The following paper was originally published in the
Proceedings of the Fourth USENIX Tcl/Tk Workshop
Monterey, CA, July 10-13, 1996

Agent Tcl: A flexible and secure mobile-agent system

Robert S. Gray
Dartmouth College
Hanover, New Hampshire 03755

Agent Tcl: A flexible and secure mobile-agent system

Robert S. Gray*
Department of Computer Science
Dartmouth College
Hanover, New Hampshire 03755

robert.s.gray@dartmouth.edu

Abstract

An *information agent* manages all or a portion of a user's information space. The electronic resources in this space are often distributed across a network and can contain tremendous quantities of data. *Mobile agents* provide efficient access to such resources and are a powerful tool for implementing information agents. A mobile agent is an autonomous program that can migrate from machine to machine in a heterogeneous network. By migrating to the location of a resource, the agent can access the resource efficiently even if network conditions are poor or the resource has a low-level interface. Telescript is the best-known mobile-agent system. Telescript, however, requires the programmer to learn and work with a complex object-oriented language and a complex security model. Agent Tcl, on the other hand, is a simple, flexible, and secure system that is based on the Tcl scripting language and the Safe Tcl extension. In this paper we describe the architecture of Agent Tcl and its current implementation.

1 Introduction

An *information agent* is charged with the task of managing all or a portion of a user's information space. The electronic resources in this space are often distributed across a network and can contain tremendous quantities of data. *Mobile agents* allow efficient access to such resources and are a powerful tool for implementing information agents. A mobile agent is a program that can migrate *under its own control* from machine to machine in a heterogeneous network. In other words, an agent can suspend its execution at any point, transport its code and state to another machine, and resume execution on the new machine. By migrating to the location of an electronic resource, an agent can access the resource locally and can eliminate the network transfer of all intermediate data. Thus the agent can access the resource efficiently even if network conditions are poor or the resource has a low-level interface. This efficiency, combined with the fact that an agent does not require a permanent connection with its home site, makes agents particularly attractive for mobile computing since roving devices often have a low-bandwidth, unreliable connection into the network. Mobile agents also ease the development, testing and deployment of distributed applications since they hide the communication channels but not the location of the computation [Whi94], they eliminate the need to detect and handle network failure except during migration, and they can dynamically distribute and redistribute themselves throughout the network. 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]. Finally, anecdotal evidence suggests that mobile agents are easier to understand than many other distributed-computing paradigms. Existing applications for mobile agents include electronic commerce, active documents and mail, information retrieval, workflow and process management, and network management [Whi94, Ous95]. Potential applications include most distributed applications, particularly those that must run on disconnected platforms or that must invoke multiple operations at each remote site [Whi94, Ous95].

The advantages and potential applications of mobile agents have led to a flurry of recent implementation work. Notable systems include Telescript from General Magic, Inc. [Whi94], Tacoma

*Supported by AFOSR contract F49620-93-1-0266 and ONR contract N00014-95-1-1204. A small section of this paper appeared in [Gra95].

from the University of Cornell [JvRS95], SodaBot from MIT [Coe94] and ARA from the University of Kaiserslautern [Pei96]. These systems suffer from a range of weaknesses. Telescript provides a complex, object-oriented language and a complex security model in which the *programmer* must carefully identify and disallow dangerous actions. Tacoma and SodaBot provide high-level scripting languages (Tcl and SodaBotL respectively) that are much easier to learn and use. In addition, Tacoma uses the Horus toolkit to provide significant fault tolerance. Tacoma, however, requires the programmer to explicitly capture state information before migration. Tacoma and SodaBot only partially address security issues — Tacoma via simple encryption and SodaBot via minimal user control over resource usage — and do not provide low-level communication mechanisms, forcing some communication to take place outside of the agent framework. Finally, although the scripting languages of Tacoma and SodaBot are sufficient for most tasks, the lack of a faster language makes them unsuitable for speed-critical applications. ARA strikes a balance between the Telescript and Tacoma extremes by providing multiple languages, a framework for incorporating additional languages, and low-level communication mechanisms. ARA, however, has not been released and does not address security issues.

