Agent Tcl: Alpha Release 1.1

Robert S. Gray*
Department of Computer Science
Dartmouth College
Hanover, NH 03755
E-mail: robert.s.gray@dartmouth.edu

December 1, 1995

Abstract

Agent Tcl is a transportable agent system. The agents are written in an extended version of the Tool Command Lanuage (Tcl). Each agent can suspend its execution at an arbitrary point, transport to another machine and resume execution on the new machine. This *migration* is accomplished with the *agent_jump* command. *agent_jump* captures the current state of the Tcl script and transfers this state to the destination machine. The state is restored on the new machine and the Tcl script continues its execution from the command immediately after the *agent_jump*. In addition to migration, agents can send messages to each other and can establish direct connections. A direct connection is more efficient than message passing for bulk data transfer. Finally, agents can use the Tk toolkit to create graphical user interfaces on their current machine. Agent Tcl is implemented as two components. The first component is an extended Tcl interpreter. The second component is a server which runs on each machine. The server accepts incoming agents, messages and connection requests and keeps track of the agents that are running on its machine. An alpha release of Agent Tcl is available for public use. This documentation describes how to obtain and compile the source code, how to run the server and how to write transportable agents.

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

Contents

1 Introduction **4**

2 Installation **6**

3 Running the server **8**

4 Utilities **11**

5 Interpreter directories **12**

6 Tcl and Tk **12**

7 Agent Tcl **14**

7.1 Agent identification 14
7.2 Registering an agent 16
7.3 Messages 22
7.4 Events 24
7.5 Migration 24
7.6 Limitations of jump and fork 34
7.7 Meetings 35
7.8 Timing 49
7.9 Masks 52
7.10 Undocumented commands 58
7.11 Summary 59

8 Agent Tk **59**

8.1 Creating a main window 59
8.2 Destroying a main window 64
8.3 Waiting for the user 64
8.4 Tk handlers for incoming messages, events and meetings 64
8.5 Summary 68

9 Advanced topics **68**

10 Future directions **69**

A Changes from previous releases **71**

A.1 Changes from 0.5 to 1.0 71
A.2 Changes from 1.0 to 1.1 71

B Known bugs **73**

B.1 Missing *masks* and *timeouts* 73
B.2 Sticky *event* handlers 73

B.3 Lost *upvar* reference 73
B.4 *gets*, *puts* and *read* 74

C Command summaries **75**
C.1 Registration 75
C.2 Migration 75
C.3 Basic communication 75
C.4 Meetings 76
C.5 Masks 76
C.6 Timing and retries 77
C.7 Information 77
C.8 Advanced 78
C.9 Miscellaneous 78

1 Introduction

An *information* agent is charged with the task of searching a collection of electronic resources for information that is relevant to the user's current needs. These resources are often distributed across a network and can contain tremendous quantities of data. One of the paradigms that has been suggested for allowing efficient access to such resources is *transportable agents*. 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. Although the idea of a program that can move from machine to machine under its own control is not new [WVF89], it is only in the last two years that production-quality systems have been implemented. The two most notable transportable agent systems are Telescript from General Magic [Whi94, Whi95b, Whi95a] and TACOMA from Cornell University [JvRS95]. Telescript is a dynamic object-oriented language that is centered around network communication. An agent written in Telescript uses the *go* instruction to migrate to a new machine. The agent continues execution on the new machine from the instruction after the *go*. A Telescript *engine* at each site accepts, authenticates and executes incoming agents, enforces security constraints, and backs up the internal state of agents to nonvolatile store in case of site failure. TACOMA takes a more general view of agents. The single abstraction is the *meet* instruction which one agent uses to invoke another agent. All services except for *meet* are provided directly by other agents. For example an agent meets with the *ag_tcl* agent in order to migrate to a new machine. Important features of TACOMA are rear guard agents which restart a remote agent if the remote site fails, broker agents that provide scheduling and directory services, and electronic cash that is used to pay for services and prevent runaway agents. TACOMA agents are written in Tcl.

The recent development of transportable agent systems has been fueled by the growing inadequacy of the traditional client/server model for modern distributed applications. The traditional client/server model assigns fixed roles to programs. A program is either a *server* which provides some service such as file transfer or a *client* which makes requests of the server. Transportable agents replace this artificial division with a *peer-to-peer* model in which agents communicate as peers and act as both clients and servers depending on their current needs and capabilities. Such a model provides far more flexibility when developing distributed applications [Lew95]. In addition traditional servers provide a fixed set of operations. All operations that are not provided in this fixed set must be performed at the client. If the server does not provide an operation that matches the client task exactly, the client must make a series of server requests, bringing intermediate data across the network on each request. If the intermediate data is not useful beyond the end of the task, a significant amount of network bandwidth has been wasted. To avoid this inefficiency, server developers tend to provide a collection of specialized operations – one operation for each client task – rather than a collection of simple primitives that can be combined into more complex operations. Providing these specialized operations violates software engineering principles and becomes intractable as the number of clients increases. Transportable agents avoid both the inefficiency and the need for specialized operations since the agents migrate *to the remote resources*. An agent executes local to the resource and returns only its final result to the client. No network resources are wasted on intermediate data. The performance gain is greatest in low bandwidth or high latency networks [Whi94].

Modern client/server techniques such as remote evaluation and SUPRA-RPC allow a program to migrate to the resource as well. However these techniques maintain the fixed client/server division since the programs are *anonymous* entities that can not communicate easily with each other. At best the programs are limited to exchanging partial results with their invoker. Transportable agents are *named* entities that communicate at will and support the peer-to-peer model. In addition remote evaluation and SUPRA-RPC require a connection between communicating machines. Transportable agents do not require a continuous connection and do not require the maintenance of state information at both the local and remote machine. This makes transportable agents more fault tolerant [WVF89] and – in combination with their low use of network resources – makes them ideally suited to mobile computing [Whi94]. Mobile computing is characterized by high latency, low bandwidth and periods of disconnection from the network.

Transportable agents are a convenient paradigm for distributed computing. They make efficient use of network resources, support the peer-to-peer model and tolerate network disconnection. In addition they

hide the communication channels but not the location of the computation [JvRS95]. The agent specifies when and where to migrate but the system handles the transmission details. This makes transportable agents easier to use than low-level facilities in which the programmer must explicitly handle communication *but* more flexible and powerful than schemes such as process migration in which the *system* decides when to move a program based on a small set of fixed criteria. Transportable agents allow the implicit transfer of information since a migrating agent carries all of its internal state along with it. This eliminates the need for a separate communication step. Many tasks – especially network management, information retrieval and workflow – fit naturally into the *jump, do and jump again* model of transportable agents. The agent migrates to a site, performs a task, migrates to a new site, performs a task that depends on the outcome of the first task and so on. Finally transportable agents are a useful extension to traditional clients and servers. Clients and servers can program each other which greatly extends the functionality that application and server developers can provide to their customers. In addition an application can *dynamically* distribute its server components when it starts executing.

Applications that have been suggested for transportable agents include distributed information retrieval, network management, active e-mail, active documents, control of remote devices and electronic shopping [Whi94, Ous95]. Our research group at Dartmouth began exploring transportable agents in the context of distributed information retrieval and attempted to find an existing system that would meet our needs. Telescript appeared to be a suitable choice but unfortunately it does not run on general platforms and the source code is not available to the research community. TACOMA was unavailable at the time and places the burden of migration squarely on the agent programmer. The agent must explicitly specify every piece of information that needs to move to the remote machine. In contrast, the *go* instruction of Telescript transparently migrates the complete internal state of an agent. The agent continues from the point of interruption. Programming such behavior into TACOMA must be done at the agent level and is nearly intractable. Other systems such as Safe-Tcl/MIME and HotJava provide only certain aspects of transportable agent behavior and only in certain contexts. Safe-Tcl/MIME allows programs to be included in mail messages (*active mail*) while HotJava allows programs to be included in World Wide Web documents (*active documents*).

Due to the limitations and unavailability of existing systems, we are developing a transportable agent system called Agent Tcl. Agent Tcl runs on generic UNIX workstations, communicates over the Internet using the standard TCP/IP protocol and reduces migration to a single instruction as in Telescript. Agent Tcl is far from complete but has progressed to the point where it is flexible and robust enough to be used in a range of applications. The transportable agents are written in an extended version of the Tool Command Language (Tcl). Each agent can suspend its execution at an arbitrary point, transport to another machine and resume execution on the new machine. This migration is accomplished with the *agent_jump* command. *agent_jump* captures the internal state of the Tcl script and transfers this state to the destination machine. The state is restored on the new machine and the Tcl script continues from the command immediately after the agent_jump. In addition to migration, agents can send messages to each other and can establish direct connections or *meetings* with each other. A direct connection is more efficient than message passing for bulk data transfer. Finally, agents can use the Tk toolkit to create graphical user interfaces on their current machine.

Agent Tcl is implemented as two components. The first component is a modified Tcl interpreter. The second component is a server which runs on each machine. The server accepts incoming agents, messages and meeting requests and keeps track of the agents that are running on its machine. All agents run with the permissions of the server so the server should run under its own account rather than as root. The server provides limited security by refusing all agents and requests that do not come from an "approved" machine. A list of approved machines is passed to the server at startup. This level of security should be sufficient for initial development work and for applications in a controlled, localized environment.

An alpha release of Agent Tcl is available for public use. This documentation describes how to obtain and compile the source code, how to run the server and how to write transportable agents. The final section discusses planned extensions to the current release. One notable extension is improved security.

Questions, comments, suggestions, critiques and bug reports should be directed to *robert.gray@dartmouth.edu*. Source code is welcome as well if you modify Agent Tcl and wish to submit your modification for potential inclusion in the official release.

2 Installation

The alpha release of Agent Tcl is packaged as a tar file

`agent.1.1.tar.gz`

which can be obtained via our World Wide Web page

`http://www.cs.dartmouth.edu/~rgray/transportable.html`

or via anonymous ftp

`ftp:://bald.cs.dartmouth.edu/pub/agents/`

Note that the tar file contains *all* necessary code. You do not need the source code for standard Tcl or Tk and you do not need to install standard Tcl or Tk on your system.

The alpha release is known to compile with *gcc 2.6* on an Intel 486 running Linux 1.2.8, an IBM RISC 6000 running AIX 3.2.5, an SGI Indy running IRIX System V.4, a DecStation 5000 running ULTRIX V4.3, a DEC alpha running OSF/1 V3.2 and an Intel Pentium running FreeBsd 2.1. These are the Unix environments to which the author has access. The alpha release should be easily portable to any Unix environment that provides TCP/IP and Berkeley sockets. Other environments will require more work.

To install Agent Tcl

1. The server provides enough security for experimentation and research, initial development and applications that run in controlled, localized environments. However the security component of Agent Tcl is far from complete. One notable deficiency is that the transportable agents run with the authority of the server. Therefore do *not* run the server as *root* or as any userid that has the authority to access or damage sensitive data. It is highly recommended that you create a new account for the Agent Tcl system and run the server from this account. This account should have only as much authority as is needed for the desired application(s). In this document I will assume that you create an account with userid *agent*.

2. Log onto the *agent* account, download agent.1.1.tar.gz, uncompress and untar. This process will create a directory called *agent.1.1*. The contents of this directory are summarized in Table 1.

3. Edit *Makefile*. Change SERVER_TCP_PORT if desired. SERVER_TCP_PORT specifies the TCP/IP port that the server uses. Change the installation directories if desired. Pay special attention to TCL_LIBRARY and TK_LIBRARY which specify the directories where the interpreters expect to find the initialization scripts. TCL_LIBRARY and TK_LIBRARY should resolve to the same directories as TCL_INSTALL and TK_INSTALL respectively. Table 2 summarizes the files that are installed during installation. Note that some of the file names are the same as for standard Tcl. Do not overwrite the standard Tcl files if standard Tcl is on your system! Once you have chosen the port and the installation directories, uncomment the appropriate set of switches (there is a BUILTIN_LIBS, CC, RANLIB and CFLAGS switch defined for each architecture).

4. Compile the system by typing

   ```
   make
   ```

5. Install the system by typing

   ```
   make install
   ```

File or subdirectory	Contents
Makefile	the top-level makefile
install-sh	an installation script
agent.terms	the licensing terms
INSTALL	instructions for quick installation
README	a brief overview of the system
TO_DO	a to-do list
NOTE	a note about porting the system
tcl7.4_stack	the modified Tcl 7.4 core
tk4.0	the Tk 4.0 core
agent-tcl	the agent interpreter (Tcl)
agent-tk	the agent interpreter (Tcl/Tk)
server	the server that runs on each machine
utility	routines that are used in the interpreters and the server
tcpip-tcl	a Tcl extension that provides simple socket management
restrict	a simple timeout facility
scripts	initialization scripts for the interpreters
examples	example agents
doc	the document that you are reading

Table 1: The contents of agent.1.1.tar.gz

File	Description	Source directory	Install directory
init.tcl	initialization of the Tcl core	scripts/agent	TCL_INSTALL
agent.tcl	initialization of the agent interpreter	scripts/agent	TCL_INSTALL
parray.tcl	array utility	scripts/agent	TCL_INSTALL
tkerror.tcl	a background error handler	scripts/agent	TCL_INSTALL
retry.tcl	the *retry* command	scripts/agent	TCL_INSTALL
file.tcl	the *get_remote_file* command	scripts/agent	TCL_INSTALL
tclIndex	Tcl index file	scripts/agent	TCL_INSTALL
*.tcl	initialization of the Tk core	scripts/tk	TK_INSTALL
tclIndex	Tcl index file	scripts/tk	TK_INSTALL
agent	the agent interpreter (Tcl)	agent-tcl	EXEC_INSTALL
agent-tk	the agent interpreter (Tcl/Tk)	agent-tk	EXEC_INSTALL
agentd	the agent server	server	EXEC_INSTALL
machine.tcl	utility to check server status	examples	EXEC_INSTALL
libtcl.a	modified Tcl 7.4 library	tcl7.4_stack	LIB_INSTALL
libtk.a	Tk 4.0 library	tk4.0	LIB_INSTALL
libagent.a	agent library	agent-tcl	LIB_INSTALL
libtcpip.a	TCP/IP library	tcpip-tcl	LIB_INSTALL
libutility.a	utility library	utility	LIB_INSTALL
librestrict.a	timeout library	restrict	LIB_INSTALL
tcl.h	header file for Tcl library	tcl7.4_stack	INCLUDE_INSTALL
tk.h	header file for Tk library	tk4.0	INCLUDE_INSTALL
tclAgent.h	header file for agent library	agent-tcl	INCLUDE_INSTALL
tclTcpip.h	header file for TCP/IP library	tcpip-tcl	INCLUDE_INSTALL
tclRestrict.h	header file for timeout library	restrict	INCLUDE_INSTALL
my_sizes.h	typedefs that are used in headers	utility	INCLUDE_INSTALL
*.tcl	examples	examples	EXAMPLE_INSTALL

Table 2: Where the files are installed

6. Add the directory that contains the *agent* and *agent-tk* interpreters to the PATH environment variable on the *agent* account – i.e. add EXEC_INSTALL to the PATH variable. If you want to be able to invoke the interpreters from other accounts, add EXEC_INSTALL to the PATH variable on those accounts and change the interpreter permissions to publically executable.

7. Finally, change the first line of *machine.tcl* and each of the example scripts so that it specifies the correct location of the *agent* or *agent-tk* interpreter. In addition, change the list of machines in *machine.tcl* to the machines on which you want to run the transportable agent system.

.

The entire installation process must be repeated for every distinct architecture on which you want to run the agent system. If these architectures share a filespace, you will have to use a different EXEC_INSTALL directory for each architecture. On duplicate architectures, simply create the agent account and make sure that the *agent* and *agent-tk* interpreters are accessible and that the directory of initialization scripts is accessible. As in standard Tcl and Tk, you can set the environment variables TCL_LIBRARY and TK_LIBRARY to override the directories that have been compiled into the agent executables. This will allow you to avoid recompilation on duplicate architectures that do not share a filespace. You will however have to replace *agent* with a script that first sets the TCL_LIBRARY environment variable and then calls the actual *agent* (and similarly for *agent-tk* and *agentd* except that you should set TK_LIBRARY as well).

3 Running the server

The server must run on every machine to which transportable agents can be *sent*. For each machine

1. Log onto the *agent* account. Remember that we want the server to run with the minimal permissions of the *agent* account since transportable agents inherit the permissions of the server.

