Agent Tcl

Robert Gray*

George Cybenko

David Kotz

Daniela Rus

Department of Computer Science

Dartmouth College

Hanover, NH 03755

{rgray,gvc,dfk,rus}@cs.dartmouth.edu

May 29, 1996

1 Overview

Agent Tcl is a simple itinerant-agent system that runs on Unix workstations and allows the rapid development of complex agents [Gra95, Gra96]. Although Agent Tcl currently lacks the features of commercial systems such as Telescript [Whi94], it is an effective platform for experimentation with itinerant agents and for the development of small to medium-sized applications. Agent Tcl agents are written in an extended version of the Tool Command Language (Tcl). Tcl

*Supported by AFOSR contract F49620-93-1-0266 and ONR contract N00014-95-1-1204

is a high-level scripting language that is both powerful and easy to learn. It was designed to "control and extend [existing] applications [and tools]" [Ous94]. This makes Tcl an ideal language for itinerant agents because most agents are concerned primarily with coordinating high-level communication and resource access. Agent Tcl agents can use all of the standard Tcl commands as well as a set of special commands that are provided as a Tcl extension. These special commands allow an agent to migrate from one machine to another, to create child agents, to communicate with other agents, and to obtain information about its current network location. In addition, Agent Tcl, like all Tcl-based systems, can be extended with user-defined commands to create a more powerful agent system — e.g., a set of text-processing commands can be made available to all agents at a particular site.

Migration is accomplished with the `agent_jump` command, which can appear anywhere within an agent. It captures the current state of the agent and transfers this state image to a server on the destination machine. The server restores the state image and the agent continues execution from the command immediately after the `agent_jump`. In other words, `agent_jump` allows the agent to suspend its execution at an arbitrary point, transport to another machine, and resume execution on the new machine at the exact point at which it left off. This approach to migration is the same as in Telescript [Whi94], but is different than in Tacoma where an agent executes *from the beginning* on each new machine and must explicitly collect state information [JvRS95]. Once an Agent Tcl agent has migrated to a machine, it can access resources and communicate with other agents on that machine. Once it finishes its local task, it migrates to the next machine.

There are two forms of inter-agent communication. The first form of communication is message passing, which uses the traditional *send* and *receive* prim-

itives [SS94]. The `agent_send` command sends a message to another agent and the `agent_receive` command receives an incoming message. The second form of communication is a direct connection, which is essentially a named message stream. An agent establishes a direct connection with another agent using the `agent_meet` command. The two agents then exchange messages over the connection. Direct connections are more efficient than message passing for long interactions and are convenient for the programmer since the agent can wait for messages on a particular connection. A message in Agent Tcl is an arbitrary string with no predefined syntax or semantics. The agents must agree on the meaning of the messages that they exchange. The base communication mechanisms were made purposely low-level to allow experimentation with a range of communication paradigms. Two paradigms have already been implemented on top of the base facilities. The first is analogous to RPC (Remote Procedure Call) [NCK96]; the second is a conversational approach that views communication between a pair of agents as an *ongoing* dialog. An agent can participate in as many simultaneous dialogs as desired. Each dialog has its own state space; incoming messages from other agents are automatically turned into events that execute within the appropriate state space.

Other commands allow an agent to create new child agents and to obtain information about its current machine, such as the identities of other agents. In addition, agents can use the Tk toolkit to interact with the user of the current machine. Tk is a Tcl extension that provides commands for creating graphical user interfaces (GUI) [Ous94]. Tk is event-driven and supports all of the standard GUI features such as windows, menus, scrollbars, and drawing areas. Event handlers can be associated with arbitrary window events as well as with agent events such as incoming messages from other agents. Since Tk allows a GUI to be written entirely in Tcl, professional-quality interfaces can be

created with a relatively small amount of time and code. An agent that wanted to interact with a user during the course of its travels would carry the necessary Tk code with it. Once it reached the user's machine, it would execute the code and present the interface.

Agent Tcl is an ongoing research project and is far from complete. Our research focuses on the security issues associated with roving code and on support for mobile computing, since agents become particularly useful when they can migrate to and from portable machines. Portable machines are often disconnected from their network and often have an unreliable, low-bandwidth connection when they are connected. By migrating to or from the machine (to interact with the user or with network resources respectively), an agent can avoid extensive use of the poor connection. In this chapter, we first describe the planned architecture of Agent Tcl and the prototype that is included on the enclosed CDROM in directory `systems/agent-tcl` (along with detailed documentation and installation instructions in subdirectory `doc`). Then we present potential uses for Agent Tcl and work through a specific programming example in which an agent collects system information from each machine that it visists. We conclude with a discussion of the weaknesses and strengths of Agent Tcl and its future outlook.

