ffMAIN: Using Tcl and the TclHttpd Web server to implement a mobile agent infrastructure

Anselm Lingnau
Fachbereich Informatik (ABVS)
Johann Wolfgang Goethe-Universität Frankfurt, Germany
E-Mail: `lingnau@tm.informatik.uni-frankfurt.de`

Abstract

Tcl is a powerful, well-known, freely available scripting language. The TclHttpd web server is an extensible web server written in Tcl. We have used Tcl and TclHttpd to implement *ffMAIN*, an infrastructure for mobile agents. This paper gives a Tcl-related overview of the architecture and implementation of *ffMAIN*, together with some experiences gained during the project.

1 Introduction

Mobile agents—programs that can move about in a computer network according to instructions in their own code—have attracted considerable research interest during the last years, both from the AI and the distributed systems communities. While the former tend to be more interested in agents as entities that can observe the world and reason about those observations, as distributed system researchers our research angle is about mobile agents as a way of structuring distributed applications, and about the kind of system support a mobile agent system would need.

In this paper, we describe some of the experiences we have had implementing an infrastructure for mobile agents based on Tcl [17] and the TclHttpd web server [20]. We give an overview of the infrastructure and its implementation, with a particular focus on its Tcl-related aspects. The paper presupposes a working knowledge of Tcl.

2 The *ffMAIN* Project

2.1 Project Goals and History

The *ffMAIN* (Frankfurt Mobile Agents Infrastructure) project was started in 1995 as part of a larger research effort in mobile communications. Our main goal at the time was to provide a research platform which would allow maximal flexibility both in the type of application and in the way of implementing mobile agents. Thus from the beginning we decided not to prejudice agent implementation, e. g., by constraining users to a single implementation language or communication mechanism. Since *ffMAIN* was intended to be used for research rather than real-world applications, the efficiency of the system was also a secondary concern.

Initially the system was based on an HTTP server written from scratch in the Perl language (the Perl HTTP server classes available today did not exist at the time) [12]. After a while we found that the program had become too complicated to develop further, and that it would be desirable to offer the customary feature set of an HTTP server in addition to the mobile agent functionality. At that point we became aware of the TclHttpd web server developed at—then—Sunlabs by Brent Welch and Stephen Uhler [20], and, since Tcl was already used in a prominent position in the project, we decided to re-implement the agent server functionality in Tcl as an extension to TclHttpd. It was possible to keep the rest of *ffMAIN* with very little modification, and indeed the agents themselves did not need to be changed at all. Work on the Perl server has long since been discontinued, and the implementation based on TclHttpd is now the primary *ffMAIN* platform.

