AGENT TCL: Targeting the Needs of Mobile Computers

DAVID KOTZ, ROBERT GRAY, SAURAB NOG, DANIELA RUS, SUMIT CHAWLA, AND GEORGE CYBENKO
Dartmouth College

Agent Tcl accommodates mobile computers with features like laptop docking, which lets an agent return to a periodically disconnected machine.



Mobile computers have become increasingly popular as users discover the benefits of having their electronic work available at all times. Using Internet resources from a mobile platform, however, is a major challenge. Mobile computers do not have a permanent network connection and are often disconnected for long periods. And when the computer *is* connected, the connection is often prone to sudden failure, such as when a physical obstruction blocks the signal from a cellular modem. In addition, the network connection often performs poorly and can vary dramatically from one session to the next, since the computer might use different transmission channels at different locations. Finally, depending on the transmission channel, the computer might be assigned a different network address each time it reconnects.

Mobile agents are one way to handle these unforgiving network conditions. A mobile agent is an autonomous program that can move from machine to machine in a heterogeneous network under its own control. It can suspend its execution at any point, transport itself to a new machine, and resume execution on the new machine from the point at which it left off. On each machine, it interacts with service agents and other resources to accomplish its task, returning to its home site with a final result when that task is finished. The sidebar "Why Mobile Agents?" describes the motivations for and benefits of these agents in more detail.

Agent Tcl is a mobile-agent system whose agents can be written in Tcl, Java, and Scheme, although the version available to the public supports only Tcl at present. Agent Tcl has extensive navigation and

1089-7801/97/$10.00©1997 IEEE

<http://www.cs.dartmouth.edu/~dfk/papers/kotz:jmobile.pdf>.
IEEE Internet Computing, 1(4):58-67, July/August 1997.

WHY MOBILE AGENTS?

Mobile agents have several strengths. By migrating to the location of a needed resource, an agent can interact with the resource without transmitting any intermediate data across the network, significantly reducing bandwidth consumption in many applications. Similarly, by migrating to the location of a user, an agent can respond to user actions rapidly. In either case, the agent can continue its interaction with the resource or user even if the network connection goes down. These features make mobile agents particularly attractive for mobile-computing applications.

Mobile agents let traditional clients and servers offload work to each other, and *change* who offloads to whom according to machine capabilities and current loads. Similarly, mobile agents allow an application to dynamically deploy its components to arbitrary network sites, and to *re*deploy those components in response to varying network conditions.

Finally, most distributed applications fit naturally into the mobile-agent model, since mobile agents can migrate sequentially through a set of machines, send out a wave of child agents to visit multiple machines in parallel, remain stationary and interact with resources remotely, or perform any combination of these three extremes. Complex, efficient, and robust behaviors can be realized with surprisingly little code. Our own experience with undergraduate programmers at Dartmouth suggests that mobile agents are easier to understand than many other distributed computing paradigms.

Although each of these strengths is a reasonable argument for mobile agents, any specific application can be implemented just as efficiently and robustly with more traditional techniques, such as queued RPC, high-level query languages, dedicated proxies within the network, automatic installation facilities, and Java applets.[1] However, mobile agents eliminate the need for these other techniques, combining their strengths into a single, general, and convenient framework. Distributed applications can be implemented efficiently and easily—even if they must exhibit extremely flexible behavior in the face of changing network conditions. For example, a search application can migrate to a dynamically selected proxy site and do its merging and filtering there, while a server can continually migrate to new machines to minimize the average latency between itself and its clients.[2]

In short, the true strength of mobile agents is that they are a uniform paradigm for distributed applications. Thus, all existing systems—Agent Tcl, Telescript,[3] Odyssey,[4] IBM Aglets,[5] and Sumatra[2]—are intended for general applications, differing only in their languages, migration and security models, and support services. Agent Tcl distinguishes itself by combining a true jump instruction (one that automatically captures the entire program state); support for multiple languages; simple but effective security mechanisms; and significant navigation, communication, and debugging tools.

REFERENCES

1. D. Chess et al., "Itinerant Agents for Mobile Computing," *IEEE Personal Comm.*, Oct. 1995, pp. 34-49.
2. M. Ranganathan et al., "Network-Aware Mobile Programs," *Proc. Usenix Tech. Conf.*, Usenix, Berkeley, Calif., 1997, pp. 91-104.
3. J.E. White, "Mobile Agents," in *Software Agents*, J.M. Bradshaw, ed., MIT Press, Cambridge, Mass., 1997, pp. 437-472.
4. http://www.genmagic.com/agents/
5. http://www.trl.ibm.co.jp/aglets/

communication services, security mechanisms, and debugging and tracking tools. In this article we focus on Agent Tcl's architecture and security mechanisms, its RPC system, and its docking system, which lets an agent move transparently among mobile computers, regardless of when they are connected to the network.

