Agent Tcl: A transportable agent system

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

robert.s.gray@dartmouth.edu

22 September 1995

Abstract

Agent Tcl is a transportable-agent system that is under development at Dartmouth College. A *transportable agent* is a named program that can migrate from machine to machine in a heterogeneous network. Such programs are a powerful tool for implementing *information agents* since the electronic resources in a user's information space are often distributed across a network and can contain tremendous quantities of data. Sending a *user-specific* program to the network location of the resource is often the most convenient and efficient alternative. The goal of Agent Tcl is to address the weaknesses of existing transportable-agent systems. Agent Tcl will run on standard hardware, support multiple languages and transport mechanisms, provide transparent migration and communication, and provide effective security and fault-tolerance in the uncertain world of the Internet. This paper describes the architecture of Agent Tcl and its current implementation and presents four information-management applications in which Agent Tcl has proven useful.

1 Introduction

An *information agent* manages a portion of a user's information space. The electronic resources in this information space are often distributed across a network and can contain tremendous quantities of data. *Transportable agents* provide efficient access to such resources. A transportable agent is a named program that can migrate from machine to machine in a heterogeneous network. The program chooses when and where to migrate. It can suspend its execution at an arbitrary point, transport to another machine and resume execution on the new machine. By migrating to the network location of the resource, the program does not need to bring intermediate data across the network and can access the resource efficiently even if the resource developer provides only simple primitives. Thus transportable agents are more efficient than the traditional client-server paradigm and allow the rapid development of distributed applications.

Transportable agents are a new research area. The few existing systems include Telescript^{TM} from General Magic and Tacoma from the University of Tromsø and the University of Cornell [Whi94, JvRS95]. These initial systems suffer from a range of weaknesses. Tacoma, for example, requires the programmer to explicitly capture state information before migration and provides no security mechanisms. Telescript requires special hardware, is not available to researchers and limits the programmer to a single language.

The goal of the Agent Tcl project at Dartmouth is to address these weaknesses. Agent Tcl will run on standard Unix platforms, support multiple languages and transport mechanisms, reduce migration to a single instruction like the Telescript *go*, provide transparent communication and provide effective security and fault-tolerance in the uncertain world of

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

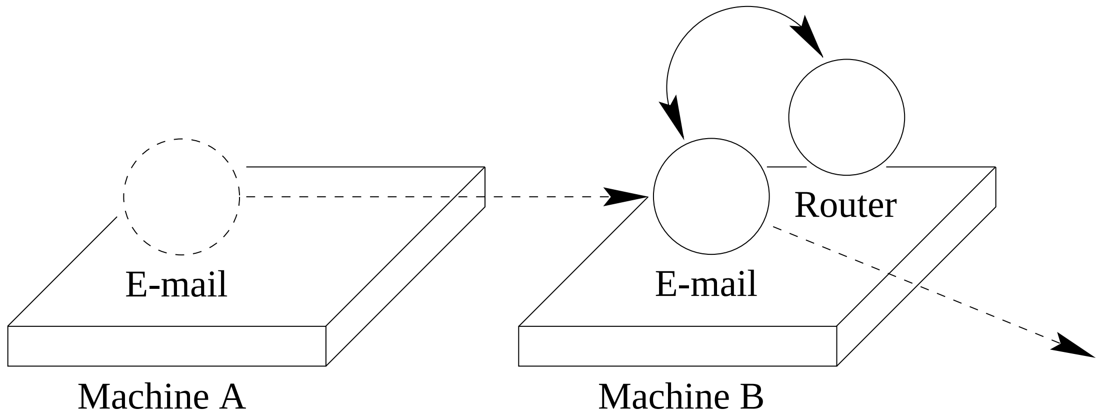


Figure 1: An example of a transportable agent. Here an active e-mail message has jumped to interact with a router and will jump again to interact with the recipient's mailbox. This figure was adapted from [Whi94].

the Internet. Although Agent Tcl is far from complete, it has been used in four information-management applications in which the relevant resources are distributed across a small network. These applications demonstrate the convenience and efficiency of transportable agents.

Section 2 discusses existing transportable-agent systems. Section 3 describes the architecture of Agent Tcl while section 4 describes the current implementation. Section 5 examines the four information-management applications. Finally section 6 covers the most important areas of future work.

2 Background

Transportable agents have been developed as an extension to and replacement of the client-server paradigm. The client-server paradigm divides programs into fixed roles. A *server* provides a set of services while a *client* requests those services. The client and server often reside on different machines which means that client-server interaction requires network communication. The most common communication mechanism is message passing. Message passing is powerful and flexible but requires the programmer to handle low-level details such as determining the network address of the server, matching responses with requests and handling communication errors [SS94]. Remote procedure call (RPC) hides these low-level details by allowing a client to invoke a server operation *using the standard procedure call mechanism* [BN84]. Most implementations of RPC use *stub* procedures.