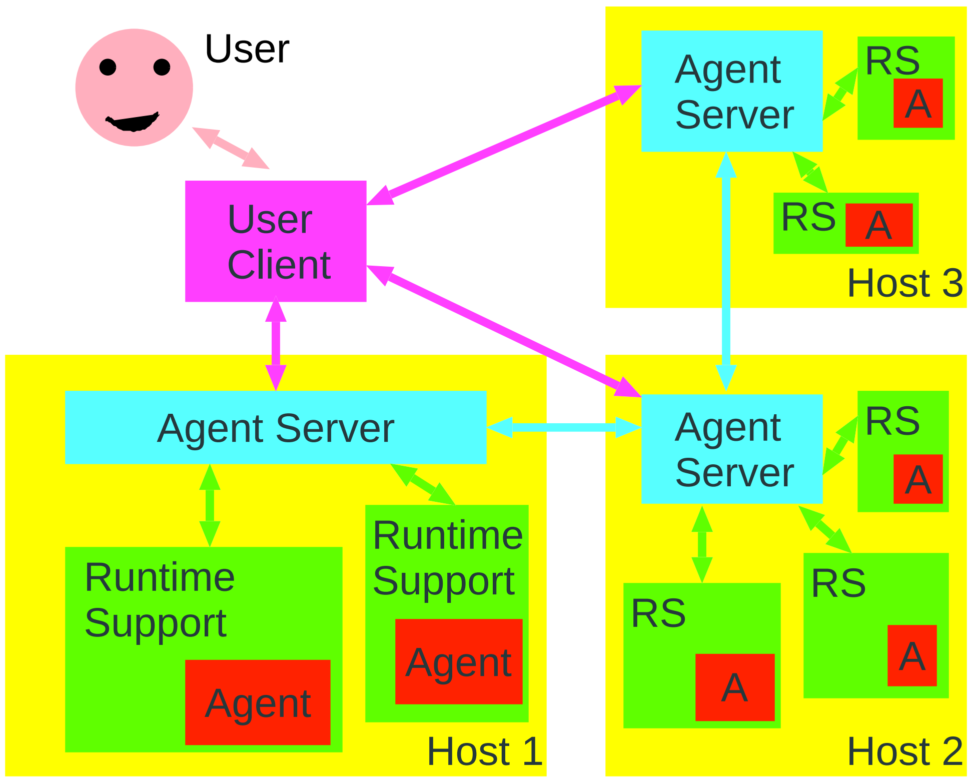

Figure 1: *ffMAIN* architecture model

2.2 Related Work

With mobile agents being an actively-researched topic, many groups have implemented mobile agent infrastructures. A survey carried out in late 1999 [5] lists close to 70 different systems in various stages of completion. It seems that the current language of choice for agent systems and agent implementation is Java, but Tcl-based systems are not entirely unheard of (e. g. Agent Tcl [4], TACOMA [8]). As far as we are aware, our effort is the only one combining a language-independent approach with a standardized protocol such as HTTP. So far attempts to establish systems based on mobile agents in the "real world" have been, on the whole, less than successful; one of the more conspicuous examples is General Magic's Telescript [21], which, among other projects, formed the basis of a PDA-based network service operated jointly with AT&T. This venture has been discontinued. Nevertheless, various companies like IKV++ (Grasshopper [6]), Mitsubishi Electric (Concordia [15]), or ObjectSpace (Voyager [16]) offer agent-based infrastructures as commercial projects. Even the Object Management Group (OMG) has adopted a proposal (MASIF) [14] for integrating mobile agents with CORBA. Even though this idea has not met with universal approval in the community, it is interesting for us to investigate how such an integration could be put into practice using available technology like our system and Pilhofer's *TclMico* Tcl-CORBA interface [19].

2.3 System Architecture

The basis of the *ffMAIN* architecture model for the agent infrastructure is the notion of an *agent server*. This is a program (like a mail server, FTP server, ...) which runs on every computer that will be accessible to mobile agents and is in charge of the agents running on that computer. Its tasks include accepting agents—either from other agent servers or from user clients—, creating the appropriate runtime environments, supervising the agents' execution (while being prepared to answer queries about their status) and terminating them if so directed. The agent server must also organize agent transport to other servers, manage communications among agents as well as among agents and users, and do authentication and access control for all agent operations. Furthermore, in a network of agent servers, every single server may be expected to participate in management operations such as agent location and the gathering of statistics.

We assume that each agent server knows about other agent servers in its “neighbourhood” and makes this information available to agents, who can use it to select a new destination when they decide to leave the current server. Such “neighbours” do not have to be physically (or topologically) close to one another—for example, an agent server specializing in bibliographic databases could tell an agent about other servers offering similar information. No single server (or agent) needs to know the topology of the whole network; if every server knows about its own vicinity, an agent will still be able to traverse an “interesting” subset of all servers by “transitive closure”. As a further refinement, the list of neighbours presented to an agent could be customized according to the origin or purpose of the agent. That way, “firewall” schemes or domain boundaries can be realized. The business of pointing agents to new servers can also be delegated to specialized “routing” agents, which can dispense more specialized or targeted information based on detailed knowledge about their clients' goals and history.

For each agent running on a server, there is a dedicated *runtime environment*. This interfaces between the agent and the server (or the host that the server runs on) by making its resources available to the agent in a controlled way.

Users interact with the agent infrastructure through a *client*. This is a program which will let a user submit an agent for execution, find out about its status, stop or recall it, and perform other operations as necessary. In the simplest case this can be a WWW browser. It is important to note that the client does not need a permanent connection to the rest of the agent infrastructure; it can, for example, reside on a mobile computer or PDA that communicates with the fixed infrastructure via a slow radio link. (We have experimented with submitting agents to *ffMAIN* from a Palm III PDA via an infrared data connection, so this is perfectly feasible.)

