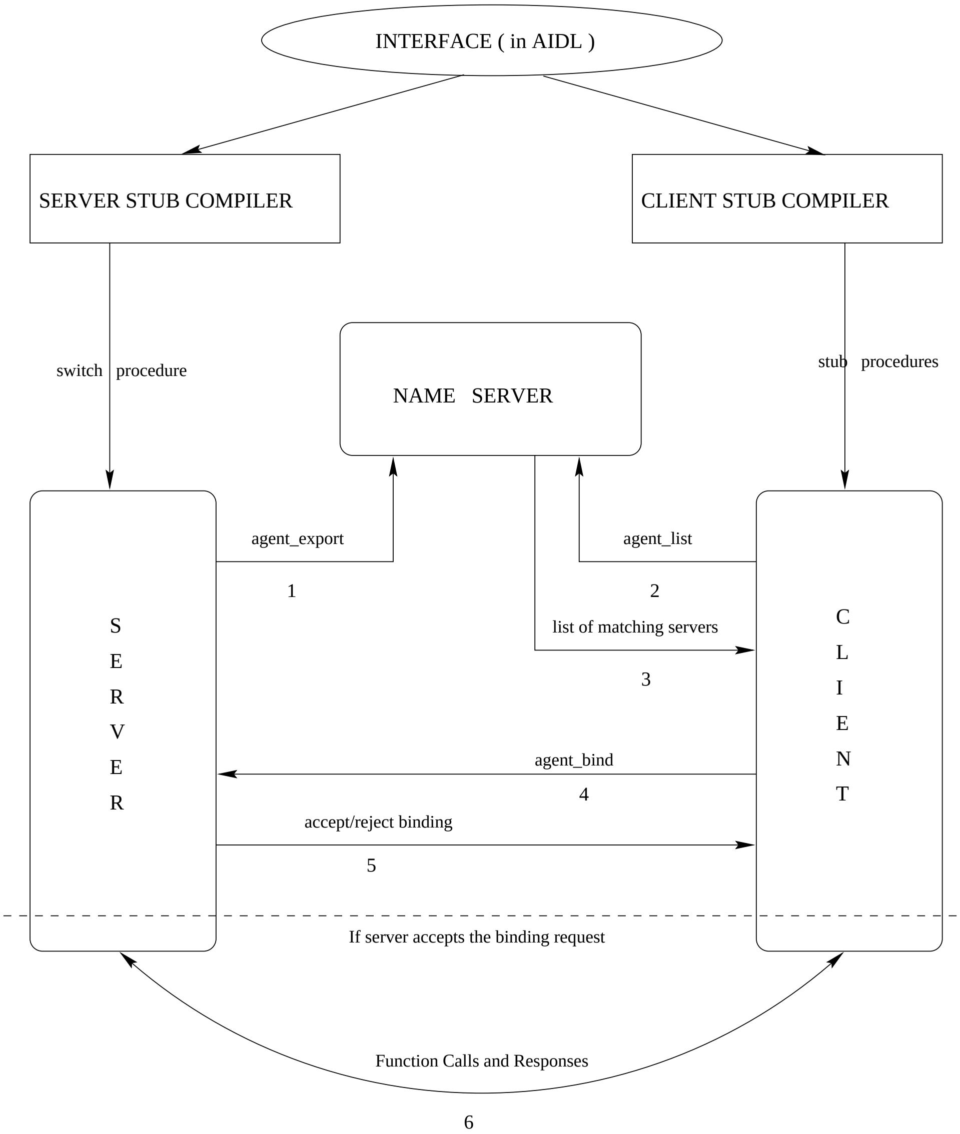


Figure 9.4: The architecture of the Agent RPC system. First a stub compiler transforms an interface definition into client and server stubs that are included in the client and server agents respectively. The server agent registers its interface with the nameservers on startup. The client agent finds the desired server agent by querying the nameservers, either by name or by interface definition. The client agent then connects to the server agent and calls its local client stubs to invoke server operations. This figure appears in [NCK96] and was used with permission.

Chapter 10

Future work

Performance analysis and performance improvements. When network conditions are good, i.e., when bandwidth and reliability are high and latency is low, traditional RPC exhibits significantly better performance than mobile Tcl agents, mainly due to the slowness of Tcl, the migration overhead, and the simplistic implementation of the agent servers and the communication subsystem. The performance of the Tcl agents is good enough, however, to suggest that a combination of faster languages and additional system engineering will lead to competitive performance under even the best network conditions. Other performance studies such as [Knu95] and [RASS97] bear out this contention.

Two faster languages, Java and Scheme, have already been added to Agent Tcl. In addition, there are several obvious improvements that can be made in the system implementation. First, the amount of dynamic memory allocation in the messaging subsystem can be reduced significantly. Second, shared memory, rather than Unix domain sockets, can be used for communication among agents on the same machine, although the resulting performance improvement will be much greater on some architectures than others. Third, every agent and every server request is currently executed in a separate process, which requires process creation, interprocess communication and interprocess synchronization, all of which have high overhead. Although we do not feel that it is worthwhile to multithread the entire system as in Ara [Pei96], the server itself should be multithreaded, eliminating most process creation and several extraneous communication steps. We will also consider limited multithreading in the interpreters themselves. For example, given that Java and Scheme are already multithreaded, it would be reasonable to run all Java agents inside one process, all Scheme agents inside another, and each Tcl agent in its own process as before. Fourth, even without any multithreading of the interpreters, migration latency can be reduced significantly with a pool of ready interpreter processes and application-level caching of needed initialization files, particularly in the case of Tcl. Finally, when both the source and target agent of a communication are on the same machine, the server should always be bypassed.

Once the system enhancements are finished, we need to reevaluate performance under a wider range of network conditions. Depending on the performance of Java agents relative to native code, we will consider either "just-in-time" compilation or software-fault-isolation of native code (or more likely of code for a low-level virtual machine that is immediately compiled into native code) [ATLLW96, WLAG93, LSW95]. Alternatively, the system could accept native code only from certain trusted users, eliminating the need for software fault isolation, but limiting untrusted agents to the slower, interpreted languages. Finally, once we have replaced PGP with a more efficient and flexible encryption subsystem, we need to compare *secure* mobile agents with *secure* versions of the traditional distributed computing paradigms. To our knowledge, no such performance comparison has been done; our own performance analysis was limited to anonymous agents that required neither encryption nor digital signatures. It seems reasonable to expect, however, that authenticating a mobile agent should involve roughly the same overhead as authenticating a cross-network call, and that we will see the same *relative* performance whether or not the agents are encrypted.

Security. The immediate task is to replace PGP with a faster and more flexible encryption library, finish the security enforcement modules for Java and Scheme, allow untrusted agents to have *limited* screen access, and eliminate the remaining denial-of-service attacks, such as an agent that sends messages to another agent as fast as possible or an agent that sits in a tight loop and uses its allocated CPU time as fast as possible. Eliminating the denial-of-service attacks requires additional resource limits and rate throttles. Once this initial work is complete, we can move on to protect a group of machines from a malicious agent and an agent from a malicious machine. To protect a group of machines, we are looking at electronic cash schemes where each resource has an associated price; agents must spend electronic cash from their finite reserve to access the desired resources and thus cannot survive forever within the network [JvRS95, DiMMTH95].[1] To protect an agent, we are looking at a combination of audit trails [CGH+95], replication and voting schemes [MvRSS96], a component model in which an agent is divided into pieces that are encrypted and signed at different times and with different keys [CGH+95], limited forms of proof-carrying or self-authenticating code [PS97], and a tamper-proof movement history embedded inside a migrating agent [MvRSS96].

[1]Most electronic cash schemes involve significant network communication. In a mobile-agent system, it is critical to eliminate as much communication as possible and defer the rest so that it does not lie on a critical path.

Fault tolerance. Although mobile agents allow an application to make minimal use of a low-quality network link, they do not eliminate all of the fault-tolerance issues associated with traditional distributed computing and introduce some of their own. Agent Tcl currently does not address any of these issues, making fault tolerance one of the most critical areas of future work. Most importantly, both migratory and stationary agents must be able to live past a machine failure, restarting with the most current state image possible as soon as the machine comes back up. Such recovery requires a persistent store in which the server can store the agent's administrative information and in which the agent itself can store its current state information. One such persistent store is the Gamelon File I/O library from the Menai corporation, which provides object-oriented persistent storage through a familiar file-access API [Men96]. Work is underway to integrate this library into the Agent Tcl system. The server's internal tables will be mirrored into the persistent store; an agent's complete state will be saved at startup or upon arrival; and initially the agent will be able to save its *complete* state at additional times of its choosing. Since the complete state image is already captured during migration, no new code needs to be written; the same state image is simply written to the persistent store rather than transmitted across the network. If the server crashes and restarts, it will reload its internal tables from the persistent store and restart all agents with their saved state images. Due to the overhead involved in writing the entire state image to disk, even if it is just written once upon agent arrival, persistent state backup will be an optional feature, so that a noncritical agent can get better performance at the risk of suddenly vanishing due to machine failure.

Requiring the agent to checkpoint its complete state is inefficient and often unnecessary. The agent should be able to perform an incremental checkpoint of its complete state or a partial checkpoint of some desired subset of its state. Incremental checkpoints save only the *changes* in the agent's state, either automatically or at agent-specified times, maintaining a complete state image in persistent store without the overhead of recapturing the entire state image. Unfortunately, incremental checkpointing require either substantial support from the interpreters or a general mechanism for incrementally capturing the state of arbitrary *native* processes, both of which are beyond the scope of the project. Partial checkpointing, however, is much more straightforward and, in its simplest form, requires no language-specific support. Each agent is allocated a fixed amount of space within the persistent store and allowed to write arbitrary data into that space. When an agent restarts after a machine crash, Agent Tcl raises an exception within the agent, so that the agent knows about the restart and can check the persistent store for previously saved data.[2]

Of course, it would also be useful to integrate partial state capture more closely with each agent language. Tcl *traces*, for example, provide a way to automatically detect and save any change in the value of a Tcl variable; Ara uses traces for just this purpose [Pei96]. Similarly, persistent objects can be implemented easily with Java's existing object serialization facilities [CH97]. Although such higher-level state capture is more convenient for the programmer, we plan to focus on the more general mechanism, and implement the language-specific mechanisms as time allows.

Allowing an agent to survive machine failure is a first step towards a robust mobile-agent system, but is not sufficient in and of itself. If a migrating agent temporarily disappears due to a machine crash, the overall application might not be able to wait

[2] Such an exception should always be raised after a machine crash so that a restored agent can take whatever action is appropriate for its task. For example, a time-dependent agent might report failure to its home site and then terminate if too much time has elapsed between crash and restart.

until the machine comes back up and the agent reappears. Therefore the application must be able to detect a machine failure so that it can spawn a new agent to carry on with the task. One approach is illustrated in Tacoma where a migrating agent leaves behind *rear guard* agents [JvRS95]. A rear guard concludes that the migrating agent has disappeared if it loses contact with the agent for a suitably long period of time. Once this happens, the rear guard sends out a new copy of the agent. In Tacoma, failed machines do not restart the agents that were executing on the machine, so the rear guards does not need to worry about the original agent's reappearance. In Agent Tcl, where a failed machine will restart the agents, any such rear guard mechanism must take the original agent's reappearance into account. One simple approach is to have a restarted agent terminate immediately if the crash lasted longer than the rear guard timeout interval. In either case, duplicate copies of the same agent are certainly possible if contact is lost due to a network rather than machine failure; preventing such duplicates is an open problem.

From a similar viewpoint, if a stationary service agent disappears due to a machine failure, having the service remain unavailable until the machine comes back up is no more reasonable than in any distributed computing environment, demanding traditional service replication with a backup service agent taking over for an unavailable primary [Mul93]. Mobile agents do allow an interesting enhancement to traditional replication schemes, however, namely on-the-fly replication in which a new copy of the service agent can be dynamically deployed to any desired network location, either in response to machine and network failures or to changing load [RASS97]. Finally, cooperating mobile agents are vulnerable to all the same communication failures as cooperating stationary processes, which means that some mobile-agent applications will need reliable group communication, transactions, voting schemes, and so on [Mul93].

As with replication, there are three interesting questions: (1) which existing techniques are the more natural fit for a mobile-agent system, (2) how can techniques intended for stationary processes be extended to handle the fact that the communicating entities are moving from machine to machine, and (3) how much of each technique can be implemented efficiently at the agent level so that the agent servers remain lightweight.

Mobile computing. Agent Tcl is meant to work seamlessly with both mobile computers and wireless networks. The existing *docking system* brings us a long ways towards this goal and will be our near-term focus [GKN+97]. The docking system must be re-implemented in a faster agent language, must be extended so that it handles inter-agent communication in addition to migration, and must actively exchange information about a machine's current network location so that as few agents as possible are actually routed through the docks.

Even with these enhancements, however, the docking system may prove too inefficient in wireless networks with rapidly changing configurations[3], since the machines will be unreachable frequently but often for only short periods. Buffering all agents and messages on some remote dock machine during these short periods of discon-

[3]An example of a "rapidly changing" network would be some number of moving vehicles and people, each with a wireless device that can communicate only with the other wireless devices (i.e., there is no central tower or satellite to serve as an intermediary). Network connectivity will change quickly as vehicles and people move in and out of range with each other. Such a network is in marked contrast to an essentially wired network, in which a laptop is unplugged from one location and later plugged in at a different location, or a tower- or satellite-based network, in which the only changes are infrequent cell handoffs. Cellular networks with small cells, where handoffs are frequent, fall somewhere in the middle.

nection could produce unacceptably high latencies. Since it is our intention to work with off-the-shelf networks rather than implementing our own network protocols, the first issue is to make all the information that the network is already exchanging about its current configuration visible to Agent Tcl, so that Agent Tcl will know a machine's current location whenever possible. In addition, the docks must be far more hierarchical and distributed, so that an agent or message can be buffered as close as possible to the suspected location of the target machine, even if this means buffering the agent or message on one or more machines that are themselves wireless. Of course, if a network already supports efficient location-independent addressing of the *physical* machines (e.g., a slowly changing network with mobile IP), the docking system becomes much simpler, since it no longer needs to keep track of temporary network locations. Instead it simply needs to buffer agents and messages at the most advantageous network locations. Although some networks can handle the buffering as well, relying on the network for buffering would eliminate several convenient features, such as the ability to wake up an agent and have it proceed with some alternative task if it is unable to reach the target machine within a specified timeout.

Standardization. There have been some recent efforts towards developing standards for mobile-agent systems, most notably by a multi-company coalition consisting of Crystaliz, General Magic, GMD FOKUS, IBM, and The Open Group. The coalition has developed a Mobile Agent Facility (MAF) specification in response to the Object Management Group's Common Facility Task Force RFP3 [MAF97]. The specification "focuses on the interoperability of different agent systems" [MAF97]. The specification does not allow an agent from one system to *execute* inside another system; such a cross-execution mechanism is simply impossible due to the wide range of programming languages that are used in different systems. Instead the specification (1) provides a uniform management interface, so that the system administrator can manage multiple agent systems via the same interface, and (2) allows an agent to locate and communicate with agents from different systems, so that clients and server providers do not have to have the same system. In the latter case, an agent would typically migrate to the system that had the same type as its home system and that was as close as possible to the desired resource, and then interact with the resource from across the network. Systems that were sufficiently similar could adopt additional standards to allow an agent to actually migrate from one system to another.[4]

Although Agent Tcl, like all current mobile-agent systems, does not support the MAF standard, it has most of the same functionality and can be extended easily to provide the rest. If the standard is accepted and gains acceptance in the developer community, supporting the full standard will allow Agent Tcl to interoperate with other systems, accelerating the acceptance of Agent Tcl and providing a much larger and more interesting environment in which Agent Tcl agents can be developed and used.

Modeling and simulation. Organizations are justifiably hesitant to install a prototype mobile-agent system on their machines, due to the security risks associated with the execution of untrusted code. Thus a mobile-agent system expands its installed base extremely slowly, with organizations first testing the system on internal, isolated networks and slowly making the system available to external agents. Although the slow spread of mobile-agent systems is acceptable in the long run, it does

[4] *At a minimum*, "sufficiently similar" means that the systems must support the same agent language, since there is not yet any adequate way to automatically translate an agent from one language to another.

significantly complicate the development and testing process, since it is nearly impossible to obtain a large testbed. Most testing and development is done in small, local networks, leaving open questions about the scalability of security and fault-tolerance algorithms. There are two partial solutions. The first solution is to construct a larger testbed by cooperating with other mobile-agent research groups, each of which is facing the same problem. Currently Dartmouth, Cornell and the University of Tromso in Norway are setting aside a few machines each. Each group's mobile-agent system will be installed on each machine, allowing experimentation across a much wider area although still on a small number of machines. We hope to extend this testbed to other universities and eventually a few commercial research labs.

The second solution is to develop a formal model of a mobile-agent system and then *simulate* system execution on as many virtual machines as desired. Although simulation would never be able to predict the performance of an actual system exactly, it would allow a relative comparison of different security and fault-tolerance strategies. Many potential strategies could be eliminated from consideration without the need to test each one within an actual network. Appropriate simulation environments are critical to future success and must be developed soon. Although other groups may be working on such simulation environments, none had been made public at the time of this writing.