The problem with message passing and RPC is that the client is limited to the operations provided at the server. If the server does not provide an operation that matches the client task exactly, the client must make a series of remote calls, bringing intermediate data across the network on each call. Transmitting the intermediate data is a waste of time and bandwidth. To avoid this inefficiency, server developers often provide specialized operations for each client. This approach becomes intractable as the number of clients grows, does not allow for unforeseen clients and violates the modern software-engineering principle of providing simple, efficient primitives rather than complex procedures. The solution is for the server to provide a set of efficient primitives and for the client to send a *subprogram* to the server. The subprogram executes at the server and returns only the final result to the client. All intermediate data transfer is eliminated, conserving bandwidth and reducing overall latency. The subprogram approach is exemplified in the Network Command Language (NCL), Remote Evaluation (REV) and SUPRA-RPC [Fal87, SG90, Sto94].

The subprograms of NCL, REV and SUPRA-RPC are limited in that they can not migrate after their initial transfer, can not communicate easily with each other, maintain the fixed client-server division and are explicitly tied to the client. Transportable agents, however, are autonomous, named programs that communicate and migrate at will. Transportable agents support the peer-to-peer model in which processes communicate as peers and act as either clients or servers depending on their current needs [Coe94]. Transportable agents do not require the maintenance of state information at both the local and remote machines and do not require a permanent connection between machines. This makes transportable agents more fault-tolerant and, in combination with their efficient use of network resources, makes them ideally suited to mobile computing [Whi94]. Transportable agents are a more natural fit for applications such as workflow, information filtering and network management in which processing must be performed on multiple machines *in sequence*. A transportable agent simply migrates through the machines in the desired order. Finally, transportable agents ease the development, testing and deployment of distributed applications since an application can dynamically distribute its components as it sees fit.

The advantages of transportable agents have led to a flurry of recent implementation work. The two most notable systems are Tacoma [JvRS95] and Telescript [Whi94, Whi95b, Whi95a]. Tacoma agents are written in Tcl/Horus which is a version of the Tcl scripting language that uses Horus to provide group communication and fault tolerance. The single abstraction in Tacoma is the *meet* operation which an agent uses to execute another agent. All other services are provided by *agents*. For example, an agent meets with the *ag_tcl* agent on a remote site in order to migrate to that site (a server at each site handles meeting requests). Tacoma, however, does not support the interruption of executing agents. The migrated agent executes from the beginning rather than the point of migration. This makes it difficult to write an agent that must preserve state information while migrating through a sequence of machines (but not impossible since state information can be explicitly collected and passed along with the code). In addition Tacoma provides no security mechanisms and its Horus component is unavailable on most platforms. Notable features of Tacoma include rear guard agents that restart lost agents, electronic cash that is used to pay for services and prevent runaway agents, and broker agents that provide scheduling and directory services.

Telescript is a General Magic product that is used in the AT&T PersonaLink^{TM} network. Telescript is an object-oriented language in which migration is viewed as the basic operation. Thus migration is reduced to a single instruction, *go*, which an agent issues whenever it wants to move to a new machine. The agent continues execution on the new machine from the statement immediately after the *go*. This transparent migration of internal state is more convenient than the *ag_tcl* agent of Tacoma. Telescript agents communicate by obtaining references to each other's objects if they are on the same machine and sending objects to each other if they are on different machines. A server at each site authenticates and executes incoming agents, enforces security constraints, handles object passing, and continuously backs up the internal state of agents in case of node failure. Unfortunately, Telescript is not available to researchers and has only been implemented on two Personal Digital Assistants (PDA). Most of the technical details of Telescript are unclear.

There are numerous other systems that exhibit aspects of transportable-agent behavior. The intelligent routers of [WVF89] move from machine to machine to accomplish a given task; the Safe-Tcl/MIME combination allows Tcl scripts to be embedded in electronic mail messages [Way95]; the HotJava browser allows Java scripts to be embedded in World Wide Web documents [Sun94]; a SodaBot application can dynamically distribute its components [Coe94]; and Postscript programs are often sent to remote displays. Only the intelligent routers provide arbitrary migration and only SodaBot provides arbitrary communication. The current status of the intelligent router work is unclear. Also notable are the object-oriented systems Obliq, SmallTalk Agents and IBM Intelligent Communications, each of which allows objects to dynamically move through a network [Car94, Way95, Rei94]. Like Telescript these systems are intrinsically tied to a specific programming language that is unnecessarily complex for many applications.

3 Architecture

Existing transportable agent systems suffer from one or more of the following weaknesses.

- Migration cannot occur at arbitrary points or requires the explicit capture of state information at the agent level.
- Communication between agents is nonexistent or difficult.
- Security mechanisms are nonexistent.
- Agents must be written in a specific and often complex language.
- Implementations only exist for nonstandard hardware.
- Portions of the implementation only run on specific Unix platforms.
- Source code is not available to the research community.