An overview of the *ffMAIN* architecture is shown in figure 1. The main design guideline for *ffMAIN* was to allow maximum flexibility concerning agent implementation, their access to the host system, and their communication. We are convinced that a general infrastructure for mobile agents must provide “mechanism, not policy” in order to gain wide acceptance.

2.4 Agents

Before examining the components of the *ffMAIN* infrastructure in more detail, we need to take a look at *ffMAIN* agents. In *ffMAIN*, a mobile agent consists of three components:

Code The program (in a suitable language) that defines the agent's behaviour.

State The agent's internal variables, etc., which enable it to resume its activities after moving to another server.

Attributes Information describing the agent, its origin and owner, its movement history, resource requirements, . . . , for use by the infrastructure. Part of this may be accessible to the agent itself, but the agent code must not be able to modify the attributes.

The difference between code and state may to a certain extent not be clear-cut. For instance, with agents implemented in Tcl, the state is a sequence of Tcl statements, so the state information could theoretically be incorporated in the agent code. However, this is not sufficiently general; for one, the code could be digitally signed for authenticity, and for another, languages like Java cannot be handled in such a straightforward manner.

One long-standing controversy in the mobile agent community is whether a system should cater for *strong* or *weak migration*. With strong migration, the complete dynamic state of an agent is captured, such that, when the agent hits a `go serverB` statement in its code it moves to server B and resumes execution immediately after that statement (which may be part of a deeply-nested procedure call). Weak migration means that, while suitable state is preserved to allow the agent to continue running, on the new server the code is entered at a different point, usually at the beginning. It is the agent programmer's responsibility to ensure that execution continues in the correct fashion for the current state of the agent; in general this can be done, e. g., by putting extra information into a global variable and starting the program with a multi-way branch according to where it left off on the previous server.

It turns out that strong migration, which is of course most convenient for the agent programmer, is feasible but difficult to implement in an agent system; weak migration is a lot easier to do but makes life harder for agent programmers. As an aside, it is possible to change a Tcl interpreter to support strong migration (this has been done, for example, for the Ara project [18]), but our stance is that it is more important to be able to use stock language implementations. The current language environments for *ffMAIN* all support weak migration only; if language implementations become wide-spread that can make the dynamic state of an agent explicit enough to be transportable it will be easy to support them in *ffMAIN*.

3 The Server

3.1 Why HTTP?

Any mobile agent infrastructure requires some protocol for agent mobility and communications. Many agent systems fall back to language-specific mechanisms such as Java RMI [7] or custom, infrastructure-specific protocols, while others propose a more general approach like CORBA [14]. We have selected HTTP, for a number of reasons:

- HTTP is a well-known, well-understood, and widely accepted protocol.
- HTTP contains all the necessary primitives to support agent mobility. For example, the `POST` method can be used to submit an agent to a server for execution, and the `GET` method caters for status requests, etc. Also, the HTTP specification [2, 3] leaves room for custom extensions, should those turn out to be needed.
- Existing WWW browsers like Netscape can be used for most of the infrastructure's user interface. This saves a lot of work and is convenient for users, who do not have to learn how to use yet another tool.
- The World-Wide Web's platform independence makes it easy to support mobile agents in a heterogeneous network. With Web access available even from PDAs or mobile phones, agent-based applications can be used from nearly everywhere.
- We can make use of ongoing research on topics like secure transmission, authentication or electronic commerce. It will be much easier to adopt solutions from the WWW community and to convince users of their merit, than to come up with independent approaches of equal quality.
- WWW-based and agent-based services can be integrated more easily if their technical basis is closely related to begin with. In this way, agent support becomes a "value-added" service that WWW providers can offer in a clean and straightforward way.

3.2 Using the TclHttpd Web Server

The TclHttpd Web Server was begun by Brent Welch and Stephen Uhler when the Tcl project was still a part of Sun Microsystems Labs. It is still being maintained and developed further by Brent Welch at Scriptics. Besides being a full-featured HTTP server, offering the usual functionality such as serving of HTML pages, running CGI scripts and so on, it allows for a number of unique and interesting extension and support facilities.