Binary messages. Agent Tcl messages are currently null-terminated ASCII strings. Although strings are an appropriate transmission unit for some applications, such as those in which the agents exchange KQML messages [GK94], strings introduce extra overhead in many other applications, such as those in which the agents exchange numeric data. The overhead comes from converting the data into a string and then *parsing* the string to recover the data. To efficiently support a wider range of communication styles, Agent Tcl must be extended so that a message can include either a string or arbitrary binary data. Helper routines would be provided to add and extract standard RPC data types [BN84] from a binary data buffer. The ability to transmit binary data would eliminate the parsing overhead, and in particular, would be a much more efficient lower-layer for the Agent RPC system [NCK96]. The messaging subsystem in Agent Tcl can already handle binary messages; we just need to implement a binary data type and the helper routines for each language.

Unmodified interpreters. The *jump* command requires modifications to the Tcl, Java and Scheme interpreters since the unmodified interpreters do not allow the capture of an executing program's (or thread's) complete state. These modifications are not a major concern yet, since both the binaries and source code of the modified Tcl and Scheme interpreters and the binaries of the modified Java interpreter can be distributed freely for academic use. However, the modified interpreters increase the size of the Agent Tcl code base and lead to a significant amount of extra work when porting Agent Tcl to a new platform or when moving to a new version of the interpreter. In addition, some users have already objected to installing another version of the interpreter on their system when the standard version is already available. The most immediate need is to create a version of Agent Tcl that does not provide the *jump* command and thus can run with the unmodified interpreters. Such a version would use migration techniques similar to those found in Aglets [Ven97], Odyssey [Gen97], Concordia [WPW+97] or Tacoma [JvRS95]. At the same time, we need to decide whether the convenience of the *jump* command outweighs the additional burden that it places on the *system* programmer. If it does, we need to work with the interpreter developers to get the necessary state capture routines added to the standard interpreters. Due to the growing interest in mobile computation, the developers are likely to agree.

Administration tools. Agent Tcl currently does not provide a machine's owner or administrator with an effective view of the agents running on that machine, supporting little more than the forcible termination of a particular agent. This does not represent any conceptual fault with the Agent Tcl system, but rather a lack of time for implementing administration tools. However, as Agent Tcl moves into wider release, such tools are becoming more and more critical. The administrator needs a graphical tool that shows the status and identity of all the agents running on the current machine and allows her to terminate, suspend, resume or adjust the security parameters for any and all agents. To support such a tool, the server needs to report on the allowances that the resource managers have assigned to each agent and how much of each allowance an agent has left; the resource managers must accept temporary or permanent security policy changes for individual agents or agent owners; and the agent core must respond to suspend and resume messages sent from the server. Aside from suspend and resume, all the necessary information and mechanisms already exist within the Agent Tcl system, but cannot be accessed through a single point of control.

Web access. One way to bring Agent Tcl into much wider use is to make the system accessible via the Web. There are two approaches. Most or all of the system can be rewritten in Java and downloaded on demand to Java-enabled web browsers, essentially turning these browsers into temporary agent servers [Ven97]. Alternatively, the system can provide a much simpler Java applet or CGI script that interfaces with an agent server on the same machine as the web server. A user would give an agent to the applet or CGI script, which would pass the agent on to the agent server [JvRS96]. Although the first approach is more flexible, it is much more complex and is mainly a matter of porting code, making it more suited to a commercial product rather than a research project. Instead we will take the second approach.

Operating system support. The security and resource scheduling mechanisms in most operating systems are a poor match for mobile agents. For example, a mobile-agent system must impose a limit on disk accesses per unit time so that an agent can not thrash the local disk. Most operating systems do not support such a limit, forcing the mobile-agent system to implement the limit itself, something that is awkward at best since the mobile-agent system is executing in application space. If the operating system did support such limits, however, and could schedule available capacity among competing entities, the mobile-agent implementation would be more efficient and much simpler.

Chapter 11

Conclusion

Agent Tcl is a simple but powerful mobile-agent system that distinguishes itself from other mobile-agent systems with (1) its combination of multiple languages, a simple migration mechanism, and both low- and high-level communication protocols, (2) its simple but effective security model, and (3) its extensive support services and tools.

- **Multiple languages**. Agent Tcl supports multiple, off-the-shelf languages, Tcl, Java and Scheme, and allows the straightforward addition of new languages. The agent programmer can select the language that is most appropriate for her task.

- **Migration**. Agent Tcl reduces migration to a single instruction, `jump`, which automatically capture the complete state of the agent and sends the state image to the new machine. The agent continues from the point of the `jump` on the new machine. Although the system programmer must implement the `jump` instruction for each supported language, once the instruction is available, the agent programmers do not need to explicitly collect state information before migration.

- **Communication**. The base Agent Tcl system provides two low-level communication mechanisms, messaging passing and direct connections (for bulk data transfer), which work the same regardless of whether the communicating agents are on the same or different machines. Higher-level communication mechanisms, such as a Remote Procedure Call (RPC) mechanism [NCK96], are implemented at the agent level on top of the two low-level services. With this approach, the agent programmer can choose from a range of communication mechanisms, but the base system remains lightweight.

- **Security**. Agent Tcl protects an individual machine against malicious agents with a straightforward security model that cleanly separates policy and enforcement. Agents are digitally signed during migration so that their owner can be identified. Resource manager agents use the identity of the agent's owner to decide which screen, network, disk, etc., accesses are allowed for that agent. Lightweight enforcement modules for each supported language enforce the manager decisions.

- **Support services**. Agent Tcl provides numerous support services, most notably (1) a debugger that tracks an agent as it moves through the network and provides traditional debugger features such as breakpoints, watch conditions and line-at-a-time execution [HK97], (2) a docking system that allows an agent to transparently migrate to or from a mobile computer [GKN+97], (3) hierarchical yellow pages that provide a keyword-indexed directory of available services [GKN+97], (4) several network sensing and planning modules that allow an agent to determine the best route through the network [GKN+97, Car97], and (5) a Mobile Agent Construct Environment (MACE) that allows a nonprogrammer to graphically construct an agent [Sha97].

The main weaknesses of Agent Tcl are its lack of fault-recovery mechanisms and its poor performance relative to traditional client-server techniques. The poor performance is due to to the large migration overhead and the slowness of Tcl (which was the only language considered in the performance analysis). In a mid- or high-performance network, an Agent Tcl agent is an attractive choice only if (1) the client-server solution would perform a hundred or more cross-network calls or (2) link reliability or the latency of each *individual* call is the overriding performance concern. The performance numbers are good enough, however, to suggest that a combination of faster languages and additional system engineering will make an Agent Tcl agent competitive with or better than the corresponding client-server solution in a much wider range of applications, including those where the client-server solution performs only a handful of cross-network calls. Two faster languages, Scheme and Java, are already in place, and we are working to make the messaging system more efficient and to eliminate most of the migration overhead.

Once performance and fault tolerance are addressed (as well as some of the less critical issues that were discussed in Chapter 10), Agent Tcl will become an attractive platform for most distributed applications, not because it makes radically new applications possible, but because applications can be implemented efficiently and easily within a single, general framework.

Bibliography

[AS94] Stephen Appleby and Simon Steward. Mobile software agents for control in telecommunication networks. *British Telecom Technical Journal*, 12(2), April 1994.

[AS97] Anurag Acharya and Joel Saltz. Dynamic linking for mobile programs. In J. Vitek and C. Tschudin, editors, *Mobile Object Systems*, Springer-Verlag Lecture Notes in Computer Science. Springer-Verlag, 1997.

[ATLLW96] Ali-Reza Adl-Tabatabai, Geoff Langdale, Steven Lucco, and Robert Wahbe. Efficient and language-independent mobile programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 127–136, 1996.

[BC95] Krishna A. Bharat and Luca Cardelli. Migratory applications. In *Proceedings of the Eighth Annual ACM Symposium on User Interface Software and Technology*, November 1995.

[BDH+94] C. Mic Bowman, Peter B. Danzig, Darren R. Hardy, Udi Manber, and Michael F. Schwartz. Harvest: A scalable, customizable discovery and access system. Technical Report CU-CS-732-94, University of Colorado, Boulder, CO, August 1994.

[BFD96] Lubomir F. Bic, Munehiro Fukuda, and Michael B. Dillencourt. Distributed computing using autonomous objects. *IEEE Computer*, 29(8):55–61, August 1996.

[BHR+97] J. Baumann, F. Hohl, N. Radouniklis, M. Straber, and K. Rothermel. Communication concepts for mobile-agent systems. In *Proceedings of the First International Workshop on Mobile Agents (MA '97)*, volume 1219 of *Lecture Notes in Computer Science*, Berlin, April 1997. Springer-Verlag.

[BN84] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[BN97] Marc H. Brown and Marc A. Najork. Distributed active objects. *Dr. Dobb's Journal*, (263):34–41, March 1997.

[BP88] Andrea J. Borr and Franco Putzolu. High performance SQL through low-level system integration. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 342–349, Chicago, Illinois, 1988. ACM Press.

[Car95] Luca Cardelli. A language with distributed scope. *Computing Systems*, 8(1):27–59, Winter 1995.

[Car97] Wilmer Caripe. Applicability of the SNMP network management framework to the network-sensing module supporting the mobile-agent planning process. Technical report, Thayer School of Engineering, Dartmouth College, Hanover, New Hampshire, 1997. In progress.

[CB97] David Chaum and Stefan Brands. "Minting" electronic cash. *IEEE Spectrum*, 34(2):30–34, February 1997. Special issue on Technology and Electronic Economy.

[CBC96] Kurt Cohen, Aditya Bhasin, and George Cybenko. Pattern recognition of 3D CAD objects: Towards an electronic yellow pages of mechanical parts. *International Journal of Intelligent Engineering Systems*, 1996.

[CGH+95] David Chess, Benjamin Grosof, Colin Harrison, David Levine, Colin Parris, and Gene Tsudik. Itinerant agents for mobile computing. *IEEE Personal Communications*, 2(5):34–49, October 1995.

[CGN96] Ting Cai, Peter A. Gloor, and Saurab Nog. DartFlow: A workflow management system on the web using transportable agents. Technical Report PCS-TR96-283, Department of Computer Science, Dartmouth College, Hanover, New Hampshire, May 1996.

[CH97] Gary Cornell and Cay S. Horstmann. *Core Java*. Sunsoft Press (Prentice Hall), 1997.

[Cha96] Phil Inje Chang. Inside the Java Web Server: An overview of Java Web Server 1.0, Java Servlets, and the JavaServer architecture. Sun Microsystems White Paper, Sun Microsystems, 1996.

[CJK95] Henry Cejtin, Suresh Jagannathan, and Richard Kelsey. Higher-order distributed objects. *ACM Transactions on Programming Languages and Systems*, 17(5), September 1995.

[CMB96] Michael Condict, Dejan Milojicic, and Don Bolinger. Toward a world-wide civilization of objects. In *Proceedings of the Seventh ACM SIGOPS European Workshop*, pages 25–32, September 1996.

[CMG95] George Cybenko, Katsuhiro Moizumi, and Robert S. Gray. Q-Learning: A tutorial and extensions. In *Proceedings of the 1995 Conference on the Mathematics of Artificial Neural Networks*, 1995.

[Coe94] Michael H. Coen. SodaBot: A software agent environment and construction system. In Yannis Labrou and Tim Finin, editors, *Proceedings of the CIKM Workshop on Intelligent Information Agents, Third International Conference on Information and Knowledge Management (CIKM 94)*, Gaithersburg, Maryland, December 1994.

[CR91] William Clinger and Jonathan Rees. Revised(4) report on the algortihmic language Scheme. *ACM Lisp Pointers IV*, pages 1–55, July 1991.

[CW97] Mary Campione and Kathy Walrath. *The Java tutorial: Object-oriented programming for the Internet.* Addison-Wesley, 1997.

[dIC96] Luiz Henrique de Figueiredo, Robert Ierusalimschy, and Waldemar Celes. Lua: An extensible embedded language. *Dr. Dobb's Journal*, 21(12), December 1996.

[DiMMTH95] Giovanna Di Marzo, Murhimanya Muhugusa, Christian Tschudin, and Jürgen Harms. The Messenger paradigm and its implications on distributed systems. In *Proceedings of the ICC'95 Workshop on Intelligent Computer Communication*, 1995.

[Dyb87] R. Kent Dybvig. *The Scheme Programming Language*. Prentice-Hall, 1987.

[EW94] Oren Etzionoi and Daniel Weld. A softbot-based interface to the Internet. *Communications of the ACM*, 37(7):48–53, July 1994.

[Fal87] Joseph R. Falcone. A programmable interface language for heterogeneous systems. *ACM Transactions on Computer Systems*, 5(4):330–351, November 1987.

[FHS97] Stephanie Forrest, Steven A. Hofmeyr, and Anil Somayaji. Computer immunology. *Communications of the ACM*, 40(10):88–96, October 1997.

[Fon93] Leonard N. Foner. What's an agent anyway: A sociological case study. Agents Memo 93-01, Agents Group, MIT Media Lab, 1993.

[FTP96] FTP Software's agent technology: A general overview. FTP Software Technical Report, FTP Software, 1996.

[Fun97] Stefan Funfrocken. How to integrate mobile agents into web servers. In *Proceedings of the WETICE '97 Workshop on Collaborative Agents in Distributed Web Applications*, Boston, Massachusetts, June 1997.

[Gai94] R. Stockton Gaines. Dixie language and intepreter issues. In *Proceedings of the USENIX Symposium on Very High Level Languages (VHLL '94)*, Sante Fe, New Mexico, October 1994.

[Gen97] *Odyssey: Beta Release 1.0*, 1997. Available as part of the Odyssey package at `http:// www.genmagic.com/ agents/`.

[GG88] D. K. Gifford and N. Glasser. Remote pipes and procedures for efficient distributed communication. *ACM Transactions on Computer Systems*, 6(3):258–283, August 1988.

[GK94] Michael R. Genesereth and Steven P. Ketchpel. Software agents. *Communications of the ACM*, 37(7):48–53, July 1994.

[GKCR97] Robert Gray, David Kotz, George Cybenko, and Daniela Rus. Agent Tcl. In William Cockayne and Michael Zyda, editors, *Itinerant Agents: Explanations and Examples with CD-ROM*. Manning Publishing, 1997. Imprints by Manning Publishing and Prentice Hall.

[GKN+97] Robert Gray, David Kotz, Saurab Nog, Daniela Rus, and George Cybenko. Mobile agents: The next generation in distributed computing. In *Proceedings of the Second Aizu International Symposium on Parallel Algorithms/Architectures Synthesis (pAs '97)*, Fukushima, Japan, March 1997.

[GM95] James Gosling and Henry McGilton. The Java language environment: A white paper. Sun Microsystems White Paper, Sun Microsystems, 1995.

[Gra95] Robert S. Gray. Agent Tcl: A transportable agent system. In James Mayfield and Tim Finin, editors, *Proceedings of the CIKM Workshop on Intelligent Information Agents, Fourth International Conference on Information and Knowledge Management (CIKM 95)*, Baltimore, Maryland, December 1995.

[Gra96] Robert S. Gray. Agent Tcl: A flexible and secure mobile-agent system. In Mark Diekhans and Mark Roseman, editors, *Proceedings of the Fourth Annual Tcl/Tk Workshop (TCL '96)*, Monterey, California, July 1996.

[Gra97] Robert S. Gray. *Agent Tcl: Release 2.0*, 1997. Available at `http://www.cs. dartmouth.edu/~agent/`.

[Gre97a] Michael Greenberg. FTP Software, Septmber 1997. Personal correspondence.

[Gre97b] Michael Greenberg. Non-authentication security modes for mobile agents. In *Proceedings of the 1997 Workshop on Mobile Agents and Security*, University of Maryland, October 1997.

[GSS94] Michael Genesereth, Narinder Singh, and Mustafa Syed. A distributed and anonymous knowledge sharing approach to software interoperation. In Yannis Labrou and Tim Finin, editors, *Proceedings of the CIKM Workshop on Intelligent Information Agents, Third International Conference on Information and Knowledge Management (CIKM 94)*, Gaithersburg, Maryland, December 1994.

[Haf95] Katie Hafner. Have your agent call my agent. *Newsweek*, 75(9), February 27 1995.

[Har95] Kenneth E. Harker. TIAS: A Transportable Intelligent Agent System. Technical Report PCS-TR95-258, Department of Computer Science, Dartmouth College, 1995.

[HBB96] David Halls, John Bates, and Jean Bacon. Flexible distributed programming using mobile code. In *Proceedings of the Seventh ACM SIGOPS European Workshop*, pages 225–231, September 1996.

[HCK95] Colin G. Harrison, David M. Chess, and Aaron Kershenbaum. Mobile agents: Are they a good idea? IBM Research Report, IBM T. J. Watson Research Center, March 1995.

[HCS97] Leon Hurst, Pádraig Cunningham, and Fergal Somers. Mobile agents—smart messages. In *Proceedings of the First International Workshop on Mobile Agents (MA '97)*, volume 1219 of *Lecture Notes in Computer Science*, Berlin, April 1997. Springer-Verlag.

[HG97] Gisli Hjalmtysson and Robert S. Gray. Dynamic C++ classes: A lightweight mechanism to update code in a running program. AT&T Research Technical Report, AT&T Research, 1997. Submitted to the 1998 USENIX Technical Conference.

[HK97] Melissa Hirschl and David Kotz. AGDB: A debugger for Agent Tcl. Technical Report PCS-TR97-306, Department of Computer Science, Dartmouth College, Hanover, New Hampshire, February 1997.