The goal of the Agent Tcl project at Dartmouth is to address these weaknesses. Agent Tcl should

- Reduce migration to a single instruction like the Telescript *go* and allow this instruction to occur at arbitrary points.
- Provide transparent communication among agents.
- Support multiple languages and transport mechanisms and allow the *straightforward* addition of a new language or transport mechanism.
- Run on general Unix platforms and port as easily as possible to non-Unix platforms.
- Provide effective security and fault-tolerance in the uncertain world of the Internet.
- Be available in the public domain.

The architecture of Agent Tcl is shown in Figure 2. The architecture builds on the server model of Telescript [Whi94], the multiple languages of Dixie [Gai94] and the transport mechanisms of two predecessors at Dartmouth [Har95, KK94]. The architecture has four levels. The lowest level consists of an API for each transport mechanism. The second level is a server that runs at each network site. The server must perform the following tasks.

- Keep track of the set of available interpreters.
- Keep track of the agents that are running on its machine and answer queries about their current status.
- Accept an incoming agent, authenticate the identity of its owner, and pass the authenticated agent to the correct interpreter.
- Provide a hierarchical namespace in which each agent has a unique name. The topmost division of the namespace specifies the network location of the agent.
- Allow agents to send messages to each other. The address of an agent is its name within the hierarchical namespace. 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 either requests or rejects the establishment of a direct connection. A direct connection is a named message stream between two agents. The first advantage of direct connections is programmer convenience since an agent can watch for messages on a particular connection rather than for any incoming message. The second advantage of direct connections is efficiency since, if allowed by the interpreter and transport mechanism, control of the connection is handed off to the interpreter, completely bypassing the server for all messages sent along the connection. The server's role in message passing is to select the appropriate transport mechanism for outgoing messages, buffer incoming messages, and create a named message stream once an agent accepts a connection request. The server passes control of the stream to the interpreter if possible; otherwise it buffers the stream messages as well.

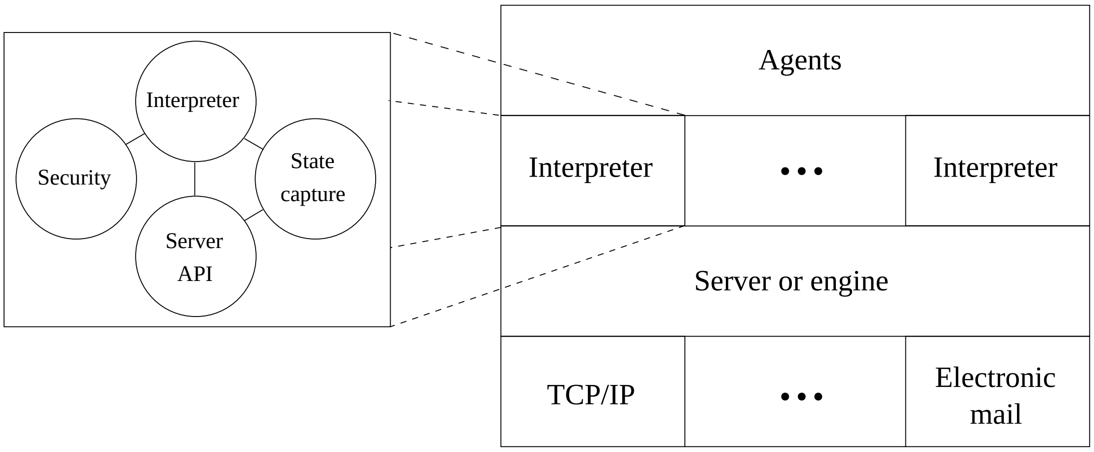


Figure 2: The architecture of Agent Tcl. The four levels consist of an API for each transport mechanism, a server that accepts incoming agents and provides agent communication, an interpreter for each supported language, and the agents themselves.

- Allow an agent to send itself or a child agent to a remote site. The server selects the appropriate transport mechanism for the outgoing agent.
- Provide access to a nonvolatile store so that agents can back up their internal state as desired.
- Restore agents from nonvolatile store in the event of node failure.

As in Tacoma all other services are provided by *agents*. Such services include planning, scheduling, dynamic blackboards, group communication, location-independent addressing and fault tolerance. With the addition of appropriate service agents, Agent Tcl can become the lowest level of more complex agent architectures such as Agent-0 [Sho93], KQML-based facilitators [GK94], the Open Agent Architecture [CCWB94] and the evolving Unified Agent Architecture [Bel95].

The third level of the Agent Tcl architecture consists of one interpreter for each available agent language. We say *interpreter* since it is expected that most of the languages will be interpreted due to security and portability constraints. 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. Some languages might allow the submission of child agents but not migration. These languages do not need the state module. Other languages such as C and C++ might support agent communication only. These languages do not need the security or state modules and can be compiled. The top level of the architecture contains the agents themselves.