2. Edit the file *agent.access* which is found in the subdirectory *scripts/start*. The current *agent.access* is

```
# Agent Tcl
# Bob Gray
# 23 August 1995
#
# agent.access
#
# This file lists the machines that are allowed to use the server.

localhost                               # localhost
felix.cs.dartmouth.edu                  # Agent 007
gogol.cs.dartmouth.edu                  # Agent 007
jaws.cs.dartmouth.edu                   # Agent 007
m.cs.dartmouth.edu                      # Agent 007
oddjob.cs.dartmouth.edu                 # Agent 007
q.cs.dartmouth.edu                      # Agent 007
spectre.cs.dartmouth.edu                # Agent 007
muir.cs.dartmouth.edu                   # DEC Alpha
sable.cs.dartmouth.edu                  # DEC Alpha
tenaya.cs.dartmouth.edu                 # DEC Alpha
tioga.cs.dartmouth.edu                  # DEC Alpha
earhart.cs.dartmouth.edu                # DEC Alpha
tuolomne.cs.dartmouth.edu               # DEC Alpha
bald.cs.dartmouth.edu                   # Linux
```

```
plum.cs.dartmouth.edu                   # SGI
gainsboro.cs.dartmouth.edu              # SGI
orchid.cs.dartmouth.edu                 # SGI
peach.cs.dartmouth.edu                  # SGI
coral.cs.dartmouth.edu                  # SGI
salmon.cs.dartmouth.edu                 # SGI
cosmo.dartmouth.edu                     # Thayer
lost-ark.dartmouth.edu                  # Thayer
temple-doom.dartmouth.edu               # Thayer
```

You should replace these machines with the machines on which you are running the transportable agent system. The server will only accept agents that come from one of the machines on the list. Thus the server is only vulnerable to (1) masquerade attacks in which a malicious machine masquerades as a machine that has crashed and (2) attacks that originate from within your own set of machines. The latter means that the server is as secure as any individual account on your machines. If the security of one of these accounts is breached, however, a malicious user can submit an arbitrary Tcl script that will run with the permissions of the server. This is why the *agent* account should have minimal permissions.

3. Edit the file *agent.languages* which is found in the subdirectory *scripts/start*. The current *agent.languages* is

```
# Agent Tcl
# Bob Gray
# 23 August 1995
#
# agent.languages
#
# This file defines the set of interpreters that are available through the
# server.  Each interpreter definition consists of:
#
# 1. Symbolic name
# 2. Executable name
# 3. argv[0]
# 4. Socket directory
# 5. STATE or NORMAL
#
# The five items must appear one per line and in the order specified.


# STATE-TCL is the Tcl interpreter that allows the capture and restoration
# of the internal state of an executing Tcl script.

STATE-TCL
/usr/contrib/bin/agent      # FILE
agent                       # ARGV0
/tmp                        # SOCKET DIRECTORY
STATE

# STATE-TK is the Tk interpreter that allows the capture and restoration of
# the internal state of an executing Tk script.

STATE-TK
/usr/contrib/bin/agent-tk   # FILE
```

```
agent-tk                    # ARGV0
/tmp                        # SOCKET DIRECTORY
STATE
```

This file specifies the location of the available interpreters. You should change the lines marked "FILE" so that they specify the location of the interpreters on your system. If you have installed the interpreters under different names, you should change the lines marked "ARGV0" so that they specify the new names. Finally, if you do not want the server to create temporary files in */tmp*, you should change the lines marked "SOCKET DIRECTORY" so that they specify the desired directory.

4. Change into the subdirectory *scripts/start*.

5. Run the server. On the author's machine this is done with the command

```
agentd -host bald.cs.dartmouth.edu \
       -log /tmp/agent.log \
       -lock /tmp/agent.lock \
       -access agent.access \
       -lang agent.languages
```

The -host parameter specifies the full Internet name of the host on which the server is being started. This parameter is necessary since gethostbyname() does not return the full Internet name on every machine. *agentd* will verify that the IP address of the specified host matches the IP address of the host returned by gethostbyname(). This eliminates the possibility of accidentally specifying the wrong machine. The -log parameter specifies the name of a file into which error messages are written during server execution. The -lock parameter specifies the name of a file that is used to synchronize access to the internal data structures of the server. The -access parameter specifies the name of the access file from step 2. The -lang parameter specifies the name of the language file from step 3. Note that the -log and -lock filenames must be fully qualified. Executing this command produces the following output on the author's machine.

```
Processing interpreter list ...

Adding "STATE-TCL" to interpreter list
Adding "STATE-TK" to interpreter list

Processing access list ...

Adding       127.0.0.1 (localhost) to access list
Adding 129.170.202.124 (felix.cs.dartmouth.edu) to access list
Adding 129.170.202.127 (gogol.cs.dartmouth.edu) to access list
Adding 129.170.202.123 (jaws.cs.dartmouth.edu) to access list
Adding 129.170.202.121 (m.cs.dartmouth.edu) to access list
Adding 129.170.202.128 (oddjob.cs.dartmouth.edu) to access list
Adding 129.170.202.122 (q.cs.dartmouth.edu) to access list
Adding 129.170.202.126 (spectre.cs.dartmouth.edu) to access list
Adding  129.170.192.42 (muir.cs.dartmouth.edu) to access list
Adding  129.170.192.72 (sable.cs.dartmouth.edu) to access list
Adding  129.170.200.32 (tenaya.cs.dartmouth.edu) to access list
Adding  129.170.192.21 (tioga.cs.dartmouth.edu) to access list
Adding  129.170.194.33 (earhart.cs.dartmouth.edu) to access list
Adding  129.170.192.41 (tuolomne.cs.dartmouth.edu) to access list
Adding  129.170.192.98 (bald.cs.dartmouth.edu) to access list
Adding  129.170.192.65 (plum.cs.dartmouth.edu) to access list
```

```
Adding  129.170.192.67 (gainsboro.cs.dartmouth.edu) to access list
Adding  129.170.192.66 (orchid.cs.dartmouth.edu) to access list
Adding  129.170.192.68 (peach.cs.dartmouth.edu) to access list
Adding  129.170.192.60 (coral.cs.dartmouth.edu) to access list
Adding  129.170.192.59 (salmon.cs.dartmouth.edu) to access list
Adding   129.170.24.57 (cosmo.dartmouth.edu) to access list
Adding   129.170.24.47 (lost-ark.dartmouth.edu) to access list
Adding   129.170.24.48 (temple-doom.dartmouth.edu) to access list

Verifying server IP address ...

OK!

Tracking status ...

The tracker agent is OFF.
```

The server should now be running in the background as a daemon. This can be verified with the ps command. On the author's machine

```
% ps ax | grep agentd
32080 pp4 S     0:00 agentd -host bald.cs.dartmouth.edu ...
32081 pp4 S     0:00 agentd -host bald.cs.dartmouth.edu ...
%
```

The first agentd maintains the internal tables. The second agentd watches for incoming agents on the TCP/IP port. Both processes must be running for the server to function correctly. Check the error log if one or both of the processes are missing. In the error log the first agentd is referred to as "agentd" and the second agentd is referred to as "socketd" since it is the socket watcher. The server can be run in the foreground by specifying the "-nodaemon" flag when starting *agentd*. All error and informational messages will be displayed on the terminal rather than written to the log file. The foreground mode is useful for server development and intense debugging.

6. To bring down the server, you need to kill all running *agentd* processes (there will be more than two if there are agents executing on the system). There is a utility called *kill.tcl* which will help with this task. *kill.tcl* is discussed in the next section.

7. If you change *agent.access* or *agent.languages*, you will have to bring down and restart the server. The server can not read its configuration files while it is running. This will be fixed soon.

4 Utilities

There are two useful utilities included with the system. The first is *kill.tcl* which is found in *scripts/start* subdirectory. *kill.tcl* forcibly terminates all of the agents that are running under the control of the local server. Note that *kill.tcl* can only kill those processes that are running under userid *agent*. This means that you might have to kill some *agent* and *agent-tk* processes by hand. *kill.tcl* will always break all connections to the server's TCP/IP socket, however, which will allow you to bring down and restart the server.

The second utility is *machine.tcl* which is found in *examples* subdirectory and was installed the EXEC_INSTALL directory during installation. First, if you did not do so during installation, change the first line of *machine.tcl* to the correct location of the *agent* interpreter and change the machines to the machines on which you are running the agent system. Now if you just type

```
machine.tcl
```

you will see a list of the machines on which the server should be running. If you type

```
machine.tcl all
```

machine.tcl will *confirm* that the server is running on each machine by sending out agents and waiting for responses. You will see informational messages as it submits the agents and receives the responses. Once *machine.tcl* has received all the responses or timed out, it will display a list of errors. Errors almost always indicate that the server is not running on the specified machine. Finally, rather than typing "all", you can list one or more machines on the command line. For example,

```
machine.tcl m q
```

confirms that the server is running on machines *m* and *q*.

5 Interpreter directories

The following sections assume that the standard Tcl/Tk interpreters, *tclsh* and *wish*, are in

```
/usr/local/bin
```

and that the corresponding agent interpreters, *agent* and *agent-tk*, are in

```
/usr/contrib/bin
```

These are the directories that are used at Dartmouth. You will need to replace all occurrences of these directories with the directories that you specified during installation.

6 Tcl and Tk

Transportable agents are written in the Tool Command Language (Tcl). Tcl was created by Dr. John Ousterhout at the University of Berkeley in 1987 and has enjoyed enormous popularity since then. Tcl is a general-purpose scripting language that has two components. The first component is a stand-alone shell similar to the C and Korn shells. The shell allows the user to interactively execute Tcl commands and scripts. The second component is a library of C functions. The library provides functions to "create" a Tcl interpreter, define new Tcl commands and submit Tcl scripts to the interpreter for evaluation. This library allows Tcl to be *embedded* inside a larger application. Any application that needs a scripting language can link in the library and let the user write Tcl scripts.

A tutorial on Tcl is beyond the scope of this documentation. Tcl is relatively easy to learn, however, and is similar to other scripting languages such as Perl and C shell. For example the following Tcl script computes factorials.

```
#!/usr/local/bin/tclsh
#
# fac.tcl
#
# This Tcl script computes factorials.

proc factorial x {

    if {$x <= 1} {
```

```
        return 1;
    }

    expr $x * [factorial [expr $x - 1]]
}

set number 0

while {$number != -1} {

  puts "Enter a nonnegative integer (-1 to quit): "
  gets stdin number

  if {$number != -1} {
    puts "$number! is equal to [factorial $number]"
  }
}
```

The two most important aspects of this script are the *command* and *variable* subsitutions. For example in the command

```
expr $x * [factorial [expr $x - 1]]
```

$x is a variable substitution and *[factorial [expr $x - 1]]* is a command substitution. *$x* will be replaced by the value of variable *x* when the command is evaluated. *[factorial [expr $x - 1]]* will be replaced by the result of the command *factorial [expr $x - 1]*.

There are three ways to execute the script. If the Tcl shell is called *tclsh* and the script is in the file *fact.tcl*, you can type *tclsh* to start the shell and then *source fac.tcl* to load and execute the script. Alternatively you can type *tclsh fac.tcl*. alternatively you can turn on the execution permissions for file *fac.tcl* and just type *fac.tcl*. This works because the first line in the script is

```
#!/usr/local/bin/tclsh
```

which tells Unix to run the script *using the tclsh shell*. In other words Unix starts up tclsh which then runs factorial. tclsh will terminate when factorial terminates.

Tk is an extension to Tcl that allows the rapid prototyping of graphical user interfaces (GUI). A tutorial on Tk is beyond the scope of this documentation. Tk is relatively easy to learn, however, since it provides *high-level* commands for creating Motif-like interfaces. For example, the following Tk script creates a button that says "Hello, World!". Clicking on the button terminates the script. This example was taken from [Ous94].

```
#!/usr/contrib/bin/wish
#
# hello.tk
#
# This is the "Hello, World!" example from [Ous94].

  # make the "Hello, World!" button

button .button -text "Hello, World!" -command exit
pack .button
```

Again there are three ways to execute the script. Type *wish* and then *source hello.tk*; type *wish hello.tk*; or turn on the execution permissions for file *hello.tk* and just type *hello.tk*.

There are numerous sources of information on Tcl and Tk. I recommend the book by Ousterhout [Ous94], the book by Welch [Wel95] and the Tcl news group *comp.lang.tcl*.

7 Agent Tcl

Standard Tcl has no notion of transportability so we have developed an extended version of Tcl called Agent Tcl that provides a special set of commands. These commands allow Tcl scripts to migrate from machine to machine and allow distributed Tcl scripts to communicate with each other. The shell that interprets this extended version of Tcl is called *agent*. *agent* is fully compatible with Tcl 7.4 except for the presence of the agent commands. To run the factorial script using the agent shell, start the agent shell by typing *agent* and then type *source fac.tcl*; type *agent fac.tcl*; or change the first line of the script to

```
#!/usr/contrib/bin/agent
```

and just type *factorial.tcl*. All transportable agents must run under the agent shell *agent* rather than the standard Tcl shell *tclsh*.

7.1 Agent identification

Each transportable agent has a controlling server which can be thought of as the agent's current home. The agent acquires a controlling server when it first starts up and acquires a *new* controlling server whenever it migrates. Whenever the agent acquires a controlling server, it is assigned a unique numeric id. Once the agent has a numeric id, it can choose a unique symbolic name if desired. Agents that want to communicate with the given agent specify its *address* as the network location of its controlling server plus either its numeric id or its symbolic name. This address changes whenever the agent migrates to a new machine and acquires a new controlling server.

More formally, each transportable agent has a 4-element identification

```
machine_name   machine_IP   symbolic_name    numeric_id
```

where *machine_name* is the full Internet name of the machine on which the controlling server is executing; *machine_IP* is the IP address of the machine; *symbolic_name* is the symbolic name of the agent; and *numeric_id* is the numeric id of the agent. For example, the agent with controlling server *bald*, symbolic name *search_agent* and numeric id *10* would have the identification

```
bald.cs.dartmouth.edu    129.170.192.98    search_agent    10
```

Note that there is substantial redundancy in this identification. Any of the following are enough to uniquely identify the sample agent:

```
bald.cs.dartmouth.edu search_agent
bald.cs.dartmouth.edu 10
129.170.192.98 search_agent
129.170.192.98 10
```

Two agents with the same controlling server will *never* have the same name or the same numeric ID.

In addition to its own identification, each agent has a *root identification*. Agents can create other agents. This leads to hierarchies of agents, each with a single agent at the top. The top-level agent is the *root agent*

Array element	Contents
registered	1 if the agent has a controlling server and 0 otherwise
toplevel	1 if the agent is a root agent and 0 otherwise
server	1 is the agent arrived via the server and 0 otherwise
actual-server	Name of the machine on which the agent is executing
actual-ip	IP address of the machine on which the agent is executing
local	4-element identification of the agent
local-server	Name of the controlling server's machine
local-ip	IP address of the controlling server's machine
local-name	Symbolic name of the agent
local-id	Numeric id of the agent
root	4-element identification of the *root* agent
root-server	Name of the root server's machine
root-ip	IP address of the root server's machine
root-name	Symbolic name of the root agent
root-id	Numeric id of the root agent

Table 3: Elements of the agent array – All elements of the array except for *actual-server*, *actual-ip*, *registered*, *server* and *toplevel* are the empty string if the agent does not have a controlling server. *local-name* is the empty string if the agent does not have a symbolic name. *root-name* is the empty string if the root agent does not have a symbolic name. Note that *local-server*, *local-ip*, *local-name* and *local-id* are the four pieces of *local* while *root-server*, *root-ip*, *root-name* and *root-id* are the four pieces of *root*. In addition note that the root information in the array will *not* change when the root agent moves or changes its name (aside from in the root agent itself). The system does not keep track of all agents with a given root.

for all of the agents in its hierarchy. The *root server* is the controlling server of the root agent. The *root identification* is the 4-element identification of the root agent.

Finally, each agent has an *actual location*. An agent does not have to execute on the same machine as its controlling server so its actual location is the name and IP address of the machine on which it is actually running.

All of this information is stored in a global array called *agent*. Table 3 summarizes the elements of the *agent* array. The array is always available inside a transportable agent. For example, to display the name of its controlling server, an agent would issue the Tcl command

```
puts $agent(local-server)
```

If you want to access the *agent* array from inside a Tcl procedure, you will have to use the *global* or *upvar* command to create a local reference to the array. This is true for all global variables in Tcl. For example,

```
proc display args {

  global agent

    # the following statement will cause an "undefined variable" error
    # if the "global" statement is removed

  puts $agent(local-server)
}
```

The *agent* array is *read-only* and updates automatically as the agent moves through the network.

7.2 Registering an agent

Registering is the process of acquiring a controlling server and selecting a symbolic name. There are five relevant commands.

- agent_begin [*machine*] [-time *seconds*]

  A root agent uses the agent_begin command to acquire a controlling server. *machine* is the name or IP address of the machine on which the desired server is running. *machine* defaults to the agent's actual machine if it is not specified. The name and IP address of the agent's actual machine are in *agent(actual-server)* and *agent(actual-ip)* respectively. *seconds* is the maximum number of seconds to wait for a response from the server. *seconds* can be a fractional number of seconds and defaults to 15 if the *-time* parameter is not specified.