2 Architecture

Agent Tcl has four main goals:

- **Reduce migration to a single instruction**, `agent_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 a new network address.

- **Provide communication mechanisms that are flexible and low-level**, but that hide all transmission details, including whether the agents are on the same or different machines.

- **Provide a high-level scripting language as the main agent language**, 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 most itinerant agents, it is ill-suited for agents that require large amounts of code or that perform speed-critical tasks.

- **Provide effective security** in the uncertain world of the Internet.

The overall goal is a simple, flexible and secure itinerant-agent system that will allow the programmer to select the most appropriate language for her task and rapidly develop even large-scale applications.

2.1 Planned architecture

The planned architecture for Agent Tcl is shown in Figure 1. The architecture builds on the server model of Telescript [Whi94], the multiple languages of ARA [Pei96] and Dixie [Gai94], and the transport mechanisms of two predecessor systems at Dartmouth [Har95, KK94]. The architecture has four levels. The lowest level is an API for the available transport mechanisms. The second level is a server that runs at each network site to which agents can be *sent*. The server performs the following tasks:

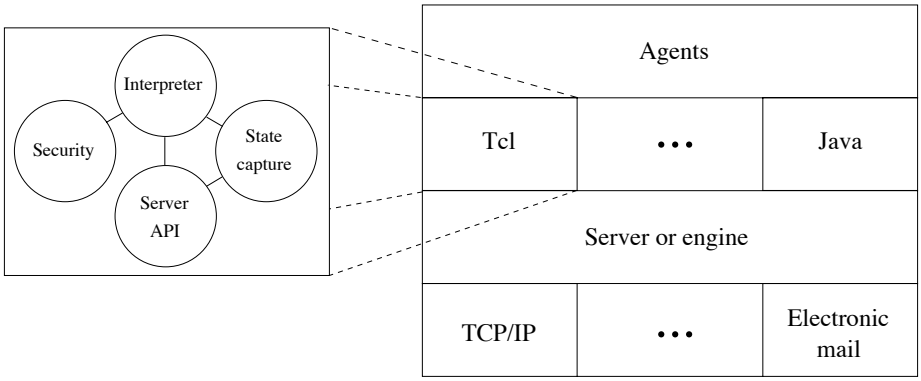

Figure 1: The architecture of Agent Tcl. The four levels consist of an API for the available transport mechanisms, a server that accepts incoming agents and mediates agent communication, an interpreter for each supported language, and the agents themselves.

- *Status.* The server keeps track of the agents that are running on its machine and answers queries about their status.

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

- *Communication.* The server provides a hierarchical namespace for agents and allows agents to send messages to each other within this namespace. The topmost division of the namespace is the symbolic name of the agent's network location. A message is an arbitrary sequence of bytes with no predefined syntax or semantics except for two types of distinguished messages. An *event* message provides asynchronous notification of an important occurrence while a *connection* message requests or rejects the establishment of a direct connection. A direct connection is a named message stream

between agents and is more convenient and efficient than message passing for long interactions (since the programmer can wait 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 restores the agents from the nonvolatile store in the event of machine failure.

As in Tacoma all other services are provided by *agents* [JvRS95]. This approach provides the most flexibility and, with sufficient engineering work on the inter-agent communication mechanisms, should be nearly as efficient as providing the services directly in the agent servers. Such services include resource discovery, group communication, fault tolerance, access control, network sensing, and location-independent communication (e.g., an agent should be able to communicate with another agent without knowing its current network location). The most important service agents in the *internal* Dartmouth prototype are *docking* agents and *resource-manager* agents. *Docking* agents support disconnected operation [GKN+96]. If an agent is unable to migrate to the desired location because of machine or network failure, the agent is added to a queue or *dock* within the network. The *dock* forwards the agent to the desired location once it becomes reachable. *Resource-manager* agents, in combination with the Pretty Good Encryption (PGP) encryption system [KPS95] and language-specific security modules such as Safe Tcl [BR], guard access to critical system resources such as the screen, disk and speaker [Gra96]. PGP authenticates incoming agents; the resource managers assign access restrictions based on this

authentication; and Safe Tcl enforces the access restrictions. In other words, the *resource-manager* agents provide the security policy, while Safe Tcl provides the enforcement mechanism. This approach means that the *same* resource managers can provide the security policy for any agent, regardless of the agent's implementation language. Only the enforcement mechanism needs to change from one language to another.