Agent Tcl is being used in experimental projects at numerous academic and industrial research laboratories, including labs at Lockheed Martin, Siemens, Cornell University, and the University of Bordeaux, and has begun to find its way into production-quality applications as well. The public release provides migration, low-level communication, and security mechanisms for protecting a machine against malicious agents. The internal version includes the docking and RPC systems, the debugging tools, additional security features, and support for Java and Scheme agents. The new components in the internal version will be available in fall 1997. The current public release and all Agent Tcl publications are online (http://www.cs.dartmouth.edu/~agent).

OVERVIEW

Like all mobile-agent systems, the main component of Agent Tcl is a server that runs on each machine. When an agent wants to migrate to a new machine, it calls a single function, agent_jump, which automatically captures the complete state of the agent and sends this state information to the server on the destination machine. The destination server starts up an appropriate execution environment (for example, a Tcl interpreter for an agent written in Tcl), loads the state information into this execution environment, and restarts the agent from the exact point at which it left off. Now the agent is on the destination machine and can interact with that machine's resources without any further network communication. In addition to reducing migration to a single instruction, Agent Tcl has several important features:

- *Simple architecture.* The simple, layered architecture supports multiple languages and transport mechanisms. The main language is Tcl. The main transport mechanism is TCP/IP.

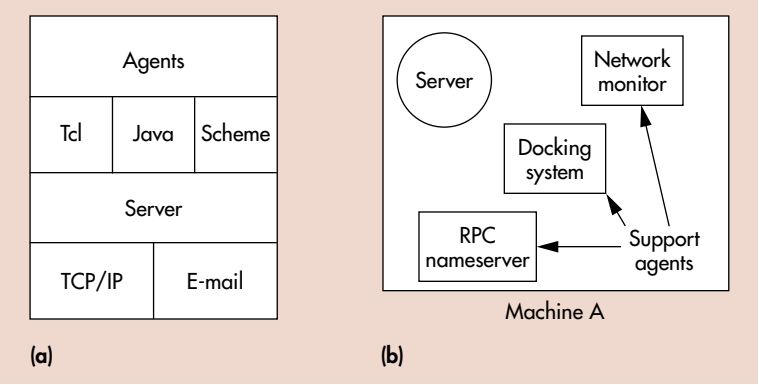

Figure 1. The architecture of Agent Tcl. (a) **The core system has four levels: transport mechanisms, a server that runs on each machine, an interpreter for each supported agent language, and the agents themselves.** (b) **Support agents provide navigation, communication, and resource management services to other agents.**

- *Security.* Agent Tcl protects individual machines against malicious agents—agents that try to access or destroy restricted information or consume too many machine resources. It also protects groups of machines controlled by a single organization.
- *Docking system*. The docking system lets an agent transparently jump off a partially connected computer (such as a mobile laptop) and return later, even if the computer is connected only briefly.
- *Interagent communication*. Agents communicate with either low-level mechanisms (message passing and streams) or high-level mechanisms (RPC) that are implemented at the agent level atop the lower level mechanisms. All communication mechanisms work the same whether or not the communicating agents are on the same machine.

ARCHITECTURE

As Figure 1 shows, Agent Tcl's architecture has a four-level core system and an agent-level support system.

Core System

At the lowest level of the core system, Figure 1a, is an interface to each available transport mechanism. The next level is the server that runs on each machine. The server keeps track of the agents running on its machine, provides the low-level interagent communication facilities (message passing and streams), receives and authenticates agents arriving from another host, and restarts an authenticated agent in an appropriate execution environment.

The third level of the architecture consists of the execution environments, one for each supported agent language. Agent Tcl supports Tcl, Java, and Scheme, so its execution environments are simply a Tcl interpreter (Tcl 7.5), a Scheme interpreter (Scheme 48), and the Java virtual machine (Sun JDK 1.2). For each incoming agent, the server starts up the appropriate interpreter.

The last level of the architecture comprises the agents themselves, which execute in the interpreters and use the facilities provided by the server to migrate from machine to machine and to communicate with other agents. Agents include both moving agents, which visit different machines to access needed resources, and stationary agents, which stay on a single machine and provide a specific service to either the user or other agents. From the system's point of view, there is no difference between the two kinds of agents, except that a stationary agent has authority to access more system resources.

To add a language to Agent Tcl, programmers simply extend the interpreter to provide

- state-capture routines that capture and restore the state of an executing program, and
- an interface to the agent servers.

In the Tcl interpreter, for example, the state-capture routines capture and restore all defined variables and procedures, the procedure-call stack, and the control stack. The interface to the servers is a set of Tcl commands, such as agent_begin and agent_jump, which are provided as a standard Tcl extension. The agent_jump command calls the state-capture routines to capture and restore the state of an executing Tcl agent. Similarly, in Java, the state-capture routines capture and restore the state of a single Java thread (including all accessible objects). The interface to the servers is a special Java class. Finally, in Scheme, the state-capture routines capture and restore the current continuation (the rest of the program), and the interface to the servers is a set of Scheme functions.