  The server on the specified machine assigns a unique numeric ID to the agent and adds the agent to its internal tables. The server is now the controlling server for the agent. agent_begin returns the new 4-element identification of the agent and fills in the root and local sections of the *agent* array. The agent is its own root so the root and local identifications are the same. The agent initially has no symbolic name so *agent(local-name)* and *agent(root-name)* are the empty string.

  Note:

  Non-root agents do not need to use agent_begin since the controlling server of a child agent is automatically the server to which the child agent was submitted. Root agents must issue agent_begin since they have no controlling server initially and can not use the other agent commands until a controlling server has been acquired.

  Examples:

  Suppose that the root agent is executing on machine *moose*. Issuing the command

  ```
  agent_begin
  ```

  registers the agent with the server on *moose*. The command returns the new identification of the agent which in this case might be "moose.cs.dartmouth.edu 129.170.194.28 {} 10". Note that the agent has no symbolic name when it is first registered. The root and local sections of the *agent* array are updated with the new identification.

  Issuing either of the commands

  ```
  agent_begin bald.cs.dartmouth.edu
  agent_begin 129.170.192.98
  ```

  registers the agent with the server on *bald*. The commands return the new identification of the agent which in this case might be "bald.cs.dartmouth.edu 129.170.192.98 {} 17". As before the agent has no symbolic name when it is first registered and the root and local sections of the array are updated with the new identification. Note that you do not have to specify the full Internet name of the machine. It is sufficient to specify any name that the local name server can map to the desired machine.

  Error messages:

  On error, agent_begin raises a standard Tcl exception that can be caught with the *catch* command. agent_begin will return one of the following error messages. The indented text below each error message describes the possible interpretations of the message.

  ```
  wrong number of arguments

    You specified the wrong number of arguments.
  ```

```
agent has been registered

  The agent already has a controlling server.

unable to convert name

  The system was unable to determine the network address of the specified
  machine.

unable to send to server

  The system was unable to contact the server within the specified number of
  seconds.  This means that the specified machine does not exist or has
  crashed, the specified machine is temporarily unreachable due to network
  failure or overload, or the server is not running on the specified machine.
  The standard response to this error is to retry the command at least once.

server unable to comply (no response)

  The server failed while handling the agent_begin.

server unable to comply (bad response)

  The server failed while handling the agent_begin.

server unable to comply (server error)

  The server does not accept requests from the agent's current machine, i.e.,
  the agent's current machine is NOT in the server's access list.
```

- agent_name *name* [-time *seconds*]

  An agent uses the agent_name command to register *name* as its symbolic name. *seconds* is the maximum number of seconds to wait for a response from the server. *seconds* can be a fractional number of seconds and defaults to 15 if the *-time* parameter is not specified.

  agent_name returns the new 4-element identification of the agent and updates the local section of the *agent* array. It updates the root section as well if the agent is a root agent.

  Examples:

  Suppose that the agent identification is currently "bald.cs.dartmouth.edu 129.170.192.98 {} 10". Issuing the command

  ```
  agent_name search_agent
  ```

  makes "search_agent" the symbolic name of the agent. The command returns the new agent identification "bald.cs.dartmouth.edu 129.170.192.98 search_agent 10". The local and root sections of the *agent* array are updated with the new symbolic name.

  Notes:

  1. It is possible in pathological situations for the server to believe that the name is in use even after the agent has terminated. In such cases the agent_force command can be used to forcibly reclaim the name. See the section on agent_force below.
  2. An agent can changes its symbolic name by calling agent_name again. The old name is removed from the server tables and can no longer be used when sending messages to the agent. There is currently no way for an agent to have multiple symbolic names.

Error messages:

On error, agent_name raises a standard Tcl exception that can be caught with the *catch* command. agent_name returns one of the following error messages. The indented text below each error message describes the possible interpretations of the message.

```
wrong number of arguments

  You specified the wrong number of arguments.

agent has NOT been registered

  The agent does not have a controlling server.

unable to send to server

  The agent was unable to contact its controlling server within the
  specified number of seconds.  This means that the controlling server has
  crashed, the controlling server's machine has crashed, or the controlling
  server is temporarily unreachable due to network contention or overload.
  The standard response to this error is to retry the command at least once.

server unable to comply (no response)

  The server failed while handling the agent_name request.

server unable to comply (bad response)

  The server failed while handling the agent_name request.

server unable to comply (server error)

  Another agent is using the symbolic name.  A simple way for a child agent
  to ensure a unique name is to prefix the name with the identification of
  the root agent.  Any other agent in the same hierarchy will have the
  same root agent and thus will know the prefix.  An alternative approach is
  to prefix the name with the agent's owner or an application-specific string.
  This latter approach works for both root and child agents.  The "server
  unable to comply error (server error)" message can also mean that the
  controlling server has crashed and been restarted.  In this case the agent
  is no longer in the server's internal tables and is effectively dead.
```

- agent_root

  An agent uses the agent_root command to become a root agent. The local section of the agent array is copied into the root section. The old root identification is overwritten so the agent is effectively disconnected from its hierarchy unless it chooses to remember the old root identification in a separate variable. agent_root returns the empty string.

  Example:

  Suppose that an agent has local identification "bald.cs.dartmouth.edu 129.170.192.98 ftp_remote 10" and root identification "moose.cs.dartmouth.edu 129.170.194.28 ftp_local 15". Issuing the command

```
agent_root
```

makes the root identification the same as the local identification. In other words the root identification becomes "bald.cs.dartmouth.edu 129.170.192.98 ftp_remote 10". The *agent* array is updated with the new root identification. The agent is now a root agent.

Error messages:

On error, agent_root raises a standard Tcl exception that can be caught with the *catch* command. agent_root returns one of the following error messages. The indented text below each error message describes the possible interpretations of the message.

```
wrong number of arguments

  You specified the wrong number of arguments.

agent has NOT been registered

  The agent does not have a controlling server.
```

- agent_end [-time *seconds*]

  An agent uses the agent_end command to notify its controlling server that it is finished with its task. The controlling server removes the agent from its internal tables. *seconds* is the maximum number of seconds to wait for a response from the controlling server. *seconds* can be a fractional number of seconds and defaults to 15 if the *-time* parameter is not specified. agent_end returns the empty string.

  Only a root agent can call agent_end. Child agents are automatically removed on termination. The reason for an explicit agent_end in a root agent is that a root agent might have sections where it needs agent services and sections where it does not. Rather than taking up server resources for its entire duration, a root agent can contain multiple agent_begin/agent_end pairs, one pair for each section where it needs agent services.

  Note:

  It is bad form for a root agent to exit without calling agent_end. Explicitly calling agent_end ensures the quickest possible reclamation of server resources.

  Example:

  Issuing the command

```
agent_end
```

  detaches the agent from its controlling server. The root and local identifications are reset to the empty string. The agent can not use any of the agent commands until it makes another call to agent_begin.

  Error messages:

  On error, agent_end raises a standard Tcl exception that can be caught with the *catch* command. agent_end returns one of the following error messages. The indented text below each error message describes the possible interpretations of the message.

```
wrong number of arguments

  You specified the wrong number of arguments.

agent has NOT been registered

  The agent does not have a controlling server.

unable to send to server
```

```
  The agent was unable to contact its controlling server within the specified
  number of seconds.  This means that the controlling server has crashed, the
  controlling server's machine has crashed, or the controlling server is
  temporarily unreachable due to network failure or overload.

server unable to comply (no response)

  The server failed while handling the agent_end request.

server unable to comply (bad response)

  The server failed while handling the agent_end request.

server unable to comply (server error)

  The controlling server has crashed and been restarted.  In this case the
  agent is no longer in the server's internal tables and is effectively dead.
```

If the agent is unable to contact its server, the agent's identifications are reset to the empty string *anyways*. The agent does not need to retry the agent_end command again.

- agent_force *identification* [-time *seconds*]

  In pathological situations an agent can terminate in such a way that the server does not remove the agent from its internal tables. This means that the server will incorrectly report that the agent's symbolic name is in use. In other situations you might want to terminate the old version of an agent when starting up a new version. The agent_force command is used in both cases.

  The agent_force command forcibly removes an agent from the server tables (and forcibly terminates the agent if the agent is running and the server has sufficient authority). *identification* specifies the agent that should be forcibly removed. *seconds* is the maximum number of seconds to wait for a response from the agent's server. *seconds* can be a fractional number of seconds and defaults to 15 if the *-time* parameter is not specified. agent_force returns "-1" if the specified agent does not exist. Otherwise agent_force returns the *complete* identification of the agent that was forcibly removed.

  Notes:

  1. For most agents, it is reasonable to have an agent_force command immediately before every agent_name command in order to ensure that the desired symbolic name is not in use. This is not true for every agent since agent_force removes all messages and meeting requests as well as the agent's symbolic name. For example, if two agents wish to exchange messages, they would obtain a controlling server using agent_begin, register their symbolic names using agent_name and then send messages to each other using agent_send. If one of the agents forcibly removes its desired symbolic name, it might also forcibly remove one or more of the messages that have been sent from the other agent.
  2. There are no security checks on the agent_force command. An agent can forcibly remove any other agent from the server tables. Therefore use the agent_force command with care and make sure that you use unique symbolic names. One way to ensure a unique symbolic name is to prefix the name with the root identification, the owner's userid or an application-specific string. agent_force will be removed or reworked as the fault-tolerance, namespace and security components of Agent Tcl are implemented and improved.
  3. An agent can forcibly remove *itself* if desired. This, however, is not a good idea.

  Example:

Most agents would use the following command sequence to obtain a controlling server and register their symbolic name. The sequence assumes that the desired symbolic name is in the Tcl variable *name*.

```
agent_begin
agent_force "$agent(local-ip) $name"
agent_name $name
```

The agent_begin command registers the agent with the server on the current machine. Then the agent_force command forcibly removes the desired symbolic name from the server tables (remember that *agent(local-ip)* contains the IP address of the controlling server). Finally the agent_name command registers the desired symbolic name.

Error messages:

On error, agent_force raises a standard Tcl exception that can be caught with the *catch* command. agent_force returns one of the following error messages. The indented text below each error message describes the possible interpretations of the message.

```
wrong number of arguments

  You specified the wrong number of arguments.

agent has NOT been registered

  The agent does not have a controlling server.

agent identification must be ...

  The identification was specified incorrectly.

unable to convert name

  The system was unable to determine the network address of the specified
  machine.

unable to send to server

  The system was unable to contact the controlling server of the specified
  agent within the given number of seconds.  This means that the specified
  machine has crashed or does not exist, the server is not running on the
  specified machine, or the specified machine is temporarily unreachable
  due to network failure or overload.  The standard response to this error is
  to retry the command at least once.

server unable to comply (no response)

  The server failed while handling the agent_force.

server unable to comply (bad response)

  The server failed while handling the agent_force.

server unable to comply (server error)

  The server does not accept connections from the agent's current machine, i.e.,
  the agent's current machine is NOT in the server's access list.
```

7.3 Messages

Agent Tcl provides a message passing facility. A message consists of a numeric code and a string.

- agent_send *destination* [*integer*] *string* [-time *seconds*]

  agent_send sends a message to another agent. *integer* is the numeric code and *string* is the string. The numeric code defaults to 0 if it is not specified. *destination* is the identity of the recipient agent. *destination* is either a standard 4-element identification

  ```
  bald.cs.dartmouth.edu  129.170.192.98  calculator_agent 100
  ```

  or any 2-element identification that uniquely identifies the recipient

  ```
  bald.cs.dartmouth.edu  calculator_agent
  bald.cs.dartmouth.edu  100
  129.170.192.98  calculator_agent
  129.170.192.98  calculator_agent
  ```

  Such a 2-element list must include either the machine name or machine IP *and* either the recipient's name or recipient's ID. It is not necessary to specify the full Internet name of the machine. Any name that the local name server maps to the desired machine is acceptable. *seconds* is the maximum number of seconds to wait for a response from the destination machine. *seconds* can be a fractional number of seconds and defaults to 15 if the *-time* parameter is not specified. agent_send returns the empty string.

  Example:

  An agent would use the following command to send the message {0, INIT_CALC} to the calculator agent on *bald*.

  ```
  agent_send "bald calculator_agent" 0 INIT_CALC
  ```

  Error messages:

  On error, agent_send raises a standard Tcl exception that can be caught with the *catch* command. agent_send returns one of the following error messages. The indented text below each error message describes the possible interpretations of the message.

  ```
  wrong number of arguments

    You specified the wrong number of arguments.

  agent has NOT been registered

    The sender does not have a controlling server.

  agent identification must be ...

    The recipient identification was specified incorrectly.

  unable to convert name

    The system was unable to determine the network address of the recipient
    machine.
  ```

```
unable to send to server

  The sender was unable to contact the recipient server within the specified
  number of seconds.  This means that the recipient machine has crashed or
  does not exist, the server is not running on the recipient machine, or the
  recipient machine is temporarily unreachable due to network failure or
  overload.  The standard response to this error is to retry the command at
  least once.

server unable to comply (no response)

  The recipient server failed while processing the message.

server unable to comply (bad response)

  The recipient server failed while processing the message.

server unable to comply (server error)

  There is no agent at the recipient server with the specified numeric ID.
  Note that no error occurs if the sender specifies a symbolic name and there
  is no agent with that name.  Instead the server buffers the message and
  transfers it to the first agent that requests the name with the agent_name
  command.  This buffering allows a set of agents to request symbolic names
  and then send messages to each other without having to worry that the
  intended recipient has not executed its agent_name command yet.  This of
  course assumes that each agent knows what names the other agents will
  request.  Therefore it is most useful when a parent agent creates several
  child agents that must communicate with each other.  The "server unable to
  comply (server error)" message can also mean that the server does not accept
  connections from the sender's current machine, i.e., the sender's current
  machine is NOT in the server's access list.
```

- agent_receive *code_var string_var* <-blocking | -time *seconds* | -nonblocking>

  agent_receive is used to receive a message. The command has blocking, nonblocking and timed forms. The blocking form waits until a message is available. Then it sets the variable *code_var* to the message code, sets the variable *string_var* to the message string, and returns the 4-element identification of the sender. The nonblocking form returns -1 if there is no message available. Otherwise it sets *code_var* and *message_var* to the message code and string respectively and returns the 4-element identification of the sender. The timed form returns -1 if no message arrives before the specified number of seconds has elapsed. Otherwise it sets *code_var* and *message_var* to the message code and string respectively and returns the 4-element identification of the sender.

  Examples:

  Suppose that agent "moose.cs.dartmouth.edu 129.170.194.28 {} 25" issues the agent_send command above and that the calculator agent on bald issues the command

  ```
  agent_receive code string -blocking
  ```

  The calculator agent sleeps until the message arrives. Then the variable *code* is set to the message code "0". The variable *string* is set to the message string "INIT_CALC". Finally agent_receive returns the identification of the sender which in this case is "moose.cs.dartmouth.edu 129.170.192.98 {} 25". If the calculator agent issues the command

```
agent\_receive code string -nonblocking
```

and the message has not arrived, the command returns -1. Otherwise the command fills in the variables and returns the sender identification as before. If the calculator agent issues the command

```
agent\_receive code string -time 5.0
```

and the message does not arrive within 5 seconds, the command returns -1. Otherwise the command fills in the variables and returns the sender identification as before.

Error messages:

On error, agent_receive raises a standard Tcl exception that can be caught with the *catch* command. agent_receive returns one of the following error messages. The indented text below each error message describes the possible interpretations of the message.

```
wrong number of arguments

  You specified the wrong number of arguments.

agent has NOT been registered

  The agent does not have a controlling server.

broken connection to server

  The agent's controlling server has crashed.  The agent is effectively dead.
```

7.4 Events

An *event* consists of a tag and a body. Both the tag and the body are arbitrary strings. Events will eventually be used for asynchronous notification of important occurrences. In the current system, however, events are exactly the same as messages except for the use of a tag rather than a numeric code. Despite the similarity, events are often a more convenient communication mechanism since the tag can be a descriptive string.

The agent_event command is used to send an event. The agent_getevent command is used to receive an event. These commands have the same syntax, semantics and error messages as *agent_send* and *agent_receive* respectively, i.e.,

agent_event *destination tag string* [-time *seconds*]

and

agent_getevent *tag_var string_var* <-blocking | -time *seconds* | -nonblocking>

7.5 Migration

An agent migrates when it moves from one machine to another. An agent can migrate to a machine only if the server is running on that machine. There are three migration-related commands. agent_jump migrates the agent, agent_fork clones the agent and agent_submit creates a new agent.

Keep two things in mind through this section.

1. A child agent created with the agent_submit or agent_fork commands executes with the authority of the agent *server*.
2. As soon as an agent migrates for the first time, it executes with the authority of the agent *server*. It will execute with the authority of the *server* until it terminates.

This, of course, will change in future releases of the system. In the meantime you must make sure that the *server* has the authority to do whatever your child or migrated agent is trying to do. In other words you must give userid *agent* the appropriate access permissions. Recall that *agent* is the userid that was created specifically for the agent servers.