The third level of the Agent Tcl architecture consists of one interpreter for each available 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). Each interpreter has four components — the interpreter itself, a security module that prevents the agent from taking malicious action, a state module that captures and restores the internal state of an executing agent, and an API that interacts with the server to handle migration, communication, and checkpointing. Adding a new language consists of writing the security module, the state-capture module, and a language-specific wrapper for the generic API. 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 security module for Tcl agents is the existing Safe Tcl extension that allows a Tcl interpreter to replace "dangerous" commands with safe equivalents that perform access checks [BR]. The state-capture module must provide two functions for use in the generic API. 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. The top level of the Agent Tcl architecture consists of the agents themselves.

2.2 Current status

The architecture has not been completely implemented. The current implementation does not provide the nonvolatile store or multiple languages and transport mechanisms (although the framework for incorporating additional languages and transport mechanisms is in place). In addition, several components were undergoing final revision and testing at the time of publication and were not ready for public release. Therefore, the CDROM contains the "stripped-down" version of Agent Tcl that was described in the introduction. This version has the following features:

- There is a single language (Tcl) and a single transport mechanism (TCP/IP). Agents can use all of the standard Tcl features, however, as well as the Tk toolkit.
- Migration, message passing, and direct connections are supported, although the syntax of direct connections is artificially tied to the TCP/IP protocol.
- The namespace is *flat* rather than hierarchical.
- The *docking* and *resource-manager* agents and the authentication subsystem are not included. This means that there is no direct support for mobile computers and that the security mechanisms are rudimentary. The security mechanisms are sufficient, however, for experimentation and for local applications — i.e., an agent server will only accept an incoming agent or message if it originated from an "approved" machine; a list of "approved" machines is given to each server at startup.
- An agent server can only provide limited status information about the agents that are running on its machine.

More complete versions are likely to be available by the time that this book appears on shelves. Interested readers should refer to the downloading instructions in the "Availability" section at the end of this chapter. Although limited, the current version has proven to be a useful tool both at Dartmouth and at several external sites. Part of its usefulness comes from the selection of Tcl as the main agent language. The rest of this section presents the rational behind the use of Tcl and the details of how a Tcl script interacts with the agent system. The subsequent sections present existing and potential applications for Agent Tcl and a specific programming example.

2.3 Tcl

Tcl is a high-level scripting language that was developed in 1987 and has enjoyed enormous popularity [Wel95]. Tcl has several advantages as an itinerant-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 easier 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. Tcl is inefficient compared to other interpreted languages and is orders of magnitude slower than optimized C [SBD94]. 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 itinerant agents tend to involve high-level resource access wrapped with straightforward control logic, a situation for which Tcl is uniquely suited. An itinerant Tcl agent is usually short even if it performs a complex task, and is usually 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 not suitable for every itinerant-agent application, however, such as an application that performs search operations against large, distributed collections of numerical data. For this reason Agent Tcl includes a framework for incorporating additional languages. We are using this framework to add support for the new Java language [Sun94]. Java is much more structured than Tcl and has the potential to run at near-native speed through "just-in-time" compilation. We expect, however, that Tcl will continue to be the main agent language and that Java will be used only for speed-critical agents (or portions of agents).

The main disadvantage of Tcl is that it provides no facilities for capturing the 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 standard Tcl interpreter. The basic problem is that the Tcl interpreter evaluates a script by making *recursive* calls to a function called 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 *interpreter's* runtime stack and is not easily accessible. Our solution adds an explicit stack to the Tcl interpreter. 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 off the stack.

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 the Tcl interpreter on the destination machine must contain the same set of C functions [Pei96]. On the other hand, the explicit stack is less efficient. Our modified Tcl core runs Tcl scripts approximately 20 percent slower than the standard Tcl interpreter, whereas ARA's modified Tcl interpreter imposes no additional overhead. 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, `captureState` and `restoreState`, 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 ignored.