Most of the interface between the interpreters and the servers is implemented in a C/C++ library and shared among all interpreters. The language-specific portion is just a set of stubs that call into this library.

Agent-Level Support

The agent servers provide low-level functionality. As Figure 1b shows, all other services are provided at the agent level by dedicated service agents. Such services include navigation, high-level communication protocols, and resource management. Both the docking system and Agent RPC (described later) are implemented entirely at the agent level.

Sample Agent

Figure 2 shows a simple Agent Tcl agent written in Tcl. The agent's task is to make a list of all users logged onto some Dartmouth machines and then show this list to its owner. The agent has several important parts:

- agent_begin. The agent registers with Agent Tcl through the server on its home machine (Bald).
- agent_jump $machine. The agent migrates sequentially through the machines of interest (Muir, Tenaya, and others not shown). It continues executing from the point of the jump on each successive machine. On each machine, the agent executes the Unix who command (exec who) to obtain the user list.
- agent_jump $agent(home). Once the agent has migrated through all the machines, it migrates one last time to return to Bald.
- # display results. Once on Bald, the agent displays the complete user list to its owner (not shown).
- agent_end. The agent tells Agent Tcl that it has finished.

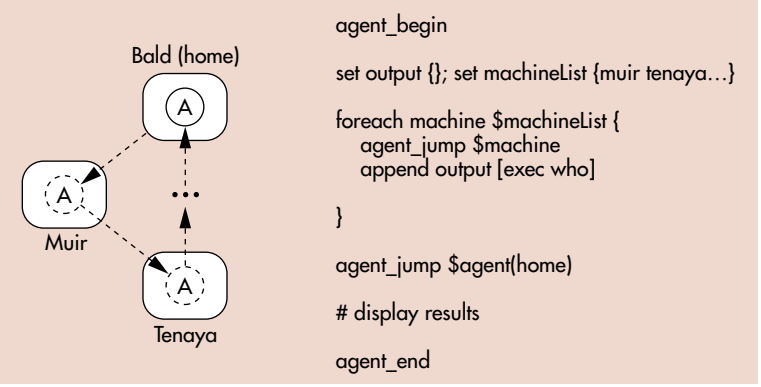

Figure 2. A simple Tcl agent that figures out which users are logged onto some set of machines. Bald, Muir, and Tenaya are machines at Dartmouth College. The agent starts on Bald.

Although this agent performs a simple task, any agent that migrates sequentially through one or more machines has the same general form. The exec who command can be replaced with any desired local processing. Some agents will need learning and reasoning capabilities. Agent Tcl does not provide such capabilities directly, but an agent is just a program written in Tcl, Scheme, or Java, so it can include and use any existing libraries.

SECURITY

Security, of course, is a critical issue in any mobile-code system. Agent Tcl currently protects machines from malicious agents (both individual machines and groups of machines under single administrative control), but does not protect agents from malicious machines. We give a brief summary of our current implementation here; a detailed description and our future plans are given elsewhere.[1,2]

The security mechanism has three major features:

- Agents and messages sent between machines are encrypted, which maintains agent privacy.
- Agents and messages sent between machines are signed, which authenticates the agent to the new host.
- Resource managers control access to system resources.

Each resource (CPU, memory, file system, screen, network, and so on) has a stationary agent that acts as a manager. For Tcl agents, each visiting agent is run in an *untrusted* Tcl interpreter, and all resource accesses are trapped into a separate, *trusted* interpreter. The trusted interpreter asks the appropriate resource manager to determine if the visiting agent should have access to the resource. For example, the memory manager might limit an agent to 100 Kbytes of memory. The trusted interpreter then enforces the manager's policy decision, either proceeding with the resource access or throwing a security exception back to the untrusted interpreter.

In addition to absolute limits on resource use, we plan to use a currency-based model in which agents purchase resources from the managers, thus limiting their total resource use even across administrative domains.[2] The prices will vary according to supply and demand and the changing priorities of agents and servers.

The same resource managers are used for all agents; only the enforcement mechanism differs among agent languages. In Java, for example, a special security manager class contacts the resource managers and enforces the policy decisions, rather than a separate trusted interpreter.

DOCKING SYSTEM

When a mobile agent tries to return to its home machine with final results, the machine might be disconnected. Thus, the agent must have some way to determine when the home machine reconnects. A simple approach is polling (try, timeout, sleep, try again, and so on), but polling wastes network resources and will fail outright if the home machine reconnects for only brief periods. For this reason, we devised a docking system, Figure 3, that pairs each laptop computer ("laptop" meaning any mobile device) with a permanently connected *dock* machine. When a mobile agent is unable to migrate to a laptop (laptopX), it waits at the laptop's dock machine (laptopX_dock). When the laptop reconnects, it notifies the dock machine of its new network address, and the dock machine forwards all waiting agents.