[HMPP96] John Hartman, Udi Manber, Larry Peterson, and Todd Proebsting. Liquid software: A new paradigm for networked systems. Technical Report TR96-11, Department of Computer Science, University of Arizonia, 1996.

[Hoh97] Fritz Hohl. Protecting mobile agents with blackbox security. In *Proceedings of the 1997 Workshop on Mobile Agents and Security*, University of Maryland, October 1997.

[JdT+95] Anthony D. Joseph, Alan F. de Lespinasse, Joshua A. Tauber, David K. Gifford, and M. Frans Kaashoek. Rover: A toolkit for mobile information access. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 156–171, Copper Mountain, Colorado, December 1995. ACM Press.

[JK96] Anthony D. Joseph and M. Frans Kaashoek. Building reliable mobile-aware applications using the Rover toolkit. In *Proceedings of the Second Annual International Conference on Mobile Computing and Networking (MOBICOM '96)*, pages 117–129, Rye, New York, November 1996.

[Joh93] Raymond W. Johnson. Autonomous knowledge agents: How agents use the tool command language. In *Proceedings of the 1993 Tcl Workshop*, 1993.

[Joh95] D. B. Johnson. Scalable support for transparent mobile host internetworking. *Wireless Networks*, 1:311–321, October 1995.

[JSvR97] Dag Johansen, Nils P. Sudmann, and Robbert van Renesse. Performance issues in Tacoma. In *Proceedings of the 3rd Workshop on Mobile Object Systems, 11th European Conference on Object-Oriented Programming*, June 1997.

[JTK97] Anthony D. Joseph, Joshua A. Tauber, and M. Frans Kaashoek. Mobile computing with the Rover toolkit. 46(3):337–352, March 1997.

[JvRS95] Dag Johansen, Robbert van Renesse, and Fred B. Scheidner. Operating system support for mobile agents. In *Proceedings of the Fifth IEEE Workshop on Hot Topics in Operating Systems (HTOS)*, pages 42–45, May 1995.

[JvRS96] Dag Johansen, Robbert van Renesse, and Fred B. Schneider. Supporting broad Internet access to TACOMA. In *Proceedings of the Seventh ACM SIGOPS European Workshop*, pages 55–58, September 1996.

[KK94] Keith Kotay and David Kotz. Transportable agents. In Yannis Labrou and Tim Finin, editors, *Proceedings of the CIKM Workshop on Intelligent Information Agents, Third International Conference on Information and Knowledge Management (CIKM 94)*, Gaithersburg, Maryland, December 1994.

[Kna96] Frederick Knabe. An overview of mobile agent programming. In *Proceedings of the Fifth LOMAPS Workshop on Analysis and Verification of Multiple-Agent Languages*, Stockholm, Sweden, June 1996.

[Knu95] Paul Knudsen. Comparing two distributed paradigms – a performance-case study. Master's thesis, Department of Computer Science, Institute of Mathematical and Physical Science, University of Tromsø, 1995.

[KPS95] Charlie Kaufman, Radia Perlman, and Mike Speciner. *Network Security: Private Communication in a Public World.* Prentice-Hall, New Jersey, 1995.

[KR95] Richard Kelsey and Jonathan Rees. A tractable Scheme implementation. *Lisp and Symbolic Computation*, 7(4), 1995.

[LDD95] Anselm Lingnau, Oswald Drobnik, and Peter Dömel. An HTTP-based infrastructure for mobile agents. *World Wide Web Journal*, (1), December 1995.

[Lew95] Ted G. Lewis. Where is client/server software heading? *IEEE Computer*, pages 49–55, April 1995.

[LO95] Jacob Y. Levy and John K. Ousterhout. Safe Tcl toolkit for electronic meeting places. In *Proceedings of the First USENIX Workshop on Electronic Commerce*, pages 133–135, July 1995.

[LO97] Danny B. Lange and Mitsuru Oshima. *The Aglet cookbook*. 1997. In progress. Selected chapters available at `http:// www.trl.ibm.co.jp/ aglets/ aglet-book/ index.html`.

[Log96] LogicWare technology: Foundation for next generation, world wide, collaborative applications. Crystaliz White Paper, Crystaliz, 1996.

[LSW95] Steven Lucco, Oliver Sharp, and Robert Wahbe. Omniware: A universal substrate for web programming. *World Wide Web Journal*, (1), December 1995.

[Luc96] Inferno: la Commedia Interativa. Lucent Technologies Technical Report, Lucent Technologies, 1996. This report is not available publically. Some information on Inferno can be found at `http:// www.lucent.com/ inferno/`.

[Lut96] Mark Lutz. *Programming Python*. O'Reilly and Associates, 1996.

[Mad96] Peter W. Madany. JavaOS: A standalone Java environment. Sun Microsystems White Paper, Sun Microsystems, 1996.

[Mae94] Pattie Maes. Agents that reduce work and information overload. *Communications of the ACM*, 37(7):30–40, July 1994.

[MAF97] Mobile Agent Facility Specification (joint submissions). Technical report, Crystaliz, General Magic, GMD FOKUS, Internal Business Machine Corporation, and The Open Group, 1997. Response to OMG's Common Facility Task Force RFP3. Draft 5 is available at `http://www.genmagic.com/agents/MAF/`.

[MBZM96] Dejan S. Milojicic, Don Bolinger, Mary Ellen Zurko, and Murray Mazer. Mobile objects and agents. The Open Group Research Institute, November 1996.

[Men96] Gamelon file I/O library. Menai Corporation White Paper, Menai Corporation, 1996.

[Mit97a] Mobile agent computing. Mitsubishi Electric ITA White Paper, Mitsubishi Electric ITA, 1997.

[MIT97b] The Scheme programming language. WWW page `http://www-swiss.ai.mit.edu/scheme-home.html`, 1997.

[Mul93] Sape Mullender, editor. *Distributed Systems.* Addison-Wesley, second edition, 1993. Lecture notes from the annual Advanced Course on Distributed Systems.

[MvRSS96] Yaron Minsky, Robbert van Renesse, Fred B. Schneider, and Scott D. Stoller. Cryptographic support for fault-tolerant distributed computing. In *Proceedings of the Seventh ACM SIGOPS European Workshop*, pages 109–114, September 1996.

[NCK96] Saurab Nog, Sumit Chawla, and David Kotz. An RPC mechanism for transportable agents. Technical Report PCS-TR96–280, Department of Computer Science, Dartmouth College, March 1996.

[OLW97] John K. Ousterhout, Jacob Y. Levy, and Brent B. Welch. The Safe-Tcl security model. Technical report, Sun Microsystems Laboratories, 1997. In progress. Draft available at `http:// www.sunlabs.com/ people/ john.ousterhout/ safeTcl.html`.

[OPL94] Tim Oates, M. V. Nagendra Prasad, and Victor Lesser. Networked information retrieval as distributed problem solving. In Yannis Labrou and Tim Finin, editors, *Proceedings of the CIKM Workshop on Intelligent Information Agents, Third International Conference on Information and Knowledge Management (CIKM 94)*, Gaithersburg, Maryland, December 1994.

[Ous94] John K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley Professional Computing Series. Addison-Wesley, Reading, Massachusetts, 1994.

[Pei96] Holger Peine. The Ara project. WWW page `http:// www.uni-kl.edu/AG-Nehmer/Ara`, Distributed Systems Group, Department of Computer Science, University of Kaiserlautern, 1996.

[PS97] Holger Peine and Torsten Stolpmann. The architecture of the Ara platform for mobile agents. In *Proceedings of the First International Workshop on Mobile Agents (MA '97)*, volume 1219 of *Lecture Notes in Computer Science*, Berlin, April 1997. Springer-Verlag.

[RAS96] M. Ranganathan, Anurag Acharya, and Joel Saltz. Distributed resource monitors for mobile objects. In *Proceedings of the Fifth International Workshop on Object Orientation in Operating Systems*, 1996.

[RASS97] M. Ranganathan, Anurag Acharya, Shamik Sharma, and Joel Saltz. Network-aware mobile programs. In *Proceedings of the 1997 USENIX Technical Conference*, pages 91–104, 1997.

[Rei94] Andy Reinhardt. The network with smarts. *Byte*, pages 51–64, October 1994.

[RGK96] Daniela Rus, Robert Gray, and David Kotz. Autonomous and adaptive agents that gather information. In *AAAI '96 International Workshop on Intelligent Adaptive Agents*, August 1996. To appear.

[Rie94] Doug Riecken. M: An architecture of distributed agents. *Communications of the ACM*, 37(7):106–116, July 1994.

[RN95] Stuart Russell and Peter Norvig. *Artificial intelligence: A modern approach.* Prentice-Hall Series on Artificial Intelligence. Prentice-Hall, Englewood Cliffs, New Jersey, 1995.

[Rog95] Adams Rogers. Is there a case for viruses? *Newsweek*, 75(9), February 27 1995.

[Rul69] Jeff Rulifson. Decode encode language (DEL). Technical Report RFC-0005, DARPA Network Working Group, 1969.

[RWWB96] Roger Riggs, Jim Waldo, Ann Wollrath, and Krishna Bharat. Pickling state in the Java system. *Computing Systems*, 9(4):291–312, Fall 1996.

[SA94] Simon Steward and Steve Appleby. Mobile software agents for control of distributed systems based on principles of social insect behaviour. In *IEEE International Conference on Communication Systems (ICCS)*. IEEE Computer Society Press, November 1994.

[Sah94] Adam Sah. TC: An efficient implementation of the Tcl language. Master's thesis, University of California at Berkeley, May 1994. Available as Technical Report UCB-CSD-94-812.

[San97] Thomas Sander. On cryptographic protection of mobile agents. In *Proceedings of the 1997 Workshop on Mobile Agents and Security*, University of Maryland, October 1997.

[Sap96] Peter Sapaty. Mobile processing in open systems. In *Proceedings of the Fith Annual High Performance Computing Conference (HPCD '96)*, 1996.

[SBH96] F. Straber, J. Baumann, and F. Hohl. Mole - a Java-based mobile-agent system. In *Proceedings of the ECOOP '96 Workshop on Mobile Object Systems*, 1996.

[Sch97a] Fred B. Schneider. Security in Tacoma Too. In *Proceedings of the 1997 DAGSTUHL Workshop on Mobile Agents*, September 1997.

[Sch97b] Fred B. Schneider. Towards fault-tolerant and secure agentry. In *Proceedings of the 11th International Workshop on Distributed Algortithms*, September 1997.

[SD95] Marvin Sirbu and J. D.Tygar. NetBill: An Internet commerce system optimized for network delivered services. In *Proceedings of 40th IEEE*

Computer Society International Conference (COMPCON 95). IEEE Computer Society Press, March 1995.

[SG90] J. Stamos and D. Gifford. Remote evaluation. *ACM Transactions on Programming Languages and Systems*, 12(4):537–565, October 1990.

[Sha97] Rohit Sharma. Mobile agent construction environment. Master's thesis, Thayer School of Engineering, Dartmouth College, 1997.

[SS94] Mukesh Singhal and Niranjan G. Shivaratri. *Advanced Concepts in Operating Systems: Distributed, Database and Multiprocessor Operating Systems*. McGraw-Hill Series in Computer Science. McGraw-Hill, New York, 1994.

[Ste94] W. Richard Stevens. *TCP/IP Illustrated, Volume 1: The Protocols*. Addison-Wesley, 1994.

[Sto94] A. D. Stoyenko. SUPRA-RPC: SUbprogram PaRAmeters in Remote Procedure Calls. *Software–Practice and Experience*, 24(1):27–49, January 1994.

[Sun97a] HotJava: The security story. Sun Microsystems White Paper, Sun Microsystems, 1997.

[Sun97b] The Java language specification: Release 1.1. Sun Microsystems White Paper, Sun Microsystems, 1997.

[Sun97c] The Java Virtual Machine specification: Release 1.1. Sun Microsystems White Paper, Sun Microsystems, 1997.

[TDiMMH94] Christian Tschudin, Giovanna Di Marzo, Murhimanya Muhugusa, and Jürgen Harms. Messenger-based operating systems. Cahier du Centre Universitaire d'Informatique no 90, University of Geneva, Switzerland, 1994.

[TK93] Anand R. Tripathi and Neeran M. Karnik. Systems-level issues for agent-based distributed computing. Technical Report TR96-049, Department of Computer Science, University of Minnesota, 1993.

[TLKC95] Bent Thomsen, Lone Leth, Frederick Knabe, and Pierre-Yves Chevalier. Mobile agents. ECRC External Report, European Computer-Industry Research Centre, 1995.

[Tsc94] Christian Tschudin. An introduction to the Mϕ messenger language. Cahier du Centre Universitaire d'Informatique no 86, University of Geneva, Switzerland, 1994.

[Tsc97] Christian F. Tschudin. Open resource allocation for mobile agents. In *Proceedings of the 1997 Workshop on Mobile Agents (MA '97)*, volume 1219 of *Lecture Notes in Computer Science*, Berlin, April 1997. Springer-Verlag.

[TSS+97] D. L. Tennenhouse, J. M. Smith, W. D. Sincoskie, D. J. Wtherall, and G. J. Minden. A survey of active network research. *IEEE Communications*, 35(1):80–86, January 1997.

[TTP+95] Douglas B. Terry, Marvin M. Theimer, Karin Petersen, Alan J. Demers, Mike J. Spreitzer, and Carl H. Hauser. Managing update conflicts in a weakly connected replicated storage system. In *Proceedings* *of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 172–183, Copper Mountain, CO, December 1995. ACM Press.

[TV96] Joseph Tardo and Luis Valente. Mobile agent security and Telescript. In *Proceedings of the 41th International Conference of the IEEE Computer Society (CompCon '96)*, February 1996.

[Ven97] Bill Venners. Under the hood: The architecture of aglets. *JavaWorld*, January 1997. JavaWorld is an online magazine. This article is available at `http:// www.javaworld.com/ javaworld/ jw-04-1997/ jw-04-hood.html`.

[Voo91] Ellen M. Voorhees. Using computerized routers to control product flow. In *Proceedings of the Twenty-Fourth Annual Hawaii International Conference on System Sciences*, pages 275–282, January 1991.

[Voy97] Voyager technical overview. ObjectSpace White Paper, ObjectSpace, 1997.

[Wat95] Terri Watson. Effective wireless communication through application partitioning. In *Proceedings of the Fifth IEEE Workshop on Hot Topcs in Operating Systems (HTOS)*, pages 24–27, May 1995.

[Way95] Peter Wayner. *Agents Unleashed: A public domain look at agent technology.* AP Professional, Chestnut Hill, Massachusetts, 1995.

[Wel95] Brent B. Welch. *Practical Programming in Tcl and Tk*. Prentice-Hall, New Jersey, 1995.

[Whi94] James E. White. Telescript technology: The foundation for the electronic marketplace. General Magic White Paper, General Magic, Inc., 1994.

[Whi95a] James E. White. Telescript technology: An introduction to the language. General Magic White Paper, General Magic, 1995.

[Whi95b] James E. White. Telescript technology: Scenes from the electronic marketplace. General Magic White Paper, General Magic, 1995.

[Whi96] James E. White. Telescript technology: Mobile agents. 1996.

[WLAG93] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 203–216, Ashville, NC, 1993. ACM Press.

[WPW+97] D. Wong, N. Paciorek, T. Walsh, J. DiCelie, M. Young, and B. Peet. Concordia: An infrastructure for collaborating mobile agents. In *Proceedings of the First International Workshop on Mobile Agents (MA '97)*, volume 1219 of *Lecture Notes in Computer Science*, Berlin, April 1997. Springer-Verlag.

[WRW96] Ann Wollrath, Roger Riggs, and Jim Waldo. A distributed object model for the Java system. *Computing Systems*, 9(4):265–290, Fall 1996.

[Wu95] Yunxin Wu. Advanced algorithms of information organization and retrieval. Master's thesis, Thayer School of Engineering, Dartmouth College, 1995.

[WVF89] C. Daniel Wolfson, Ellen M. Voorhees, and Maura M. Flatley. Intelligent routers. In *Proceedings of the Ninth International Conference on Distributed Computing Systems*, pages 371–376. IEEE, June 1989.

[YD96] Zhonghua Yang and Keith Duddy. CORBA: A platform for distributed object computing. *ACM Operating Systems Review*, 30(2):4–31, April 1996.

[ZPMD97] Deborra J. Zukowski, Apratim Purakayastha, Ajay Mohindra, and Murthy Devarakonda. Metis: A thin-client application framework. In *Proceedings of the Conference on Object-oriented Technologies and Systems*. USENIX Association, 1997.

Appendix A

Performance data - Base performance

Tables A.1 through A.15 contain the timing data from the experiments that were done to measure Agent Tcl's base performance. Each data point is an average of between 2 and 180 measured times, where each measured time was obtained by timing 100 or 1000 iterations of the event in question and then dividing by 100 or 1000.[1] More iterations and trials were done for those experiments in which each iteration took a short time; fewer iterations and trials were done for those experiments in which each iteration took a longer time. Figure A.1 shows the distribution of the standard deviations for the data points, where each deviation is expressed as a percentage of the corresponding average. The data and the experiments are discussed further in Chapter 7.

[1]Between 4 and 360 times were obtained for each data point. All times above the median were thrown out, and the remaining 2 to 180 times were averaged. The intent was to keep only those times that correspond to periods of light network load.