The advantages of Tcl are strong and the disadvantages are either easily overcome or do not affect most 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 itinerant-agent systems such as Tacoma [JvRS95] and ARA [Pei96].

2.4 Tcl scripts as agents

An Agent Tcl agent is a Tcl script that runs on top of the modified Tcl interpreter and a Tcl extension. The modified interpreter provides the explicit stack and the state-capture routines. The Tcl extension provides the set of commands that the script uses to migrate, communicate, and create child agents. Due to the nature of Tcl extensions, these commands are tightly integrated with the normal Tcl commands, and, in fact, appear to be a part of the Tcl language itself. Internally each command uses the generic server API to contact an agent

server, transfer an agent, message, or request, and wait for a response. The main difference between the current and planned implementations 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.

The most important agent commands are `agent_begin`, `agent_submit`, `agent_jump`, `agent_send`, `agent_receive`, `agent_meet`, `agent_accept`, and `agent_end`. An agent uses the `agent_begin` command to register with a server and obtain an identifier in the flat namespace. An identifier currently consists of the IP address of the server, a unique integer, and an optional symbolic name that the agent specifies later with the `agent_name` command. The `agent_submit` command is used to create a child agent on a particular machine. The `agent_jump` command migrates an agent to a particular machine. The `agent_jump` command captures the internal state of the agent, packages the state image for transport, and sends the state image to the destination server. The server accepts the state image, selects a new 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_send` and `agent_receive` commands are used to send and receive messages. The `agent_meet` and `agent_accept` commands are used to establish a direct connection between agents. For direct connections, 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. The agent server will take on more of the responsibility for establishing a direct connection as additional transport mechanisms are added.

3 Examples

Itinerant agents are best viewed as a tool for developing distributed applications rather than as an enabling technology. Their advantage lies not so much in making a particular distributed application possible but rather in unifying a programming model and improving the performance for distributed applications. Performance can be a matter of network utilization, completion time, programmer convenience, or just the ability to continue interacting with the user during a period of network disconnection. Like most itinerant-agent systems, therefore, Agent Tcl is intended for and is a useful tool in general distributed applications.

Some potential applications for Agent Tcl come from existing applications of other itinerant-agent systems. The Telescript system, for example, is currently used in active mail, network and platform management, and electronic commerce [Whi94]. In active mail, a program to be embedded inside a mail message. The program is executed when the mail message is received or viewed. This embedded program can be a Telescript agent. In one platform-management application, a Telescript agent is used to perform software updates. The agent carries the necessary files onto the machine. It installs the files itself and then disappears. Owners of a Sony MagicLink or a Motorola Envoy have received several software updates this way (the MagicLink and Envoy are two personal digital assistants that are based around Telescript and the MagicCap operating system). In several electronic-commerce applications, a Telescript agent leaves

a personal digital assistant (PDA), searches multiple electronic catalogs for a certain product, and returns to the PDA with the best purchase price and the corresponding vendor.

The Tacoma system is used most visibly in StormCast, a distributed system for weather simulation in which the data volumes are so immense as to make data movement impractical [JvRS95]. The use of itinerant agents allows new simulation operations to be rapidly constructed and deployed to the data locations.

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. When a user connects his laptop to the conference's machines, an agent 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.

Java applets also suggest many potential applications. A Java applet is (usually) an interactive, graphical application that is automatically brought to and executed on a user's machine when the user visits the applet's enclosing web page. Existing Java applets include stock tickers, games, and language tutorials. Java applets would be intolerably slow if they controlled the screen from a remote location; dynamic deployment allows them to control the screen efficiently without the need for pre-installation. Itinerant agents can play the same role as Java applets by carrying interface code to the user's location.

Agent Tcl is well suited to most of these applications, although it would have some trouble with the network and platform-management applications of Telescript since Tcl has no direct capabilities for working with binary data. Agent Tcl is being used in three information-retrieval applications. The first involves searching distributed collections of technical reports; the second, med-

ical records [Wu95]; and the third, three-dimensional drawings of mechanical parts [CBC96]. In each application, there is a collection of "documents" at one or more network sites. Each collection provides a set of low-level search primitives. Agents use these primitives to perform a multi-step search at each site. Since the agents move to the location of the collection and do not transfer intermediate results, the multi-step searches can be performed efficiently even though only low-level primitives are available. In addition, since the agent does not need to be in continuous contact with the user's machine, it can continue its task even if the user's machine becomes temporarily unreachable.