4 Agent Tcl Version 1.0

Agent Tcl is far from complete but an alpha release is available [Gra95a]. The alpha release supports a single language (Tcl) and a single transport mechanism (TCP/IP). It provides migration, message passing, direct connections and rudimentary security. No service agents have been implemented and the namespace is *flat* rather than hierarchical. Here we briefly discuss Tcl and then the details of the alpha release.

4.1 Tcl

Tcl is a high-level scripting language that was developed in 1987 and has enjoyed enormous popularity [Ous94]. Tcl has several advantages as a transportable-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. We feel that it is critical to start with a simple, imperative language and explore the range of applications that such a language can support. 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 transportable Tcl agents. Finally, Tcl can be extended with user-defined commands which allows a resource to provide a package of Tcl commands that are used to access the resource. This is more efficient than encapsulating the resource within an agent and will be an attractive alternative in certain applications.

Tcl has several disadvantages however. Tcl is inefficient compared to most other interpreted languages [Ous95] and is ten thousand times slower than optimized C [SBD94]. In addition Tcl is *not* object-oriented and provides no code modularization aside from procedures. This makes it difficult to write and debug large scripts. Fortunately several groups are working on object-oriented extensions to Tcl and on faster Tcl interpreters [Sah94]. There are also efficient and structured alternatives to Tcl such as the new Java language [Sun94]. The lack of efficiency and structure has not been an issue so far since our agents are small and rely on existing tools at each site for intensive processing. As the agents grow in size, it will be necessary to consider an extended version of Tcl or a different language.

The final 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 was straightforward, but it required the modification of the Tcl core. The basic problem is that the Tcl core evaluates a script by making *recursive* calls to the main evaluation procedure Tcl_Eval. For example, the handler for the *while* command calls Tcl_Eval in order to evaluate the body of the loop. The solution was to add an explicit stack. The handlers are split 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 the next subhandler and for specifying when the command is finished and can be popped. Figure 3 illustrates this process for the *while* command.

The stack is not quite enough to handle command substitutions, but the details of command substitutions are beyond the scope of the paper. Once the stack was added and command substitutions were handled properly, it was trivial to write procedures that save and restore the internal state of a Tcl script. These procedures are the heart of the migration facilities.

4.2 Agent Tcl Version 1.0

The architecture of the alpha release is shown in Figure 4. The architecture has two components. The first component is the server that runs at each network site. The server is implemented as two cooperating processes. The first process is the *socket watcher* which watches a Unix socket for incoming agents, messages and requests. A message is either a generic message or a connection message. Events have not been implemented. Requests consist of asking for a name in the namespace, removing a name from the namespace, and getting the next available message. The namespace in the alpha release is flat rather than hierarchical. The second server process is the *agent tabler* which keeps track of the agents that are running on its machine and buffers incoming messages until the destination agent receives them. The *socket watcher* forks a handler for each incoming agent, message or request. The handler interacts with the *agent tabler* and takes the appropriate action.

The second component of the architecture consists of the modified Tcl core and a Tcl extension that provides the commands *agent_begin*, *agent_name*, *agent_submit*, *agent_send*, *agent_receive*, *agent_meet*, *agent_accept* and *agent_end*. Internally each command uses the server API to contact the server, transfer an agent, message or request, and wait for an acknowledgement. Here the main difference between the alpha release and the proposed architecture is that when migrating, creating a child agent or sending a message, the alpha release bypasses the local server and interacts directly with the destination server using