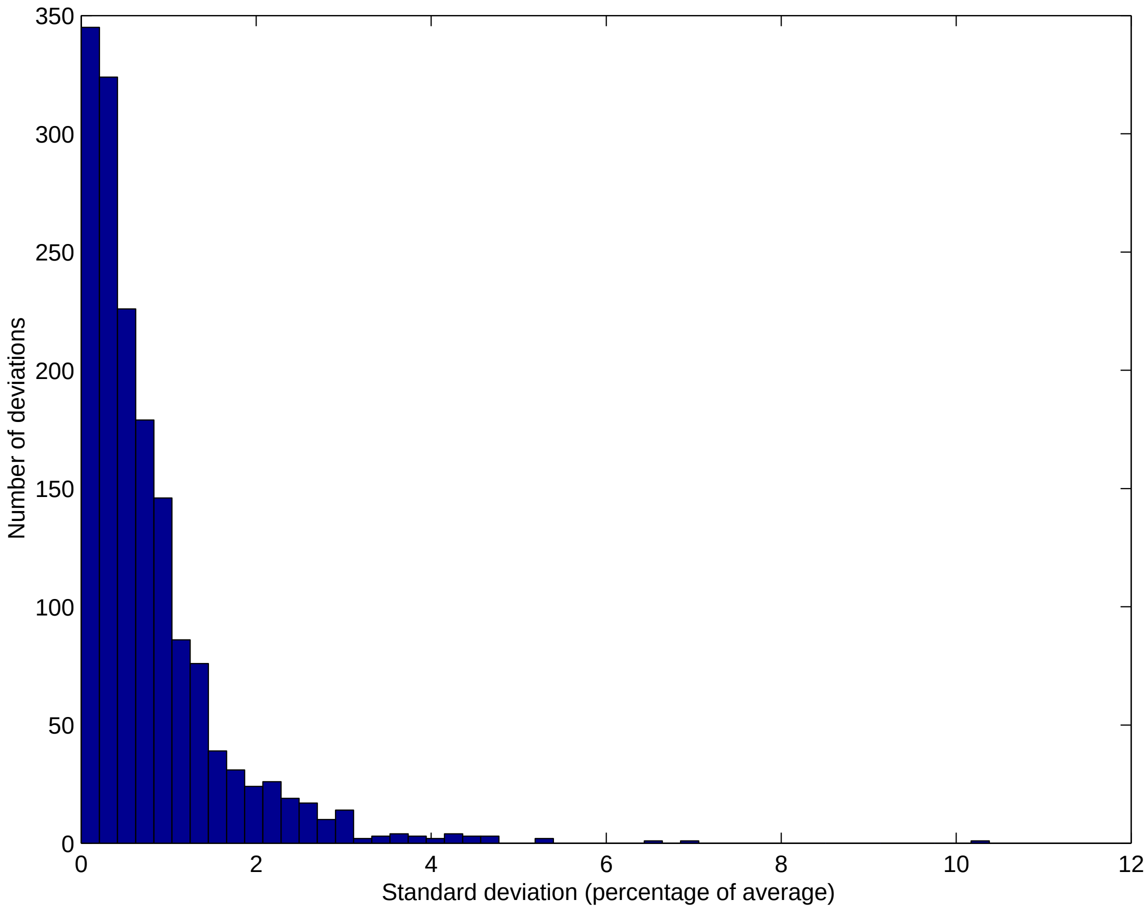


Figure A.1: Histogram of the standard deviations for the average times in Tables A.1 through A.15. Each standard deviation is expressed as a percentage of the corresponding average.

Request size (bytes)	Response size (bytes): 64	128	256	512
64	0.07	0.08	0.08	0.08
128	0.07	0.07	0.08	0.08
256	0.08	0.07	0.07	0.08
512	0.08	0.08	0.08	0.08
1024	0.10	0.09	0.09	0.10
2048	0.14	0.14	0.14	0.15
4196	0.22	0.22	0.22	0.23
8192	0.35	0.35	0.35	0.36
16384	0.60	0.60	0.61	0.61
32768	1.13	1.14	1.15	1.16
65536	2.30	2.30	2.31	2.32

Request size (bytes)	1024	2048	4196	8192
64	0.10	0.14	0.22	0.34
128	0.09	0.14	0.22	0.34
256	0.09	0.14	0.22	0.35
512	0.10	0.15	0.23	0.36
1024	0.10	0.16	0.24	0.37
2048	0.16	0.19	0.28	0.41
4196	0.24	0.28	0.35	0.48
8192	0.37	0.42	0.49	0.61
16384	0.63	0.67	0.74	0.87
32768	1.17	1.21	1.28	1.41
65536	2.33	2.39	2.45	2.61

Request size (bytes)	16384	32768	65536
64	0.57	1.03	2.20
128	0.58	1.04	2.21
256	0.58	1.04	2.21
512	0.59	1.05	2.23
1024	0.60	1.07	2.24
2048	0.64	1.12	2.30
4196	0.71	1.21	2.38
8192	0.85	1.34	2.53
16384	1.09	1.57	2.82
32768	1.63	2.08	3.27
65536	2.85	3.33	4.48

Table A.1: Time in milliseconds for two processes on the same machine to exchange a request and response over a Unix domain socket.

Response size (bytes)

Request size (bytes)		64	128	256	512
	64	0.21	0.22	0.24	0.27
	128	0.23	0.23	0.24	0.28
	256	0.24	0.24	0.26	0.30
	512	0.27	0.28	0.29	0.32
	1024	0.35	0.35	0.37	0.40
	2048	0.50	0.51	0.53	0.56
	4096	0.79	0.80	0.82	0.86
	8192	1.37	1.39	1.41	1.44
	16384	2.60	2.61	2.63	2.67
	32768	4.99	5.01	5.02	5.06
	65536	12.19	12.21	12.23	12.26

Request size (bytes)		1024	2048	4196	8192
	64	0.34	0.49	0.79	1.37
	128	0.35	0.50	0.80	1.38
	256	0.36	0.51	0.81	1.39
	512	0.39	0.55	0.85	1.42
	1024	0.46	0.62	0.92	1.50
	2048	0.64	0.78	1.06	1.64
	4096	0.93	1.07	1.34	1.93
	8192	1.51	1.66	1.94	2.51
	16384	2.74	2.89	3.18	3.76
	32768	5.14	5.29	5.59	6.20
	65536	12.34	12.54	12.82	13.51

Request size (bytes)		16384	32768	65536
	64	2.61	5.08	12.33
	128	2.62	5.08	12.34
	256	2.64	5.12	12.37
	512	2.68	5.15	12.39
	1024	2.75	5.22	12.47
	2048	2.89	5.38	12.62
	4096	3.19	5.68	12.91
	8192	3.76	6.26	13.55
	16384	4.98	7.53	14.86
	32768	7.49	9.92	17.25
	65536	14.82	19.12	24.42

Table A.2: Same as the previous experiment except that we are using the messaging subsystem from Agent Tcl.

		Response size (bytes)			
		64	128	256	512
	64	0.19	0.19	0.19	0.20
	128	0.19	0.19	0.19	0.20
	256	0.19	0.19	0.19	0.20
	512	0.20	0.20	0.20	0.21
Request	1024	0.21	0.21	0.22	0.23
size	2048	0.26	0.26	0.27	0.28
(bytes)	4196	0.37	0.38	0.38	0.39
	8192	0.59	0.59	0.60	0.61
	16384	0.93	0.93	0.93	0.94
	32768	1.73	1.74	1.75	1.76
	65536	3.43	3.44	3.45	3.46

		1024	2048	4196	8192
	64	0.21	0.26	0.37	0.57
	128	0.21	0.27	0.37	0.58
	256	0.22	0.27	0.38	0.58
	512	0.23	0.28	0.39	0.59
Request	1024	0.24	0.29	0.40	0.61
size	2048	0.29	0.34	0.45	0.64
(bytes)	4196	0.41	0.45	0.57	0.75
	8192	0.63	0.66	0.77	0.96
	16384	0.96	0.99	1.10	1.30
	32768	1.78	1.82	1.94	2.15
	65536	3.48	3.53	3.64	3.84

		16384	32768	65536
	64	0.91	1.72	3.37
	128	0.92	1.72	3.36
	256	0.92	1.74	3.38
	512	0.93	1.74	3.39
Request	1024	0.95	1.76	3.42
size	2048	0.98	1.81	3.47
(bytes)	4196	1.09	1.93	3.59
	8192	1.30	2.14	3.80
	16384	1.64	2.48	4.17
	32768	2.50	3.32	5.01
	65536	4.20	5.05	6.70

Table A.3: Time in milliseconds for two processes on the same machine to exchange a request and response over a TCP/IP socket.

Response size (bytes)

Request size (bytes)		64	128	256	512
	64	0.35	0.35	0.37	0.41
	128	0.35	0.36	0.38	0.42
	256	0.37	0.38	0.40	0.43
	512	0.41	0.42	0.44	0.47
	1024	0.48	0.49	0.51	0.55
	2048	0.64	0.65	0.67	0.70
	4096	0.98	0.99	1.01	1.04
	8192	1.64	1.65	1.67	1.70
	16384	2.96	2.96	2.99	3.02
	32768	5.76	5.76	5.79	5.83
	65536	13.14	13.14	13.17	13.21

Request size (bytes)		1024	2048	4196	8192
	64	0.48	0.63	0.97	1.64
	128	0.48	0.64	0.98	1.65
	256	0.51	0.66	1.00	1.67
	512	0.54	0.70	1.04	1.71
	1024	0.62	0.78	1.11	1.78
	2048	0.78	0.93	1.25	1.93
	4096	1.11	1.26	1.59	2.26
	8192	1.77	1.92	2.25	2.91
	16384	3.09	3.25	3.58	4.24
	32768	5.90	6.05	6.39	7.06
	65536	13.30	13.48	13.82	14.51

Request size (bytes)		16384	32768	65536
	64	3.01	5.89	13.32
	128	3.02	5.89	13.34
	256	3.04	5.91	13.36
	512	3.08	5.95	13.41
	1024	3.15	6.02	13.51
	2048	3.30	6.16	13.65
	4096	3.64	6.50	13.96
	8192	4.29	7.16	14.62
	16384	5.62	8.49	15.99
	32768	8.44	11.27	18.79
	65536	15.96	20.70	26.39

Table A.4: Same as the previous experiment except that we are using the messaging subsystem from Agent Tcl.

		Response size (bytes)			
		64	128	256	512
Request size (bytes)	64	0.29	0.30	0.31	0.34
	128	0.30	0.30	0.31	0.35
	256	0.31	0.32	0.33	0.35
	512	0.33	0.34	0.35	0.36
	1024	0.38	0.38	0.40	0.42
	2048	0.47	0.48	0.49	0.52
	4096	0.81	0.82	0.83	0.85
	8192	1.31	1.31	1.32	1.35

		1024	2048	4196	8192
Request size (bytes)	64	0.39	0.51	0.84	1.36
	128	0.40	0.52	0.85	1.36
	256	0.41	0.53	0.86	1.38
	512	0.43	0.55	0.88	1.40
	1024	0.45	0.59	0.92	1.44
	2048	0.57	0.67	1.00	1.51
	4096	0.90	1.01	1.33	1.88
	8192	1.39	1.51	1.87	2.38

Table A.5: Time in milliseconds to make an RPC call when the client and server are on the same machine.

Request size (bytes)	Response size (bytes): 64	128	256	512
64	2.11	2.36	2.80	3.58
128	2.61	2.84	3.26	4.07
256	2.80	3.03	3.44	4.21
512	3.62	3.84	4.25	5.06
1024	5.20	5.41	5.92	6.74
2048	7.43	7.74	8.19	8.93
4196	9.91	10.26	10.73	11.29
8192	14.23	14.59	14.75	15.26
16384	24.65	24.83	25.40	25.45
32768	47.76	48.03	48.99	49.21
65536	91.39	92.72	93.06	95.71

Request size (bytes)	1024	2048	4196	8192
64	5.15	7.46	9.25	14.14
128	5.67	7.95	9.65	14.29
256	5.92	8.19	9.94	14.37
512	6.59	8.97	10.61	15.27
1024	8.21	10.55	12.17	17.07
2048	10.50	12.88	14.45	19.10
4196	12.97	15.36	17.00	21.38
8192	17.01	19.24	21.31	25.64
16384	27.11	29.21	31.77	35.82
32768	50.35	52.85	54.57	59.25
65536	96.30	97.39	99.88	104.92

Request size (bytes)	16384	32768	65536
64	22.30	60.12	107.15
128	22.95	61.11	109.00
256	22.90	60.37	107.59
512	23.50	60.16	110.15
1024	25.49	61.77	110.92
2048	27.62	60.53	113.67
4196	29.72	67.41	113.08
8192	33.64	71.09	114.25
16384	43.83	78.26	140.78
32768	69.61	102.96	160.42
65536	113.82	147.83	195.18

Table A.6: Time in milliseconds for two processes on different machines to exchange a request and response over a TCP/IP socket.

		Response size (bytes)			
		64	128	256	512
	64	2.50	2.69	3.14	3.97
	128	3.00	3.12	3.64	4.36
	256	3.13	3.32	3.75	4.56
	512	3.99	4.14	4.58	5.46
Request size (bytes)	1024	5.69	5.83	6.24	7.17
	2048	8.34	8.55	8.92	9.72
	4096	10.59	10.92	11.54	12.07
	8192	15.55	15.58	16.38	16.93
	16384	27.38	27.53	27.74	28.96
	32768	54.00	53.78	53.25	54.98
	65536	103.38	102.60	104.16	106.35

		1024	2048	4196	8192
	64	5.62	8.00	9.99	15.14
	128	6.00	8.42	10.15	15.45
	256	6.22	8.55	10.46	15.66
	512	7.09	9.51	11.67	16.51
	1024	8.75	11.16	13.09	18.28
Request size (bytes)	2048	11.53	13.81	15.96	20.75
	4096	13.81	16.05	18.15	23.37
	8192	18.64	21.06	22.90	28.46
	16384	30.11	32.49	34.41	39.88
	32768	56.33	57.99	61.47	65.95
	65536	105.51	107.35	110.39	115.24

		16384	32768	65536
	64	24.19	60.25	113.40
	128	24.54	60.74	112.48
	256	25.13	60.63	114.81
	512	25.95	62.20	116.88
	1024	27.62	69.24	112.90
Request size (bytes)	2048	30.20	72.37	111.88
	4096	32.77	73.17	114.87
	8192	38.03	74.94	141.85
	16384	48.96	92.73	148.10
	32768	74.34	109.62	167.90
	65536	125.00	164.53	210.57

Table A.7: Same as the previous experiment except that we are using the messaging subsystem from Agent Tcl.

Request size (bytes)	Response size (bytes): 64	128	256	512
64	2.50	2.72	3.13	3.96
128	2.74	2.95	3.37	4.22
256	3.14	3.36	3.76	4.61
512	3.96	4.19	4.59	5.40
1024	5.60	5.81	6.22	7.04
2048	7.80	8.03	8.44	9.25
4096	10.34	10.68	10.98	11.78
8192	14.55	14.86	15.21	15.95

Request size (bytes)	1024	2048	4196	8192
64	5.67	7.46	9.73	13.81
128	5.84	7.68	9.97	14.08
256	6.25	8.11	10.36	14.45
512	7.09	8.99	11.17	15.31
1024	8.68	10.56	12.78	17.01
2048	10.89	12.73	15.06	19.33
4096	13.46	15.31	17.53	21.64
8192	17.64	19.48	21.75	25.94

Table A.8: Time in milliseconds to make an RPC call when the client and server are on different machines.

Response size (bytes)

Request size (bytes)		64	128	256	512
	64	0.90	0.93	0.97	1.06
	128	0.93	0.95	1.00	1.09
	256	0.98	1.01	1.04	1.14
	512	1.07	1.09	1.13	1.19
	1024	1.26	1.28	1.32	1.40
	2048	1.62	1.67	1.71	1.79
	4096	2.33	2.37	2.41	2.51
	8192	3.81	3.85	3.89	3.98
	16384	6.94	6.97	7.02	7.12
	32768	13.09	13.13	13.18	13.27
	65536	30.90	30.95	31.00	31.09

Request size (bytes)		1024	2048	4196	8192
	64	1.27	1.62	2.34	3.85
	128	1.29	1.67	2.38	3.88
	256	1.33	1.71	2.43	3.92
	512	1.40	1.79	2.51	4.01
	1024	1.55	1.95	2.70	4.21
	2048	1.99	2.28	3.05	4.58
	4096	2.69	3.04	3.72	5.29
	8192	4.17	4.54	5.26	6.73
	16384	7.31	7.68	8.41	9.88
	32768	13.48	13.85	14.58	16.11
	65536	31.30	31.70	32.47	34.04

Request size (bytes)		16384	32768	65536
	64	6.93	13.11	31.07
	128	6.98	13.17	31.10
	256	7.03	13.20	31.16
	512	7.11	13.30	31.23
	1024	7.29	13.48	31.42
	2048	7.67	13.87	31.88
	4096	8.40	14.60	32.59
	8192	9.90	16.12	34.18
	16384	12.98	19.35	37.36
	32768	19.17	27.17	43.16
	65536	37.14	43.23	59.98

Table A.9: Time in milliseconds for two agents on the same machine to exchange a request and response over an Agent Tcl meeting.