Rather than base our mobile agent services on a dedicated HTTP-server-like program (which we had tried), a CGI-based system (which would be over-complicated and slow) or a C extension to a popular server like Apache [1], we decided to use TclHttpd to (re-)implement the *ffMAIN* server. Part of the reason for this is that Tcl was already being used to great effect for agent implementation within the *ffMAIN* project, and another important factor was that TclHttpd is a reasonably small and simple program to understand and extend.

Most of today's WWW servers support some kind of "native" extension interface—for example, the Apache server can be extended through loadable modules written in C or a similar language.

The TclHttpd server can be extended through Tcl code, and it offers various methods for making such extensions available for use. In particular, one interesting approach, *application-direct URLs*, is to map URLs in a particular sub-tree directly to Tcl procedure calls and their parameters within the server. For example, a URL of the form

`http://www.mydomain.com/foo/bar?baz=quux`

would be transformed into a call to the Tcl procedure `bar` with parameter `baz` set to `quux`. Our agent extension uses this feature extensively.

Another big advantage of TclHttpd is that it can be examined and modified while it is running. It offers a debugging mode which essentially consists of a Tcl command line prompt, where a developer can enter instructions to look at the values of variables or to change part of the code. It is also easy to re-load the implementation of a module after a bug has been fixed. This makes for very short turnaround time during development and is a major convenience. Much of this functionality is also accessible through a WWW interface (which should be suitably authorized).

Within TclHttpd, our mobile agent extension is implemented as a set of Tcl modules comprising approximately 1500 lines of code. This includes the procedures that accept agents, construct a run-time environment, handle status requests and agent communication and so on, as well as a certain amount of support code which is concerned with reading *ffMAIN* configuration files (which were brought over from the previous implementation) and a Tk-based graphical user interface for the server to aid in debugging agents. The *ffMAIN* extension is distributed as a set of extra Tcl files and a patch designed to be applied to a stock TclHttpd distribution.

3.3 Agent Execution

How does *ffMAIN* run an agent? Users can post agents to the URL `http://www.myserver.com/create`. These agents need to be in a special multipart MIME format with the agent code, state and attributes in separate MIME parts; a program called `asubmit` is provided to construct a suitable multipart content from given agent source code. This transfer format is then parsed by the server, which also examines the agent attributes to determine its resource requirements and the implementation language used. The agent server creates a suitable run-time environment (usually by forking an interpreter for the implementation language and initializing it to run the agent) and launches the agent.

While the agent is executing on an agent server, its status can be accessed via an URL of the form `http://www.myserver.com/visit`. This produces a list of all the agents currently running on the server. During its execution, an agent is assigned a *local ID* by the server, and a "visitor URL" of the form `http://www.myserver.com/visit/a123` returns a more detailed status page for that agent.

3.4 Agent Mobility

Agents can "decide" to move to a different server by executing an appropriate command (which will usually be furnished by the runtime environment). The runtime environment is responsible for regenerating the agent's MIME multipart transfer representation, together with the up-to-date state, which is then sent to the agent server. (In most cases, the agent code can be re-used unchanged, but in Tcl it would be possible for an agent to rewrite itself. For perfect results, it would therefore be necessary to look at the actual running code using Tcl's introspection facilities, and to reconstruct a suitable code file. This would mean that all the agent code would have to be inside Tcl procedures, and a conventional procedure—e. g., `main`—would need to be called as part of the agent launch process. This is currently not implemented, both for reasons of simplicity and also because it would interfere with a digital signature on the code.)

The agent server contacts the destination server at `http://www.yourserver.com/move` and forwards the agent's transfer representation there; the destination server proceeds essentially as described in the previous section to start the agent.