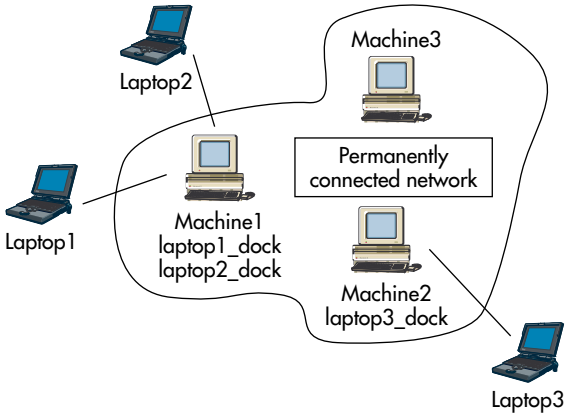

Figure 3. Docking system. Each laptop is paired with a permanently connected machine, where agents wait for the laptop to reconnect. Here "laptop" refers to any partially connected machine.

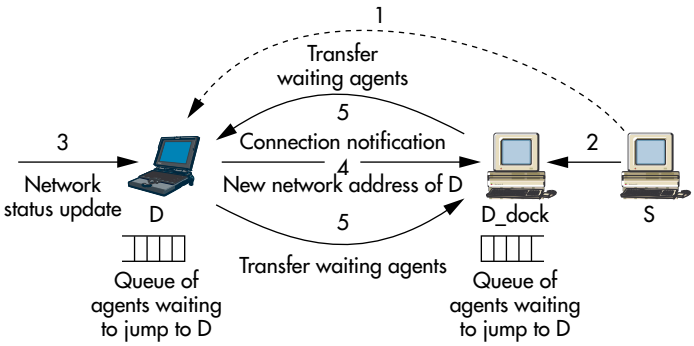

Figure 4. Jumping to or from a laptop.

Although only certain machines act as docks, all machines have a *dock master,* an agent that maintains a queue of waiting agents. The queue is always stored on disk rather than in memory. On the dock machine, the queue contains the agents waiting to visit the laptop; on the laptop itself, the queue contains the agents waiting to leave the laptop.

Application

Figure 4 illustrates how the docking system works in an application. The figure depicts the following sequence of events (numbers in parentheses correspond to the numbers in the figure). An agent wants to jump from a source machine *S* to a disconnected destination laptop *D,* so it executes the command agent_jump~D. The agent_jump command tries but fails to contact laptop *D* directly (1). Once the agent_jump command discovers that *D* is disconnected, it contacts the dock master on *D*'s dock machine *D*_dock and transfers the agent to this dock master (2). The dock master adds the agent to the queue of agents waiting to jump to *D.* When *D* reconnects to the network (3), the agent system on *D* detects the reconnection and notifies the dock master on *D*_dock (4). The dock master on *D*_dock transfers all waiting agents to *D,* where they resume execution (5). If *D* has changed its network address, the new address is included in the notification, so that waiting agents can be transferred to the new address. Agents trying to reach *D* at the old address will fail, jump to *D*_dock, and eventually reach *D* at its new address.

If the agent is trying to *leave* the disconnected laptop *D,* it again executes the agent_jump command, which detects that the laptop is disconnected, saves the state of the agent to disk, and informs the local dock master. The local dock master continually monitors network status, and when the laptop reconnects to the network, the dock master transfers the waiting agent to the desired destination.

A more complex case is when both the agent's source *S* and destination *D* are laptops. The two laptops might never be connected to the network at the same time. If *S* is disconnected, the dock master on *S* saves the agent's state on disk. When the dock master on *S* detects network reconnection, it tries to transfer the agent to *D*. If *D* is unreachable, it tries to transfer the agent to *D*'s dock. If *D*_dock is unreachable, perhaps because of a temporary problem on the Internet, the dock master on *S* tries to transfer the agent to *S*_dock. If *S*_dock is also unreachable, the dock master will try the entire process again at a later time. If *S*_dock can be reached, the agent is sent to *S*_dock, and the dock master on *S*_dock will periodically attempt to transfer the agent to either *D* or *D*_dock. The agent may reside at *D*_dock until *D* connects and notifies the dock master at *D*_dock of *D*'s new location. Once at *D*, the agent continues executing.

Multidestination Jumps

We are extending our docking system to support multidestination jumps, which are useful when an agent wants to visit multiple hosts (D_1, D_2, …, D_N) but in no particular order. The agent may be searching all sites for information or visiting one of a replicated set of servers. The dock master on *S* first tries to transfer the agent to one of the final destinations by trying each in order (D_1, D_2, ..., D_N). If all destinations are unreachable, the *S* dock master transfers the agent to *S*_dock. The dock master at *S*_dock periodically tries to reach the destinations until one of the transfers succeeds. *S*_dock does not transfer the agent to a dock machine $D_K$dock, so that it does not prematurely commit to a destination that may rarely connect (although this issue is an open research topic). When the agent awakes (returns from its call to agent_jump), it checks its actual destination and proceeds with its task.