Response size (bytes)

Request size (bytes)		64	128	256	512
	64	3.04	3.26	3.70	4.60
	128	3.47	3.69	4.13	5.03
	256	3.71	3.93	4.36	5.26
	512	4.62	4.84	5.27	6.14
	1024	6.42	6.64	7.08	7.96
	2048	9.17	9.41	9.84	10.74
	4096	11.79	12.00	12.44	13.33
	8192	17.63	17.82	18.27	19.20
	16384	31.17	31.37	31.79	32.71
	32768	59.75	60.10	60.57	61.36
	65536	118.30	118.58	119.19	120.19

Request size (bytes)		1024	2048	4196	8192
	64	6.38	8.92	11.20	17.25
	128	6.80	9.34	11.62	17.70
	256	7.03	9.58	11.85	17.96
	512	7.93	10.47	12.75	18.85
	1024	9.70	12.25	14.58	20.64
	2048	12.50	14.96	17.26	23.39
	4096	15.10	17.61	19.98	26.10
	8192	20.96	23.48	25.88	31.92
	16384	34.53	36.86	39.37	45.46
	32768	63.19	65.85	68.09	74.14
	65536	121.76	124.54	127.09	133.63

Request size (bytes)		16384	32768	65536
	64	28.39	69.11	139.27
	128	28.79	70.81	136.79
	256	29.03	68.90	136.91
	512	29.90	70.56	139.59
	1024	31.71	71.18	139.77
	2048	34.49	71.35	141.44
	4096	37.20	70.57	140.61
	8192	43.00	84.97	148.11
	16384	56.61	96.97	161.01
	32768	85.32	127.80	186.25
	65536	145.13	185.54	263.28

Table A.10: Time in milliseconds for two agents on different machines to exchange a request and response over an Agent Tcl meeting.

Response size (bytes)

		64	128	256	512
	64	3.87	3.92	4.00	4.20
	128	3.90	3.96	4.06	4.26
	256	4.01	4.09	4.18	4.34
	512	4.21	4.24	4.32	4.52
Request	1024	4.57	4.62	4.71	4.88
size	2048	5.36	5.41	5.50	5.68
(bytes)	4096	6.84	6.89	6.99	7.15
	8192	9.82	9.88	9.97	10.09
	16384	16.10	16.16	16.26	16.47
	32768	28.45	28.51	28.59	28.79
	65536	63.65	63.68	63.76	63.94

		1024	2048	4196	8192
	64	4.59	5.37	6.82	9.85
	128	4.63	5.42	6.90	9.89
	256	4.74	5.52	7.02	9.98
	512	4.91	5.69	7.18	10.18
Request	1024	5.26	6.03	7.51	10.59
size	2048	6.04	6.80	8.31	11.35
(bytes)	4096	7.50	8.31	9.72	12.83
	8192	10.56	11.36	12.82	15.76
	16384	16.83	17.57	19.09	22.08
	32768	29.16	29.95	31.43	34.47
	65536	64.43	65.31	66.80	70.04

		16384	32768	65536
	64	16.21	28.46	63.26
	128	16.23	28.50	63.36
	256	16.38	28.67	63.43
	512	16.57	28.82	63.61
Request	1024	16.93	29.21	64.10
size	2048	17.67	29.96	64.86
(bytes)	4096	19.16	31.51	66.41
	8192	22.18	34.49	69.56
	16384	28.18	40.72	75.74
	32768	40.67	54.92	87.96
	65536	76.16	88.75	120.76

Table A.11: Time in milliseconds for two agents on the same machine to exchange a request and response using the `send` and `receive` primitives.

Response size (bytes)

Request size (bytes)		64	128	256	512
	64	28.28	31.15	33.50	33.28
	128	27.48	33.22	37.53	37.49
	256	27.96	32.87	37.58	34.15
	512	28.08	31.56	33.72	33.59
	1024	29.68	33.12	33.80	34.62
	2048	32.76	35.40	36.52	36.77
	4096	36.04	38.62	38.70	39.38
	8192	43.28	44.78	44.98	46.17
	16384	60.11	61.35	61.85	61.95
	32768	95.96	95.69	96.87	96.87
	65536	172.47	172.75	172.57	174.42

Request size (bytes)		1024	2048	4196	8192
	64	34.72	199.87	200.01	200.01
	128	34.67	199.84	200.01	200.00
	256	35.45	199.86	200.01	200.02
	512	35.17	199.84	200.01	200.01
	1024	35.69	199.89	200.01	200.01
	2048	38.49	199.89	200.01	200.01
	4096	40.12	199.86	200.01	200.01
	8192	46.38	199.92	200.01	200.01
	16384	62.40	199.91	200.01	200.01
	32768	96.95	199.91	200.01	200.01
	65536	173.45	202.63	202.81	245.62

Request size (bytes)		16384	32768	65536
	64	200.02	200.85	400.47
	128	200.03	200.05	399.87
	256	200.02	200.05	398.67
	512	200.02	200.44	399.67
	1024	200.01	200.04	400.07
	2048	200.02	200.45	399.88
	4096	200.02	200.05	399.86
	8192	200.02	200.04	399.86
	16384	200.02	200.05	399.87
	32768	200.22	242.65	400.07
	65536	399.82	400.05	401.28

Table A.12: Time in milliseconds for two agents on different machines to exchange a request and response using the `send` and `receive primitives`.

Agent size (bytes)	Time (ms)
64	126.59
128	126.49
256	126.75
512	126.93
1024	127.30
2048	128.12
4096	130.03
8192	133.18
16384	139.67
32768	154.14
65536	181.04

Table A.13: Time in milliseconds to submit an empty agent to the same machine and receive a dummy result.

Response size (bytes)

Agent size (bytes)	64	128	256	512
64	118.03	118.51	118.48	119.00
128	119.01	119.32	119.30	119.73
256	119.68	119.97	120.09	120.56
512	120.63	120.78	120.80	121.25
1024	122.67	122.64	122.63	122.91
2048	125.27	125.30	125.26	125.65
4096	130.57	130.66	130.66	131.14
8192	143.03	143.07	142.82	143.33
16384	168.56	168.43	168.33	168.74
32768	221.85	221.52	221.85	221.86
65536	323.90	325.41	323.70	323.79

Agent size (bytes)	1024	2048	4196	8192
64	119.44	197.77	200.89	201.37
128	120.18	195.20	200.85	201.34
256	121.00	195.53	200.85	201.36
512	121.57	201.15	200.85	201.30
1024	123.58	197.42	200.85	201.35
2048	126.09	201.59	200.84	201.28
4096	131.46	197.61	200.78	201.25
8192	143.67	197.19	200.63	201.16
16384	169.34	202.76	200.40	234.10
32768	222.57	388.82	399.84	400.36
65536	326.14	398.68	398.95	399.41

Agent size (bytes)	16384	32768	65536
64	203.21	386.12	410.94
128	203.22	386.08	410.76
256	203.21	386.13	410.83
512	203.21	386.06	410.98
1024	203.22	386.12	410.88
2048	204.88	386.12	410.92
4096	219.22	405.70	410.82
8192	382.91	405.68	444.14
16384	402.43	405.48	590.42
32768	402.18	405.03	589.94
65536	465.32	600.38	623.70

Table A.14: Time in milliseconds to submit an empty agent to a remote machine and receive a dummy result.

	Final size (bytes)			
	64	128	256	512
64	390.55	401.24	401.48	401.55
128	390.59	401.86	401.52	401.76
256	390.45	401.52	401.59	401.41
512	390.83	401.34	401.55	401.62
Initial size (bytes) 1024	390.62	401.21	401.34	401.76
2048	390.79	401.00	401.31	401.90
4096	391.17	401.10	401.21	401.90
8192	409.76	401.07	401.48	401.66
16384	410.38	401.62	401.41	401.83
32768	584.55	575.41	575.83	578.86
65536	608.38	601.69	601.41	601.66

	1024	2048	4196	8192
64	401.66	401.90	402.07	404.03
128	401.55	402.00	401.97	404.00
256	401.62	401.59	401.55	403.76
512	401.72	402.03	401.93	403.90
Initial size (bytes) 1024	401.83	401.69	402.38	403.83
2048	401.38	401.72	402.45	403.69
4096	401.76	401.07	401.79	403.93
8192	401.59	401.86	401.97	403.93
16384	401.86	401.90	402.21	466.41
32768	584.72	589.86	595.17	603.62
65536	601.55	602.34	602.55	648.62

	16384	32768	65536
64	406.07	597.79	827.66
128	405.86	597.83	833.62
256	405.93	597.90	839.48
512	406.03	597.79	837.86
Initial size (bytes) 1024	406.93	597.83	835.17
2048	411.48	597.72	842.93
4096	428.10	597.90	837.07
8192	585.48	610.93	832.69
16384	585.66	797.55	855.55
32768	605.72	797.48	854.41
65536	782.83	941.59	1048.07

Table A.15: Time in milliseconds for an agent to jump to a remote machine and then jump back with a dummy result.

Appendix B

Performance data - Migration versus client/server

Tables B.1 through B.8 contain the data from the experiments that were done to compare mobile agents with traditional client/server computing. Each data point is an average of between 4 and 23 measured times, where each measured time was obtained by timing 100 or 1000 iterations of the event in question and then dividing by 10, 100 or 1000.[1] More iterations and trials were done for those experiments in which each iteration took a short time; fewer iterations and trials were done for those experiments in which each iteration took a longer time. Figure B.1 shows the distribution of the standard deviations for the data points, where each deviation is expressed as a percentage of the corresponding average. The data and the experiments are discussed further in Chapter 7.

[1]Between 8 and 46 times were obtained for each data point. All times above the median were thrown out, and the remaining 4 to 23 times were averaged. The intent was to keep only those times that correspond to periods of light network load.

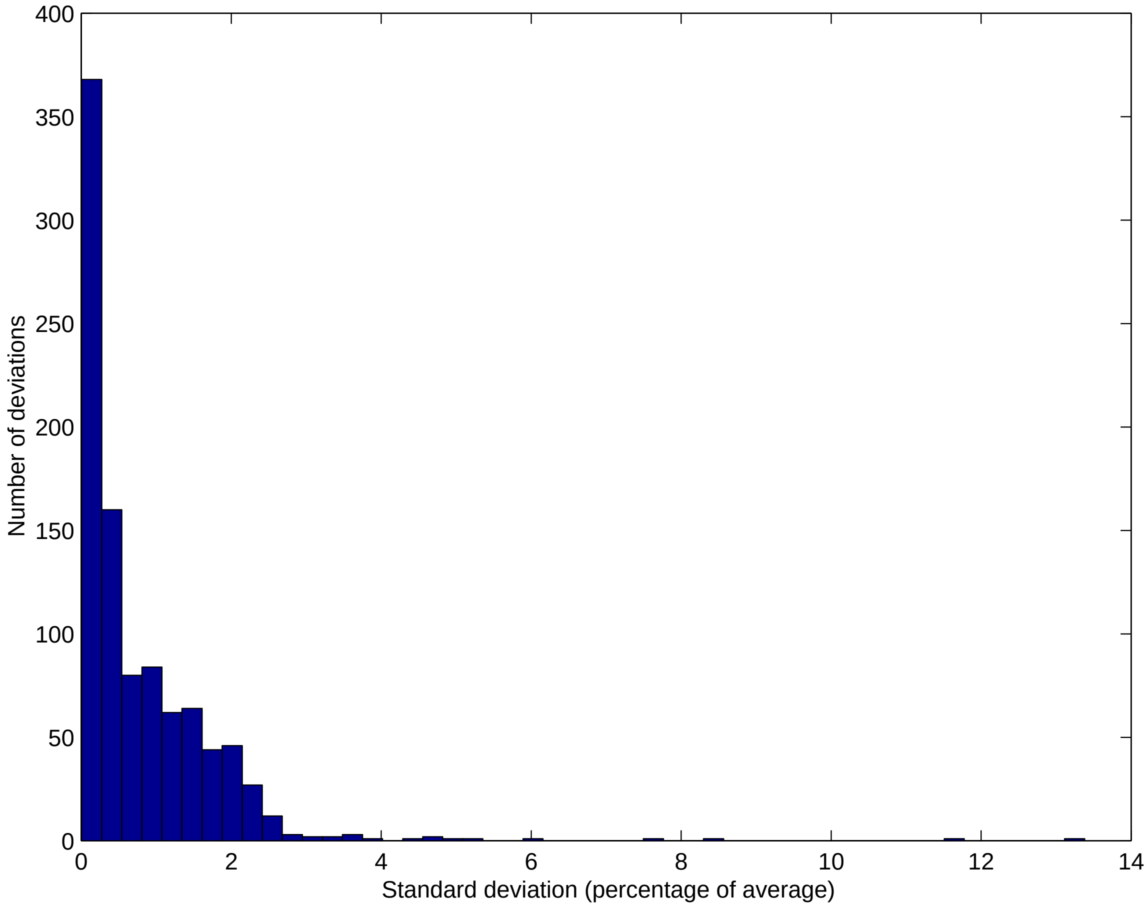


Figure B.1: Histogram of the standard deviations for the average times in Tables B.1 through B.8. Each standard deviation is expressed as a percentage of the corresponding average.

Response size (bytes)

Request size (bytes)	64	128	256	512
64	1.61	1.66	1.73	1.89
128	1.66	1.70	1.77	1.93
256	1.73	1.76	1.83	1.99
512	1.87	1.91	1.98	2.12
1024	2.18	2.22	2.29	2.41
2048	2.74	2.78	2.85	2.96
4096	3.83	3.89	3.97	4.11
8192	6.04	6.09	6.17	6.32
16384	10.77	10.81	10.88	11.04
32768	20.02	20.07	20.13	20.28
65536	45.74	45.81	45.91	46.07

Request size (bytes)	1024	2048	4196	8192
64	2.18	2.75	3.84	6.11
128	2.23	2.80	3.88	6.15
256	2.29	2.87	3.96	6.22
512	2.43	3.01	4.11	6.39
1024	2.71	3.30	4.41	6.67
2048	3.27	3.81	4.95	7.22
4096	4.39	4.96	5.94	8.34
8192	6.62	7.18	8.26	10.51
16384	11.32	11.92	13.00	15.23
32768	20.58	21.18	22.25	24.55
65536	46.31	46.97	48.19	50.55

Request size (bytes)	16384	32768	65536
64	10.83	20.02	45.99
128	10.87	20.04	46.06
256	10.95	20.22	46.11
512	11.10	20.31	46.25
1024	11.37	20.58	46.60
2048	11.97	21.17	47.06
4096	13.04	22.25	48.44
8192	15.29	24.59	50.73
16384	19.95	29.38	55.49
32768	29.24	38.33	63.54
65536	55.19	64.36	87.46

Table B.1: Time in milliseconds for two agents on the same laptop to exchange a request and response over an Agent Tcl meeting.

Response size (bytes)

		64	128	256	512
Request size (bytes)	64	6.09	6.19	6.31	6.65
	128	6.20	6.25	6.38	6.70
	256	6.34	6.42	6.53	6.84
	512	6.68	6.75	6.86	7.15
	1024	7.28	7.35	7.44	7.77
	2048	8.35	8.42	8.58	8.90
	4096	10.61	10.67	10.81	11.14
	8192	15.18	15.25	15.41	15.72
	16384	24.33	24.42	24.55	24.90
	32768	42.93	42.97	43.11	43.40
	65536	94.16	94.20	94.37	94.77

		1024	2048	4196	8192
Request size (bytes)	64	7.24	8.35	10.58	15.23
	128	7.32	8.46	10.68	15.32
	256	7.47	8.58	10.82	15.47
	512	7.73	8.89	11.14	15.78
	1024	8.33	9.48	11.60	16.35
	2048	9.42	10.52	12.79	17.43
	4096	11.69	12.79	14.93	19.70
	8192	16.31	17.40	19.60	24.06
	16384	25.48	26.61	28.78	33.40
	32768	44.01	45.09	47.34	51.96
	65536	95.41	96.61	99.07	103.69

		16384	32768	65536
Request size (bytes)	64	24.44	42.95	93.07
	128	24.54	43.08	93.16
	256	24.67	43.17	93.34
	512	24.95	43.43	93.64
	1024	25.54	44.03	94.27
	2048	26.62	45.14	95.51
	4096	28.89	47.45	97.87
	8192	33.45	52.05	102.59
	16384	42.49	63.45	111.80
	32768	61.13	82.77	129.98
	65536	112.78	131.60	178.04

Table B.2: Time in milliseconds for two agents on the same laptop to exchange a request and response using the `send` and `receive` primitives.