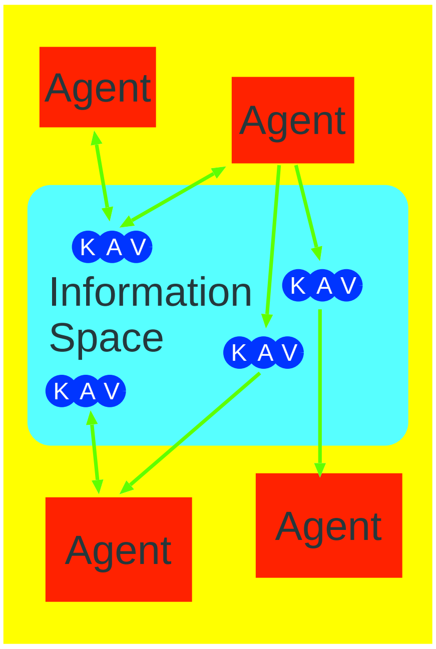

Figure 2: The agent information space

One important topic in connection with agent mobility is how to keep track of an agent's current location. In *ffMAIN*, the server where an agent first enters the infrastructure is declared that agent's *home server*, and its URL is passed on as part of the agent's attributes. Once an agent arrives at a new server and is successfully started, the home server is notified and the agent's visitor URL, including its local ID on the current server, is sent along. An agent's home server implements a "home URL" of the form `http://www.myserver.com/home/a123`, accesses to which are automatically redirected (using appropriate HTTP messages) to the agent's visitor URL on the current server. The local ID in the home URL is the agent's original local ID on the home server, and the infrastructure takes care to re-assign this local ID to the agent whenever it returns to the home server, to avoid confusion. The home URL is also returned to the submitter of the agent when the agent is first entered into the system, so they can use it for finding out about their agents.

3.5 Agent Communication

In *ffMAIN*, agents can communicate by means of an "information space". This is maintained locally by each agent server on behalf of the agents in its charge. The information space contains triples consisting of an item's *key* (or name), an *access control list*, and its *value* (see figure 2). Values are arbitrary chunks of binary information, which are not interpreted by the agent server.

Agents may write items to the information space and read them either destructively or non-destructively; all these operations are atomic and serialized in order to avoid race conditions and inconsistency. For each item, the operations can be enabled for specific agents, groups of agents or all agents according to its access control list. More advanced interaction schemes (such as RPC) can be implemented on top of these primitives, or agents can employ them to negotiate use of different communication methods such as sockets or shared memory (given suitable access privileges to the host's relevant resources). The agent server volunteers data items of general interest such as a list of agents currently running on the server (so agents can get in touch with one another).

The information space decouples agents from one another very thoroughly. In particular, it is not directly possible to damage another agent by writing items into the information space; other agent systems implement agent communication, e. g., through Java message invocations, which is potentially more dangerous.

There are some conventions governing the naming of information space items. For example, names of the form `a123:foo`, where `a123` is a local agent ID, are reserved to the agent with that local ID. There is a simple RPC-like protocol which allows other agents to write items whose names are like `a123:req-a456`. This means that agent 456 wants agent 123 to perform a service on its behalf. By convention, the service result is returned in an item called `a123:repl-a456`, which agent 456 may destructively remove. Information space items whose names don't look like local IDs are free for every agent to use. This makes it possible to implement multicast-like schemes or "blackboard systems" [10].

Agents can find out about changes to information space items by registering interest in particular item names or ranges of item names. They will then be notified when new items appear or existing items are changed. For instance, in the previous example the service-providing agent 123 would register interest in the item name range `a123:req-*`, and the agent server would notify it when new items were introduced in that part of the information space. The agent could then retrieve the request item, examine it and act on it. Service-providing agents usually offer one or more *protocols* for interaction; an agent can advertise the protocols it supports to the agent server, which makes that information available to others via the list of currently-running agents.

Finally, it is possible to access the information space from outside the agent infrastructure, using a WWW browser. To make this more convenient, information space items can be assigned a MIME type, which is then returned to WWW browsers. This means that arbitrary agents can use the information space to communicate with human users, by presenting some information for their perusal or even by putting up forms for them to fill out, whose results are then written back to the information space where the agents can pick them up. There is a special type of agent called an *inbox agent*, whose purpose is to manage incoming agent connection requests on behalf of a user, much like an e-mail client manages incoming e-mail. An inbox agent could be set up to refuse some agents' communication requests outright (say, those that offer pesky marketing research questionnaires), wait for explicit confirmation by the user for others, or pop up browser windows with the appropriate content immediately for the really important stuff [11].