- agent_submit *machine* [-procs *name name ...*] [-vars *name name* [-time *seconds*] ...] -script *script*

  agent_submit submits the Tcl script *script* to the server on machine *machine*. *seconds* is the maximum number of seconds to wait for a response from the server. *seconds* can be a fractional number of seconds and defaults to 15 if the *-time* parameter is not specified.

  agent_submit raises an error if it is unable to contact the server within the specified number of seconds. Otherwise the specified script becomes an agent under the server's control and executes on the server's machine. The agent that issues the agent_submit command is the *parent* of the new child agent. If the child agent needs to access variables or procedures that are defined in the parent, these variables and procedures should be listed after the *-vars* and *-procs* flags respectively. The variables and procedures are sent to the destination server along with the script. Note that a transmitted variable is a *copy* of the parent variable. Changes in one are never seen in the other. In addition each transmitted variable becomes a *global* variable in the child agent.

  The agent_submit command returns the 4-element identification of the new child agent. In the child agent the root section of the *agent* array is the same as in the parent while the local section is set to the child's identification. In other words the child has the same root agent as its parent and its own unique identification.

  Every Tcl script has a result. The result of the script is the result of the last command executed in that script. When the child agent finishes executing, a message is automatically sent to the root agent. The message code indicates the way in which the child agent terminated. The two possible codes are

```
0 = normal
1 = error
```

  On normal termination the message string is the script result. On error the message string is a three-element list where the first element is the error string, the second element is the contents of the Tcl variable *errorCode* and the third element is the contents of the Tcl variable *errorInfo*. These automatic messages are received with the agent_receive command just like any message.

  Note:

  An automatic message is *not* sent if the child agent explicitly issues the *exit* command.

  Example:

  The following agent asks for an integer and then computes the factorial of the integer on a *different machine*. For clarity the script does not perform any error checking.

```
#!/usr/contrib/bin/agent
#
# fac.remote.tcl
#
# This agent asks the user for an integer and then computes the factorial
# of that integer on a DIFFERENT machine.

proc factorial x {

    if {$x <= 1} {
        return 1;
    }
```

```
    expr $x * [factorial [expr $x - 1]]
}

  # ask the user for a machine

puts "Enter the name of the remote machine:"
gets stdin machine

  # get a controlling server

agent_begin

  # compute factorials

set number 0

while {$number != -1} {

  puts "Enter a nonnegative integer (-1 to quit): "
  gets stdin number

  if {$number != -1} {

      # compute the factorial on the remote machine.  We submit the script
      # "factorial $number" along with procedure "factorial" and variable
      # "number" since the script needs these two things

    agent_submit $machine -vars number -procs factorial \
                          -script {factorial $number}

      # the result of the submitted agent is the result of the last
      # command executed in that agent -- i.e. the return value of
      # procedure "factorial".  When the submitted agent ends, this
      # result is automatically sent to the root agent (which is this
      # agent).  We just wait until we receive the message.

    agent_receive code result -blocking

      # output the result

    puts "$number! is equal to $result\n"
  }
}

  # done

agent_end
```

Error messages:

On error, agent_submit raises a standard Tcl exception that can be caught with the *catch* command. agent_submit returns one of the following error messages. The indented text below each error message describes the possible interpretations of the message.

```
wrong number of arguments

  You specified the wrong number of arguments.

must specify a script

  You did not specify a script.

"-script" must be followed by a script

  You did not specify a script.

procedure ... does not exist

  One of the procedures listed after the -procs flag does not exist.

variable ... does not exit

  One of the variables listed after the -vars flag does not exist.

... is immobile

  The indicated variable or procedure is "immobile" and can not be transferred
  to a remote machine.

unable to send to server

  The destination server did not respond within the specified number of
  seconds.  This means that the destination machine has crashed or does not
  exist, the server is not running on the destination machine, or the
  destination machine is temporarily unreachable due to network failure or
  overload.  The standard response to this error is to retry the command at
  least once.

server unable to comply (no response)

  The destination server failed while handling the agent_submit.

server unable to comply (bad response)

  The destination server failed while handling the agent_submit.

server unable to comply (server error)

  The destination server does not accept connections from the agent's current
  machine, i.e., the agent's current machine is NOT in the server's access
  list.
```

Note that a child agent is *never* created if agent_submit fails.

- agent_jump *machine* [-time *seconds*]

  agent_jump migrates the agent to machine *machine*. The agent server must be running on *machine*. *seconds* is the maximum number of seconds to wait for a response from the server. *seconds* can be a fractional number of seconds and defaults to 15 if the *-time* parameter is not specified.

agent_jump suspends the execution of the agent, captures the internal state of the agent, and transmits the internal state to the server. The server restores the state image and assigns a new identification to the agent. The agent then resumes execution at the statement immediately after the agent_jump. agent_jump returns "SAME" if the destination machine is the same as the current machine and "JUMPED" if the destination machine is a different machine. The local section of the *agent* array is updated with the new local identification. The root section remains the same unless the agent is a root agent. In this case the root section is updated with the new identification as well.

Notes:

1. An agent loses its symbolic name when it jumps since the symbolic name might be in use on the destination machine. The agent must request the symbolic name after it jumps.
2. An agent loses its connection with the *console* when it jumps and can not regain this connection even when it returns to the local machine. An agent application that needs to use the *console* must be written as two agents. One agent is stationary and maintains the console connection. This stationary agent submits a mobile agent using agent_submit. The mobile agent migrates through the network as desired and returns its results to the stationary agent. Note that a *Agent Tk* agent can regain its screen connection (by contacting the X server when it returns to the machine). An alternative, therefore, is to write the agent in Tk and use a GUI for all interaction.
3. agent_jump can not be used when Tcl is in interactive mode. agent_jump makes no sense in interactive mode since the user can not interactively enter Tcl commands if the agent has migrated to a remote machine.

Examples:

The following two agents illustrate agent_submit and agent_jump. Both agents execute the *who* command on multiple machines and present a list of users to the agent's owner. The first agent submits one agent *to each machine*. Each child agent executes the *who* command on its machine and returns the list of users to the parent.

```
#!/usr/contrib/bin/agent
#
# who.tcl
#
# This agent executes the "who" command on multiple machines.  It submits one
# child agent FOR EACH MACHINE.  Each child executes the "who" command on its
# machine and returns the list of users.

  # Procedure WHO executes the "who" command

proc who args {

  global agent
  set users [exec who]
  return "$agent(local-server):\n$users\n"

}

  # list of machines -- make sure that you replace these with your machines!

set machines "bald.cs.dartmouth.edu \
              cosmo.dartmouth.edu \
              lost-ark.dartmouth.edu \
              temple-doom.dartmouth.edu \
              moose.cs.dartmouth.edu \
```

```
              muir.cs.dartmouth.edu \
              tenaya.cs.dartmouth.edu \
              tioga.cs.dartmouth.edu \
              tuolomne.cs.dartmouth.edu"

  # register the agent

if {[catch {agent_begin}]} {
  return -code error "ERROR: unable to register on $agent(actual-server)"
}

  # catch any error

if {[catch {

    # submit the "who" agents

  set submitted 0

  foreach m $machines {

    set script "agent_submit $m -procs who -script {who}"

    if {[catch $script]} {
      puts "ERROR: unable to submit to $m"
    } else {
      incr submitted
    }

  }

    # wait for each child to automatically send back the list of users

  puts "\nWHO'S WHO on our computers\n"

  for {set i 1} {$i <= $submitted} {incr i} {
    set source_id [agent_receive code message -blocking]
    puts $message
  }

} error_message]} then {

    # make sure that we clean up on error

  agent_end

  return -code error -errorcode $errorCode -errorinfo $errorInfo $error_message

}

  # clean up

agent_end
```

The second agent submits a *single* child agent that migrates from machine to machine. The child executes the *who* command on each machine and appends the users to a growing list. This list is carried along with the child agent as it migrates and is returned to the parent when the child agent finishes.

```
#!/usr/contrib/bin/agent
#
# who.jump.tcl
#
# This agent executes the "who" command on multiple machines.  It submits
# a SINGLE child agent.  The child jumps from machine to machine and
# executes the WHO command on each machine.

  # Procedure WHO is the child agent that does the jumping.

proc who machines {

  global agent

    # start with an empty list

  set list ""

    # jump from machine to machine

  foreach m $machines {

      # if we do not jump successfully append an error message
      # otherwise append the list of users

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

  return $list
}

  # list of machines -- make sure that you replace these with your machines!

set machines "bald.cs.dartmouth.edu \
              cosmo.dartmouth.edu \
              lost-ark.dartmouth.edu \
              temple-doom.dartmouth.edu \
              moose.cs.dartmouth.edu \
              muir.cs.dartmouth.edu \
              tenaya.cs.dartmouth.edu \
              tioga.cs.dartmouth.edu \
              tuolomne.cs.dartmouth.edu"

  # register the agent
```

```
if {[catch {agent_begin}]} {
  return -code error "ERROR: unable to register on $agent(actual-server)"
}

  # catch any error

if {[catch {

    # submit the agent that does the jumping

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

    # wait for the list of users that is automatically returned from the
    # child agent

  agent_receive code message -blocking

    # output the list of users

  puts "\nWHO'S WHO on our computers\n\n$message"

} error_message]} then {

    # make sure that we clean up on error

  agent_end

  return -code error -errorcode $errorCode -errorinfo $errorInfo $error_message

}

  # clean up

agent_end
```

Both of these agents produced the following output during a test run.

```
WHO'S WHO on our computers

bald.cs.dartmouth.edu:
rgray    tty6     Aug 20 21:30
rgray    ttyp1    Aug 20 21:32 (:0.0)

cosmo.dartmouth.edu:


lost-ark.dartmouth.edu:
megumi     ttyq0        Aug 21 08:32
gvc        ttyq3        Aug 15 14:32

temple-doom.dartmouth.edu:
```

```
megumi     ttyq0        Aug 21 08:33

moose.cs.dartmouth.edu:
rgray    ttyp0   Aug 21 08:56 (bald.cs.dartmouth)

muir.cs.dartmouth.edu:
hershey     :0           Aug 16 11:06
hershey     ttyp2        Aug 16 11:06
hershey     ttyp3        Aug 16 11:06

tenaya.cs.dartmouth.edu:
rgray

tioga.cs.dartmouth.edu:
mdengler    :0           Aug 11 19:23
olson       ttyp3        Aug 11 11:52

tuolomne.cs.dartmouth.edu:
rgray
```

Error messages:

On error, agent_jump raises a standard Tcl exception that can be caught with the *catch* command. agent_jump returns one of the following error messages. The indented text below each error message describes the possible interpretations of the message.

```
wrong number of arguments

  The command takes one argument

agent has NOT been registered

  The agent does not have a controlling server.

unable to send to server

  The destination server did not respond within the specified number of
  seconds.  This means that the destination machine has crashed or does not
  exist, the server is not running on the destination machine, or the
  destination machine is temporarily unreachable due to network failure or
  overload.  The standard response to this error is to retry the command at
  least once.

server unable to comply (no response)

  The destination server failed while handing the agent_jump.

server unable to comply (bad response)

  The destination server failed while handling the agent_jump.

server unable to comply (server error)

  The destination server does not accept connections from the agent's current
```

```
machine, i.e., the agent's current machine is NOT in the server's access
list.
```

Note that the agent *never* changes network location when an error occurs. Instead it continues running on the current machine.

- agent_fork *machine*

  agent_fork is analogous to Unix fork. The command creates a copy of the agent on the specified *machine*. The agent server must be running on *machine*. *seconds* is the maximum number of seconds to wait for a response from the server. *seconds* can be a fractional number of seconds and defaults to 15 if the *-time* parameter is not specified.

  The server assigns an identification to the new child. Then the parent and child continue execution from the point at which the fork occurred. agent_fork returns the string "CHILD" in the child agent and the 4-element identification of the child in the parent agent. In the child the local section of the *agent* array is set to the child's identification while the root section is the same as in the parent. In the parent the *agent* array is unchanged.

  Notes:

  1. Initially the child has no symbolic name. The child can request a symbolic name with the agent_name command if desired.
  2. agent_fork can not be used when Tcl is in interactive mode. agent_fork makes no sense in interactive mode since (1) the user can not interactively enter commands to the child if the child is on a remote machine and (2) the parent and child would compete for user input if the child is on the same machine.

  Examples:

  The following Tcl script is a skeleton for any agent that performs a fork.

```
#!/usr/contrib/bin/agent

 # change this to the desired machine

set machine tioga.cs.dartmouth.edu

 # register the agent

agent_begin

 # fork

if {[agent_fork $machine] == "CHILD"} {

    # child processing here

} else {

    # parent processing here
    # wherever you put the agent_end, make sure that only the parent does it

  agent_end

}
```

Error messages:

On error, agent_fork raises a standard Tcl exception that can be caught with the *catch* command. agent_fork returns one of the following error messages. The indented text below each error message describes the possible interpretations of the message.

```
wrong number of arguments

  The command takes one argument.

agent has NOT been registered

  The agent does not have a controlling server.

unable to send to server

  The destination server did not respond within the specified number of
  seconds.  This means that the destination machine has crashed or does not
  exist, the server is not running on the destination machine, or the
  destination machine is temporarily unreachable due to network overload or
  error.  The standard response to this error is to retry the command at least
  once.

server unable to comply (no response)

  The destination server failed while handling the agent_fork.

server unable to comply (bad response)

  The destination server failed while handling the agent fork.

server unable to comply (server error)

  The destination server does not accept connections from the agent's current
  machine, i.e., the agent's current machine is NOT in the server's access
  list.
```

Note that a child agent is *never* created if agent_fork fails.

7.6 Limitations of jump and fork

agent_fork and agent_jump capture the internal state of a Tcl script and transmit this state to a remote machine. Currently these commands capture most *but not all* of the internal state.

1. Masks are not captured. You must recreate any desired masks after a call to agent_jump or agent_fork. See the "Mask" section below. This problem will be fixed soon.
2. Timeouts are not captured. The *restrict* command will not work if the enclosed script jumps or forks. See the "Timing" section below. This problem will be fixed soon.
3. Do not fork or jump inside a variable trace or inside a procedure that the *lsort* command is using to compare list elements. You will get the error message

```
script is not interruptable
```

if you try.

4. Do not fork or jump from inside the *history* commands. You will get the error message

   ```
   script is not interruptable
   ```

   if you try.

5. The following portions of the state are simply ignored:

   - deletion callbacks,
   - open files,
   - linked variables,
   - variable traces,
   - command traces,
   - interrupt handlers,
   - child processes,
   - array searches,
   - user-defined math functions,
   - history lists
   - and the internal state of *all Tcl extensions*.

   Most of these are inherently nontransportable due to their close ties to a particular machine, to underlying C code or to the size of internal Tcl tables. Work on saving and restoring the state of Tcl extensions is in progress. The expected solution is to allow Tcl extensions to register a state handler with the Tcl core. The Tcl core will call these handlers during state transfer. Each handler is responsible for saving and loading the state of its extension. Note that you can use all of the listed constructs; the limitation is that you can not create or define the construct on one machine and then transmit it to another machine.

All of these Tcl features are described in [Ous94] and [Wel95]. The most important thing to note is that it is nearly impossible to use them accidentally. If you do not know what they are *or* do not think that you are using them, you are not using them.

7.7 Meetings

A *meeting* is a direct connection between two agents. A direct connection provides more efficient data transfer than message passing.

- agent_meet *recipient*

  The agent_meet command is used to establish a meeting with a recipient agent. *recipient* is the 4-element identification of the recipient (or any 2-element shorthand that uniquely identifies the recipient). agent_meet waits until the recipient has accepted or rejected the meeting. There are three ways for the recipient to accept the meeting.

  1. The recipient can wait for the meeting request with the get_meeting command and then accept the request with the accept_meeting command.
  2. The recipient can wait for and automatically accept the request with the agent_accept command.
  3. The recipient can issue an agent_meet command at the same time as the source agent (specifying the source agent as *its* recipient).

The first two techniques are useful for agents that must accept meetings from unknown sources. The third technique is useful when a pair of agents wants to establish a meeting. There is only one way for the recipient to reject the meeting.

1. The recipient must wait for the meeting request with the get_meeting command and then reject the request with the reject_meeting command.

agent_meet raises a standard Tcl exception if the recipient refuses the meeting and returns a socket descriptor if the recipient accepts the meeting. A socket descriptor is an integer greater than or equal to 1 and can be thought of as a meeting identification number in this context. Writing to the socket descriptor with the tcpip_write command sends a string to the recipient agent. Reading from the socket descriptor with the tcpip_read command receives a string from the recipient agent.