Response size (bytes)

Request size (bytes)		64	128	256	512
	64	152.83	160.34	277.01	359.67
	128	166.67	168.34	288.67	369.34
	256	187.51	196.00	303.17	395.00
	512	268.01	279.17	400.34	476.34
	1024	390.84	400.34	517.51	600.17
	2048	691.00	719.83	830.34	897.34
	4196	1117.17	1124.17	1239.33	1322.00
	8192	1820.00	1724.17	1840.00	1930.00
	16384	3151.34	3170.51	3276.00	3361.51
	32768	6679.01	6313.00	6439.84	6502.17
	65536	12008.50	12067.84	12165.17	12218.67

Request size (bytes)		1024	2048	4196	8192
	64	431.51	566.34	950.18	1647.67
	128	442.34	572.17	958.67	1665.18
	256	465.67	601.00	985.34	1684.17
	512	549.18	674.01	1051.84	1750.17
	1024	672.17	787.17	1167.51	1872.01
	2048	976.17	1097.67	1478.17	2189.51
	4196	1397.84	1530.50	1889.68	2594.17
	8192	1996.35	2123.17	2503.51	3200.51
	16384	3429.68	3570.38	3931.84	4650.01
	32768	6539.17	6692.51	7079.51	7798.51
	65536	12292.01	12436.34	12800.34	13493.51

Request size (bytes)		16384	32768	65536
	64	3081.34	5957.01	11735.85
	128	3094.18	5975.01	11738.01
	256	3115.34	6002.17	11759.52
	512	3187.68	6068.18	11831.18
	1024	3295.18	6174.52	11942.18
	2048	3619.17	6491.00	12254.85
	4196	4029.67	6879.41	12674.35
	8192	4639.84	7496.84	13264.17
	16384	6069.85	9124.51	14702.49
	32768	9175.68	12076.68	17943.85
	65536	14945.67	17802.01	23629.68

Table B.3: Time in milliseconds for two processes on different laptops to exchange a request and response over a 28.8 Kb/s modem link.

Final size (bytes)

Initial size (bytes)		64	128	256	512
	64	1521.50	1556.00	1545.50	1591.67
	128	1560.50	1571.50	1578.00	1616.50
	256	1550.33	1595.50	1599.00	1616.83
	512	1611.67	1659.50	1657.00	1702.50
	1024	1708.50	1747.50	1751.50	1799.50
	2048	1926.83	1948.50	1958.33	1985.33
	4096	2319.17	2328.83	2367.17	2394.17
	8192	3087.17	3115.67	3129.33	3169.50
	16384	4584.17	4620.17	4647.83	4662.67
	32768	7462.00	7501.20	7465.20	7533.80
	65536	13382.80	13399.00	13438.60	13435.40

Initial size (bytes)		1024	2048	4196	8192
	64	1747.00	1916.83	2284.00	3039.50
	128	1752.00	1920.67	2286.17	3048.50
	256	1760.67	1934.17	2303.67	3049.00
	512	1813.17	2003.67	2408.00	3126.00
	1024	1910.50	2100.00	2488.17	3227.67
	2048	2106.00	2312.17	2711.00	3424.17
	4096	2529.00	2709.17	3098.67	3824.33
	8192	3287.50	3471.67	3829.17	4604.67
	16384	4797.83	4985.33	5387.50	6104.17
	32768	7631.60	7830.80	8292.60	9058.60
	65536	13545.60	13693.20	14066.60	14716.20

Initial size (bytes)		16384	32768	65536
	64	4563.17	7402.00	13280.33
	128	4577.17	7427.17	13322.17
	256	4599.00	7434.67	13350.83
	512	4648.17	7465.00	13378.50
	1024	4735.67	7561.67	13449.00
	2048	4924.00	7735.50	13664.17
	4096	5359.00	8160.33	13999.00
	8192	6086.33	9032.83	14704.33
	16384	7442.67	10499.40	16191.00
	32768	10492.40	13310.40	18963.60
	65536	16112.20	18947.00	24746.00

Table B.4: Time in milliseconds for an agent to jump from one laptop to another (and back) over a 28.8 Kb/s modem link.

Request size (bytes)	Response size (bytes): 64	128	256	512
64	4.57	5.02	5.95	7.80
128	5.03	5.48	6.40	8.24
256	5.94	6.38	7.32	9.15
512	7.80	8.25	9.16	10.94
1024	11.46	11.91	12.84	14.68
2048	38.38	38.83	39.39	41.23
4196	72.65	75.18	70.58	72.59
8192	100.75	103.43	98.05	99.78
16384	160.18	159.09	153.01	153.37
32768	263.06	269.82	245.23	246.88
65536	484.59	505.97	461.75	462.89

Request size (bytes)	1024	2048	4196	8192
64	11.45	18.67	33.76	61.92
128	11.90	19.11	67.59	62.37
256	12.82	20.05	35.09	63.45
512	14.66	21.88	37.02	65.27
1024	18.33	25.53	40.64	68.85
2048	44.87	52.09	75.10	95.33
4196	76.79	84.10	104.19	126.90
8192	103.59	110.48	131.57	154.88
16384	156.34	157.24	180.92	207.67
32768	250.44	257.56	272.84	300.77
65536	466.53	473.60	491.67	517.13

Request size (bytes)	16384	32768	65536
64	119.63	234.35	459.58
128	120.08	234.80	460.04
256	121.00	235.73	460.95
512	122.84	237.55	462.80
1024	126.49	241.23	466.44
2048	153.14	268.12	493.31
4196	185.03	300.55	529.73
8192	211.33	328.32	555.15
16384	266.11	382.78	612.49
32768	358.51	494.06	723.30
65536	574.51	694.40	942.16

Table B.5: Time in milliseconds for two processes on different laptops to exchange a request and response over a 2.4 Mb/s wireless link.

Final size (bytes)

		64	128	256	512
	64	358.08	361.92	364.54	364.00
	128	362.00	362.62	365.00	370.69
	256	363.54	366.23	363.08	365.85
	512	367.54	365.54	368.54	370.54
Initial	1024	370.77	370.69	372.62	371.00
size	2048	381.69	381.46	381.23	384.69
(bytes)	4096	399.92	397.62	401.54	402.77
	8192	435.23	434.62	436.69	438.46
	16384	507.00	505.08	507.69	506.62
	32768	662.31	652.77	653.77	662.08
	65536	954.15	958.85	959.23	959.08

		1024	2048	4196	8192
	64	378.08	377.92	400.00	436.31
	128	369.15	384.23	397.46	433.85
	256	374.54	383.31	401.00	434.08
	512	373.77	384.38	403.00	433.77
Initial	1024	380.15	385.38	407.31	446.23
size	2048	388.85	400.15	416.00	454.08
(bytes)	4096	406.92	417.69	445.46	469.62
	8192	444.15	455.77	473.77	508.00
	16384	519.85	521.62	548.08	576.77
	32768	662.54	674.62	693.15	734.54
	65536	962.92	980.00	994.62	1028.69

		16384	32768	65536
	64	505.54	652.77	958.31
	128	506.92	658.54	946.85
	256	508.08	659.00	956.54
	512	511.46	667.31	960.08
Initial	1024	515.46	665.54	962.62
size	2048	525.00	674.69	974.23
(bytes)	4096	549.77	693.15	996.92
	8192	582.54	730.08	1028.85
	16384	652.15	810.62	1103.92
	32768	804.85	961.69	1244.54
	65536	1097.54	1246.69	1553.77

Table B.6: Time in milliseconds for an agent to jump from one laptop to another (and back) over a 2.4 Mb/s wirless link.

Response size (bytes)

		64	128	256	512
	64	0.72	0.83	1.05	1.48
	128	0.83	0.94	1.15	1.59
	256	1.05	1.15	1.37	1.80
	512	1.48	1.59	1.80	2.23
Request	1024	2.36	2.46	2.68	3.11
size	2048	3.43	3.54	3.75	4.19
(bytes)	4196	5.35	5.45	5.67	6.10
	8192	8.62	8.73	8.95	9.38
	16384	16.22	16.39	16.60	16.96
	32768	51.12	51.91	50.18	42.19
	65536	70.48	71.45	70.69	69.29

		1024	2048	4196	8192
	64	2.35	3.22	5.11	8.60
	128	2.46	3.33	5.21	8.70
	256	2.68	3.55	5.42	8.93
	512	3.11	3.98	5.86	9.36
Request	1024	3.97	4.84	6.71	10.20
size	2048	5.08	5.94	7.80	11.29
(bytes)	4196	6.97	7.81	9.69	13.15
	8192	10.25	11.07	12.92	16.42
	16384	17.87	18.68	20.33	24.06
	32768	44.27	44.09	42.76	47.31
	65536	69.44	71.14	72.45	77.43

		16384	32768	65536
	64	16.17	51.59	70.63
	128	16.32	51.54	71.74
	256	16.42	51.03	72.66
	512	16.89	50.39	72.41
Request	1024	17.71	43.36	73.85
size	2048	18.70	46.08	75.96
(bytes)	4196	20.60	45.02	74.91
	8192	23.88	49.23	78.63
	16384	31.50	55.60	87.76
	32768	54.28	81.61	113.48
	65536	84.16	110.71	140.13

Table B.7: Time in milliseconds for two processes on different laptops to exchange a request and response over a 10 Mb/s Ethernet link.

Final size (bytes)

Initial size (bytes)	64	128	256	512
64	335.60	336.50	337.80	335.40
128	337.30	335.40	335.50	338.80
256	337.40	334.70	340.90	338.80
512	337.90	336.00	335.40	339.70
1024	339.80	338.10	343.80	342.00
2048	339.20	341.60	339.50	347.20
4096	350.30	346.60	351.20	351.30
8192	368.50	370.30	364.10	371.30
16384	400.00	396.20	395.90	399.80
32768	457.40	458.00	457.90	462.50
65536	562.22	562.33	565.56	563.67

Initial size (bytes)	1024	2048	4196	8192
64	342.30	339.90	350.10	366.50
128	336.50	344.20	351.20	366.10
256	338.50	342.50	348.10	369.80
512	340.00	345.10	353.40	369.40
1024	341.10	348.10	352.50	373.60
2048	346.40	348.70	360.60	378.00
4096	355.30	355.00	362.60	382.10
8192	368.70	377.80	383.70	398.80
16384	403.20	403.60	413.10	428.30
32768	461.60	468.50	469.80	490.20
65536	563.78	572.33	573.78	597.67

Initial size (bytes)	16384	32768	65536
64	397.30	457.50	564.50
128	399.60	459.60	567.30
256	399.70	459.20	565.60
512	401.10	458.60	566.00
1024	399.70	463.80	568.90
2048	405.90	465.70	570.10
4096	410.60	471.40	580.20
8192	428.20	492.40	598.20
16384	461.40	521.80	626.80
32768	519.00	589.30	691.00
65536	620.56	686.00	802.44

Table B.8: Time in milliseconds for an agent to jump from one laptop to another (and back) over a 10 Mb/s Ethernet link.

Appendix C

A tutorial on Agent Tcl

This appendix is a tutorial on how to write Tcl agents for Agent Tcl. First, we give a brief overview of the Tcl scripting language and the special agent commands. Then we write two versions of the "who" agent.

C.1 Tcl and Tcl agents

Tcl has two components. The first component is a shell, usually called `tclsh`, that is used to execute stand-alone Tcl scripts and interactive commands. The second component is a library of C functions. The library provides functions to "create" a Tcl interpreter, define new Tcl commands in the interpreter, and submit Tcl scripts to the interpreter for evaluation. This library allows Tcl to be *embedded* inside a larger application; any application that needs a scripting language can include the library and allow its users to write Tcl scripts.

A tutorial on Tcl is beyond the scope of this document. Tcl is easy to learn, however, and is similar to other scripting languages such as Perl and the various Unix shells. The following Tcl script, for example, asks the user for a number and then displays the factorial of that number. The script keeps asking for numbers until the user enters `Q` to stop. For now, we simply examine the key features of the script; we describe how to actually run the script in the next section.

```
    # Procedure ``factorial'' recursively computes a factorial.

proc factorial x {

    if {$x <= 1} {
        return 1
    }

    return [expr $x * [factorial [expr $x - 1]]]
}

    # Repeat until the user enters "Q" to quit.

set number ""

while {$number != "q"} {

        # Get the integer for which we want the factorial
        # (or "Q" to quit).

    puts -nonewline \
      "Enter a nonnegative integer (or \"Q\" to quit): "
    gets stdin number

        # Convert to lowercase in case it's a "Q".

    set number [string tolower $number]

        # Compute the factorial if we're not quitting.

    if {$number != "q"} {
```

```
        puts "$number! is equal to [factorial $number]"
    }
}
```

There are several important things to note about Tcl in general. First, Tcl stores all data as strings. The `number` variable, for example, can be used to hold both a number and the letter `Q` because Tcl stores numbers as strings. Commands that expect numbers, such as `expr` (which evaluates general mathematical expressions), convert the given strings into an internal numeric representation.

Second, Tcl has no fixed grammar that "defines" the language [Ous94]. The Tcl interpreter does not treat the `while` construct above, for example, as a reserved word, followed by an expression, followed by a repeatedly-executed subprogram. Instead the Tcl interpreter treats the construct as a command name, `while`, followed by two argument strings; the curly bracket characters, `{` and `}`, represent nothing more than a kind of string *quotation*. The two arguments are passed to the handler for the `while` command which interprets them as it sees fit. The standard `while` handler does, in fact, treat the first argument as an expression, and if the expression is true, passes the second argument back to the Tcl interpreter for evaluation as a Tcl script. If the `while` handler is replaced, however, the behavior of the `while` command changes. Thus, although many Tcl commands look and act like traditional programming constructs, it is important to remember that Tcl parses everything as a command name and arguments.

Finally, there are two types of special syntactic constructs that can appear inside the argument strings. These constructs are called *substitutions*. In the command `expr $x * [factorial [expr $x - 1]]`, for example, `$x` is a variable substitution, and `[expr $x - 1]` is a command substitution. When the command is parsed, `$x` will be replaced with the *contents* of variable `x`, and `[expr $x - 1]` will be replaced with the *result* of executing the command `expr $x - 1`, namely the value of `$x - 1`. The quotation characters around the string determine whether these substitutions are actually performed. Curly brackets, for example, mean that substitutions are *not* performed and that the string is passed unchanged to the command handler. Double quotes (") or no quotes means that substitutions are performed. In the `while` command, above, we use curly brackets around the first argument, `$number != "q"`, so that the string is passed unchanged to the `while` handler. The variable substitution `$number` is then performed once per iteration, each time that the `while` handler checks the value of the expression. If we had used double quotes instead, the variable substitution would have been performed when the `while` command was first parsed, and the string passed to the `while` handler would have been `"" != "q"`. This expression is always true so the loop would have run forever. Proper quoting is the most difficult aspect of Tcl; it will be easier if you remember that the Tcl interpreter parses everything as a string, and that the different quotation characters affect the parsing process.

Keeping these three points in mind, it becomes straightforward to understand the script. First, the `proc` command is used to create a new command called `factorial` that takes a single argument `x` and computes `x!` by making recursive calls to itself. Then, the `puts` and `gets` commands are used to interact with the user and obtain a number; the `factorial` command is called with this number as its argument; and `puts` is used to display the factorial result. The `while` command repeats this process until the user enters `Q` rather than a number. This script highlights the main features of Tcl but uses only a small fraction of the Tcl commands. More information on Tcl can be found in the books by Ousterhout [Ous94] and Welch [Wel95], in the Tcl `man` pages, and in the `comp.lang.tcl` usenet group.

In addition to the standard Tcl commands, Agent Tcl agents use a special set of commands to migrate from machine to machine and to communicate with other agents. These commands are provided as a Tcl extension, but can be treated as a native part of the Tcl language when writing an agent. In the remainder of this section, we briefly define each command. In the next section, we use the commands to develop two agents. The commands can be divided into three main categories. The first category of commands allow an agent to register itself with an agent server and to obtain an identifier in the agent namespace.

- `agent_begin [`*`machine`*`]`. The `agent_begin` command registers the agent with the agent server on the specified machine (or on the local machine if no machine is specified) and returns the agent's new identifier within the agent namespace. In the current system, this identifier consists of the symbolic name of the server, the IP address of the server, a symbolic name that the agent chooses for itself, and a unique integer that the server assigns to the agent. So if an agent issues the command `agent_begin bald`, for example, the command might return the four-element Tcl list `bald.cs.dartmouth.edu 129.170.192.98 {} 15`. The `129.170.192.98` is the IP address of `bald`. The empty curly brackets indicate that the agent initially has no symbolic name; a symbolic name can be chosen at a later time with the `agent_name` command. The `15` is the integer id that the server on `bald` has assigned to the new agent; this integer is unique among all agents executing on `bald` but not among all agents everywhere. The agent's current identifier is stored in element `local` of the global Tcl array `agent`. This array is always available inside an Agent Tcl script and is read-only; it contains other useful information as we will see in the programming examples below. Once the agent has issued the `agent_begin` command, it can use the other agent commands.

- `agent_name` *name*. The `agent_name` command selects a symbolic name for the agent. If the agent in the example above issues the command `agent_name`

`FtpAgent`, its complete name will become

`bald.cs.dartmouth.edu 129.170.192.98 FtpAgent 15`.

- `agent_end`. An agent calls the *agent_end* command when it is finished with its task and no longer requires agent services.

The second category of commands allow an agent to migrate from machine to machine and to create child agents.

- `agent_jump` *`machine`*. An agent calls the `agent_jump` command when it wants to migrate to a new machine. This command captures the internal state of the agent and sends the state to the agent server on the specified machine. The server restores the state and the agent continues execution immediately after the `agent_jump`. Certain components of the state, such as open files and child processes, are intrinsically tied to a specific machine and are not transferred to the new machine. The agent receives a new 4-element identification when it jumps, which again is stored in element `local` of the global Tcl array `agent`. The agent also loses its symbolic name when it jumps and must request it again if needed.

- `agent_fork` *`machine`*. The `agent_fork` command is roughly analogous to Unix `fork`. It creates a copy of the agent on the specified machine. Both the original agent and the copy continue execution from the point of the `agent_fork`. The `agent_fork` command returns the 4-element identification of the copy to the original agent and the string `CHILD` to the copy.