For agents that desire more control over the jumping process, we provide hooks to allow agents to query the status of the current machine's network connection, to request a failure notification rather than being blocked when a destination cannot be reached immediately, and to request that the jump go as far toward the destination as possible and then wake up the agent.

Performance

To determine the docking system's overhead, we measured the time needed for an agent to jump onto a laptop from a nearby host. The laptop was a 66-MHz Intel 486 running Red Hat Linux; the nearby host was a 100-MHz Intel Pentium running FreeBsd 2.1; the two machines were connected via a 10-Mbps Ethernet (with no intervening routers). In one set of experiments, the laptop was still connected, and the agent jumped directly onto it. In the second set of experiments, the laptop was still connected, but we forced the agent to go through the laptop's dock machine. Under normal operation, of course, the agent goes through the dock machine only if the laptop is disconnected, but forcing the agent to go through the dock was the easiest way to measure the docking overhead.

When the agent was carrying 8 Kbytes of code and data, a direct jump onto the laptop took 0.3 second, due mainly to the cost of starting up a new Tcl interpreter for the incoming agent. An indirect jump (agent going through the dock machine) took 1.6 seconds. The extra time came from the need to connect to the dock master, transfer the agent, save the agent to disk, and get the agent off the disk for transmission to its final destination. In addition, all agents are currently written in Tcl, which is slower than most other interpreted languages. Rewriting the dock masters in Java and providing a pool of ready interpreters will reduce these times significantly because a pool of ready interpreters eliminates the need to start a new interpreter for each incoming agent.

Benefits and Limitations

The docking system has several advantages. The agents depart from or arrive at the laptop as soon as possible and do not miss any transmission opportunities (because there is no polling). In addition, because waiting agents are saved on disk, they survive machine crashes and do not occupy precious memory and CPU time. Finally, all the state of a waiting agent has already been captured, so the agent is ready for transfer as soon as the network is connected.

On the down side, if an agent is *running* on a machine when the machine goes down, the agent is lost. If an agent is running on a machine, and the machine becomes disconnected from the network for a long period, the agent remains in exile. Finally, the dock for a given host named X.domain is the host named X_dock.domain. Although this allows immediate identification of a machine's dock, it also means that the machine must have a permanent name, even if the host gets a new network address at every machine restart. We are working to address these disadvantages.

INTERAGENT COMMUNICATION

Agent Tcl provides message passing and byte streams at its lowest level. Higher-level communication mechanisms, which make many applications much easier, are implemented at the agent level using message passing or streams. Our most important high-level mechanism, Agent RPC,[3] is similar to traditional RPC.[4] Agent RPC lets two agents communicate through the procedure-call abstraction. The agents can be on the same or different machines, but usually will be on the same machine, since most client agents jump to the same location as the desired service.

Programmers using Agent RPC begin by writing an interface in AIDL (Agent Interface Definition Language). The interface specifies the procedures a server agent provides to its clients. The programmer presents the AIDL specification to a stub compiler, which generates the Tcl procedures (called stubs) that let the client and server agents communicate. The client and server agents simply include these stubs with their application-specific code. In addition to accepting client requests, the server stubs register the server with a *nameserver agent*, which client agents consult when searching for a needed service.

Although this basic structure is no different from that of traditional RPC systems, Agent RPC offers two unique advantages:

- *Flexible interface language*. AIDL allows both default and position-independent parameters in interface definitions.
- *Client-server bindings*. Bindings (or connections between compatible client and server agents) are based on interface matching rather than on names. Thus, a client agent can obtain the desired service from any server that supports the appropriate interface, rather than only servers that have a particular service name and version. In addition, a client agent can have multiple, simultaneous bindings to the same or different servers. Finally, a server agent can accept or reject a bind request according to any security information the agent system provides, such as the authenticated identity of the client agent's owner.

AIDL and Bindings

AIDL aims to support extensibility and flexible matching. To illustrate, we present a running example that begins with the interface definition:

```
{constant {version 1} {service travel_agent}
                      {category airlines}}
{list_flights      {{Origin CityCode}
```

```
                {Destination CityCode}}
                {flights flight_list}}
{buy_ticket     {{payment_form sale_type credit_card}
                {flight flight_number}}
                {success boolean}}
{refund_ticket  {{flight flight_number}}
                {success boolean}}
```

The constants convey the version number for this server (version 1), its service type (travel_agent), and its service specialization (airlines). The names of these constants are not meaningful to the stub compilers or the nameserver, but they are meaningful to the client and server agents.

Following the constants are three procedure definitions, list_flights, buy_ticket, and refund_ticket. Each procedure has a list of parameters and a return value. Each parameter and return value has a name, a type, and an optional default value.