Examples:

Here are two agents that meet with each other to exchange status information. The agents assume that they have the same controlling server. The first agent is the email agent.

```
#!/usr/contrib/bin/agent
#
# email.tcl
#
# This is the "email_agent".  It uses agent_meet to establish a direct
# connection with the "file_agent" and exchange status information.  It
# assumes that the "file_agent" has the same controlling server.

  # register the agent on the current machine

if {[catch agent_begin]} {
  return -code error "unable to register the agent on $agent(actual-server)"
}

if {[catch {

    # this is the "email_agent"

  agent_name email_agent

    # meet with the "file_agent"

  set sockfd [agent_meet "$agent(local-ip) file_agent"]

    # exchange status information

  tcpip_write $sockfd "{NEW_MESSAGES 0} {IN_BOX_MESSAGES 10}"
  set status [tcpip_read $sockfd -blocking]
  tcpip_close $sockfd

    # display the status information from the other agent

  puts "$agent(local):\nThe file agent says ... $status\n"

} error_message]} then {

    # make sure that we clean up on error
```

```
  agent_end

  return -code error -errorcode $errorCode -errorinfo $errorInfo $error_message

}

  # clean up

agent_end
```

The second agent is the file agent.

```
#!/usr/contrib/bin/agent
#
# file.tcl
#
# This is the "file_agent".  It uses agent_meet to establish a direct
# connection with the "email_agent" and exchange status information.  It
# assumes that the "email_agent" has the same controlling server.

  # register the agent on the current machine

if {[catch agent_begin]} {
  return -code error "unable to register the agent on $agent(actual-server)"
}

if {[catch {

    # this is the "file_agent"

  agent_name file_agent

    # meet with the "email_agent"

  set sockfd [agent_meet "$agent(local-ip) email_agent"]

    # exchange status information

  tcpip_write $sockfd "{FREE_SPACE 65536 KB}"
  set status [tcpip_read $sockfd -blocking]
  tcpip_close $sockfd

    # display the status information from the other agent

  puts "$agent(local):\nThe email agent says ... $status\n"

} error_message]} then {

    # make sure that we clean up on error

  agent_end
```

```
  return -code error -errorcode $errorCode -errorinfo $errorInfo $error_message

}

  # clean up

agent_end
```

Running these two scripts in two different terminal windows on the author's machine produced the following output.

```
% email.tcl
bald.cs.dartmouth.edu 129.170.192.98 email_agent 12:
The file agent says ... {FREE_SPACE 65536 KB}
%

% file.tcl
bald.cs.dartmouth.edu 129.170.192.98 file_agent 13:
The email agent says ... {NEW_MESSAGES 0} {IN_BOX_MESSAGES 10}
%
```

A scan though the code should convince you that the agents did indeed exchange their status strings.

Note:

If an agent specifies the meeting recipient by symbolic name and there is no recipient with that name, the system will buffer the meeting request and transfer the request to the first agent that registers the name (just like when sending message and events). Thus the two agents above will work even if one agent issues its agent_meet command before the other has issued its agent_name command.

Error messages:

On error agent_meet raises a standard Tcl exception that can be caught with the *catch* command. agent_meet returns one of the following error messages. The text below each error message describes the possible interpretations of the message (the *source* agent is the agent issuing the agent_meet command; the *recipient* agent is the agent that accepts or rejects the meeting; the *source server* is the controlling server of the source agent; the *recipient server* is the controlling server of the recipient agent; etc.).

```
wrong number of arguments

  You specified the wrong number of arguments.

agent has NOT been registered

  The source agent does not have a controlling server.

unable to send to server

  The recipient machine has crashed or does not exist, the server is not
  running on the recipient machine, the recipient machine is temporarily
  unreachable due to network failure or overload, the source machine has
  crashed, the source server has crashed, or the source machine is temporarily
  unreachable due to network failure or overload.  The standard response to
  this error is to retry the command at least once.

server unable to comply (bad response)
```

The source or recipient server failed while attempting to complete the meeting.

server unable to comply (no response)

The source or recipient server failed while attempting to complete the meeting.

server unable to comply (server error)

The recipient does not exist or the source server has failed and been restarted since the call to agent_begin. In the latter case the source agent is no longer in the server tables and is effectively dead. Remember that if the recipient is specified by name, the meeting request will be buffered and transferred to the first agent that requests that name. The "server unable to comply (server error)" message can also mean that the destination server does not accept connections from the agent's current machine, i.e., the agent's current machine is NOT in the server's access list.

recipient has refused the request for a meeting

The recipient has refused the meeting request. The errorCode variable is set to REFUSED.

unable to create socket

The system was unable to open a local socket.

unable to connect socket

The system was unable to connect the local socket to a socket on the recipient machine.

protocol error: expected ... but got ...

The meeting commands are implemented with lower level primitives and obey a specific protocol. This error means either that there is a bug in the meeting commands or that the recipient is using custom meeting commands that do not obey the protocol. The errorCode variable is set to PROTOCOL.

an agent can not meet with itself

The agent is attempting to meet with itself.

unable to bind socket

The system was unable to associate a port with the local socket.

unable to listen to socket

```
  The system was unable to associate a connection queue with the local
  socket.

unable to accept

  The other agent is attempting to connect to the local socket.  The system
  was unable to accept the connection.

agent identification must be ...

  The recipient was specified incorrectly.

unable to convert name

  The system was unable to determine the network address of the recipient
  machine.
```

- tcpip_write *sockfd string*

  The tcpip_write command is used to send a string along a direct connection. *sockfd* is the socket descriptor returned from the agent_meet, accept_meeting or agent_accept commands. *string* is the string that should be sent to the agent on the other end of the connection. tcpip_write returns the empty string.

  Examples:

  In the file agent above, the command

```
tcpip_write $sockfd "{FREE_SPACE 65536 KB}"
```

  sends the string

```
{FREE_SPACE 65536 KB}
```

  along the direct connection to the email agent at the other end.

  Error messages:

  On error, tcpip_write raises a standard Tcl exception that can be caught with the *catch* command. tcpip_write returns one of the following error messages. The indented text below each error message describes the possible interpretations of the message.

```
wrong number of arguments

  You specified the wrong number of arguments.

unable to write

  The socket descriptor is invalid or has become disconnected.  A socket
  becomes disconnected if the agent on other end closes its end of the
  connection, if the agent on the other end terminates, or if the machine on
  the other end crashes.
```

- tcpip_read *sockfd* <-nonblocking | -time *seconds* | -blocking>

  tcpip_read is used to read a string from a direct connection. *sockfd* is the socket descriptor returned by the agent_meet, accept_meeting or agent_accept commands. tcpip_read has three forms. The *blocking*

form waits until a string is available on the connection and then returns the string. The *timed* form raises a Tcl exception if a string does *not* arrive before the sepcified number of seconds has elapsed. Otherwise the *timed* form returns the string. The *nonblocking* form raises a Tcl exception if no string is available immediately. Otherwise the *nonblocking* form returns the string.

Example:

Continuing with the example above, the email agent issues the command

```
set status [tcpip_read $sockfd -blocking]
```

in order to receive the status string from the file agent. The *status* variable is set to

```
{FREE_SPACE 65536 KB}
```

which is the string that the file agent sent using the tcpip_write command.

Error messages:

On error, tcpip_read raises a standard Tcl exception that can be caught with the catch command. tcpip_read returns one of the following error messages. The indented text below each error message describes the possible interpretations of the message.

```
timeout

  The -nonblocking option was specified and no string was available OR the
  -time option was specified and no string was available before the specified
  number of seconds expired.  The errorCode variable is set to TIMEOUT.

wrong # of arguments

  You specified the wrong number of arguments.

unable to read

  The socket descriptor is invalid or has become disconnected.  A socket
  becomes disconnected if the agent on other end closes its end of the
  connection, if the agent on the other end terminates, or if the machine
  on the other end crashes.
```

- tcpip_close *sockfd*

  tcpip_close is used to close a direct connection. *sockfd* is the socket descriptor returned by the agent_meet, accept_meeting or agent_accept commands. Once the connection is closed, all tcpip_read and tcpip_write commands on that connection will fail.

  Note:

  tcpip_close will close the given descriptor even it does not refer to a meeting. Be sure that you specify the correct descriptor! This will be fixed soon.

  Examples:

  As soon as the *email agent* and the *file agent* have exchanged status strings, they close the direct connection with the command

  ```
  tcpip_close $sockfd
  ```

Issuing this command is actually unnecessary since all of an agent's direct connections are closed automatically when the agent terminates. Explicitly issuing the command is good form, however, and conserves system resources if the agent continues executing after it has finished with the direct connection.

Error messages:

On error, tcpip_close raises a standard Tcl exception that can be caught with the *catch* command. tcpip_close returns one of the following error messages. The indented text below each error message describes the possible interpretations of the message.

```
wrong number of arguments

  You specified the wrong number of arguments.
```

- get_meeting *location_var* <-blocking | -nonblocking>

  The agent_meet command is used when one agent wishes to meet with a specific second agent. The get_meeting, accept_meeting, reject_meeting and agent_accept commands are used when an agent wishes to meet with any agent that requests a meeting.

  The get_meeting command is used to receive a *meeting request*. The command has both a blocking and a nonblocking form. The blocking form waits until a meeting request is available. Then it sets *location_var* to a 3-element list

```
machine_name machine_IP port_number
```

  that specifies the *actual* network location of the requesting agent. This is necessary since an agent might not be on the same machine as its controlling server. Finally the blocking form returns the 4-element identification of the requester. The nonblocking form returns -1 if there is no meeting request available. Otherwise it sets *location_var* and returns the requester identification as in the blocking case.

  Examples:

  A long example appears under the accept_meeting command.

  Error messages:

  On error, get_meeting raises a standard Tcl exception that can be caught with the *catch* command. get_meeting returns one of the following error messages. The indented text below each error message describes the possible interpretations of the message.

```
wrong number of arguments

  You specified the wrong number of arguments.

agent has NOT been registered

  The agent does not have a controlling server.

unable to send to server

  The controlling server has crashed, the controlling server's machine has
  crashed, or the controlling server is temporarily unreachable due to network
  failure or overload.  The standard response to this error is to retry the
  command at least once.

server unable to comply (no response)
```

```
  The controlling server failed while handling the get_meeting.

server unable to comply (bad response)

  The controlling server failed while handling the get_meeting.

server unable to comply (server error)

  The controlling server has failed and been restarted since the call to
  agent_begin.  In this case the agent is no longer in the server tables and
  is effectively dead.

protocol error: expected REQUEST but got ...

  The meeting commands are implemented with lower level primitives and obey
  a specific protocol.  This error means either that there is a bug in the
  meeting commands or that the requester is using customized meeting commands
  that do not obey the protocol.  The errorCode variable is set to PROTOCOL.
```

- reject_meeting *requester_id*

  The reject_meeting command is used to reject a meeting request. *requester_id* is the identification of the requesting agent, i.e., the identification that was returned from the get_meeting command. reject_meeting returns the empty string.

  Examples:

  A long example appears under the accept_meeting command.

  Error messages:

  On error, reject_meeting raises a standard Tcl exception that can be caught with the *catch* command. reject_meeting returns one of the following error messages. The indented text below each error message describes the possible interpretations of the message.

```
wrong number of arguments

  You specified the wrong number of arguments.

agent has NOT been registered

  The agent does not have a controlling server.

unable to send to server

  The requester's machine has crashed or does not exist, the server is not
  running on the requester's machine, or the requester's machine is
  temporarily unreachable due to network failure or overload.  The standard
  response to this error is to retry the command at least once.

server unable to comply (no response)

  The requester's server failed while handling the rejection.

server unable to comply (bad response)
```

```
  The requester's server failed while handling the rejection.

server unable to comply (server error)

  The specified requester does not exist OR the requester's server does not
  accept connections from the agent's current machine, i.e., the agent's
  current machine is NOT in the server's access list.

identification must be ...

  The requester's identification was specified incorrectly

unable to convert name

  The system was unable to determine the network address of the requester's
  machine.
```

- accept_meeting *requester_id requester_loc sockfd*

  The accept_meeting command is used to accept a meeting request. *requester_id* is the identification of the requesting agent, i.e., the identification that was returned from the get_meeting command. *request_loc* is the 3-element list

```
machine_name machine_IP port_number
```

  that identifies the *actual* network location of the requesting agent, i.e. the 3-element list that the get_meeting command put into *locaton_var*. *sockfd* is a socket descriptor to use for the meeting. If *sockfd* is 0 or ANY, accept_meeting will choose an arbitrary TCP/IP port. Otheriwse accept_meeting will use the TCP/IP port that has already been bound to *sockfd*, i.e., *sockfd* must refer to a valid, bound socket. In nearly all instances it is reasonable to let the system choose the TCP/IP port. Thus *sockfd* should almost always be ANY. accept_meeting returns the socket descriptor for the meeting. As before this socket descriptor is an integer greater than or equal to 1 and can be thought of as a meeting identification number. The tcpip_read, tcpip_write and tcpip_close commands are used to exchange information and terminate the meeting.

  Examples:

  The following example shows a client that sends a meeting request to a server. The server either accepts or rejects the meeting (it is not much of a server since it asks a human user whether to accept or reject each meeting rather than making the decision automatically). If the server accepts the meeting, the server and client exchange a pair of strings. The server code is

```
#!/usr/contrib/bin/agent
#
# server.tcl
#
# This is the "server_agent".  It waits for a meeting request from any client
# using the get_meeting command.  Then it asks the user whether to accept
# or reject the meeting.  The client and server exchange two strings if the
# user accepts the meeting.

  # register the agent on the current machine

if {[catch agent_begin]} {
  return -code error "ERROR: unable to register the agent on $agent(actual-server)"
```

```
}

if {[catch {

    # this is the "server_agent"

  agent_force "$agent(local-ip) server_agent"
  agent_name server_agent

    # process meeting requests forever

  while {1} {

      # wait for the next meeting request

    set source_id [get_meeting source_loc -blocking]

     # ask the user to accept or reject the meeting

    puts "$source_id requests access to the server"
    puts "Accept (Y/N)?"
    gets stdin answer

      # on acceptance establish the connection and exchange the status strings
      # on rejection send a rejection notice

    if {($answer == "Y") || ($answer == "y")} {
      set sockfd [accept_meeting $source_id $source_loc ANY]
      set status [tcpip_read $sockfd -blocking]
      puts "The client $source_id says ...\n$status"
      tcpip_write $sockfd "No problem!"
      tcpip_close $sockfd
    } else {
      reject_meeting $source_id
    }
  }

} error_message]} then {

    # make sure that we clean up on error

  agent_end

  return -code error -errorcode $errorCode -errorinfo $errorInfo $error_message

}

  # clean up

agent_end
```

The client code is

```
#!/usr/contrib/bin/agent
```

```
#
# client.tcl
#
# This is the "client_agent".  It uses agent_meet to establish a direct
# connection with the "server_agent" and exchange status information.  It
# assumes that the "server_agent" has the same controlling server.

  # register the agent on the current machine

if {[catch agent_begin]} {
  return -code error "ERROR: unable to register the agent on $agent(actual-server)"
}

if {[catch {

    # meet with the "server_agent"
    # agent_meet will raise an exception if the server rejects the meeting

  set sockfd [agent_meet "$agent(local-ip) server_agent"]

    # exchange the strings

  tcpip_write $sockfd "Thank you for accepting the meeting!"
  set status [tcpip_read $sockfd -blocking]
  tcpip_close $sockfd

    # display the status information

  puts "$agent(local):\nThe server says ...\n$status"

} error_message]} then {

    # make sure that we clean up on error

  agent_end

  return -code error -errorcode $errorCode -errorinfo $errorInfo $error_message

}

  # clean up

agent_end
```

The following output shows two clients attempting to meet with the server. The server rejects the first meeting and accepts the second. The client output on the author's machine is

```
% client.tcl
recipient has refused the request for a meeting
% client.tcl
bald.cs.dartmouth.edu 129.170.192.98 {} 13:
The server says ...
No problem!
%
```

The corresponding server output is

```
% server.tcl
bald.cs.dartmouth.edu 129.170.192.98 {} 12 requests access to the server
Accept (Y/N)?
N
bald.cs.dartmouth.edu 129.170.192.98 {} 13 requests access to the server
Accept (Y/N)?
Y
The client bald.cs.dartmouth.edu 129.170.192.98 {} 13 says ...
Thank you for accepting the meeting!
```

Error messages:

On error, accept_meeting raises a standard Tcl exception that can be caught with the *catch* command. accept_meeting returns one of the following error messages. The indented text below each error message describes the possible interpretations of the message.

```
wrong number of arguments

  You specified the wrong number of arguments.

agent has NOT been registered

  The agent does not have a controlling server.

unable to send to server

  The requester's machine has crashed or does not exist, the server is not
  running on the requester's machine, or the requester's machine is
  temporarily unreachable due to network failure or overload.  The standard
  response to this error is to retry the command at least once.

server unable to comply (bad response)

  The requester's server failed while attempting to complete the meeting.

server unable to comply (no response)

  The requester's server failed while attempting to complete the meeting.

server unable to comply (server error)

  The specified requester does not exist OR the requester's server does not
  accept connections from the agent's current machine, i.e., the agent is NOT
  in the server's access list.

unable to create socket

  The system was unable to open a local socket for the meeting.

unable to bind socket

  The system was unable to associate a port with the local socket.
```

```
unable to listen to socket

  The system was unable to associate a connection queue with the local
  socket.

unable to accept

  The requester is attempting to connect to the local socket.  The system
  was unable to accept the connection.

agent identification must be ...

  The requester's identification was specified incorrectly.

unable to convert name

  The system was unable to determine the network address of the requester's
  machine.
```

- agent_accept *id_var loc_var sockfd* <-blocking | -nonblocking>

  agent_accept combines the get_meeting and accept_meeting commands. It has both a blocking and nonblocking form. The blocking form waits until a meeting request is available, accepts the meeting request, sets *id_var* to the 4-element identification of the requesting agent, sets *loc_var* to the actual network location of the requesting agent, and returns a socket descriptor for the meeting. The system chooses an arbitrary TCP/IP port for the meeting if *sockfd* is 0 or ANY. Otherwise the system uses the TCP/IP port that has already been bound to *sockfd*, i.e., *sockfd* must refer to a valid, bound socket. As before, it is nearly always reasonable to specify "ANY". The nonblocking form returns -1 if a meeting request is not available. Otherwise it proceeds as in the blocking case.