- `agent_submit` *`machine`* `-procs` *`names`* `-vars` *`names`* `-script` *`script`*. The `agent_submit` command creates a completely new agent. The parameters to `agent_submit` are a machine, a list of Tcl variables, a list of Tcl procedures, and a startup script. A new agent is created on the specified machine. This agent contains copies of the specified variables and procedures and begins execution by evaluating the startup script. The `agent_submit` command returns the 4-element identification of the new agent.

The final category of commands allow agents to communicate with each other.

- `agent_send` *`id code string`*. The `agent_send` command sends a message to another agent. A message consists of an integer *code* and an arbitrary *string*. The recipient agent is specified by its 4-element *id* or by any subset of the 4-element id that uniquely identifies the agent, such as the server name and the unique integer. The recipient receives the message using the `agent_receive` command, or if it is using Tk, by establishing an event handler for incoming messages using the `mask` command.

- `agent_event` *`id tag string`*. The `agent_event` command is a variant of `agent_send` that sends a *tag* and a *string* rather than an integer *code* and a *string*. A tag is just an arbitrary string itself. The advantage of `agent_event` is that the recipient can associate event handlers with specific tags using the `mask` command. The event handler is called automatically whenever a message arrives with the corresponding tag. If the recipient is not using Tk or chooses not to use event handlers, it must receive these tagged messages with the `agent_getevent` command.

- `agent_meet` *`id`*. The `agent_meet` command is used to request a meeting with the specified recipient. The recipient accepts the connection request either by issuing its own `agent_meet` command or with the `agent_accept` command. Once the connection request has been accepted, and the meeting has been established, arbitrary data can be sent along the connection using the `tcpip_read` and `tcpip_write` commands. The names of these commands reflect the current link between direct connections and TCP/IP; they should be changed but have been left alone for backward compatibility. Meetings are more efficient than the two message-passing variants since they bypass the agent servers.

There are several miscellaneous commands that do not fall into the three main categories. The `agent_info` command, for example, is used to obtain information from a server about the agents executing on its machine; the `retry` command retries a block of Tcl code until no error occurs or the maximum number of tries has been reached; and the `restrict` command imposes resource restrictions on an arbitrary block of Tcl code. The Agent Tcl documentation describes these commands, along with all of the commands listed above, in more detail.

C.2 Programming examples

The Unix `who` command lists all the users who are logged into a machine. In this section, we develop two versions of an agent that will travel from machine to machine, execute the Unix `who` command on each machine, and then return to the home site and show the complete list of users to its owner. These examples are a simplistic use of an agent, but they illustrate the general structure of itinerant agents, they do not require support agents at each network site, and they fit conveniently on a few pages while demonstrating most of the agent commands. As you work through these examples, you should keep in mind that the application-specific section of each agent—i.e., the invocation of the Unix `who` command—can be replaced with any desired processing.

The first step in developing the examples is to install the Agent Tcl system on two or more machines (the examples work with only one machine but are somewhat boring). Detailed installation instructions are included in the Agent Tcl documentation. Once the Agent Tcl system is installed, you will have three executable files, `agentd`, `agent` and `agent-tk`. `agentd` is the agent server, `agent` is the agent interpreter, and `agent-tk` is the agent interpreter that includes the Tk toolkit. You should start the server `agentd` on each machine on which you installed the Agent Tcl system. Again detailed instructions are included in the Agent Tcl documentation.

Once the server is running on each machine, you can execute Agent Tcl agents or any Tcl script that is fully compatible with Tcl 7.4 and Tk 4.0. Tcl scripts that require Tcl 7.5 and Tk 4.1 will not work with this version of Agent Tcl. There are three ways to execute a Tcl script using the agent interpreters. Suppose that the factorial script above is in a file called `factorial.tcl`. The first execution method is to start the agent interpreter by typing `agent` at the Unix prompt. Then you type `source factorial.tcl` at the Tcl prompt. You will return to the Tcl prompt after the factorial script finishes executing; you can type in additional Tcl commands or type `exit` to leave the agent interpreter and return to the Unix prompt. The second execution method is to type `agent factorial.tcl` at the Unix prompt; you will return to the Unix prompt when the factorial script has finished executing. The third execution method is to turn on the Unix execution permissions for file `factorial.tcl` and add the line

```
#!/usr/local/bin/agent
```

at the beginning of `factorial.tcl`. This assumes that the `agent` interpreter is in directory `/usr/local/bin`; you will need to change this line if you installed `agent` is in a different directory. Then you simply type `factorial.tcl` at the Unix prompt; you will return to the Unix prompt once the factorial script finishes executing. If the agent uses Tk, you use the same three execution methods, only with `agent-tk` rather than `agent`. Since the Agent Tcl system uses a modified Tcl interpreter, you must execute agents with either `agent` or `agent-tk`. It is *impossible* to execute an agent with the standard Tcl interpreters, `tclsh` and `wish`, even if you recompile them so that they include the agent extension.

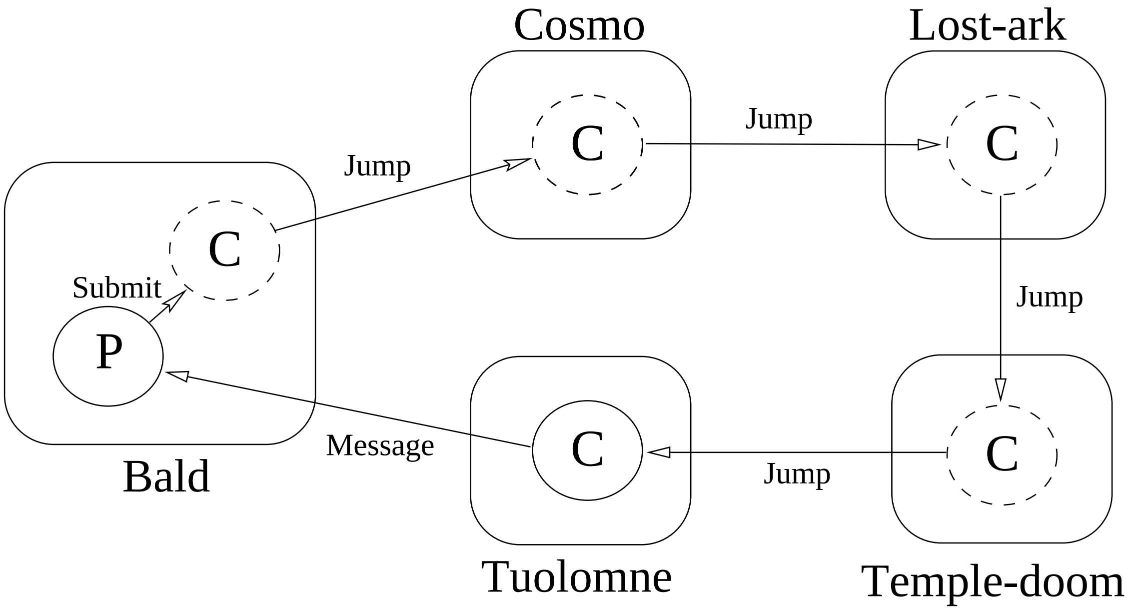


Figure C.1: The first version of the "who" agent. The parent agent (P) submits a child agent (C) that migrates through a sequence of machines and executes the Unix `who` command on each. Then the child (C) sends the complete list of users to the parent (P) for display to the user. In the specific case shown, the child agent (C) migrates through four machines at Dartmouth, `cosmo`, `lost-ark`, `temple-doom`, and `tuolomne`.

Now we develop the two versions of the "who" agent. The first version is text-based. It asks the user for a list of machines. Then it submits a single child agent using the `agent_submit` command. This child agent migrates through the specified machines using the `agent_jump` command, executes the Unix `who` command on each machine, and records the users of each machine. Once the child agent finishes, it sends the complete list of users to its parent using the `agent_send` command. The parent displays the list of users and exits. Figure C.1 illustrates the behavior of this agent.

The Tcl code for this agent is actually quite simple. You can enter the code using any standard Unix text editor. Once you have entered the code, you should save it in a file with extension `.tcl`. The discussion below assumes that you use the filename `who.tcl`. The Tcl code for the agent appears below. The code is interspersed with discussion. The code is indented and appears in a fixed-width font; the discussion is flush with the left margin and appears in the normal font. Make sure that you do not type in the discussion as part of the agent. In addition, certain lines end with a backslash (\) which is the Tcl line-continuation character. There should not be any spaces or tabs after these backslashes. The first piece of code is simply a comment header.

```
#!/usr/local/bin/agent
#
# who.tcl
#
# This agent executes the "who" command on multiple machines.
# It submits a SINGLE child agent.  The child jumps from
# machine to machine and executes the WHO command on each
# machine.  Then the child returns the complete list of users
# to the parent for display.
```

The first line specifies the location of the `agent` interpreter. This line allows you to execute the agent simply by typing `who.tcl` at the Unix prompt. You will have to change this line if you installed `agent` in a different directory. The other lines are comments which are indicated by a pound sign (`#`).

The second piece of code is the procedure that implements the *child agent*.

```
  # Procedure ‘who’ is the child agent that does the jumping.

proc who machines {

  global agent
```

```
    # start with an empty list

  set list ""

    # loop through the machines and jump to each

  foreach m $machines {

      # if we do not jump successfully, append an error message

      # otherwise append the list of users

    if {[catch {agent_jump $m} result]} {

      append list "$m:\nunable to JUMP here ($result)\n\n"

    } else {

      set users [exec who]

      append list "$agent(local-server):\n$users\n\n"

    }

  }

    # send back the list of users and finish

  agent_send $agent(root) 0 $list

  exit

}
```

There are several important things to note about this procedure. First, the procedure takes a single argument `machines` which contains the list of machines that the child agent should visit. For the purposes of the examples, a Tcl list is just a string that contains one or more whitespace-separated substrings—e.g., the string `bald cosmo lost-ark` is a Tcl list that contains three elements, `bald`, `cosmo` and `lost-ark`. Second, the command `global agent` tells the Tcl interpreter that we want to access the global array `agent` from inside the procedure; this array contains information about the location of the agent. Third, the `foreach` command loops through each element in the list of machines; the variable `m` is set to the next machine on each iteration. Fourth, the `agent_jump` command is used to jump onto each machine `m`. The `agent_jump` command is enclosed within a `catch` command. Tcl commands raise *exceptions* if an error occurs; these exceptions are caught with the `catch` command. If the `agent_jump` command fails, the `catch` command catches the exception, puts the associated error message in the variable `result`, and returns 1. The *if* clause of the `if` statement is executed and the agent records an error message. If `agent_jump` succeeds, the `catch` command returns 0. The *else* clause is executed so the agent invokes the Unix `who` command and records the list of users. Finally, once the child agent has migrated through each machine, it sends the list of users (and error messages) back to its parent using the `agent_send` command.

When agents create other agents, a parent-child hierarchy arises with a single agent at the top. The agent at the top is called the *root* agent and, in both itself and all of its descendents, its 4-element identification is found in element `root` of the `agent` array. Thus, since the parent of the child agent is also the root agent in this case, we can just send the list of users to `agent(root)`. A current limitation of the Agent Tcl system is that it does not record the complete parent-child hierarchy. If we wanted to send the message to the parent and the parent was not a root agent, we would have to explicitly record the 4-element identification of the parent in an auxiliary variable before creating the child agent.

The next piece of code is the start of the parent agent. It asks for the list of machines and registers the agent with the agent server.

```
# get the machines
```

```
puts -nonewline "Please enter the list of machines: "
gets stdin machines

  # register the agent

if {[catch {agent_begin} result]} {
  return -code error "ERROR: unable to register on \
    $agent(actual-server) ($result)"
}
```

The `gets` and `puts` commands let the user enter the list of machines. The `agent_begin` command registers the agent with the server on the local machine. The `agent_begin` command is enclosed within a `catch` command in case the server is not available on the local machine for some reason (element `actual-sever` of the `agent` array always contains the name of the local machine). The agent cannot use any of the other agent commands until it successfully registers using the `agent_begin` command.

The final piece of code is the rest of the parent agent. It creates the child agent, waits for the child agent to send the message containing the list of users, and finally displays the list of users.

```
  # catch any error

if {[catch {

    # submit the child agent that does the jumping

  agent_submit $agent(local-ip) -vars machines -procs who \
    -script {who $machines}
```

```
    # wait for the list of users

  agent_receive code message -blocking

    # output the list of users

  puts "\nWHO'S WHO on our computers\n\n$message"

    # cleanup

  agent_end

} error_message]} then {

    # cleanup on error

  agent_end

    # throw the error message up to the next level

  return -code error -errorcode $errorCode \
    -errorinfo $errorInfo error_message
}
```

First, the parent creates the child agent using `agent_submit`. The child agent is specified with the `-script` parameter and consists only of a call to procedure `who` with parameter `machines`. Since the child makes this call, it must have copies of procedure `who` and variable `machines`, so this procedure and variable are specified after the `-procs` and `-vars` parameters respectively. Once the child agent is created, the parent waits for the child's message using the `agent_receive` command. The `-blocking` parameter indicates that the agent will wait until the message arrives rather than timeout. Once the message arrives, the integer code is placed in variable `code` and the string is placed in variable `string`. Finally, the `puts` command displays the list of users and the `agent_end` command ends the agent. This whole sequence is enclosed in a `catch` command in case an error occurs. The agent is now complete and can be run with any of the three methods described above. So if you type `agent who.tcl` at the Unix prompt, you will see the request

```
Please enter the list of machines:
```

You should type in the desired machine names with one or more spaces between names. The agent server must be running on the specified machines. As an example, if the agent were executed at Dartmouth and you entered the same machine names shown in Figure C.1 (as well as one machine that does not exist), you might see the output

```
Please enter the list of machines:
cosmo lost-ark xxx temple-doom tioga

WHO'S WHO on our computers

cosmo.dartmouth.edu:


lost-ark.dartmouth.edu:
lwilson    ttyq0       Apr 29 08:16
pascalb    ttyq2       Apr 29 09:11
pascalb    ttyq3       Apr 29 09:11

xxx:
unable to JUMP here (unable to get IP address of "xxx")

temple-doom.dartmouth.edu:
rgray      ttyq0       Apr 29 08:55
```

```
rgray      ttyq2        Apr 29 09:08

tioga.cs.dartmouth.edu:
rgray       ttyp2       Apr 29 09:07
```

There will be a short delay before the child agent finishes its travels and the list of users is displayed. Note that the nonexistent machine `xxx` causes no difficulties due to the `catch` command surrounding the `agent_jump` command. Detecting and handling errors when the agent is moving is no more difficult than when the agent is stationary. Uncaught errors cause the agent to terminate, although an error message will be automatically sent to the *root* agent

The second version of the "who" agent expands on the first. First, it uses the Tk toolkit to display a window in which the user enters the names of the machines. Then, the agent itself jumps from machine to machine and executes the Unix `who` command on each machine. Once the agent has migrated through each machine, it jumps again to return to its home machine where it displays a second window that contains the results. As an additional feature, the agent leaves behind a tracker agent on the home machine; the agent communicates with the tracker agent to provide a continuous update of its current status and network location. This behavior is shown in Figure C.2. A sample screen dump is shown in Figure C.3. This agent is much longer so you will probably want to use the copy in `systems/agent-tcl/book-examples/winwho.tcl` rather than typing it in yourself. All of the code should be placed in one file although logically there are two agents (the "who" agent creates the "tracker" agent just before it starts to migrate). The first piece of the "who" agent is again a comment header. The only difference is that the first line must specify the location of the `agent-tk` interpreter rather than the `agent` interpreter.

```
#!/usr/contrib/bin/agent-tk
```

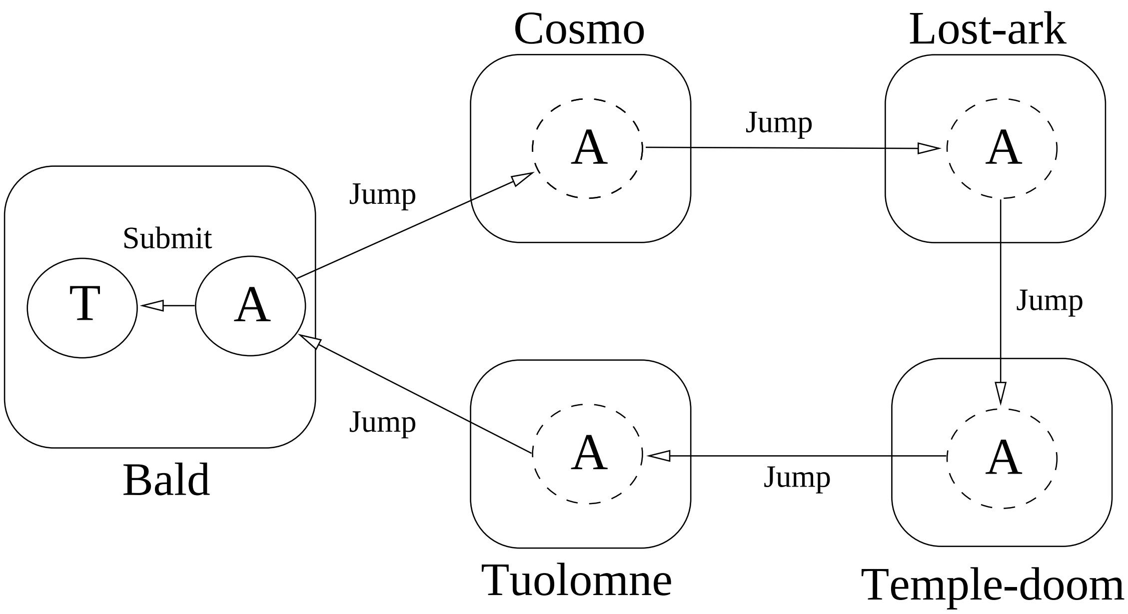


Figure C.2: The second version of the “who” agent. The agent (A) migrates through the machines itself, returns to the home machine, and displays the list of users in a Tk window. Before it begins migrating, the agent (A) creates a child agent that will serve as a tracker (T). The agent (A) communications with the tracker (T) as it migrates to provide a continuous update of its location.

```
#
# who.tk
#
# This agent executes the "who" command on multiple machines.
# It displays a Tk window in which the user enters a list of
# machines.  Then it jumps from machine to machine and executes
# the Unix "who" command on each machine.  Finally it returns
# to the home machine and displays a Tk window that contains
# the complete list of users.  While traveling, it leaves
# behind a tracker agent; it communicates with the tracker
# agent to display continuous information about its progress.
```