Agent Tcl is a mobile-agent system that is under development at Dartmouth College [Gra95]. Agent Tcl, like ARA, attempts to strike a balance among existing systems. Agent Tcl uses the flexible scripting language Tcl as its main language but provides a framework for incorporating additional languages. Agent Tcl provides migration and communication primitives that do not require the programmer to explicitly capture state information and that hide the actual transport mechanisms *but* that are low-level enough to be used as building blocks for a range of protocols. Agent Tcl uses the simple Safe Tcl security model to protect a machine from a malicious agent and agents from each other. Agent Tcl allows agents to migrate from machine to machine *or* remain stationary and access resources from across the network, to create child agents to perform subtasks, and to communicate with other agents on the local and remote machines. It is intended as a general environment for distributed applications, both in the Tcl/Tk and larger computing communities, with the application developer selecting the migration, communication and creation strategy that is best for the given network, resources and task. Although Agent Tcl is far from complete, it is in active use at several sites and has been used in several information-management applications. These applications demonstrate the convenience and efficiency of mobile agents.

Section 2 presents the Agent Tcl architecture. Section 3 describes the selection of Tcl as the "main" agent language and the current implementation. Section 4 discusses the security concerns associated with mobile code, our current Safe Tcl security mechanisms, and the security mechanisms that must be added to provide sufficient protection for both the machines and the agents. Finally, Sections 5 and 6 briefly examine several information-management applications and highlight future work.

2 Architecture

Agent Tcl has four main goals:

- Reduce migration to a single instruction like the Telescript *go* and allow this instruction to occur at arbitrary points. The instruction should not require the programmer to explicitly capture state information and should hide the actual transport mechanisms.
- Provide transparent communication among agents. The communication primitives should be flexible and low-level but should hide the actual transport mechanisms.
- Support multiple languages and transport mechanisms and allow the *straightforward* addition of a new language or transport mechanism.
- Provide effective security in the uncertain world of the Internet.

The architecture of Agent Tcl is shown in Figure 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 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. The server performs the following tasks:

- *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 its owner,

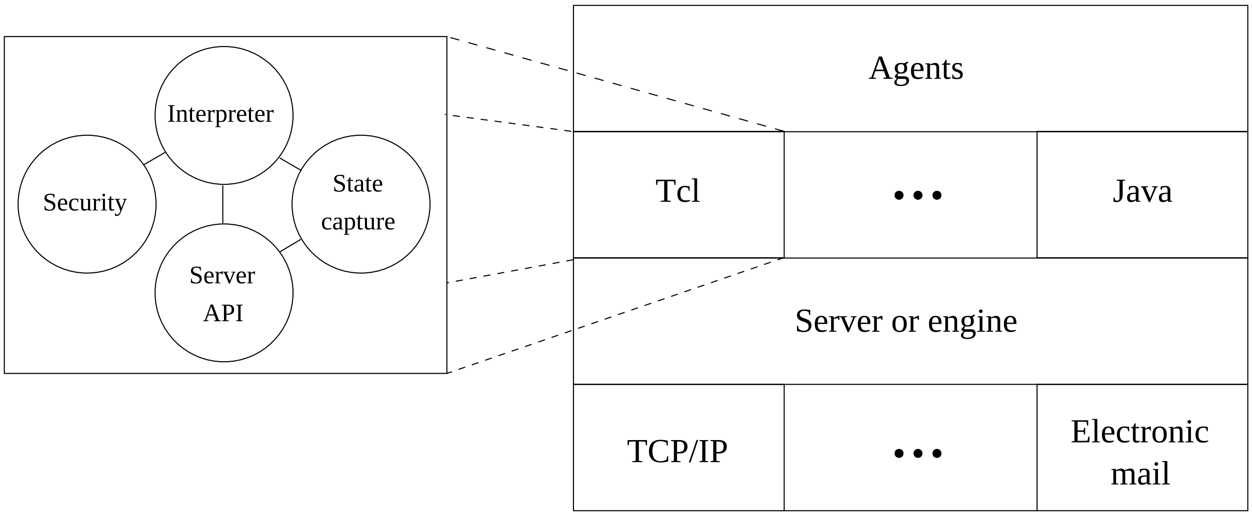


Figure 1: The architecture of Agent Tcl. The four levels consist of an API for the available transport mechanisms, a server that accepts incoming agents and mediates agent communication, an interpreter for each supported language, and the agents themselves.