Agent Tcl is also being used in several workflow applications [CGN96], although these applications are less mature than the information-retrieval applications. In one application, an agent carries an electronic form from machine to machine so that the appropriate people can fill out their sections of the form; the form is presented using Tk. In a second application, Agent Tcl handles purchase orders. An independent traveling salesperson carries a laptop with software that helps to select vendors and products and to place orders. Agents are sent to search vendor catalogs for products that meet customer needs. When a product and vendor is selected, an agent travels to the vendor's computers where it interacts with billing, inventory, and shipping agents to arrange the purchase. In both cases, the agents can continue working even while the laptop is disconnected. This application is easier to implement with some "support" agents for mobile computing that are not included on the enclosed CDROM, although a simple implementation can be created without these support agents. Agent Tcl is also being used outside Dartmouth, most notably to execute complex queries against remote databases.

4 Language design

Agent Tcl agents are written in the Tool Command Language (Tcl). 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 chapter. 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 curely 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 `man` pages that are included on the CDROM, 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 direct connection 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 direct connection 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. Direct connections 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 a timeout

on an arbitrary block of Tcl code. The documentation on the enclosed CDROM describes these commands, along with all of the commands listed above, in more detail.

5 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 compilation and installation instructions are included on the CDROM. 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. Detailed server instructions are also included on the CDROM.

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.

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`

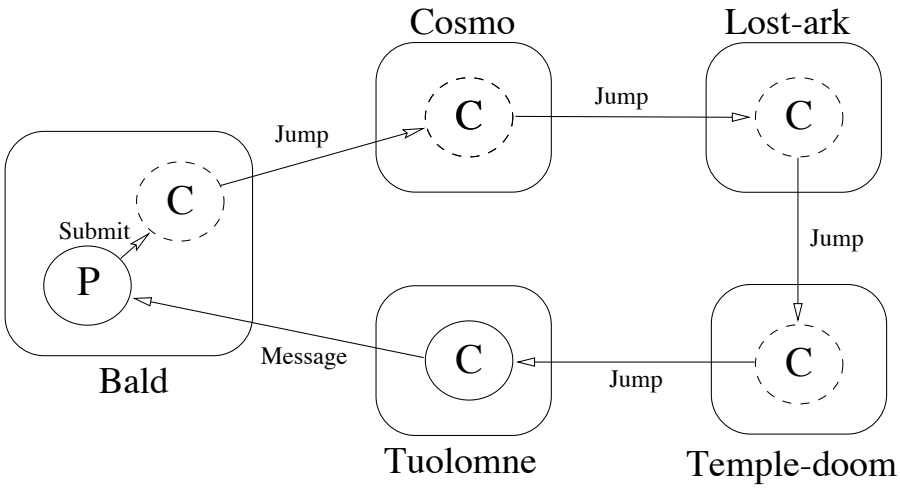

Figure 2: The first version of the "who" agent. The parent agent (P) submits a child agent (C) that migrates through a sequence of machines and executes the Unix `who` command on each. Then the child (C) sends the complete list of users to the parent (P) for display to the user. In the specific case shown, the child agent (C) migrates through four machines at Dartmouth, `cosmo`, `lost-ark`, `temple-doom`, and `tuolomne`.

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 2 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`. If you do not want to enter the code yourself, it is included on the CDROM in file `systems/agent-tcl/book-examples/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 can not 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 2 (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 3. A sample screen dump is shown in Figure 4. This agent is much longer so you will probably want to use the copy in `systems/agent-tcl/book-examples/winwho.tcl` rather than