  Examples:

  The following server is the same as the server above except that it *accepts* every meeting.

```
#!/usr/contrib/bin/agent
#
# server.two.tcl
#
# This is the second version of the "server_agent".  It waits for a meeting
# request from any client and then automatically accepts the meeting request
# with the agent_accept command.

  # register the agent on the current machine

if {[catch agent_begin]} {
  return -code error "ERROR: unable to register the agent on $agent(actual-server)"
}

if {[catch {

    # this is the "server_agent"

  agent_force "$agent(local-ip) server_agent"
  agent_name server_agent
```

```
    # process meeting requests forever

  while {1} {

      # wait for the next meeting request

    set sockfd [agent_accept source_id source_loc ANY -blocking]

      # exchange the status strings

    set status [tcpip_read $sockfd -blocking]
    puts "The client $source_id says ...\n$status"
    tcpip_write $sockfd "No problem!"
    tcpip_close $sockfd
  }

} error_message]} then {

    # make sure that we clean up on error

  agent_end

  return -code error -errorcode $errorCode -errorinfo $errorInfo $error_message

}

  # clean up

agent_end
```

On error, agent_accept raises a standard Tcl exception that can be caught with the *catch* command. The possible error messages consist of all the error messages for get_meeting plus all the error messages for accept_meeting.

7.8 Timing

There are four commands that are related to time.

- agent_elapsed

  agent_elapsed returns the number of seconds that have elapsed since the agent started executing on its *current machine*. The number of seconds will always contain a fractional part since it is reported to the resolution of the system clock (or to the number of decimal places specified in the tcl_precision variable if this number of decimal places is smaller).

  Example:

  The author typed agent_elapsed at random intervals in an interactive shell and got the following output.

```
agent-tcl> agent_elapsed
2.930000
agent-tcl> agent_elapsed
6.360000
agent-tcl> agent_elapsed
```

```
63.900000
agent-tcl>
```

Error messages:

On error, agent_elapsed raises a standard Tcl exception that can be caught with the *catch* command. agent_elapsed returns one of the following error messages. The indented text below each error message describes the possible interpretations of the message.

```
wrong number of arguments

  You specified the wrong number of arguments.
```

- agent_sleep *seconds*

  agent_sleep puts the agent to sleep for the specified number of seconds.

  Note:

  Nothing will wake the agent up except for the expiration of the specified number of seconds. If you want to sleep for a certain number of seconds *or* until something arrives from another agent, you should use the agent_select command.

  Example:

  The following command will put the agent to sleep for 10 seconds.

```
agent_sleep 10.0
```

  Error messages:

  On error, agent_sleep raises a standard Tcl exception that can be caught with the *catch* command. agent_sleep returns one of the following error messages. The indented text below each error message describes the possible interpretations of the message.

```
wrong number of arguments

  You specified the wrong number of arguments.

number of seconds must be a real number

  You must specify a number.

number of seconds must be 0 or greater

  You can not go backwards in time.
```

- retry *interval* multiplier *iterations script*

  The retry command attempts to execute the specified *script*. If an error occurs during script execution, the command waits for *interval* seconds and then attempts to execute the script again. If another error occurs, the command increases the sleep time by a factor of *multiplier* and attempts to execute the script once again. This process continues until one of four events occurs.

  1. The script executes successfully. The retry command returns the result of the script.
  2. The script has not executed successfully but the maximum number of *iterations* has been performed. The retry command throws the TCL_ERROR code with the interpreter result set to the return value from the script.

3. The script issues the *break* command at its top level. The retry command throws the TCL_BREAK code with the interpreter result set to the return value from the script.
4. The script issues the *continue* command at its top level. The retry command throws the TCL_CONTINUE code with the interpreter result set to the return value from the script.

Example:

The following Tcl fragment tries to jump to the machine *earhart.cs.dartmouth.edu*. It starts with a sleep time of 1, increases the sleep time by a factor of 2 on each iteration and makes at most 10 attempts.

```
retry 1 2 10 {

  agent_jump earhart.cs.dartmouth.edu

}
```

If the agent_jump succeeds, the retry command returns the result of the agent_jump, i.e., either "JUMPED" or "SAME" depending on whether earhart is a different machine than the agent's current machine. If agent_jump does not succeed within the specified number of iterations, the retry command throws the TCL_ERROR code with the interpreter result set to the error string from the most recent execution of agent_jump.

Error messages:

There are several error messages but there are only two possible errors. Either you specified the wrong number of arguments or you did not specify a positive real number for *interval*, *multiplier* or *iterations*.

- restrict {{wall *seconds*}} *script*

The restrict command is the beginning of a permit system that will allow a machine to restrict an agent's use of resources and that will allow an agent to temporarily restrict *its own* use of resources. The restrict command specifies the maximum number of *seconds* that a *script* can take. If the script finishes before the specified number of seconds has elapsed, the restrict command returns the result of the script. If the script does *not* finish before the specified number of seconds has elapsed, the script is *aborted* and the restrict command throws an exception.

Notes:

1. Timeout information is not captured when a script forks or migrates. Thus the restrict command will not work if the enclosed script jumps or forks. This problem will be fixed soon.
2. The restrict command is currently limited in that the script can only be aborted *between* Tcl commands. Thus the script might actually execute for much longer than the specified number of *seconds* if one of the commands in the script takes a long time to complete. Note that all of the builtin Tcl, Tk and agent commands do *not* have this problem since the command handlers explicitly check for timer expiration when necessary. Consider the following script for example.

   ```
   restrict {{wall 5.0}} {

     agent_receive code string -time 10.0

   }
   ```

   If no message arrives, the restrict command will throw a Tcl exception after 5 seconds even though the agent_receive command specifies a timeout of 10 seconds.
3. The restrict command is *not* a transaction mechanism. The state of the Tcl script is *not* rolled back if the script is aborted. It is currently the programmer's responsibility to handle any necessary cleanup and any inconsistencies that might arise due to an asynchronous abort.

4. You can nest as many restrict commands as desired. If any of the specified timeouts are exceeded, the corresponding restrict command throws a Tcl exception.

Error messages:

On error the restrict command raises a standard Tcl exception that can be caught with the *catch* command. The restrict command returns one of the following error messages. The indented text below each error message describes the possible interpretations of the message.

```
permit violation: wall time exceeded

  The timeout expired before the script finished.  The script was aborted at a
  nondeterministic point.  The errorCode variable is set to "PERMIT WALL".

wrong number of arguments

  You specified the wrong number of arguments.

permit must have the form {name value}

  You specified the timeout incorrectly.

unknown permit type

  The only supported permit type is "wall" which specifies a timeout.

wall time must be 0 or greater

  You can not go backwards in time.
```

7.9 Masks

Masks allow an agent to specify those agents with which it is willing to communicate *at the current time*. A mask is just a list of agent identifications (except that any element of an identification can be replaced with the word "ANY"). Each mask is identified with a unique integer *handle*. An agent can define as many masks as desired. At any given time one of those masks is marked as the current *message* mask; one is marked as the current *event* mask; and one is marked as the current *meeting* mask. The handles of the current message, event and meeting masks are stored in a global array called *mask*.

Element of *mask* array	Contents
message	handle of the current message mask
meeting	handle of the current meeting mask
event	handle of the current event mask

The *mask* array is always available inside an agent. As with the *agent* array, however, if you want to access it from inside a procedure, you will have to create a local reference to the array with either the *global* or *upvar* command. For example, the following procedure prints the handle of the current message mask.

```
proc display args {
  global mask
  puts $mask(message)
}
```

When an agent first starts, the *mask* array contains three default masks. The default masks accept *all* incoming communication. You can change the current meeting, event or message mask simply by setting

the appropriate element of the *mask* array to the handle of the desired mask. Once you have the desired mask in place, you use the agent communication commands as usual. The masks determine which messages, events and meeting requests are reported and which are (temporarily) ignored. Specifically,

1. agent_receive will only return a message if the sender identification matches one of the entries in the current message mask. If the sender identification does *not* match one of the entries, the message is buffered internally until the message mask is changed.
2. agent_getevent will only return an event if the sender identification *and* the event tag matches one of the entries in the current event mask. If the sender identification and event tag do *not* match one of the entries, the event is buffered internally until the event mask is changed.
3. get_meeting will only return a meeting request if the identification of the sender matches one of the entries in the current meeting mask. If the sender identification does *not* match one of the entries, the meeting request is buffered internally until the meeting mask is changed.
4. agent_accept will only accept those meeting requests for which the identification of the sender matches one of the entries in the current meeting mask. If the sender identification does *not* match one of the entries, the meeting request is buffered internally until the meeting mask is changed.
5. agent_meet is *unaffected* by the current meeting mask. This makes sense since agent_meet takes the desired recipient as an argument. It is assumed that you want to meet with this recipienet regardless of the current meeting mask.

The most important thing to note in all of these cases is that an incoming communication is *not* thrown away if it does not match one of the entries in the appropriate mask. Instead it is buffered internally until such a time that it does match one of the entries.

It is critical to note that *masks* are currently *not* transferred when an agent migrates and are not present in a forked child. You must recreate the masks after an agent_fork or agent_submit. This will be fixed soon.

There are nine specific commands for working with masks. All of the commands return the error message

```
wrong number of arguments
```

if you specify the wrong number of arguments;

```
mask handle must be an integer 0 or greater
```

if you specify an invalid handle; and

```
no mask with that handle
```

if the specified handle is not associated with a mask. These error messages are not listed under every command.

- mask new

  This command creates a new mask and returns a unique integer handle. The new mask is initialized so that it accepts *no* incoming communication.

  Example:

  Issuing the command

  ```
  set handle [mask new]
  ```

will set the *handle* variable to the unique integer id of a newly created mask.

- mask delete *handle*

  This command deletes the mask with the given *handle*. You can delete one of the masks specified in the *mask* array but you should replace it immediately since agent_receive, agent_getevent, get_meeting and agent_accept will return the error message

  ```
  no mask in the "mask" array
  ```

  if the required mask is not specified in the *mask* array. The entry in the *mask* array will contain the string "NONE" until the deleted mask is replaced.

  Example:

  Issuing the command

  ```
  mask delete $handle
  ```

  will delete the mask that was created in the previous example. Issuing the command

  ```
  mask delete $mask(message)
  ```

  will delete the current message mask. The contents of the mask array will become

  ```
  mask(message) =   NONE
  mask(meeting) =   4
  mask(event)   =   10
  ```

  You should put a mask handle into *mask(message)* before the next call to agent_receive. Note that it is possible to delete the current message mask without actually referring to the *mask* array (since you refer to a mask with its handle). Be aware of which masks are currently in the *mask* array.

- mask display *handle*

  This command returns the entries of the mask with the specified *handle*.

  Examples:

  If we create a new mask and then immediately display its entries,

  ```
  set handle [mask new]
  set entries [mask display $contents]
  puts $entries
  ```

  we see

  ```
  NONE
  ```

  since a new mask initially accepts *no* incoming communcation. If we display the entries of one of the masks in the *mask* array right after the agent starts,

  ```
  set entries [mask display $mask(message)]
  puts $entries
  ```

  we see

```
ALL
```

since the masks in the *mask* initially accept *all* incoming communication. An example in which the mask accepts only some communication is shown below.

Note:

The *mask display* command is a good debugging tool if you are having a problem with masks.

- mask add *handle* ALL

  This command makes a mask accept *all* incoming communication. The command returns the empty string.

  Example:

  The following commands were typed interactively in the *agent* shell.

```
agent> mask new
3
agent> mask display 3
NONE
agent> mask add 3 ALL
agent> mask display 3
ALL
agent>
```

  Note:

  Any entries that were in the mask are lost. If you want to make a mask accept any incoming communication *without* losing the current mask entries, you should issue the command

```
mask add $handle {ANY ANY ANY ANY}
```

  instead. You can later remove the {ANY ANY ANY ANY} entry with the *mask remove* command.

- mask remove *handle* ALL

  This command removes every entry from the mask. The mask will not accept any incoming communication. The command returns the empty string.

  Example:

  The following commands were typed interactively in the *agent* shell.

```
agent> mask display $mask(message)
ALL
agent> mask remove $mask(message) ALL
agent> mask display $mask(message)
NONE
agent>
```

- mask add *handle* {*id* [-tag *tag*]}

  This command adds an entry to the mask with the specified *handle*. An entry consists of an *id* and an optional *tag*. The *id* specifies the sender identification and can have any of the following forms:

```
machine_name machine_IP agent_name agent_id

machine_name agent_name agent_id

machine_name agent_name

machine_name agent_id
```

In addition the reserved word "ANY" can be used in place of any part of the *id* to indicate that you do not care about that part of the *id*. This will become clearer in the examples. The optional *tag* is used in event masks to indicate that you are only interested in events with a particular tag (you can specify a tag in any mask but it is ignored if the mask is not used as an event mask). If a tag is specified in an event mask entry, an incoming event will match that entry only if it has the exact same tag and the appropriate sender identification.

Examples:

The following is a sequence of examples that give the flavor of mask entries. Here we assume that we are modifying an event mask so that we can use the optional *tag* parameter. All of the examples that do not involve a *tag* apply to any kind of mask.

```
  # get a new mask

set handle [mask new]

  # an entry that will accept any event sent from an agent on machine "bald"

mask add $handle {bald.cs.dartmouth.edu ANY}

  # an entry that will accept any event with the tag "QUERY"

mask add $handle {ANY -tag QUERY}

  # an entry that will accept any event from the "smart_agent" on muir

mask add $handle {muir.cs.dartmouth.edu smart_agent}

  # an entry that will accept an event from the "server_agent" on muir
  # as long as the event has the tag "STOP" (113 is the numeric id of
  # the server_agent).

mask add $handle {muir.cs.dartmouth.edu 129.170.192.42 sever_agent 113 -tag STOP}

  # an entry that will accept any event sent from the agent's root agent

mask add $handle $agent(root)

  # an entry that will accept an event from any "mail_agent"

mask add $handle "ANY mail_agent"

  # now let's look at the mask

puts [mask display $handle]
```

The *puts* command outputs

```
{bald.cs.dartmouth.edu 129.170.192.98 ANY ANY}
{ANY ANY ANY ANY -tag QUERY}
{muir.cs.dartmouth.edu 129.170.192.42 smart_agent ANY}
{muir.cs.dartmouth.edu 129.170.192.42 sever_agent 113 -tag STOP}
{bald.cs.dartmouth.edu 129.170.192.98 {} 9}
{ANY ANY mail_agent ANY}
```

Note how the *mask display* command shows all four elements of the *id* even if you specified fewer than four elements when adding the entry. The missing pieces are filled in with "ANY" (or with an IP address if the machine name was specified but the IP address was not).

Notes:

1. If either the machine name or the IP address is "ANY", *both* of them will be transparently set to "ANY".
2. Consider the following command sequence:

   ```
   agent-tcl> mask display 3
   ALL
   agent-tcl> mask add 3 "bald name_agent"
   agent-tcl> mask display 3
   {bald 129.170.192.98 name_agent ANY}
   agent-tcl>
   ```

   If you add an entry to a mask that is currently set to "ALL", the mask will reset so that it contains *only* the added entry.

Error messages:

The *mask add* command returns a variety of error messages if the new entry is specified incorrectly. These error messages should be self explanatory.

- mask remove *handle* {*id* [-tag *tag*]}

  This command removes an entry from the mask. You specify the entry in the same way as in the *mask add* command. The *mask remove* command removes any mask entries that match the specified entry *exactly*, i.e., only those entries that have "ANY" in the same positions and match exactly in all other postions. Then the command returns the empty string.