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 hierarchical namespace for agents and allows agents to send messages to each other within this namespace. The topmost division of the namespace is the network location of the agent. 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 (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 restores the agents from the nonvolatile store in the event of machine failure.

As in Tacoma all other services will be provided by *agents*. Such services include navigation, network sensing, group communication, fault tolerance, location-independent addressing, and access control. The most important service agents in our implemented prototype are *resource manager* agents which guard access to critical system resources such as the screen, network and disk. These resource managers are discussed in the security section.

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 an 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 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.

3 Tcl and Agent Tcl

The architecture has not been completely implemented. The current implementation does not provide event messages or the nonvolatile store and has a single language (Tcl), a single transport mechanism (TCP/IP), and a *flat* rather than hierarchical namespace. It does provide migration, message passing and direct connections, and has sufficient security mechanisms to protect a machine from a malicious agent and to protect agents from each other. Incoming agents are authenticated using Pretty Good Privacy (PGP) [KPS95]; *resource manager* agents assign access restrictions based on this authentication; and Safe Tcl enforces these restrictions as the agent executes [BR]. Here we discuss the selection of Tcl as the main agent language and the details of the base system. We discuss security in the next section.

3.1 Tcl

Tcl is a high-level scripting language that was developed in 1987 and has enjoyed enormous popularity [Wel95]. 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 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 a high-level, interpreted language so it 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 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 mobile-agent application, however, such as performing 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 *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` in order 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 2 illustrates this process for the `while` command.

It is important to note that the modified Tcl core is 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