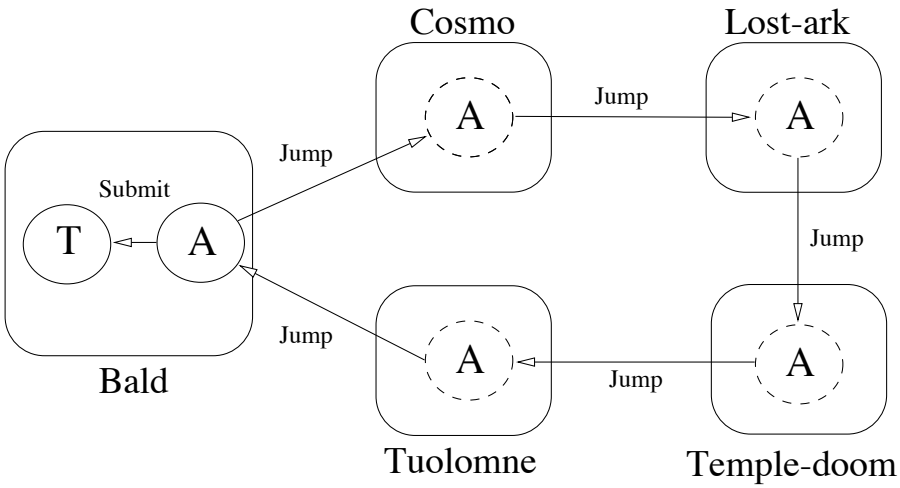

Figure 3: The second version of the "who" agent. The agent (A) migrates through the machines itself, returns to the home machine, and displays the list of users in a Tk window. Before it begins migrating, the agent (A) creates a child agent that will serve as a tracker (T). The agent (A) communications with the tracker (T) as it migrates to provide a continuous update of its location.

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
#
# 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 4. Procedure `DisplayList` creates the output window in which the list of users is displayed; the output window is the bottom window in Figure 4. 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 can not 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 4. 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 4; 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 does not include a visual debugger, but several debugging strategies are discussed in the documentation, and each is reasonably effective. One of the best is illustrated by the second "who" agent — i.e., a moving agent continually reports its status to some specified tracker agent. To report Tcl exceptions, the main body of the agent can be surrounded with a `catch`

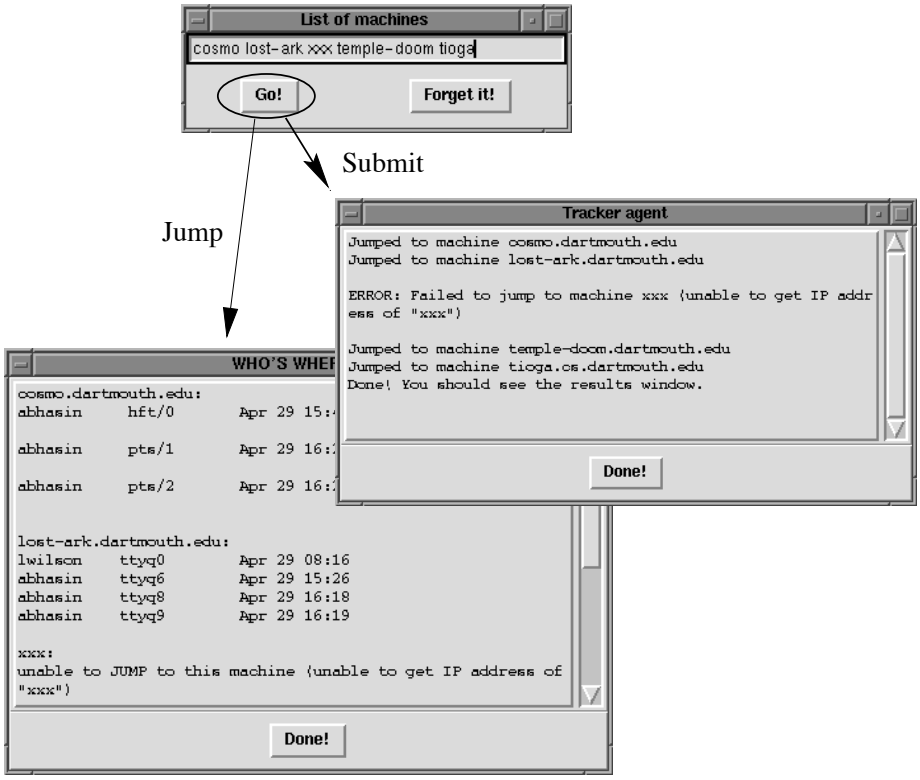

Figure 4: A sample run of the second "who" agent. The first window that the user sees is the entry box at top where the machine names are entered. Once the machine names are entered, the agent uses `agent_submit` to create the tracker agent in the middle. Then the agent jumps from machine to machine, 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.

command; if this `catch` command catches an error, the complete error message can be sent to the tracker (as well as the error location since Tcl maintains a stack trace in the global variable `errorInfo`). Once the agent is debugged, the tracking code can be removed.