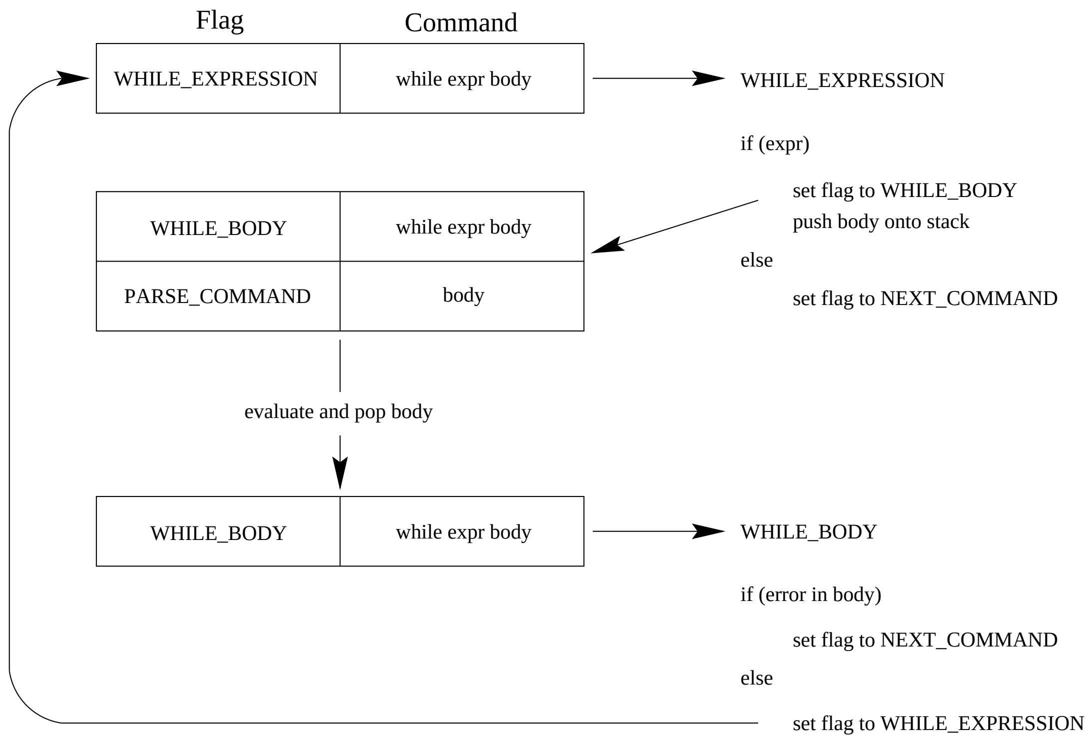


Figure 3: An example of how the stack works. The command stack is on the left and the two subhandlers for the *while* command are on the right. Tcl_Eval calls the first subhandler when the *while* command is first encountered. The first subhandler evaluates the loop expression, and if the expression is true, it pushes the body of the loop onto the stack. Tcl_Eval evaluates the body and then calls the second subhandler. The second subhandler checks for errors, and if no error has occurred, Tcl_Eval calls the first subhandler again in order to perform the next iteration of the loop. Note how each subhandler sets a flag that indicates which subhandler should be called next. The flag NEXT_COMMAND means that the command is finished and can be popped.

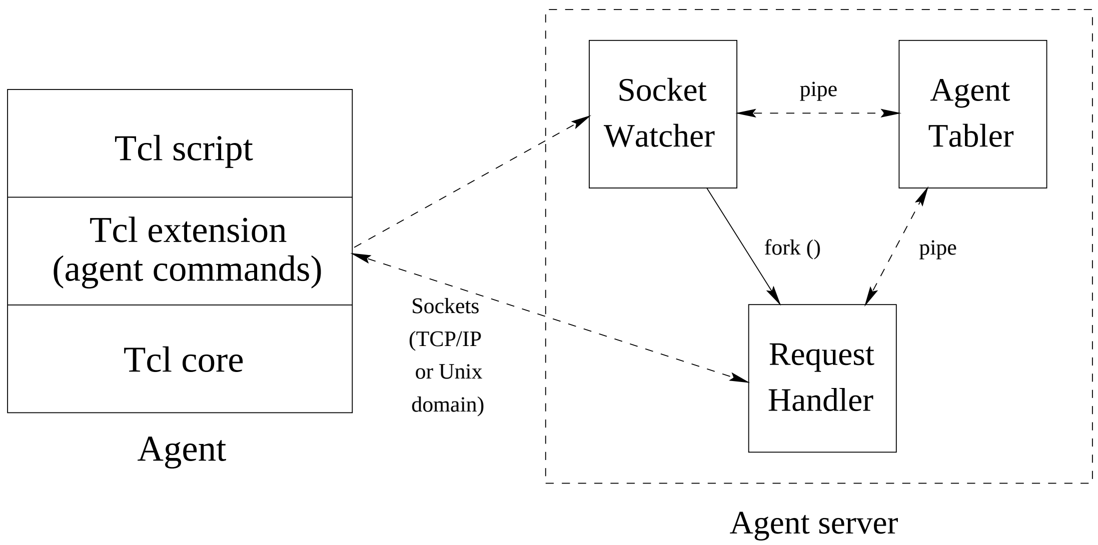


Figure 4: The architecture of the alpha release

TCP/IP. This approach was adopted to simplify the implementation and will change as additional transport mechanisms are added.

An agent is just a Tcl script that runs on top of the modified Tcl core. The agent uses the *agent_begin* command to obtain a name in the namespace. The *agent tabler* selects a unique name and returns this name to the agent. A name in the alpha release consists of the IP address of the server's machine, a unique integer and an optional string that the agent specifies with the *agent_name* command. The *agent_submit* command is used to create a child agent on a particular machine; *agent_submit* passes a Tcl script to the *socket watcher* on the destination machine. The *socket watcher* forks a handler which gets a name for the new agent from the *agent tabler* and then starts a Tcl interpreter to execute the agent. The *agent_jump* command migrates an agent to a particular machine; *agent_jump* captures the internal state of the agent and sends the state image to the *socket watcher* on the destination machine. The *socket watcher* forks a handler which gets a 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_send* command communicates with the recipient's server while *agent_receive* communicates with the agent's own server. The *agent_meet* and *agent_accept* commands are used to establish a direct connection. A direct connection is a named, message stream. Direct connections are not required for communication but are more efficient since they bypass the server. The *agent_meet* command requests a direct connection while *agent_accept* either accepts or rejects the connection. The two commands first exchange a round of messages as shown in Figure 5. The source agent uses *agent_meet* to send a connection request to the destination agent. The destination agent uses *agent_accept* to get the connection request and to send either an acceptance or rejection. An acceptance includes a TCP/IP port number. The source agent connects to that port on the recipient's machine. The two agents can then send arbitrary messages along the connection. The connection protocol will work even if the two agents simultaneously issue *agent_meet*. In this case the agent with the lower IP address and numeric id selects the TCP/IP port and the other agent connects to that port. The server will take on more of the responsibility for establishing a connection when additional transport mechanisms are added.