  Error messages:

  The *mask remove* command returns a variety of error messages if the entry is specified incorrectly. These error messages should be self explanatory.

- mask_swap <meeting | message | event> *handle*

  This is a convenience procedure for changing the current message, meeting or event mask. In each of the three cases the procedure sets the current mask to the mask with the specified *handle* and returns the handle of the old mask.

  Example:

  The command

  ```
  set old [mask_swap message $handle]
  ```

  is equivalent to

```
set old $mask(message)
set mask(message) $handle
```

Error messages:

mask_swap can return the following error message (in addition to the generic error messages that were listed above).

```
can't read "mask(...)": no such element in array

  The second argument to mask_swap must be "meeting", "message" or "event".
```

- mask_replace <message | meeting | event> *handle*

  This is a convenience procedure for changing the current meeting, message or event mask. In each of the three cases the procedure sets the current mask to the mask with the specified *handle* and *deletes* the old mask.

  Example:

  The command

```
mask_replace message $handle
```

  is equivalent to

```
mask delete $mask(message)
set mask(message) $handle
```

  Error messages:

  mask_replace can return the following error message (in addition to the generic error messages that were listed above).

```
can't read "mask(...)": no such element in array

  The second argument to mask_replace must be "meeting", "message" or "event".
```

7.10 Undocumented commands

Many experimental commands have been added to Agent Tcl in the last few weeks. These commands are stable but it was impossible to document them all before the public release. The undocumented commands are

- tcpip_read *sockfd* to *fd*
- tcpip_write *sockfd* from *fd*
- agent_info -ids *machine* [-time *seconds*]
- agent_info -names *machine* [-time *seconds*]
- agent_info *id* [-time *seconds*]
- agent_select *fd_list* <-nonblocking |-time *seconds* | -blocking>
- agent_disk

- agent_transfer *machine filename* [-time *seconds*]
- crypt *key salt*
- get_remote_file *machine remote_name local_name*
- glue var *name* [*name* ...]
- glue proc *name* [*name* ...]

A brief description of these commands appears in Appendix C. Complete documentation for these commands will be available by December 8. At that time you can obtain an updated copy of this document via our web site

```
http://www.cs.dartmouth.edu/~rgray/transportable.html
```

or via anonymous ftp

```
ftp://bald.cs.dartmouth.edu/pub/agents/doc.1.1.ps.gz
```

In the meantime many of these commands are self-evident. You are also welcome to contact the author.

7.11 Summary

All of the example agents are included in the source distribution. The best way to get a feel for Agent Tcl is to understand the examples and then try the commands in interactive mode except for agent_jump and agent_fork which can not be used in interactive mode). Appendix C summarizes the commands.

8 Agent Tk

The *agent-tk* interpreter is analogous to the *agent* interpreter except that it includes the Tk commands. This allows the programmer to write a GUI that spawns child agents or to write an agent that migrates to a remote machine and displays a GUI to interact with the user of that machine.

8.1 Creating a main window

There are two differences between the *agent-tk* interpreter and the standard *wish* interpreter. First *agent-tk* does not create a main window on startup. Instead a script must explicitly request a main window using the *main create* command. The reason is that an agent might not need to interact with the user on certain machines or at certain times. Thus it requests a main window only when needed. For example, the "Hello, World!" script from [Ous94] should be rewritten as

```
#!/usr/contrib/bin/agent-tk

# hello.two.tk
#
# This is the "Hello, World!" example from [Ous95] except that we use "main
# create" to get a main window.

  # create the main window

main create -name Hello -display :0
```

```
  # make the "Hello, World!" button

button .button -text "Hello, World!" -command exit
pack .button
```

This script creates the window

Clicking on the "Hello, World!" causes the script to terminate.

An agent can use the *main create* command to create a main window on its current machine. Then it can use all of the standard Tk commands to fill in the window and interact with the user. For example here is an agent that jumps to a remote machine and displays the "Hello, World!" window on that machine.

```
#!/usr/contrib/bin/agent-tk
#
# rhello.tk
#
# This agent jumps to a remote machine and displays the "Hello, World!"
# window on that machine.  It gets the name of the remote machine from the
# user.

  # ask the user for a machine

puts "Enter the name of the remote machine:"
gets stdin machine

  # register the agent and jump to the specified machine

agent_begin
agent_jump $machine

  # create the main window

main create -name Hello -display :0

  # make the "Hello, World!" button

button .button -text "Hello, World!" -command {set done 1}
pack .button

  # wait for the user to interact with the "Hello, World!" button

tkwait variable done

  # we're done

agent_end
```

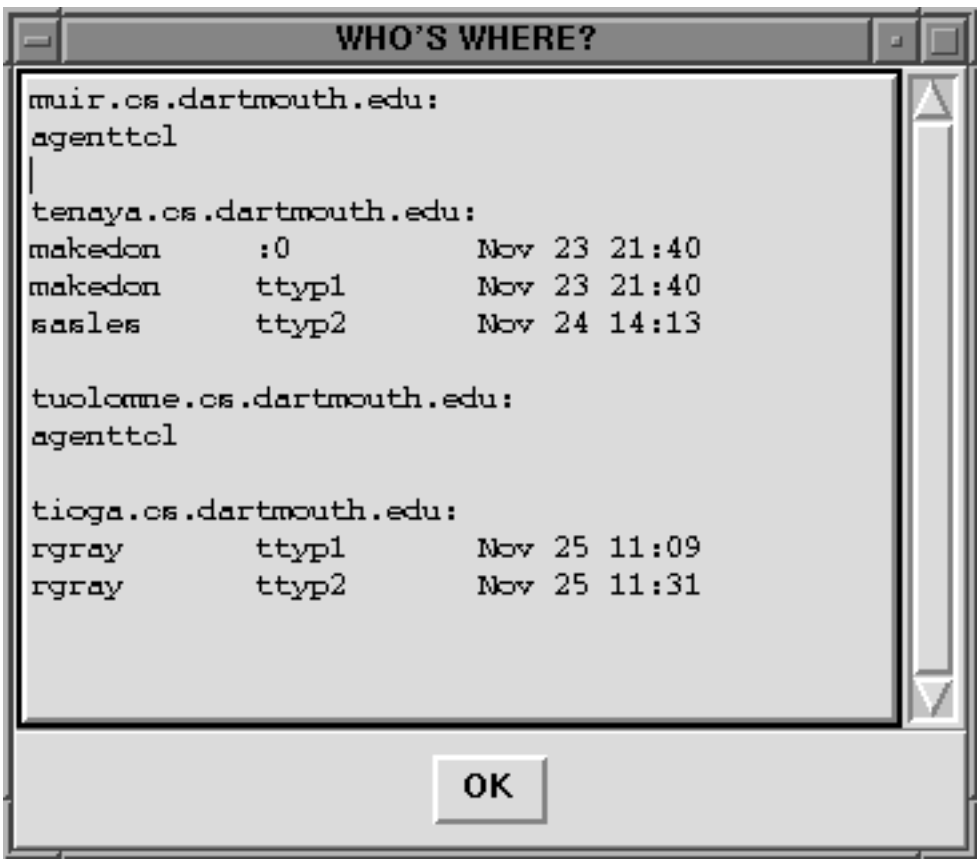
```
exit
```

As a final example, here is a rewrite of the "who" agent that executes the Unix *who* command on each machine and then jumps back to its home machine and displays the user list in a window.

```
#!/usr/contrib/bin/agent-tk
#
# who.tk
#
# This agent jumps from machine to machine and executes the Unix "who" command
# on each machine.  Then it returns to its home machine and displays the user
# list in a window.

  # list of machines -- make sure that you replace these with your machines

set machines "muir.cs.dartmouth.edu \
      tenaya.cs.dartmouth.edu \
      tuolomne.cs.dartmouth.edu \
      tioga.cs.dartmouth.edu"

  # Procedure WHO executes the who command on each machine.

proc who machines {

  global agent

  set list ""

     # jump from machine to machine

  foreach m $machines {

      # if we do not jump successfully append an error message
      # otherwise append the user list

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

  return $list
}

  # register the agent

if {[catch {agent_begin}]} {
  return -code error "ERROR: unable to register on $agent(actual-server)"
}

  # remember the home machine
```

```
set home $agent(local-ip)

  # execute who on each machine

set users [who $machines]

  # return to the home machine

agent_jump $home

  # make the main window and show the results

main create -name "WHO'S WHERE?" -display :0

  # make the two frames

frame .top -relief raised -bd 1
frame .bot -relief raised -bd 1
pack .bot -side bottom -fill both
pack .top -side bottom -fill both -expand 1

  # make a text box that will hold the results

text .text -relief raised -bd 2 -yscrollcommand ".scroll set"
scrollbar .scroll -command ".text yview"
pack .scroll -in .top -side right -fill y
pack .text -in .top -side left -fill both -expand 1

  # make the "DONE" button

button .done -text OK -command "set done 1"
pack .done -in .bot -side left -expand 1 -padx 3m -pady 2m

  # fill in the text area

.text delete 1.0 end
.text insert end $users

  # wait for the user to finish looking at the results

tkwait variable done

  # we're done

agent_end
exit
```

This agent creates the window



Clicking on "OK" terminates the agent.

Options:

The *main create* command takes several optional arguments. Specifically

main create [-display *display*] [-name *name*] [-geometry *geometry*] [-sync]

display is the name of the display on which to create the main window. *name* is the name of the application. This name will be used in the Tk *send* command and will be the initial title for the main window. *gemoetry* is the initial geometry for the main window. The geometry is specified in the normal Tk fashion. The *-sync* option specifies that the agent should use synchronous mode when talking to the X server.

Notes:

1. It is a good idea to always specify *-display :0* when an agent wants to create a main window on its current machine. Otherwise the agent will use the contents of the DISPLAY environment variable. These contents are not necessarily correct after a jump or submit.

2. As with all X applications the agent must have authority to access the X server on the specified machine. *main create* will raise a Tcl exception if the X server is not running or if the X server denies access. Currently access can not be controlled on an agent-by-agent basis but must instead be granted or revoked using the normal X mechanisms such as "xhost". In particular, this means that you must grant screen access to either every incoming agent or no incoming agents, i.e., either allow userid *agent* to access the screen or do not allow userid *agent* to access the screen. Recall that *agent* is the userid that was created specifically for the agent servers.

3. Only the *after*, *fileevent*, *tkwait* and *update* commands are available before *main create* is issued. All other Tk commands will fail with an "undefined command" error. All Tk commands are available as soon as *main create* is issued.

8.2 Destroying a main window

The *main destroy* command destroys the main Tk window and all its children. *main destroy* is simply a synonym for

```
destroy .
```

main destroy takes no arguments.

Note:

An Agent Tk script continues running even after the main window is destroyed. The script must expicitly issue the *exit* command in order to terminate. The reason that the script keeps running is that an agent might want to jump to another machine after cleaning up on the current machine or might want to perform additional processing that does not require user interaction.

8.3 Waiting for the user

The second difference between *agent-tk* and *wish* relates to event loops. *wish* executes the script specified on the command line and then automatically enters an event loop. *agent-tk* does the same thing *except in* child agents that were created using agent_submit and agent_fork *or* agents that have migrated using agent_jump. These agents terminate if they reach the end of the script. Thus the programmer must explicitly enter an event loop using the *tkwait* command. The *who.tk* and *rhello.tk* agents above use the *tkwait* command in order to allow the user to view the results for as long as desired. Without the *tkwait* command these agents would terminate immediately and the user would never see the window.

Note:

You can not migrate to another machine or fork from inside an event handler, i.e., agent_jump and agent_fork will fail when called from inside an event handler. Combined with the required use of the *tkwait* command, this means that most Agent Tk agents will perform the following sequence of steps.

1. Use the *main create* command to create a main window
2. Fill in the window and estalish event handlers
3. Use the *tkwait* command to enter an event loop and wait for the user to finish
4. Use the *main destroy* command to destroy the main window and its children
5. Jump to a new machine and repeat if desired

8.4 Tk handlers for incoming messages, events and meetings

Tk is based around an event-driven style of programming in which the programmer defines *handlers* for *events* such as mouse clicks, key presses, file I/O and so on. Agent Tk allows the programmer to define handlers for incoming messages, events and meeting requests as well as the normal Tk events. The mechanism for defining the handlers is the *mask* command that was discussed above. For example the command

```
mask add $mask(message) "bald.cs.dartmouth.edu smart -handler smartMessage"
```

specifies that procedure *smartMessage* should be called whenever a message arrives from the *smart* agent on bald.cs.dartmouth.edu. Similarly

```
mask add $mask(event) "ANY ANY ANY ANY -tag STOP -handler stopEvent"
```

specifies that procedure stopEvent should be called whenever an event arrives with the tag "STOP". Finally

```
mask add $mask(meeting) "$agent(local-ip) ANY -handler localMeeting"
```

specifies that procedure *localMeeting* should be called whenever an agent on the same machine requests a meeting.

Notes:

1. The masks are manipulated with the *mask* command exactly as before. The difference is that you can specify the *-handler* option when adding an entry to the mask.
2. You can have multiple handlers for the same incoming item simply by adding multiple entries to the mask. The entries might be exactly the same except for the name of the handler or one might be a superset of the other. If multiple entries match an incoming item, the corresponding handlers are called in the order in which the entries were added to the mask. If one of the handlers issues the *break* command at its top level, all subsequent handlers are skipped. This is consitent with standard Tk semantics.
3. The handlers are just normal Tcl procedures with specific arguments. The handler for an incoming message must have the format:

   ```
   proc name {source code string} {

     # arbitrary Tcl code

   }
   ```

   The *source* parameter is set to the 4-element identification of the sending agent. *code* is set to the message code. *string* is set to the message string. Note that the procedure should *not* call agent_receive since the message has been received implicitly. The procedure should simply handle the message as desired.

   The hander for an incoming event must have the format:

   ```
   proc name {source tag string} {

     # arbitrary Tcl code

   }
   ```

   The *source* parameter is set to the 4-element identification of the sending agent. *tag* is set to the event tag. *string* is set to the event string. Note that the procedure should *not* call agent_getevent since the event has been received implicitly. The procedure should simply handle the event as desired.

   The handler for an incoming meeting request must have the format:

   ```
   proc name {source actual status} {

     # arbitrary Tcl code

   }
   ```

   The *source* parameter is set to the 4-element identification of the sending agent. *actual* is set to the 3-element list that specifies the actual location of the requesting agent as well as a TCP/IP port.

status is set to the current status of the meeting. Unless you are writing your own protocol for meeting establishment (which is beyond the scope of this documentation) you should ignore *actual* and treat any *status* other than "REQUEST" as an error. The procedure should not call agent_meet, get_meeting or agent_accept since the meeting request has been received implicitly. Instead the procedure should simply call reject_meeting or accept_meeting as desired.

4. Entries with a handler and entries with no handler can coexist in the same mask. If a handler is specified, the handler is called when the incoming item arrives. If no handler is specified, the incoming item must be explicitly received with the agent_receive, agent_getevent or get_meeting, commands. If an item matches multiple entries in a mask and only some of those entries have a handler, the handlers take precedence, i.e., the handlers are called and the item can *not* be received with agent_receive, agent_getevent or get_meeting.

5. If you replace or add to a mask such that a pending item is now matched by at least one entry, the corresponding handlers are added to the event queue *immediately*. Thus, if you want to associate multiple handlers with the same pending item, you should construct a new mask and then replace the meeting, message or event mask in a single step (using mask_replace or mask_swap or simply by setting the appropriate element of the *mask* array).

6. If an uncaught error occurs inside a handler, Agent Tk calls the "tkerror" procedure. This is normal Tk semantics.

Default Handlers:

In addition to associating a handler with each mask entry, you can associate a handler with the entire mask. This handler is a "default" handler which is called if there are no other matching handlers inside the mask. To specify the default handler, you should use the command

mask handler *handle name*

where *handle* is the mask handle and *name* is the name of the handler. The handler must have one of the three specific formats discussed above. To display the current default handler, use the command

mask handler *handle*

where *handle* is the mask handle. To remove the current default handler, use the command

mask nohandler *handle*

where *handle* is the mask handle.