6 Pros/cons/advantages

Agent Tcl involves several tradeoffs. Like Tacoma [JvRS95] and ARA [Pei96], Agent Tcl uses the simple scripting language, Tcl, as the main agent language. Other itinerant-agent systems such as Telescript [Whi95] and Java [Sun94] require the programmer to use a complex, object-oriented language even for simple agents. In addition, few systems other than Tacoma [JvRS95] and Visual Obliq [BC96] provide a graphical toolkit that is as high-level and flexible as the Tk toolkit. Agent Tcl, therefore, allows much more rapid development of small- to medium-sized applications. Tcl, however, is slow compared to other scripting languages and is much slower than interpreted bytecodes and native machine code. In addition, Tcl provides no code modularization aside from procedures. Agent Tcl, therefore, can not be used in speed-critical or large applications. Searching a large, distributed collection of numerical data or performing intensive mathematical calculations, for example, would be intolerably slow without at least some low-level support at each site. Developing a mobile, full-featured word processor would involve too much Tcl code to be practical (although the application would potentially be fast enough with careful Tk programming). Java, Telescript, and ARA, which compile their agents into interpreted bytecodes, are the only reasonable choice for such applications, although even these systems would be too slow for such things as distributed scientific computing.

Agent Tcl provides simple, flexible migration and communication primitives. Like Telescript, Agent Tcl provides the *jump* primitive, which captures the complete state of the agent and transparently sends the state to the destination machine. Tacoma, on the other hand, requires the programmer to explicitly collect state information in a "briefcase" and then submit this briefcase along with the migrating agent; the agent starts execution from the beginning and must use the information in the briefcase to determine which task to perform next. Both approaches are equally powerful, but the *jump* primitive is more convenient. There is the potential to overuse *jump* and write hard-to-understand code — e.g., calling a procedure might unexpectedly move the agent to a new location because there is a *jump* buried in the code. This problem is much less severe than the historic *goto* problem, however, since there are no unexpected changes in control flow, and it appears that the problem is not severe enough to outweigh the convenience.

Agent Tcl's communication primitives hide all the transmission details but are low-level enough to efficiently support a range of higher-level communication services. Some systems, such as SodaBot [Coe94], provide a specific high-level communication paradigm (e.g., actor-based, declarative logic, etc.) that is inappropriate for many applications. The programmer is either locked into this paradigm or forced to communicate outside of the agent framework. Agent Tcl's communication primitives have two drawbacks, however. First, if a higher-level communication protocol is desired, it must be implemented on top of the low-level primitives. Second, there is no common "language" that every agent understands. The flexibility of low-level primitives outweighs these drawbacks. We expect that several standard, high-level communication protocols will eventually be provided as part of the Agent Tcl system; RPC and

dialog-based mechanisms have already been implemented but are not included on the CDROM. In addition, we might require agents to understand one simple, common protocol for exchanging status information, but allow them to use any other protocol that they saw fit.

Agent Tcl's main weakness is that it does not provide the features of more mature systems. Agent Tcl lacks the visual debugging tools of Java and Telescript. A simple visual debugger for Agent Tcl exists, however, and is being tested. Similarly, the version of Agent Tcl on the CDROM does not provide the security features of Telescript. Telescript authenticates all incoming agents and assigns access restrictions based on this authentication. The development version of Agent Tcl, however, does exactly this using PGP and Safe Tcl (the development version will be released in mid to late 1996). Agent Tcl's security model, in which resource managers assign access restrictions based on the agent's identification, is simpler than the Telescript model. Telescript agents communicate by exchanging references to each other's objects. Handling the security problems that arise when agents call into each other's objects requires awkward class syntax and "paranoia" programming on the part of the agent programmer [TV96]. Exchanging object references has the additional drawback of making it difficult to include new languages in a Telescript system. One of our main research areas is to expand on existing security mechanisms so that the system protects agents and groups of machines in addition to individual machines.