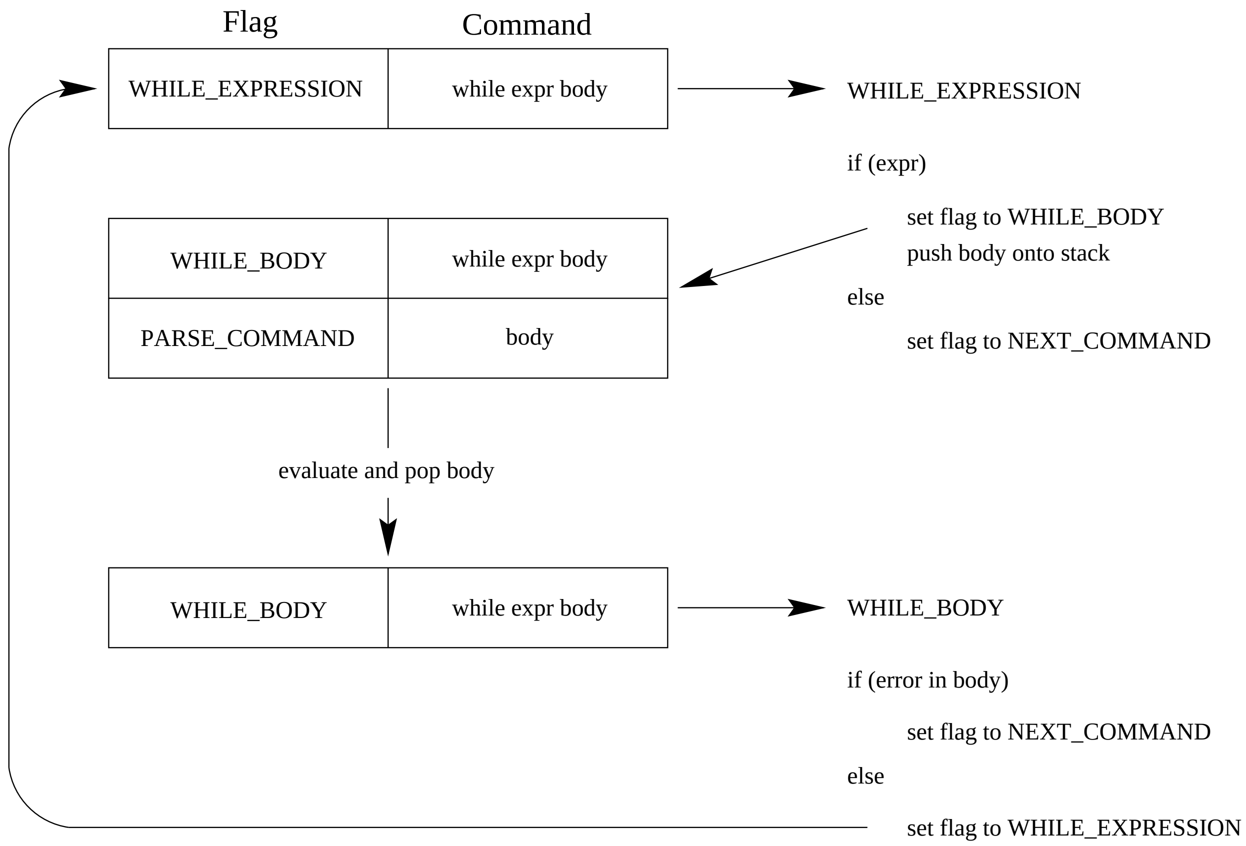


Figure 2: An example of how the stack works. The command stack is on the left and the two subhandlers for the `while` command are on the right. A subhandler sets the `NEXT_COMMAND` flag when the `while` command has finished and should be popped.

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.

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 programs approximately 20 percent slower than the standard Tcl core, whereas ARA's modified Tcl core imposes little additional overhead. It appears that this 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.

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 mobile-agent systems such as Tacoma [JvRS95] and ARA [Pei96].

3.2 Agent Tcl

The current implementation of Agent Tcl has two components. The first component is the server that runs at each network site. The server accepts, authenticates and starts incoming agents, buffers incoming messages, provides the flat namespace, and answers queries about the status of the agents that are running on its machine. The server is implemented as two cooperating processes. One process watches the network while the other maintains a table of running agents.

The second component consists of 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 uses the server API to contact an agent server, transfer an agent, message or request, and wait for a response. The main difference between the current implementation and the proposed 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 a name in the flat namespace. A name 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_submit` command accepts a Tcl script, encrypts and digitally signs the script, and sends the script to the destination server. The server authenticates this agent, selects a name for the agent, and starts a Tcl interpreter in which to execute the agent. If the agent wants a symbolic name as well as a unique, 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, encrypts and digitally signs the state image, and sends the state image to the destination server. The server authenticates this agent, selects a new name 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. A direct connection is a named message stream. Direct connections are not required for communication but are more efficient and convenient as noted above. 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.

4 Security in Agent Tcl

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 harm the machine or access privileged resources. 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. Whether the agents and machines are actively malicious or programmed poorly, the end effect is the same. A mobile-agent system must provide security mechanisms that detect and prevent malicious actions. Without strong security mechanisms, a mobile-agent system will justifiably never be accepted and used. 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 access permissions based on this authentication, and prevent any violation of the access permissions.

- *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. Clearly 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 detect tampering as soon as the agent migrates from a malicious machine back to an honest machine and terminate or fix the agent if tampering has occurred. In addition we must ensure (1) that the sensitive information never passes through an untrusted machine in an unencrypted form, (2) that 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, 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 terminated.

All of these problems have been considered in the mobile-agent literature [LO95, CGH+95, TV96] although 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] and Safe Tcl [BR]. First we present the current implementation and then potential solutions for the remaining two security problems.

4.1 Authentication

Authentication in Agent Tcl is based on PGP (Pretty Good Privacy) which is in widespread use despite controversies over export restrictions and patents [KPS95]. PGP encrypts a file or mail message using the IDEA private-key algorithm and a randomly chosen private key, encrypts the private 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 MD-5 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 small modifications. In the current implementation we run PGP as a separate process, save the data to be encrypted into a file, ask the PGP process to encrypt the file, and then transfer the file to the destination server. This structure 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].

When an agent registers with a server using the `agent_begin` command, the registration request is digitally signed used the owner's private key, encrypted using the server's public key, and sent to the server. The server makes sure that the agent's owner is allowed to register on its machine and records the authenticated identity of the agent's owner. Then the IDEA private key is used as a session key for all further communication between the agent and its newly registered server. The session key is needed to prevent a malicious program from contacting the server and masquerading as an existing agent. When the agent and its registered server are on the same machine (which is the predominant case), we do not actually encrypt with the session key since there is no possibility of message interception; instead the session key is simply included in the message and compared against the server's recorded session key. A sequence number is included in the messages to prevent replay attacks.