4.3 Example

Figure 6 shows an example agent. The agent submits a child agent that jumps from machine to machine and executes the Unix *who* command on each machine. The child returns the list of users to the parent which then displays the list to the user. The "who" agent illustrates the general form of any agent that migrates through a sequence of machines and highlights the *agent_jump* command which captures and transfers an agent's complete internal state.

5 Applications

We are using Agent Tcl in a range of information-management applications. The first application is an "alert" agent that monitors a specified set of remote resources and notifies its owner of any change in resource status. The agent notifies its owner via electronic mail. Figure 7 shows an "alert" agent that monitors a set of files and notifies the user if the size of a file changes significantly. The agent creates one child agent for each remote filesystem. Each child monitors one or more directories in its filesystem and sends a message to the parent when the size of a file changes significantly. The parent then contacts the user's "mail" agent to send the email message.

Agent Tcl has also been used in three information-retrieval applications. The first application involves technical reports [Cai95]; the second involves text-based medical records [Wu95]; and the third involves three-dimensional drawings of mechanical parts [BC95]. In all three cases the "documents" are distributed across a small network. An agent is sent to each network site. Each agent finds the relevant documents at its site and returns the relevant documents to the home site for fi-

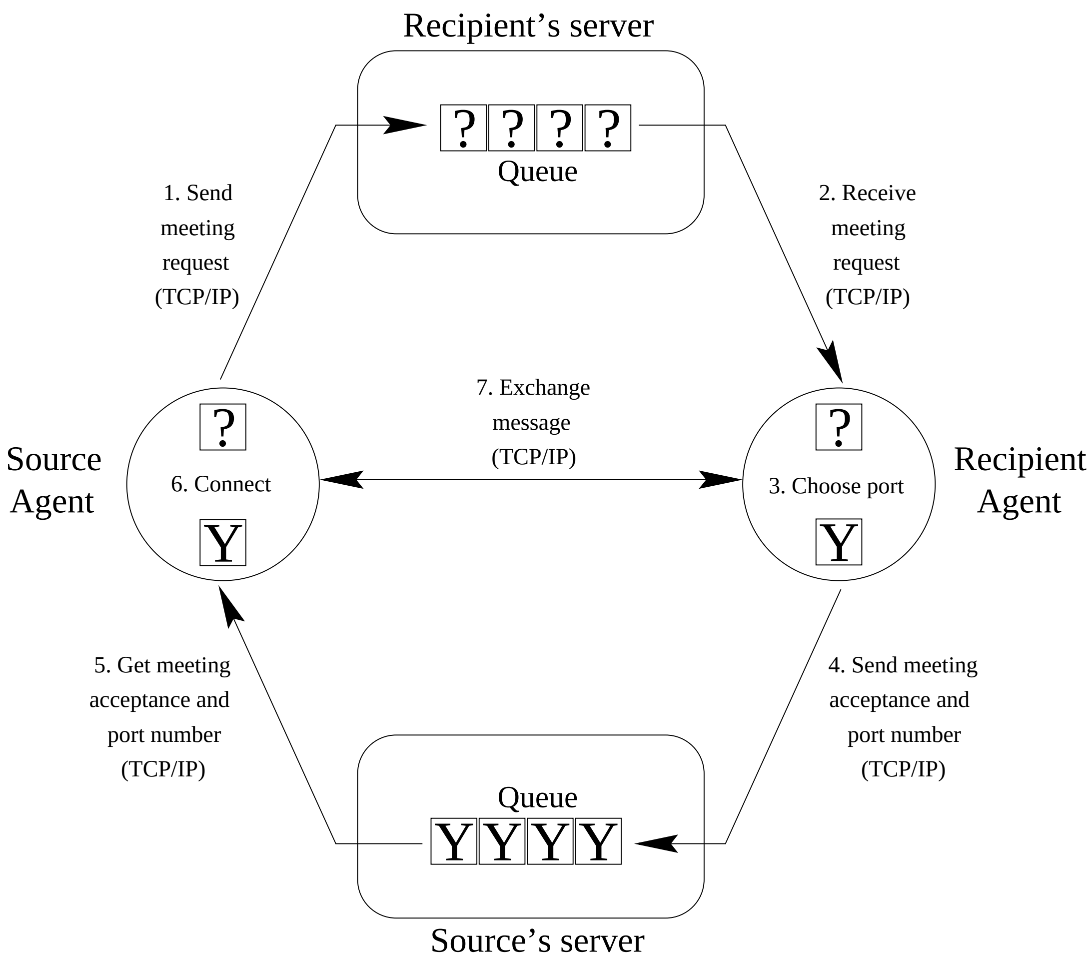
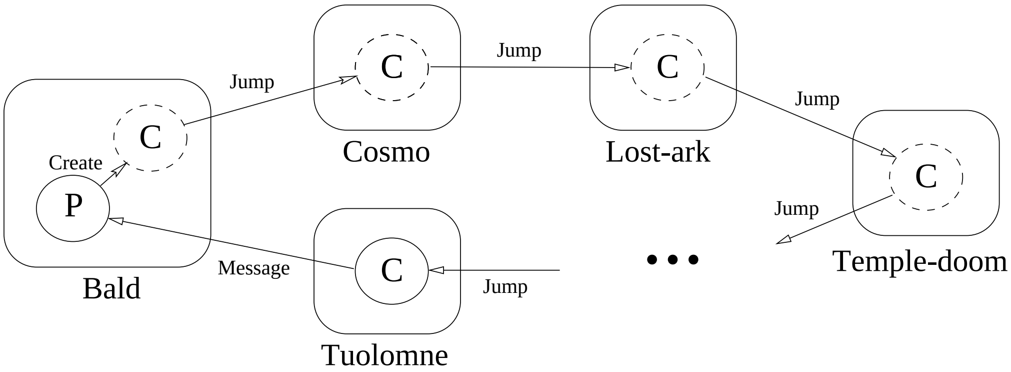