The second piece of the "who" agent are procedures `GetMachines` and `DisplayList`. Procedure `GetMachines` creates the window in which the user enters the machine names; this window is the top window in Figure C.3. Procedure `DisplayList` creates the output window in which the list of users is displayed; the output window is the bottom window in Figure C.3. Procedure `GetMachines` is called before the agent starts migrating; procedure `DisplayList` is called when the agent returns to the home machine with the list of users. These procedures use standard Tk commands and do not use any agent commands, so we do not describe them in detail. The only nonstandard commands are `main create` and `main destroy`, which create and destroy a main window for the application. The standard Tk interpreter, `wish`, automatically creates a main window. Agents, however, do not always need a main window so we introduce the command `main create` to explicitly create the main window when desired. In addition, an agent can not migrate if it is currently displaying a window. For this reason `main destroy` is used to destroy the main window before migration. Unlike `wish`, destroying the main window does not terminate the agent. Because of the need to destroy windows before migrating—and because agents cannot jump from inside a Tk event handler—Tk agents make heavy use of the `tkwait` command. The agent displays the desired interface, uses `tkwait` to stay in the event loop until the agent needs to migrate, and then destroys the interface and jumps to the next machine. This approach imposes a useful structure on the agent and is more convenient than it might seem.

```
    # Procedure GetMachines creates the Tk window in which the
    # user enters the list of machines.  It returns "OK" if the
    # user enters a list of machines and selects the "GO" button
    # It returns "FORGET" if the user selects the "FORGET" button.

  proc GetMachines {} {

      # The global variable "machines" holds the list of machines
      # and the global variable "status" is either "GO" or
      # "FORGET" depending on which button the user hits.  The
      # global variable "display" holds the name of the display
      # --- e.g., # "cosmo.dartmouth.edu:0".

    global display
    global machines
    global status

      # create the main window

    main create -name "List of machines" -display $display

      # fill in the main window with an entry box and two buttons

    entry .entry -width 40 -relief sunken -bd 2 \
      -textvariable machines
```

```
button .go -text "Go!" -command {set status GO}
button .forget -text "Forget it!" -command {set status FORGET}
pack .entry -side top -fill x -expand 1
pack .go -side left -padx 3m -pady 3m -expand 1
pack .forget -side left -padx 3m -pady 3m -expand 1
bind .entry <Return> {set status GO}
focus .entry

  # wait for the user to fill in the entry box correctly,
  # first making sure that the "status" variable does not yet
  # exist

catch {unset status}

while {![info exists status]} {

    # wait for the user to hit a button

  tkwait variable status

    # if the user hit button "GO", see if the entry box is
    # filled in

  if {($status == "GO") && ([string trim $machines] == "")} {
     tk_dialog .t "No machine!" \
       "You must enter at least one machine name!" error 0 OK
     unset status
  }
}

  # return the status --- e.g., "GO" or "FORGET" --- but first
```

```
    # destroy the window

  main destroy
  return $status
}

  # Procedure DisplayList creates the window in which the list
  # of users is displayed.  The "users" argument contains the
  # list of users.

proc DisplayList users {

    # The global variable "display" contains the name of the
    # display and the global variable "status" will be set to
    # DONE when the user finishes looking at the results.

  global display
  global status

    # create the main window

  main create -name "WHO'S WHERE?" -display $display

    # make the placeholder frames

  frame .top -relief raised -bd 1
  frame .bot -relief raised -bd 1
  pack .bot -side bottom -fill both
  pack .top -side bottom -fill both -expand 1

    # make a text box that will hold the list of users
```

```
    text .text -relief raised -bd 2 -width 60 \
      -yscrollcommand ".scroll set"
    scrollbar .scroll -command ".text yview"
    pack .scroll -in .top -side right -fill y
    pack .text -in .top -side left -fill both -expand 1

      # make the "DONE" button

    button .done -text "Done!" -command {set status DONE}
    pack .done -in .bot -side left -expand 1 -padx 3m -pady 2m

      # fill in the text area

    .text delete 1.0 end
    .text insert end $users

      # wait for the user to finish looking at the results, first
      # making sure that the "status" variable does not yet exist

    report "Done! You should see the results window."
    catch {unset status}
    tkwait variable status
    main destroy
  }
```

The next piece of the "who" agent is actually the tracker agent that displays the progress of the "who" agent through the network. The "who" agent uses the `agent_event` command to send tagged messages back to the tracker. Rather than explicitly receiving these messages with the `agent_getevent` command, the tracker uses the `mask` command to establish two message handlers. These handlers are automatically called when a tagged message arrives. Procedure `messageHandler` is automatically called if the message tag is `MESSAGE`. The `source` parameter is filled in with the 4-element identification of the sender; the `tag` parameter is filled in with the message tag; and the `string` parameter is filled in with the message string. Similarly procedure `errorHandler` is called if the message tag is `ERROR`. Procedure `Tracker` is the main body of the tracker agent. It creates a simple text window, establishes the two message handlers using the `mask` command, and calls `tkwait` to sit in the event loop. The two handlers are automatically called whenever a message arrives and simply insert the status information into the text window. This text window is the middle window in Figure C.3. The tracker agent illustrates that agents can use the Tk event model effectively. In fact Tk agents should almost always establish event handlers for incoming messages; otherwise the agent will not respond to user events while it sits at an `agent_receive` or `agent_getevent` command (or it will have to continuously poll). Procedure `LeaveTracker` actually starts up the tracker agent using `agent_submit`; it is called by the "who" agent just before the "who" agent starts migrating. The procedure returns the 4-element identification of the tracker so that the "who" agent knows where to send its status messages.

```
  # Procedure errorHandler, messageHandler and Tracker make up
  # the tracker agent.  Procedure LeaveTracker starts the
  # tracker agent and returns either the 4-element id of the
  # tracker or the string "FAILED".

proc messageHandler {source tag string} {
  .text insert end "$string\n"
}

proc errorHandler {source tag string} {
```

```
  .text insert end "\nERROR: $string\n\n"
  bell
}

proc Tracker {} {

    # The global variable "display" holds the name of the
    # display.   The global variable "status" will be set to
    # DONE when the user decides to exit.  The global array
    # "mask" --- which is available inside every agent ---
    # specifies event handlers.

  global display
  global status
  global mask

    # create the tracker window

  main create -name "Tracker agent" -display $display

   # make the placeholder frames

  frame .top -relief raised -bd 1
  frame .bot -relief raised -bd 1
  pack .bot -side bottom -fill both
  pack .top -side bottom -fill both -expand 1

    # make a text box that will hold the list of users

  text .text -relief raised -bd 2 -width 60 \
    -yscrollcommand ".scroll set"
```

```
  scrollbar .scroll -command ".text yview"
  pack .scroll -in .top -side right -fill y
  pack .text -in .top -side left -fill both -expand 1

    # make the "DONE" button

  button .done -text "Done!" -command {set status DONE}
  pack .done -in .bot -side left -expand 1 -padx 3m -pady 2m

    # turn on the event handlers

  mask add $mask(event) "ANY -tag MESSAGE \
    -handler messageHandler"
  mask add $mask(event) "ANY -tag ERROR -handler errorHandler"

    # wait for the user to finish looking at the results, first
    # making sure that the variable "status" does not yet exist

  catch {unset status}
  tkwait variable status
  main destroy
}

proc LeaveTracker {} {

  global agent
  global display

    # try to submit the tracker agent

  if {[catch {
```

```
    set tracker [
      agent_submit $agent(local-ip) -vars display \
        -procs errorHandler messageHandler Tracker \
        -script {Tracker; exit}
    ]
  } result]} {
    set tracker FAILED
  }
  return $tracker
}
```

The next piece of the "who" agent is procedure `who`, which routes the agent through the specified machines using `agent_jump` and executes the Unix `who` command on each. This procedure is almost the same as the `who` procedure from the first version. The only difference is that it reports its current location and status to the tracker agent by calling the `report` and `reportError procedures`. These two procedures use `agent_event` to send a tagged message back to the tracker. When the tracker receives the tagged message, either procedure `messageHandler` or procedure `errorHandler` is automatically called, and the status information is inserted into the tracker window.

```
  # Procedure who executes the Unix "who" command on each
  # machine.  Procedure report sends normal information back to
  # the tracker agent whereas Procedure reportError sends error
  # information back to the tracker agent.

proc report message {
```

```
    # The global variable "tracker" holds the 4-element id of
    # the tracker agent.

  global tracker

    # send the message, ignoring errors

  catch {
    agent_event $tracker MESSAGE $message
  }
}

proc reportError error {

    # The global variable "tracker" holds the 4-element id of
    # the tracker agent.

  global tracker

    # send the message, ignoring errors

  catch {
    agent_event $tracker ERROR $error
  }
}

proc who machines {

  global agent
  global tracker
```

```
    # start with an empty list

  set list ""

    # jump from machine to machine

  foreach m $machines {

      # if we do not jump successfully, append an error message
      # otherwise append the list of users

    if {[catch "agent_jump $m" result]} {
      reportError "Failed to jump to machine $m ($result)"
      append list \
        "$m:\nunable to JUMP to this machine ($result)\n\n"
    } else {
      report "Jumped to machine $agent(actual-server)"
      set users [exec who]
      append list "$agent(local-server):\n$users\n\n"
    }
  }

  return $list
}
```

The last piece of the "who" agent simply calls the procedures above. First, the "who" agent calls procedure `GetMachines` to get the machine names from the user; the machine names are stored in the global variable `machines`. Once the machine names have been obtained, the agent calls `agent_begin` to register the agent with the local agent server, and then calls procedure `LeaveTracker` to start up the tracker agent.

Then the "who" agent jumps through the specified machines by calling procedure `who`; procedure `who` returns the list of users. Once procedure `who` is finished, the agent calls `agent_jump` one more time to return home. Once the agent is home, it calls procedure `DisplayList` to show the list of users in an output window. Finally the agent calls `agent_end` and exits.

```
  # remember the display

if {![info exists env(DISPLAY)]} {
  set display ":0"
} else {
  set display $env(DISPLAY)
}

  # get the list of machines

if {[GetMachines] == "FORGET"} {
  exit
}

  # register the agent with an agent server and remember the
  # home machine

if {[catch {agent_begin} result]} {
  puts "Unable to register on $agent(actual-server) ($result)"
  exit
}

set home $agent(local-ip)

  # try to leave behind the tracker agent
```

```
set tracker [LeaveTracker]

if {$tracker == "FAILED"} {
  puts "Unable to leave behind the tracker agent!"
  exit
}

  # jump from machine to machine, executing the "who" command on
  # each machine, and then jump back home

set users [who $machines]
agent_jump $home

  # display the results

DisplayList $users

  # done

exit
```

The agent is now complete. It can be run with any of the three methods discussed above except that you must use `agent-tk` rather than `agent`. One important note is that, if you followed the installation instructions carefully (which is highly recommended), an agent will start running under a special userid as soon as it jumps for the first time. On most Unix machines, you will need to use the `xhost` command (or equivalent) to allow this special userid to create windows on your screen; otherwise the agent will not be able to create the output and tracker windows. The reference documentation for your Unix machine will have more details about screen access.

Once the agent starts executing, you will first see the entry form where you enter the names of the machines. Once you hit "GO!" to send the agent on its way, the entry form will disappear, and the tracker window will appear. Lines will appear in the tracker window one at a time as the "who" agent makes its ways through the network and reports back its current location. Finally the "who" agent will return and the output window will appear showing the list of users. A sample run is shown in Figure C.3; the machine names are the same as were used before.

Although these two versions of the "who" agent perform a simple task, they use most of the agent commands and can serve as building blocks for more complex agents. There is no reason for the agent to be self-contained, for example. There might be service agents on each machine with which the agent communicates as it migrates. These service agents should be given well-known names with the `agent_name` command so that client agents can communicate with them easily. In one of our information-retrieval applications, for example, there is an agent named `TechReports` on each machine which provides a low-level search interface to a collection of technical reports. Agents, migrating from collection to collection, combine the low-level search primitives into complex queries.

One area of difficulty for new agent programmers is debugging a moving agent. Agent Tcl includes a visual debugger called "agdb" that tracks an agent as it moves through the network, monitors its communication with other agents, and provides traditional debugger features such as breakpoints, watch conditions, and line-at-a-time execution.

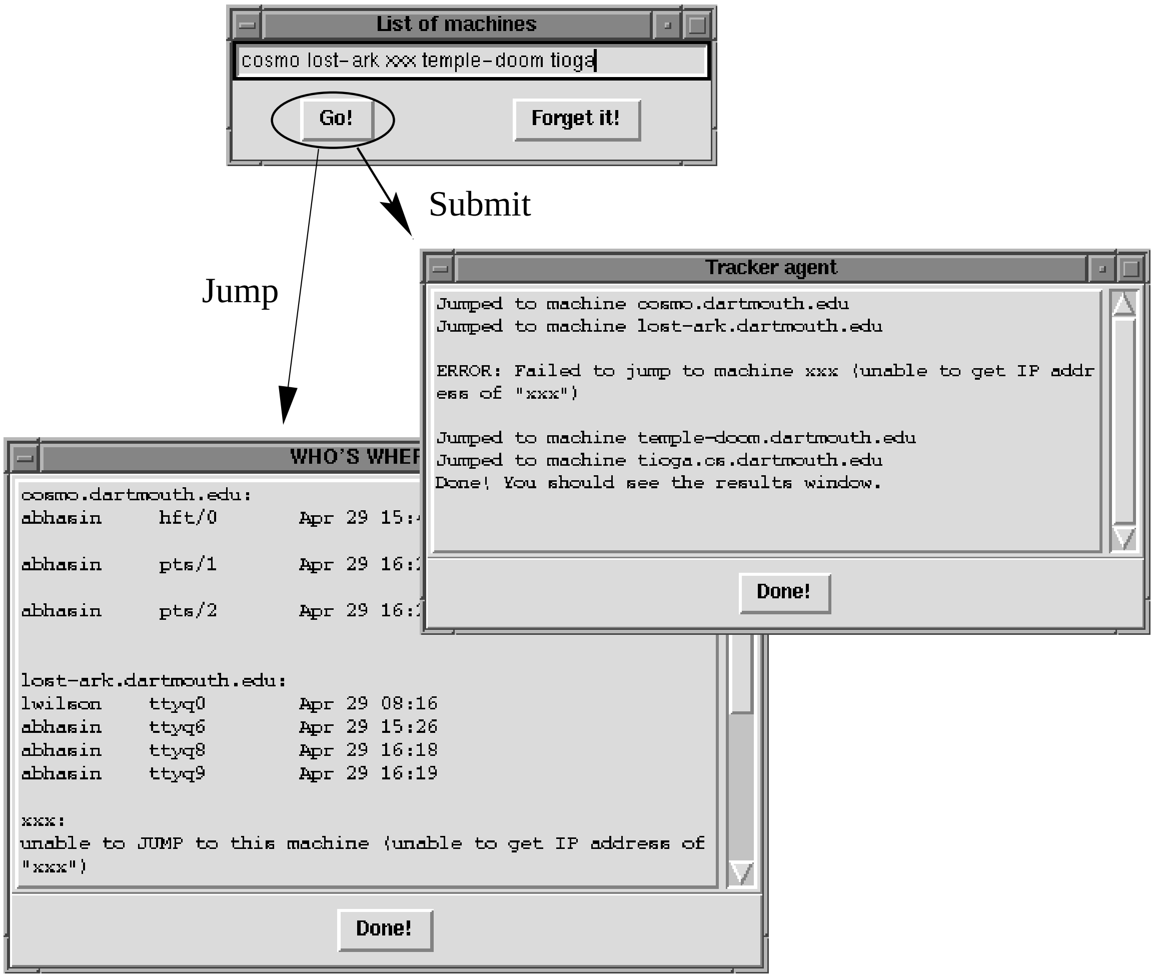


Figure C.3: A sample run of the second "who" agent. The first window that the user sees is the entry box at top where the machine names are entered. Once the machine names are entered, the agent uses `agent_submit` to create the tracker agent in the middle. Then the agent jumps from machine to machine, eventually returning to the starting machine and displaying the list of users at bottom. As the agent migrates, it communicates its position to the tracker agent; the text in the tracker window appears one line at a time.