When an agent migrates using the `agent_jump` command, it is digitally signed with the current server's private key and encrypted with the recipient server's public key. As in Telescript, we digitally sign using the *server*'s private key since the owner's private key is unavailable once the agent leaves its home machine [TV96]. This approach requires the servers to have a high degree of trust in each other, so we will eventually adopt the Itinerant Agent solution [CGH+95], in which as much of the agent as possible is encrypted with the owner's private key on creation and remains encrypted throughout the agent's lifetime. The identity of the agent's owner is included in the migration message. The recipient server chooses whether to believe that identity based on its trust in the sending server. If the server accepts the agent, it records the apparent identity of the agent's owner, the authenticated identity of the sending server, and its degree of confidence that the owner's identity is valid. A session key is used for all further agent-server communication as in the `agent_begin` case. The same steps occur when a child agent is created with the `agent_submit` command except that a Tcl script is encrypted rather than a Tcl state image. The same steps also occur when an agent sends a message to an agent on another machine. In the case of a direct connection, the IDEA private key from the acceptance message becomes the session key for the direct connection. A sequence number associated with the direct connection prevents replay attacks.

There are two weaknesses with the current implementation. First, there is no automatic distribution mechanism for the PGP public keys. Each server must already *know* all possible public keys so that it can authenticate incoming agents. An automatic distribution mechanism must be added when we start to use Agent Tcl in wide-area networks. Second, the system is vulnerable to replay attacks in which an attacker replays a migrating agent or any message sent from one agent to another (outside of a direct connection). An obvious solution is for each server to have a distinct sequence number for all servers with which it is in contact.

4.2 Authorization and enforcement

Once the identity of an agent's owner has been determined, the system must impose access restrictions on 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 only be accessed through another agent. *Builtin* resources are directly accessible through language primitives for reasons of efficiency or convenience or simply by definition. Builtin Tcl/Tk resources include the screen, the file system, wall-clock time and CPU time.

For indirect resources, the agent that controls the resource enforces the relevant access restrictions. For each message from another agent, the local server attaches to the message a 5-tuple that contains the apparent identity of the agent's owner, the apparent identity of the sending server, a flag that indicates whether the owner could be authenticated, a flag that indicates whether the sending server could be authenticated, and a numerical confidence level that represents how much trust the local server places in the sending server. The agent uses this 5-tuple along with its own internal access lists to respond appropriately to the incoming message.

For builtin resources, security is maintained using Safe Tcl and a set of *resource manager* agents. Safe Tcl is a Tcl extension that is designed to allow the safe execution of untrusted Tcl scripts [BR]. 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 from which all dangerous commands have been removed. 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 the 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. The idea with this type of security risk is to restrict the number of times per second that the agent can *initiate* the event itself, or to restrict the agent to its own window in which it can do whatever it wants but whose size and position it can not affect.

Although the dangerous commands have been removed from the untrusted interpreter, we do not want to deny all access to the resources associated with these commands. Thus, instead of removing a dangerous command entirely, Safe Tcl can replace the command with a *link* to a command in the trusted interpreter. This trusted command either severely restricts the functionality of the original command or examines the command arguments and the identity of the script's owner to determine if the command should be allowed.

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. These secure versions check a set of access lists to see if the command is allowed. In the current implementation there is an access list for wall-clock and CPU time, the screen, the network, the file system, and external programs. Each access list is a set of *(name, quantity)* pairs where *name* specifies the name of the required resource and *quantity* specifies the number of instances of that resource (if applicable). The *screen* access list, for example, might contain the pair *(toplevel, 5)*, which indicates that the agent can have no more than five toplevel windows. The *program* access list might contain the pair *(ls, ())* which indicates that the agent is allowed to execute the Unix `ls` program. Initially the access lists are empty except that the agent is given a minimal amount of wall-clock and CPU time (our modified Tcl interpreter aborts a script if the script exceeds the time limits). To obtain additional time or to obtain access to other builtin resources, the agent must explicitly or implicitly ask a *resource manager* agent for permission. There are five resource managers in the current system. These managers correspond to the five access lists and control access to wall-clock and CPU time, the screen, the network, the file system, and external programs.