The named parameters and the default values make interface matching more flexible. The process has the following steps:

- A server agent provides the AIDL description of its interface to the nameserver.
- A client agent provides an AIDL description of the interface it needs.
- The nameserver looks for a match between the client's desired interface and the interfaces the servers support. Two interfaces match if the server's AIDL matches all the functions and constants described in the client's AIDL. Two constants match if they have the same name and value (or if they have the same name and the server's value falls within the client's set of values). Two function descriptions match if they have the same function name, same parameter names and types (in any order), and same return name and type. If the server function provides a default value for a parameter not mentioned in the client's function description, the functions still match. If the server provides additional functions, the interfaces still match.

Thus, a client searching for the interface defined below will find all servers that support the interface defined earlier:

```
{constant {category airlines} {service travel_agent}}
{list_flights   {{Destination CityCode} {Origin CityCode}}
                {flights flight_list}}
{buy_ticket     {{flight flight_number}}
                {success boolean}}
```

In this example, the client does not care about the version number, uses the default value for the sale_type parameter to the buy_ticket function, and does not need the refund_ticket function. The client will find all servers that have a different version number, all servers whose buy_ticket function does not even have a sale_type parameter, and all servers that provide only the list_flights and buy_ticket functions. In other words, a client can find and use all servers that provide the functions it needs, regardless of whether those servers provide additional functions as well.

This flexibility is important in the dynamic world of the Internet, where clients and servers are not implemented by the same parties, where older or simpler clients must interact with newer or more complex servers, and so forth. It is easy to add more functions, constants, or parameters to a server and still support clients that expect an older, simpler interface.

The stub compiler compacts and sorts the interface definitions, so that the constants, the functions, and the parameters within each function are sorted by name. This sort lets the nameserver quickly compare two interfaces for a match. The generated client stubs pack the parameters (into a byte stream) in sorted order, and the generated server stubs unpack the parameters in sorted order. Thus, there is no explicit sorting step, which saves considerable time.

Performance

We measured the performance of Agent RPC using two machines: Bald, a 200-MHz Intel Pentium running Linux version 2.0, and q, a 100-MHz Intel Pentium running FreeBsd version 2.1. The two machines were connected with a 10 Mbps Ethernet and one intervening router. The server agent was always on Bald. The client agent was on either machine. For both client locations, we measured the end-to-end wall clock time for a remote procedure call that had a single parameter, an empty server procedure, and no return data. We repeated the experiments for various parameter sizes.

Table 1 presents the average timing results. The first column is the size of the single parameter. The second column is the total time needed for the RPC call. The third column shows the time needed for the client stub to pack the procedure parameters into a byte stream. The fourth column is the time needed for the server stub to unpack the parameters, invoke the actual procedure, and pack the (void) return value. The last column is the percentage of time actually spent transmitting data from one agent to the other. As expected, this percentage is significantly smaller when the two agents are on the same machine. The total RPC time is just the sum of the client stub, server stub, and communication times.

Communication time (last column) shows the time to make a local procedure call (one in the same program) with the same data. Because Tcl is inherently slow, this measure is a good baseline for evaluating results with remote procedure calls. When the client and server agents are on the same machine, and when the parameter size is zero, the remote procedure call takes 200 times longer than the local call. In all other cases, it takes 40 to 140 times longer than a local

Table 1. Agent RPC performance. All times are in milliseconds, and are averages of more than 1,000 trials. The processors passed the data bytes as a single parameter. Bald and q are the names of the two client machines in the experiment.

Data size (bytes)	RPC time		Client stub time		Server stub time	Communication time				Local call time	
	Same machine	Different machine	Bald	q		Same machine		Different machine		Bald	q
0	5.1	6.9	1.1	1.8	2.2	1.8	35.8%	3.0	43.0%	0.024	0.049
64	5.8	8.3	1.2	1.9	3.0	1.7	29.8%	3.5	42.3%	0.062	0.099
256	6.6	9.5	1.3	2.0	3.6	1.8	26.9%	4.0	41.7%	0.100	0.148
1,024	9.6	14.9	1.6	2.4	5.9	2.2	22.9%	6.6	44.6%	0.233	0.321

call. This ratio is fairly common in RPC systems. In Agent RPC, however, we plan to significantly reduce it by using a faster interprocess communication mechanism when the two agents are on the same machine and implementing the parameter packing routines in C instead of in Tcl. Communication was 23 to 43 percent of the total time in all cases. Thus the overhead imposed by Tcl and our software is only two to four times greater—not unreasonable given Tcl's slow interpretation speed. Of course, Tcl will be too slow for certain applications; for those, the client and server agents could be written in Java or Scheme, either of which is 10 to 1,000 times faster than Tcl, depending on the application.

SEARCH APPLICATIONS

Agent Tcl is used primarily in distributed information retrieval applications. Our most full-featured application is a mobile Tcl agent that searches distributed collections of technical reports. The mobile agent starts on the user's home machine, typically a laptop, where it asks the user for a free-text query. The agent then travels to the network site of each collection, where it interacts with a dedicated information retrieval agent to retrieve relevant documents. As it travels, the agent merges and organizes the results from each collection.