Example:

The following agent has two event handlers. *stopEvent* is called when the agent receives an event with the tag "STOP". *otherEvent* is called when the agent receives any other event.

```
#!/usr/contrib/bin/agent-tk
#
# handler.tk
#
# This agent illustrates how to associate event handlers with incoming
# messages, events or meeting requests.  Procedure stopEvent is called
# when the agent receives an event with the tag "STOP".  Procedure
# otherEvent is called when the agent receive any other event.

set events 0

  # an event with tag "STOP" has arrived

proc stopEvent {source tag string} {
```

```
  puts "Received STOP so stopping ...."
  exit
}

  # some other event arrived

proc otherEvent {source tag string} {

  global events

  puts "Event $events"
  puts "Source ==> $source"
  puts "Tag    ==> $tag"
  puts "String ==> $string"
  puts ""

  incr events
}

  # register the agent

agent_begin

  # get our symbolic name

catch {agent_force "$agent(local-ip) handler"}
agent_name handler

  # set up the handlers

set m [mask new]
mask handler $m otherEvent
mask add $m "ANY ANY -tag STOP -handler stopEvent"
mask_replace event $m

  # fall off the end into the event loop -- this is NOT a child or migrated
  # agent so it behaves in the normal Tk fashion
\end{vebratim}

The following agents tests the previous agent by sending it two arbitrary
events and then the STOP event.

\begin{verbatim}
#!/usr/contrib/bin/agent-tk
#
# driver.tk
#
# This agent tests handler.tk.

  # register the agent

agent_begin
```

```
  # send some events to handler.tk

agent_event "$agent(local-ip) handler" QUERY "This is a query,"
agent_event "$agent(local-ip) handler" DOCUMENT "This is a document request."
agent_event "$agent(local-ip) handler" STOP "This is a stop."

  # we're done

agent_end
exit
```

Running handler.tk and then driver.tk produced the following output on one of the Dartmouth machines.

```
Event 0
Source ==> tioga.cs.dartmouth.edu 129.170.192.21 {} 33
Tag    ==> QUERY
String ==> This is a query,

Event 1
Source ==> tioga.cs.dartmouth.edu 129.170.192.21 {} 33
Tag    ==> DOCUMENT
String ==> This is a document request.

Received STOP so stopping ....
```

Note that handler.tk does not actually use any Tk commands but does use the event-handling facilities of Tk.

Deficiencies:

There is currently no way to associate a handler with an *established* meeting. This means that there is no way to create a handler that is called whenever a string arrives on a meeting connection.

8.5 Summary

All of the example agents are included in the source distribution. The best way to get a feel Agent Tk is to understand the examples and then try the commands in interactive mode. For example you might establish a handler for messages that are sent from yourself and then send messages to yourself. Appendix C summarizes the commands.

9 Advanced topics

Three advanced topics are not covered in this documentation.

1. Agent Tcl has a C/C++ API that is similar to the Tcl API. This API allows the programmer to access agent facilities from inside C code and to embed agent functionality inside a larger application. The relevant C definitions appear in my_sizes.h, tcl.h, tk.h, tclTcpip.h, agentId.h, tclAgent.h and tkAgent.h. These header files are found in the INCLUDE_INSTALL directory.

2. Existing Tcl extensions can be used with Agent Tcl and Agent Tk since they are fully compatible with Tcl 7.4 and Tk 4.0 respectively. The compilation process is the same as for standard Tcl and Tk except that the extension must be linked with libagent.a, librestrict.a, libtcpip.a and libutility.a in addition

to libtcl.a and libtk.a. These libraries are found in the LIB_INSTALL directory. Make sure that you use the versions of libtcl.a and libtk.a that come with the agent system, make sure that you do the final linking with a C++ compiler, and remember that the internal state of a Tcl extension will not be captured when an agent migrates.

3. The meeting commands – agent_meet, get_meeting, reject_meeting, accept_meeting and agent_accept – are written in Tcl and use the low-level commands agent_req, agent_getreq, tcpip_socket, tcpip_bind, tcpip_listen, tcpip_accept, tcpip_connect and tcpip_getport. These low-level commands can be used directly if desired. Reading through agent.tcl will impart the flavor of these commands. agent.tcl is found in the TCL_LIBRARY directory.

Readers who are interested in these topics are urged to contact the author.

10 Future directions

Agent Tcl is far from complete and is under continuous development. The author works on Agent Tcl full-time and has part-time programming support from several project members. The immediate future of Agent Tcl consists of adding

1. Time outs on the meeting commands
2. Multiple languages and transport mechanisms

   We plan to add an e-mail transport mechanism as well as Scheme and Java interpreters. The two interpreters are potentially long-term projects but are included here since we hope that they can be added quickly.
3. Point-to-point authentication and permits

   Permits specify which actions an agent is allowed to take on its current machine. We hope to find a generic permit mechanism that will work across interpreters.

Three months have been allocated for these items. There will be another alpha release as soon as they have been implemented.

The long-term future of Agent Tcl consists of addressing five open research problems.

1. Debugging

   Agents have proven to be easy to write but difficult to debug if the first attempt contains major bugs. How can the programmer debug an agent quickly and effectively?
2. Privacy

   An agent might contain sensitive information or might reveal sensitive information through its external behavior. How can an agent prevent a malicious third party – i.e., a malicious server – from obtaining this information?
3. Security related to transportability

   An agent can migrate through an arbitrary sequence of machines. This raises numerous security issues that are not addressed in schemes such as Safe-Tcl. Safe-Tcl sends a code fragment from the local machine to a remote machine after which the code fragment travels no farther. An agent, however, can migrate from the first remote machine to a second remote machine. As an example, consider an agent that migrates to machine A and then from machine A to machine B. How can machine B verify the identity of the agent's *original* sender and verify that machine A has not modified the agent in a malicious way?

4. Track moving agents

   This is a lower-level issue than resource discovery. If agent A is communicating with agent B, can A transparently continue communicating even if B migrates to a new machine? In other words the problem is to transparently track a known agent rather than to find a previously unknown agent that can perform the desired task.

5. Network awareness

   How can an agent discover the current state of the network and modify its actions as appropriate?

These five issues make up the programming portion of the author's thesis and will take at least a year. There will be another alpha release as each component is implemented.

References

[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.

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

[Ous94] John K. Ousterhout. *Tcl and the Tk toolkit.* 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.

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

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

[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.

A Changes from previous releases

This section summarizes the changes from previous releases. Note that release 1.1 is the first public release.

A.1 Changes from 0.5 to 1.0

1. An agent identification now consists of four elements – the full Internet name of the machine, the IP address of the machine, the agent's symbolic name and the agent's numeric id. For example,

   ```
   bald.cs.dartmouth.edu  129.170.192.98  ftp_agent  16
   ```

   All commands that require an agent identification will accept either the full 4-element identification or any 2-element shorthand that uniquely identifies the agent. For example

   ```
   bald.cs.dartmouth.edu  ftp_agent
   bald.cs.dartmouth.edu  10
   129.170.192.98         ftp_agent
   129.170.192.98         10
   ```

   In addition it is not necessary to specify the full Internet name. Any name that the local name server maps to the desired machine is acceptable.

2. The *machine* argument to agent_begin is optional. agent_begin will register with the server on the current machine if the *machine* argument is not specified.

3. agent_receive returns the identification of the sender rather than the message code. The message code and string are stored in the two variables whose names are provided as command arguments. In addition agent_receive now has both blocking and nonblocking forms. The desired form is selected with the *-blocking* and *-nonblocking* flags.

4. On success agent_jump returns *SAME* if the destination machine is the same as the agent's current machine and *JUMPED* if the destination machine is a different machine.

5. agent_jump and agent_fork will return an error message if they are called in interactive mode.

6. The agent_name, agent_root, agent_meet, agent_accept, get_meeting, reject_meeting and accept_meeting commands are new.

7. The tcpip_read, tcpip_write and tcpip_close commands are new.

8. The agent_req, agent_getreq, tcpip_socket, tcpip_bind, tcpip_listen, tcpip_accept and tcpip_connect commands are new. These commands are low-level primitives that are used to implement the meeting commands.

A.2 Changes from 1.0 to 1.1

1. Most agent commands now have a timeout parameter *-time*.

2. The *restrict* command is new.

3. The *retry* command is new.

4. The *agent_elapsed* and *agent_sleep* commands are new.

5. The *mask* command is new.

6. The *agent_select* command is new.
7. The *glue* command is new. The *proc* command takes the optional flag *glue*.
8. The *agent_info* command is new.
9. The *agent_event* and *agent_getevent* commands are new.
10. The *agent_disk* and *agent_transfer* commands are new.
11. The *tcpip_read* and *tcpip_write* commands can now read and write entire files as well as Tcl strings.
12. The *tcpip_read* command forces you to explicitly specify *-blocking*, *-nonblocking* or *-time* rather than defaulting to *-blocking*.
13. The *exec* command takes the optional parameter *-noPipeError*.
14. Agents can now use Tk commands.

B Known bugs

There are fours known bugs that will be fixed as soon as possible.

B.1 Missing *masks* and *timeouts*

Masks and timeouts are not transferred when an agent migrates or forks. Thus (1) you must recreate any desired masks after a call to agent_jump or agent_fork *and* (2) you should not call agent_jump or agent_fork from inside the *restrict* command. These problems are not as limiting as they might appear.

B.2 Sticky *event* handlers

You can not fork or jump from inside a Tk event handler. This problem is not as limiting as it might appear.

B.3 Lost *upvar* reference

An *upvar* reference to an *element* of the *env* array disappears when the agent jumps and is not present in a forked child. For example, the procedure

```
proc run_around machines {

  upvar #0 env(DISPLAY) display

  set list ""

  foreach m $machines {
    agent_jump $m
    append list "The current display is $display.\n"
  }
}
```

will fail with the error message

```
can't read "display": no such variable
```

since the *upvar* reference to *env(DISPLAY)* disappears when the agent jumps. The procedure should be rewritten as

```
proc run_around args {

  set list ""

  foreach m $machines {
    agent_jump $m
    upvar #0 env(DISPLAY) display
    append list "The current display is $display.\n"
  }
}
```

In other words, an *upvar* reference to an *element* of the *env* array must be recreated after a call to agent_jump or agent_fork. All other *upvar* references will work as expected and do not need to be recreated.

B.4 *gets*, *puts* and *read*

Internally Agent Tcl uses the SIGIO signal to notify an agent that a message has arrived from another agent. Unfortunately, if the SIGIO occurs in the middle of a blocked *gets*, *puts* or *read* command, the command will fail with the error message

```
Interrupted system call
```

and will set the *errorCode* variable to

```
POSIX EINTR ...
```

A temporary workaround is to reissue the command whenever the *errorCode* variable starts with "POSIX EINTR". For example,

```
  # loop until there is no error

while {[catch {gets $fd} line]} {

    # An error has occured so see if errorCode starts with POSIX EINTR.
    # Loop around and issue the command again if errorCode starts with
    # POSIX EINTR.  Otherwise handle some other error.

  if {![string match "POSIX EINTR*" $errorCode]} {

      # handle other errors here

  }
}

  # now the next line in the file is in the variable "line"
```

This code fragment can be placed inside a procedure so that you do not have to type it over and over.

C Command summaries

This appendix summarizes the agent commands.

C.1 Registration

These commands register an agent and its symbolic name, turn an agent into a root agent and forcibly terminate an agent.

Command	Description
agent_begin [*machine*] [-time *seconds*]	Acquire a controlling server Returns the new *id* of the agent
agent_name *name* [-time *seconds*]	Acquire a symbolic name Returns the new *id* of the agent
agent_root	Turn the agent into a root agent Returns the empty string
agent_force *id* [-time *seconds*]	Forcibly terminate an agent Returns -1 if the agent does not exist Otherwise returns the complete *id* of the terminated agent
agent_end [-time *seconds*]	Tell the controlling server that the agent has finished Returns the empty string

C.2 Migration

These commands migrate an agent to a remote machine and create new agents.

Command	Description
agent_submit *machine* [-procs *name name* ...] [-vars *name name* ...] [-time *seconds*] -script *script*	Create an agent Returns the *id* of the new child agent
agent_fork *machine* [-time *seconds*]	Clone the agent Returns "CHILD" to the new child agent Returns the *id* of the new child agent to the parent
agent_jump *machine* [-time *seconds*]	Migrate the agent Returns "SAME" if *machine* is the same as the current machine Otherwise returns "JUMPED"

C.3 Basic communication

These commands send and receive messages and events.

Command	Description
agent_send *id* [*code*] *string* [-time *seconds*]	Send a message to an agent Returns the empty string
agent_event *id tag string* [-time *seconds*]	Send an event to an agent Returns the empty string
agent_receive *code_var string_var* <-nonblocking | -time *seconds* | -blocking]	Receive a message Returns -1 if the timeout expires before a message arrives Otherwise sets *code_var* and *string_var* to the message code and string and returns the *id* of the sender
agent_getevent *tag_var string_var* <-nonblocking | -time *seconds* | -blocking]	Receive an event Returns -1 if the timeout expires before a message arrives Otherwise sets *tag_var* and *string_var* to the event tag and string and returns the *id* of the sender

C.4 Meetings

These commands set up a meeting between two agents.

Command	Description
agent_meet *id*	Meet with an agent Returns the meeting *sockfd*
get_meeting <-blocking | -nonblocking>	Get a meeting request Returns -1 if nonblocking and no request was available Otherwise returns the *id* of the requester
reject_meeting *requester_id*	Reject a meeting request Returns the empty string
accept_meeting *requester_id*	Accept a meeting request Returns the meeting *sockfd*
agent_accept *id_var* <-blocking | -nonblocking>	Get and accept a meeting request Returns -1 if nonblocking and no request was available Otherwise sets *id_var* to the *id* of the requester and returns the meeting *sockfd*
tcpip_write *sockfd string*	Write a string onto a meeting connection Returns the empty string
tcpip_write *sockfd* from *fd*	Write a file onto a meeting connection Returns the empty string
tcpip_read *sockfd* < -nonblocking | -time *seconds* -blocking >	Read a string from a meeting connection Returns the string
tcpip_read *sockfd* to *fd* < -nonblocking | -time *seconds* -blocking >	Read a file from a meeting connection Returns the empty string
tcpip_close *sockd*	Close a meeting connection Returns the empty string

C.5 Masks

These commands create and modify masks.

Command	Description
mask new	Create a new empty mask Returns the mask *handle*
mask delete *handle*	Delete a mask Returns the empty string
mask add *handle* ALL	Add every possible entry to a mask Returns the empty string
mask add *handle* {*id* [-tag *tag*] [-handler *name*]}	Add a new entry to a mask Returns the empty string
mask remove *handle* ALL	Remove every entry from a mask Returns the empty string
mask remove *handle* {*id* [-tag *tag*] [-handler *name*]}	Remove an entry from the mask Returns the empty string
mask display *handle*	Get the contents of a mask Returns the contents of the mask
mask handler *handle name*	Associate a default handler with the mask Returns the empty string
mask handler *handle*	Get the default handler Returns the empty string if there is no default handler Otherwise returns the name of the handler
mask nohandler *handle*	Remove the default handler Returns the empty string
mask_swap < meeting | message | event > *handle*	Associate a mask with an incoming queue Returns the *handle* of the old mask
mask_replace < meeting | message | event > *handle*	Associate a mask with an incoming queue Deletes the old mask Returns the empty string

C.6 Timing and retries

These commands provide timing facilities and a generic retry mechanism.

Command	Description
agent_sleep *seconds*	Sleep for the specified number of seconds Returns the empty string
agent_elapsed	Get the number of seconds that have elapsed since the agent started executing on the current machine Returns the number of seconds
restrict {{wall *seconds*}} *script*	Restrict the script to the specified number of seconds Returns the empty string
retry *delay multiplier attempts script*	Retry until the *script* succeeds or the maximum number of attempts have been made

C.7 Information

These commands get information about existing agents.

Command	Description
agent_info -ids *machine* [-time *seconds*]	Get the numeric id of every agent registered on *machine* Returns the list of numeric ids
agent_info -names *machine* [-time *seconds*]	Get every symbolic name registered on *machine* Returns the list of symbolic names
agent_info *id* [-time *seconds*]	Get the status of a particular agent Returns -1 if the agent does not exist Otherwise returns the agent status This command is *not* implemented yet.

C.8 Advanced

These commands provide an agent version of the Unix *select* command and save and restore state images to and from disk.

Command	Description
agent_select *fd_list* [-nonblocking | -time *seconds* |-blocking]	Wait for incoming messages, meeting requests and events and for incoming strings on a meeting connection Returns the list of ready descriptors
agent_disk	Save the current state to disk Returns a *filename* to the caller Returns "JUMPED" when the state is restored
agent_transfer *machine filename* [-time *seconds*]	Transfer the saved state to *machine* Returns the empty string

C.9 Miscellaneous

These commands create a main window for a Tk application, perform DES encryption, get a file from a remote machine and mark variables and procedures as *immobile*.

Command	Description
main create [-display *display*] [-geometry *geometry*] [-name *name*] [-sync]	Create a main Tk window Returns the empty string
main destroy	Destroy the main Tk window and its children Returns the empty string
crypt *key salt*	Encrypt a constant string using DES and the given *key* and *salt* Returns the encrypted constant string
get_remote_file *machine* *remote_name local_name*	Retrieve a file from a remote machine Returns the empty string
glue var *name* [*name* ...]	Mark the specified variables as immobile Returns the empty string
glue proc *name* [*name* ...]	Mark the specified procedures as immobile Returns the empty string