An agent uses the `require` command to explicitly ask a resource manager for access. The `require` command takes the symbolic name of the resource manager — e.g., *screen* — and a list of *(name, quantity)* pairs which specify the desired access permissions — e.g., *(toplevel, 5), (screen_area, 30 percent)*. The `require` command is actually just a link to a procedure in the trusted interpreter. This procedure contacts the appropriate resource manager and passes the list of access requests to the resource manager. The procedure waits for the response and then adds each access request to the internal access lists, indicating for each whether the request was granted or denied.

To implicitly ask a resource manager for access, an agent simply calls a command that uses the resource. For example, if the agent issues the command `exec ls`, the `exec` procedure in the trusted interpreter checks the *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 aborts with a security error. Otherwise the command contacts the *program* resource manager and either proceeds or aborts depending on the manager's response. Although these implicit access restrictions are convenient, the agent should use the `require` command whenever possible so that it can determine whether a required resource is available before it tries to use the resource.

Our implementation does not yet provide a safe version of all dangerous commands. For example, an agent that arrives from another machine can not use the `source` and `send` commands (the `send` command will probably never be available since it is difficult to make secure and agents should communicate within the agent framework anyways). In addition, the "annoyance" security threats have not been eliminated. Rather than restricting the use of all associated commands, we plan to provide each agent with a virtual screen in which it can do whatever it wants but that only the user can move and resize. Although the annoyance threats remain, Agent Tcl currently protects the machine well using the simple kernel-user model of Safe Tcl. No direct access to system resources is possible. There is no way for an agent to subvert the *resource-manager* system since there is no way for the agent to modify the access lists contained in the trusted interpreter; it is possible for the agent to contact a resource manager directly, but this accomplishes nothing since the response (1) will correctly grant or deny access and (2) even so will not be added to the access lists. In our case, Safe Tcl is the mechanism for enforcing the policy provided by the resource managers. When Java is added to the system, the existing Java security mechanisms will be used to enforce the same policy provided by the same resource managers.

In addition to the resource managers, Agent Tcl includes a *console* agent, which is used primarily on machines that have a specific owner. The *console* agent has two purposes. First, it tracks all of the agents that are running on the machine, and allows the machine's owner to deny entry to incoming agents and to terminate running agents. Second, it provides a pathway through which a resource manager can ask the owner whether an agent should be able to perform a particular action. The owner will eventually be able to specify exactly those situations in which she should be asked.

4.3 Future security work

There are three areas of future work. First, we plan to add a hierarchical system of resource managers. This will become particularly important as we move towards the Telescript model in which there are multiple virtual *places* per machine [Whi94]. Each place might have its own security policy while the machine has an overall security policy. Second, we must protect an agent from malicious machines. Here we are exploring the suggestions from [CGH+95] in which an agent is divided into components and each component is encrypted and signed separately for all or part of the journey. This scheme allows immediate detection of blatant tampering, such as dropping part of the agent or inserting an entirely new procedure, and prevents blatant theft of sensitive data. In addition we plan to record an audit trail that can be analyzed to determine the point at which a failed agent might have been modified inappropriately. For more subtle modification threats, such as modifying a piece of data that changes on every machine and thus must be unencrypted, solutions are less clear and may be impossible. Third, we must protect a group of machines from a malicious agent. Here we are looking at a *currency-based* resource-allocation scheme in which an agent's owner gives the agent a finite currency supply from her own finite currency supply. The currency does not have to be tied to legal currency, but it should be impossible to spend a currency unit more than once. The agent must spend currency in order to access resources and must divide its own currency among its children. The agent and its children will eventually run out of currency and terminate. Such currency schemes already exist in the context of electronic commerce [Way95].