The agent does not actually travel sequentially through the collection sites. Instead, it sends out child agents to search the collections in parallel. More specifically, the agent makes two decisions:

- If the home machine is connected to the network with an unreliable or low-bandwidth link, the agent first migrates to some dynamically selected proxy site within the network. This eliminates any use of the low-quality link except for the initial transmission of the agent and the final transmission of the merged query results.
- If the information retrieval agents provide a low-level interface to the collections, the agent sends a child agent to each collection. The child agents can perform a multistep query using only local communications; only the final query results are sent back to the main agent. On the other hand, if the information retrieval agents provide a high-level interface and the query requires only a single operation, the agent does not send out child agents. It simply interacts with the collections from across the network, avoiding the migration overhead.
- In either case, once the main agent has results from each collection, it merges and filters the results and carries the final list of relevant reports back to the home machine. If the home machine is disconnected, the agent goes through the docking system. Once the agent is back on its home machine, it displays the list of reports to the user. If the user wants to read a specific report, the agent retrieves the complete text.

Benefits and Limitations

Using a mobile agent in this and other search applications has several advantages:

- *Task continuation*. Because the agent migrates onto a proxy site, it can continue its task even if the home machine disconnects. For example, a user can launch the mobile agent from her laptop, disconnect the laptop, fly across the country (or walk down the hall to a conference room), and then have the agent immediately return with the final results when she reconnects.
- *Minimal connection use*. The agent merges and filters the documents at the proxy site, so the use of the connection between the network and the laptop is minimal. This is critical if the connection is a low-bandwidth wireless or modem link.
- *No application-specific support.* Neither the proxy sites nor the document collections need to provide any application-specific support. In fact, the document collections can provide an extremely low-level interface, such as a single operation that just returns the complete text of a specific technical report. By migrating to the collection, the agent can still perform its search efficiently, since all resource accesses are then local. Thus, once the agent system is installed at a site, developers can efficiently imple-

ment numerous distributed applications without any additional software support at the service sites, which makes life much easier for the service providers.

- *Straightforward code*. Even though the agent exhibits relatively complex behavior, it was extremely easy to write, since the communication mechanisms work the same regardless of whether two agents are on the same machines. Basically, the agent just asks the agent system about the current network link and then jumps to a proxy site if that network link has low bandwidth or is expected to go down. Then, using its knowledge of its query and the collection interfaces, the agent will either send out child agents or interact with the collections remotely. The code to perform the query is the same in both cases.

On the downside, because the agent is written in Tcl, it uses significant CPU time at each collection site. In addition, migration overhead in the current system is large. Thus, if the link between the home machine and network has high bandwidth and stays connected, the mobile agent takes more total time than a traditional implementation. With a high-quality network, even though the agent always eliminates intermediate network transmissions when performing a multistep query, the data amount is not large enough for the transmission time to outweigh the CPU and migration time. On the other hand, if the link between the home machine and network is going up and down or has low bandwidth, the agent takes less time, since it transmits minimal data across the link and can proceed even if the home machine is unavailable.

Reducing the migration time should make the agent competitive in all cases, and we are currently doing the necessary implementation and experimental work to verify this belief. Of course, agents that do a large amount of processing per resource access will need to be written in a faster language, such as Java.

Other Applications

Other information-retrieval applications of Agent Tcl include searching for products that meet a customer's needs, searching for a particular mechanical part (with a CAD drawing of the part provided as input), and searching for medical records that match given criteria. In addition, a joint project between Lockheed Martin and the US Army uses Agent Tcl (and a second, proprietary mobile agent system) to propagate tactical information between the battlefield and command headquarters (and to retrieve information relevant to the current situation from various online sources). Lockheed and the Army developed the application over the course of six Military Intelligence Brigade exercises. In all cases, the agents eliminated intermediate data transmission, continued with their task even when the home machine was disconnected, and performed efficiently without application-specific support at each network site.

Agent Tcl allows the rapid development of efficient, robust distributed applications, particularly when mobile computers are involved. Despite its current capabilities, we see several areas for future work. The two most important are network sensing and service directories. To best plan its route through the Internet, an agent needs information about the network's current state, such as its bandwidth, latency, and connectivity. We are developing sensing agents that glean this information from recent past communications with remote hosts.[5]

Once an agent can roam the network, it needs to know where to go to find relevant services. We are constructing a distributed "yellow pages" infrastructure in which agents can advertise their services and client agents can look for agents that meet their needs.[5] These "yellow pages" are similar to the RPC nameservers except that they are hierarchical and can contain arbitrary service descriptions. Eventually the RPC interface definitions will be included in the "yellow page" entries, eliminating the need for the separate RPC nameservers.