The version of Agent Tcl on the CDROM also does not include direct support for mobile computing; both Telescript and MSA [TLKC95] provide such support. We have implemented a flexible system of support agents for mobile computing, however, and are successfully using these agents in several appli-

cations [GKN+96]. Agent Tcl does not provide the fault tolerance of Tacoma which uses "rear-guard" agents and the Horus toolkit [JvRS95]. Although these fault-tolerance mechanisms are not incompatible with Agent Tcl, we do not plan to add them as part of our research work. Agent Tcl does not yet support multiple languages. Work on incorporating Java, however, is progressing well. Finally, from an architectural standpoint, Agent Tcl is inefficient since the server and each agent run as separate processes, rather than in an integrated execution environment such as ARA or Telescript. We do not plan to change this in the near future.

Agent Tcl, therefore, is best-suited for experimentation with itinerant-agent ideas and for the development of small- to medium-sized applications in which at least some low-level support is available at each site. Agent Tcl agents combine the low-level services at each site into complex operations, coordinate their efforts with other agents, and handle unexpected error conditions. The flexibility of Agent Tcl allows such agents to be developed rapidly.

Availability

Directory `systems/agent-tcl` on the enclosed CDROM contains the version of Agent Tcl described here (along with complete documentation and installation instructions in subdirectory `doc`). By the time that you read this, however, it is likely that a new version of Agent Tcl will be available. The new version will provide security mechanisms to protect a machine against malicious agents, full compatibility with Tcl 7.5 and Tk 4.1 (as opposed to 7.4 and 4.0), and one or more of the higher-level communication mechanisms such as the RPC analog. All of these are present in the internal Dartmouth version and are undergoing

final testing. The version may also include a visual debugger and support for Java agents; work on both is in progress. Readers who are interested in the new version should consult the WWW site `http://www.cs.dartmouth.edu/~agent` for release dates and downloading instructions.

Acknowledgements

Agent Tcl represents the work of many people. Many thanks to Saurab Nog and Sumit Chawla for developing the RPC system; to Joe Edelman for creating the dialog-based communication mechanism; to Fred Henle and Scott Silver for providing an agent tracker and the basic encryption services; to Melissa Hirschl for implementing an agent debugger; to Keith Kotay and Ken Harker for their work on the Dartmouth ancestors of Agent Tcl; to Brian Brewington, Aditya Bhasin, Kurt Cohen, Yunxin Wu, and Katsuhiro Moizumi for writing the first Agent Tcl applications; and to all the members of the agents research group and the CS 188 topics course who have developed several applications as well as the "service" agents that support resource discovery and mobile computing. Much of their work was undergoing final revision at the time of publication and is not included on the CDROM; interested readers are urged to visit the WWW site listed above. Many thanks also to Bob Sproull of Sun Microsystems and Gisli Hjalmtysson of AT&T Bell Labs for extensive discussion; to the Navy and Air Force for their gracious financial support (AFOSR contract F49620-93-1-0266 and ONR contract N00014-95-1-1204); and to the external users of Agent Tcl, especially Gregory Jorstad of Lockheed Martin's Artificial Intelligence Lab, who have provided invaluable feedback.

References

[BC96] Krishna A. Bharat and Luca Cardelli. Migratory applications. SRC Research Report, Systems Research Center, Digital Equipment Corporation, February 1996.

[BR] N. S. Borenstein and M. Rose. Safe Tcl. Available at `ftp://ftp.fv.com/pub/code/other/safe-tcl.tar.Z`.

[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. To appear.

[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, Deptartment of Computer Science, Dartmouth College, May 1996.

[Coe94] Michael D. 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.

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

[GKN+96] Robert Gray, David Kotz, Saurab Nog, Daniela Rus, and George Cybenko. Mobile agents for mobile computing. Technical Report PCS-TR96-285, Dept. of Computer Science, Dartmouth College, May 1996. Submitted to ACM MobiCom '96.

[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.

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

[JvRS95] Dag Johansen, Robbert van Renesse, and Fred B. Scheidner. Operating system support for mobile agents. In *Proceedings of the 5th IEEE Workshop on Hot Topics in Operating Systems (HTOS)*, pages 42–45, 1995.

[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.

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

[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.

[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.

[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.

[SBD94] Adam Sah, Jon Blow, and Brian Dennis. An introduction to the Rush language. In *Proceedings of the 1994 Tcl Workshop*, June 1994.

[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.

[Sun94] The Java language: A white paper. Sun Microsystems White Paper, Sun Microsystems, 1994.

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

[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.

[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.

[Whi95] James E. White. Telescript technology: Scenes from the electronic marketplace. General Magic White Paper, General Magic, 1995.

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