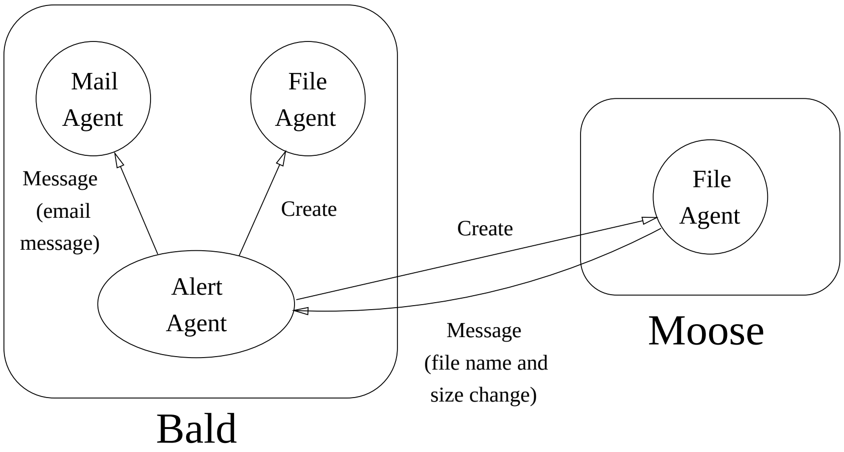
Figure 5: The protocol for establishing a direct connection. Here the source agent issues the *agent_meet* command while the recipient agent issues the *agent_accept* command. The *agent_accept* command can either block and wait for a connection request or can poll and return immediately. A *?* indicates a connection request and a *Y* indicates a connection acceptance.




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

proc who machines {

  global agent

  set list ""

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

  foreach m $machines {

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

  return $list
}

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

  # get a name from the server

agent_begin

  # submit the child agent that jumps

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

  # wait for and output the list of users

agent_receive code string -blocking
puts $string

  # agent is done

agent_end
```

```
bald.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 6: 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 a
  # file has changed size; then send an alert message to the user by
  # asking the user's email agent to send a message to its owner

while {1} {

  agent_receive code string -blocking
  set alert [construct_alert $string]
  agent_send $email_agent {SEND OWNER $alert}