Other areas of future work include rewriting some of the service agents in the much faster Java language, making the agent servers more efficient, and extending the security model to protect agents from malicious machines. We are also investigating multidestination jump support, and are integrating our interagent message-passing with the docking system so that messages go through docks when necessary. We are adding a persistent store so that an agent may leave most of its data (such as the results of a database search) at one host, carry a small amount of its data along with it, and yet be able to remotely access the stored data if necessary. Finally, in cooperation with other groups, we are continuing to develop applications that demonstrate the effectiveness of mobile agents in different network environments. ■

ACKNOWLEDGMENTS

We thank the students who helped construct the support agents described in this article: Ting Cai, Saurab Nog, Vishesh Khemani, and Jun Shen built the dock system; Saurab Nog and Sumit Chawla built the RPC system; Katya Pelekhov implemented the technical report search agent; and Saurab Nog and David Hofer maintained the Agent 007 research lab. We also thank the students of CS88/188 (fall 1995) for their many discussions.

This research was supported by ONR contract N00014-95-1-1204, AFOSR contract F49620-93-1-0266, and Air Force Multidisciplinary University Research Initiative grant F49620-97-1-0382.

REFERENCES

1. R.S. Gray. "Agent Tcl: A Flexible and Secure Mobile-Agent System," *Proc. Tcl/Tk Workshop*, Usenix, Berkeley, Calif., 1996, pp. 9-23.
2. R.S. Gray, "Agent Tcl: A Flexible and Secure Mobile-Agent System," doctoral dissertation, CS Dept., Dartmouth College, Hanover, N.H., 1997.
3. S. Nog, S. Chawla, and D. Kotz, "An RPC Mechanism for Transportable Agents," Tech. Report PCS-TR96-280, CS Dept., Dartmouth College, Hanover, N.H., 1996.
4. A. Birrell and B. Nelson, "Implementing Remote Procedure Calls,"

ACM Trans. Computer Systems, Feb. 1984, pp. 39-59.
5. D. Rus, R. Gray, and D. Kotz, "Transportable Information Agents," in *Proc. Ann. Conf. Autonomous Agents,* ACM Press, New York, 1997, pp. 228-236.

David Kotz is an associate professor of computer science at Dartmouth College, where his research interests include parallel operating systems and architecture, multiprocessor file systems, transportable agents, parallel computer performance monitoring, and parallel computing in computer science education. Kotz received a PhD in computer science from Duke University. He is a member of the ACM, IEEE Computer Society, USENIX, and Computer Professionals for Social Responsibility.

Robert Gray is a research professor in the Thayer School of Engineering at Dartmouth College, where his main interests are the performance, security, and fault tolerance of mobile-agent systems. He is the lead programmer for the Agent Tcl project. Gray received a PhD in computer science from Dartmouth College.

Saurab Nog is a software design engineer at Microsoft, where his interests are mobile agents and their applications, operating systems, and computer networks. Nog received an MS in computer science from Dartmouth College and a BTech in computer science and engineering from the Indian Institute of Technology.

Daniela Rus is an assistant professor of computer science at Dartmouth College, where she cofounded and codirects the Transportable Agents Laboratory. She also founded and directs the Dartmouth Robotics Laboratory. Her research interests include mobile agents, robotics, and information capture and access. Rus received a PhD in computer science from Cornell University. She holds an NSF Career award and is a member of Phi Beta Kappa.

Sumit Chawla is a PhD candidate in computer science at Dartmouth College, where his research interests include image compression, fast medical image acquisition, and parallel computing. He received an MS in computer science from Dartmouth and a BA in computer science and mathematics from Knox College.

George Cybenko is the Dorothy and Walter Gramm Professor of Engineering at Dartmouth College, where his current interests are advanced information retrieval and management systems. He is principal investigator of the Transportable Agents and Wireless Computing project, which is part of the DoD's Multidisciplinary University Research Initiative. He is also editor-in-chief of *IEEE Computational Science and Engineering*. Cybenko received a PhD in electrical engineering from Princeton University.

Readers can contact Kotz at Dartmouth College, 6211 Sudikoff Laboratory, Hanover, NH 03755; dfk@cs.dartmouth.edu; or Gray at Dartmouth College, Thayer School of Eng., 8000 Cummings Hall, Hanover, NH 03755; rgray@cs.dartmouth.edu.

GET CONNECTED

with a new publication from IEEE Computer Society

Features
- Peer-reviewed articles report the latest developments in Internet-based applications and enabling technologies
- Essays, interviews, and roundtable discussions address the Internet's impact on engineering practice and society.
- Columnists provide tutorials and commentary on a range of topics.
- A companion webzine, *IC Online*, supports discussion threads on magazine content, archives of back issues, and links to other sites.

To subscribe at the half-year special rates
- Send check, money order, credit card number to IEEE Computer Society, 10662 Los Vaqueros Circle, PO Box 3014, Los Alamitos, CA 90720-1314.
  - $14 for members of the IEEE Computer or Communications Societies (membership number: _____)
  - $17 for members of other IEEE societies (membership number: _____)

Name
Company Name
Address
City State Zip Code
Country

http://computer.org/internet/

IEEE COMPUTER SOCIETY

THE INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, INC.