5 Applications

Figure 3 shows the "who" agent which illustrates the agent commands. The agent's task is to determine which users are logged onto a set of machines. The agent uses `agent_submit` to create a child agent. The child agent jumps from machine to machine using `agent_jump` and executes the Unix `who` command on each machine. The child then sends the list back to its parent with `agent_send`. The parent has been waiting for the list with `agent_receive` and displays the list to the user.

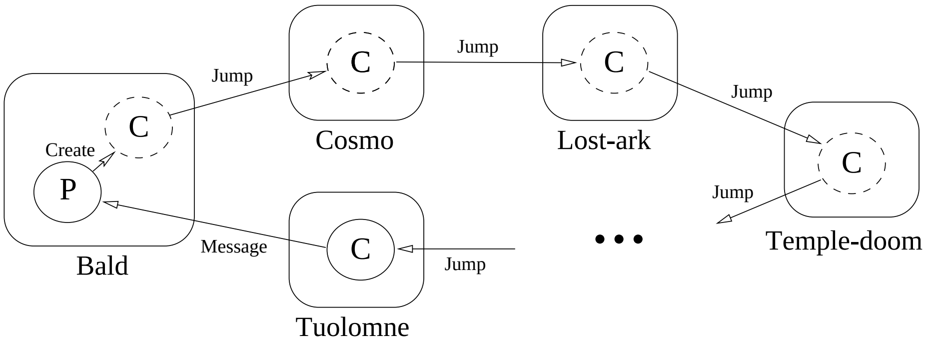
Although its task is simple and can be accomplished easily without a mobile agent, the "who" agent illustrates the general form of any agent that migrates through a sequence of machines. Existing Agent Tcl agents that fall into this category are a workflow agent that carries an electronic form from user 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.

Like the "who" agent, 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. In addition, the workflow and medical agents are extremely easy to implement within the agent framework. 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 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. The `agent_jump` command is convenient since it automatically captures this state. The `agent_jump` command does impose a moderate execution overhead on the Tcl interpreter; this overhead can be made much smaller, however, and can even be reduced to near zero with the ARA solution [Pei96].

Another example is our "alert" agent that monitors a specified set of remote resources and notifies its owner of any change in resource status. Figure 4 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 email 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.

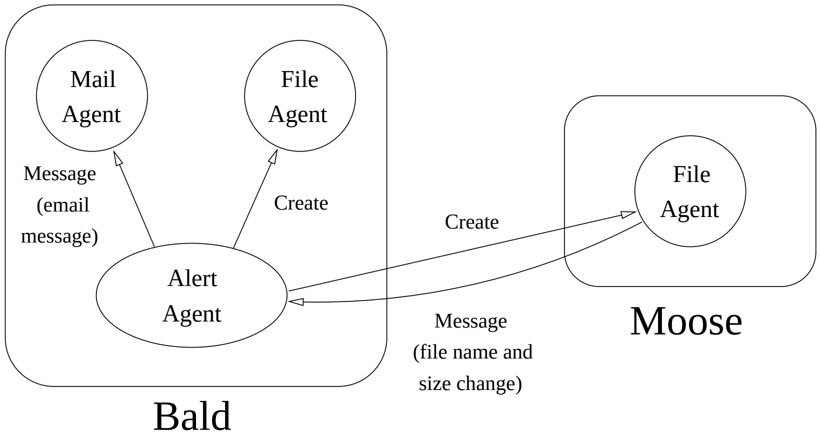
A hybrid of the two examples is our text-retrieval agent that searches distributed collections of text documents. This agent is designed to be launched from a mobile device. It first obtains the query from




```
  # procedure WHO is the child agent that does the jumping

proc who machines {

  global agent

  set list ""

    # jump from machine to machine and execute the Unix who command on each machine

  foreach m $machines {

    if {catch "agent_jump" $m"} {
      append list "$m:\n unable to JUMP to this machine"
    else {
      set users [exec who]
      append list "$agent(local-server):\n$users\n\n"
    }
  }

  agent_send $agent(root) $list
  exit
}

set machines "bald cosmo lost-ark temple-doom moose muir tenaya tioga tuolomne"

  # get a name from the server

agent_begin

  # submit the child agent that jumps

agent_submit $agent(local-ip) -vars machines -procs who -script {who $machines}

  # wait for and output the list of users

agent_receive code string -blocking
puts $string

  # agent is done

agent_end
```

```
bald.cs.dartmouth.edu:
rgray    ttyp2    Sep  5 21:24 (:0.0)
rgray    tty6     Sep  7 07:14

cosmo.dartmouth.edu:
gvc         pts/0      Aug 23 10:11

...
```