}
```

Figure 7: The "alert" agent monitors a set of files and sends an email message to the user when the size of a file changes significantly. A simplified version of the "alert" agent appears at bottom. The network location of the various agents is shown at top.

nal processing and display. Each site provides a small set of retrieval primitives that can be combined into complex queries.

Transportable agents provide efficient execution in all of these applications even though the desired operations are not provided at the remote sites and must be built up from low-level primitives. This is particularly important for the medical records and mechanical parts since the queries are complex and varied and it is unreasonable to expect a remote archive to support all possible queries as atomic operations. In addition transportable agents led to extremely short development times since no application-specific code had to be installed at the remote sites (all of the necessary primitives were already available for local use).

6 Future work

The first phase of future work is to implement the remainder of the proposed architecture and certain low-level services that are critical in a production-quality system [Gra95b]. Eight components must be implemented. The first three are primarily programming tasks while the last five address open research issues.

- *Events.* An *event* provides asynchronous notification of an important occurrence. Events are needed to support interrupts and event-driven languages such as Tcl/Tk.

- *Multiple languages and transport mechanisms.* We plan to incorporate (1) a functional or declarative language such as Lisp or Scheme to complement the imperative language Tcl and to provide better support for applications in the artificial intelligence community, and (2) an electronic-mail transport mechanism to support Personal Digital Assistants (PDA). In addition it might become necessary to include a more efficient or structured imperative language.

- *Hierarchical namespace.* A hierarchical namespace will prevent naming conflicts and will provide support for higher-level services since we plan to follow the example of Telescript and allow the administrator or developer to associate a distinguished agent with each node in the hierarchy. The distinguished agent is notified of important events that occur within its portion of the namespace. A typical event would be an agent attempting to enter the namespace. The event is allowed only if the distinguished agent approves.

- *Nonvolatile store.* The nonvolatile store backs up the internal state of executing agents in case of node failure. The problem is to provide sufficient fault tolerance while maintaining efficient execution. We need an incremental backup mechanism that is used only at "key" points within each agent.

- *Security.* There are two levels of security. The first level involves the standard point-to-point issues of authenticating the sender of an agent and preventing the agent from performing malicious actions. The second level involves all of the security holes that appear when an agent migrates through a *sequence* of machines. For example, if an agent migrates from its home machine to machine A and then to machine B, machine B must be able to authenticate the *original* sender of the agent and must be able to verify that machine A did not modify the agent in a malicious way.

- *Privacy.* An agent might carry sensitive information about its owner in order to make effective decisions. This information must be hidden from other agents and from malicious machines. The latter appears impossible so some information must always remain at the home site.

- *Network awareness.* An agent should be able to discover the current state of the network and use this information to select a migration strategy that takes it through the required resources as quickly as possible while maintaining application constraints.

- *Track moving agents.* This is a lower-level issue than resource discovery. Here we are concerned with the fact that an agent should be able to transparently continue communicating with a second agent even if the second agent changes its network location. We hope that the distinguished agents will form the basis for this functionality. For example, if we notify a distinguished agent whenever a message is sent to a *nonexistent* name in its portion of the namespace, it becomes straightforward to implement virtual names. The distinguished agent would forward all messages that are sent to a moving agent's virtual name.

The second phase of future work is to identify the higher-level services such as planning and scheduling that are required in many applications and to implement agents that provide these services.

7 Conclusion

We have described a transportable-agent system called Agent Tcl that will address the main weaknesses of existing systems. Agent Tcl will run on standard hardware, support multiple languages and transport mechanisms, provide transparent migration and communication, and provide effective security and fault tolerance in the uncertain world of the Internet. 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 1.0 is available at `http://www.cs.dartmouth.edu/~rgray/transportable.html`.

Acknowledgements

Many thanks to my advisor, Professor George Cybenko, and to Professor David Kotz and Professor Daniela Rus for reading the various incarnations of this paper and providing helpful criticism; to Ting Cai, Yunxin Wu, Aditya Bhasin and Kurt Cohen for implementing the information-retrieval agents; and to the Air Force and Navy for their gracious financial support (AFOSR contract F49620-93-1-0266 and ONR contract N000014-95-1-1204).

References

[BC95] Aditya Bhasin and Kurt Cohen. A distributed database of mechanical parts. Technical report, Thayer School of Engineering, Dartmouth College, 1995. In progress.

[Bel95] Marc Belgrave. The Unified Agent Architecture: A white paper. Available at `http://www.ee.mcgill.ca/~belmarc/agent_root.html`, 1995.

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

[Cai95] Ting Cai. A technical report agent. Technical report, Department of Computer Science, Dartmouth College, 1995. In progress.

[Car94] Luca Cardelli. Obliq: A language with distributed scope. Digital White Paper, Digital Equipment Corporation, Systems Research Center, 1994.

[CCWB94] Phillip R. Cohen, Adam Cheyer, Michelle Wang, and Soon Cheol Baeg. An open agent architecture. In *Proceedings of the AAAI Spring Symposium*, 1994.

[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.

[Fal87] Joseph R. Falcone. A programmable interface language for heterogeneous systems. *ACM*

Transactions on Computer Systems, 5(4):330–351, November 1987.

[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.

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

[Gra95a] Robert S. Gray. *Agent Tcl: Alpha Release 1.0*, 1995. Available at `http://www.cs.dartmouth.edu/~rgray/transportable.html`.

[Gra95b] Robert S. Gray. Ph.d. Thesis Proposal: Transportable agents. Technical Report PCS-TR95-261, Department of Computer Science, Dartmouth College, 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.

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

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

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

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

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

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

[Sho93] Yoav Shoham. Agent oriented programming. *Journal of Artificial Intelligence*, 1993.

[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.

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

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

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

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

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

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

[Wu95] Yunxin Wu. Retrieval and organization of distributed medical records. Technical report, Thayer School of Engineering, Dartmouth College, 1995. In progresss.

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