Figure 3: The "who" agent submits a child agent that jumps from machine to machine and executes the Unix `who` command on each machine. The Tcl code is in the middle (the `agent` array holds the current location of the agent and is updated automatically as the agent migrates). The path of the agents through the network is shown at top. A fragment of the output appears at bottom.




```
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 4: The "alert" agent monitors a set of files and sends an email 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.

the user and then jumps to a permanently connected machine somewhere in the network. It then spawns one child agent for each collection. The child agents travel to the remote collections, perform the query using the available retrieval tools, and return to the permanently connected machine with the query results. The original agent then *discards all duplicates* and carries the results back to the mobile device. This approach allows the agent to carry on its retrieval work even when the mobile device is disconnected and minimizes the total number of bytes transferred across the low-bandwidth connection between the mobile device and the network (each document entry consists of the title, author and abstract so it takes only a few duplicates to add up to the agent's code size). In addition, there is no need to provide high-level search operations at each collection; since the child agents move to the collections, they can perform their search efficiently even if they must combine low-level primitives into the desired search operation.

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.

6 Future directions

The first area of future work is to finish the proposed architecture. We must add the hierarchical namespace, the nonvolatile store, and multiple languages and transport mechanisms. We are specifically interested in Java, Lisp, electronic mail and HTTP. Work on Java is in progress. In addition, we must finish the resource managers and add the security mechanisms that will protect an agent from a malicious machine and a group of machines from a malicious agent. Finally, we must extend our existing application agents so that they use the available security information.

The second area of future work is to add support agents. The resource managers that specify the security policies are one type of support agent. An effective mobile-agent system requires several more. We are in the process of identifying and constructing the necessary agents. Work on agents that provide directory services, navigation services, network-sensing tools, high-level communication services, and graphical construction tools is in progress.

The third area of future work is to experimentally compare the performance of mobile agents against traditional client/server solutions and to formally characterize when an agent should remain stationary and when and how far it should migrate. Such a characterization must consider such things as network latency and bandwidth, relative machine speeds, code sizes, and data volumes.

7 Conclusion

Agent Tcl is a secure mobile-agent system that gains much of its flexibility and simplicity from use of the high-level scripting language Tcl. Although implementation work is not complete, Agent Tcl is in active use and has allowed the rapid development of efficient, distributed applications.

Availability

Agent Tcl version 1.2 will be available at `http://www.cs.dartmouth.edu/~agent` near the end of the summer; we are finishing the resource managers, writing the new documentation, improving the interface between Agent Tcl and PGP, and reworking the session-key implementation so that it does not require modifications to PGP. Agent Tcl version 1.1 is available now; version 1.1 uses Tcl 7.4, provides limited security and is somewhat slower. Agent Tcl runs on standard Unix platforms.

Acknowledgements

Many thanks to Professor David Kotz for reading the various incarnations of this paper and providing helpful criticism; to the anonymous reviewers for their constructive feedback; to Professor George Cybenko and Professor Daniela Rus for their support and encouragement; to Saurab Nog, Ting Cai, Yunxin Wu, Aditya Bhasin, Kurt Cohen and Scott Silver for their implementation work; and to the Air Force and Navy for their gracious financial support (ONR contract N00014-95-1-1204 and AFOSR contract F49620-93-1-0266).

References

[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.

[CGH$^+$95] David Chess, Benjamin Grosof, Colin Harrison, David Levine, Colin Parris, and Gene Tsudik. Itinerant agents for mobile computing. Technical Report RC 20010, IBM T. J. Watson Research Center, March 1995. Revised October 17, 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, 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.

[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.

[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*, 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-Hall, New Jersey, 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.

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

[Ous95] John K. Ousterhout. Scripts and agents: The new software high ground. Invited Talk at 1995 Winter USENIX Conference, January 1995.

[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.

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

[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.

[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.

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