When an agent leaves a particular server, the information space items that it has created are removed automatically. This makes it impossible for agents to be a nuisance to others by leaving huge data items on all accessible servers. It would also be easy to impose "quotas" on the amount of data agents can deposit in the information space, but this has not been implemented yet.

4 Implementing Agents

4.1 Language Independence

One of the major design goals of *ffMAIN* was to allow programmers to write agents in more-or-less arbitrary languages. Until now, most *ffMAIN* agents have been written in Tcl, but we have also developed experimental runtime environments for Java and Perl. The basic requirement that a language must meet so that a *ffMAIN* environment can be written for it is that it must allow communication via the Unix `stdin` and `stdout` predefined files. The server uses the Expect extension to Tcl [9] to handle input/output to the runtime environment.

All the current runtime environments work by creating a new process per agent. This process contains an interpreter (or virtual machine) for the language in question, as well as custom library support to access the agent communication facilities and other *ffMAIN* services. The advantage of this approach is that agents are protected from one another, as well as the system from all agents, by the usual Unix process restrictions—for example, they all execute in separate address spaces and can be controlled using standard process limits. The downside is that this limits the number of agents that a server can run simultaneously to the number of simultaneous processes in the system (usually somewhere in the thousands). Also, depending on the system, processes are a fairly expensive and ponderous resource.

It would be possible to use language-specific runtime environment server processes that can run several agents at once, using a mechanism like threads. Instead of forking a new process for every new agent, the agent server could hand the agent off to the appropriate language-specific server. However, agents would not be as well-protected. In Tcl, the obvious idea would be to run every agent in its own Tcl interpreter within a single Tcl shell. This method could be implemented simply

`agent attrib` a	Query agent attributes
`agent moveto` s	Move to server s
`agent put` n v `...`	Put v into the information space under name n. Additional optional arguments include MIME content type and access control list
`agent get` n	Retrieve item n from the information space
`agent dget` n	Retrieve and remove item n from the information space
`agent subscribe` r [c]	Register interest in information space item range r; use Tcl code c as a call-back if specified. Returns magic cookie i
`agent unsubscribe` i	Undoes `agent subscribe` command identified by magic cookie i
`agent fetch` i	Wait for a notification on `agent subscribe` command identified by magic cookie i
`agent commands` [p]	Return subcommands supported by `agent` (matching glob pattern p)
`agent version`	Retrieve name and version number of the runtime environment
`agent find ...`	Convenience function: Locates an agent matching different criteria in the list of currently-running agents; returns that agent's local ID
`agent request` i v	Convenience function: Writes v to agent i as a request, using the simple RPC protocol

Table 1: Runtime support commands for agents

by exchanging the Tcl runtime environment, with no changes to the rest of the system including the agents themselves. Of course both types of support could be offered at the same time.

Besides actually executing the agent, an agent runtime environment must allow communication between the agent server and the agent. Usually the runtime environment makes available a set of commands or built-in procedures that let the agent send or retrieve items to or from the server's information space, move itself to another server and so on. Most of the agent support commands are handed off to the agent server via the standard I/O pipes, with the runtime environment performing any necessary format translation. For efficiency, the direct connections between the agent server and the runtime modules do not use full-blown HTTP but an abbreviated version that packs most of the headers onto a single line and avoids superfluous verbiage as much as possible. Otherwise it would be very inefficient to pass small amounts of data such as change notifications and simple information space values across the connection. Of course the server also accepts full-blown HTTP for the agent functionality since that is what is used when it is accessed via an HTTP browser; the abbreviated protocol is used exclusively on an agent's I/O pipes.

4.2 Agents in Tcl

Tcl was the first agent implementation language supported by *ffMAIN*, and in many ways it is still the most important environment used in our experiments. Mostly this is due to the malleable nature of the language and its excellent introspection facilities. For example, it is straightforward to examine all the variables being used in an agent and to construct a representation of its state from this information. Also, Tcl agents can be very compact, and this increases the savings obtained through sending agent code rather than raw data across the network. In one of our experiments, a typical Tcl agent was only half the size of a similar agent implemented in Java [13]. This is a particular advantage when agents originate from mobile devices.

At the moment, the Tcl runtime environment supports only weak migration. The reason for this is that the current Tcl implementation makes it difficult to restore the state of a running program if it is in the middle of a nested procedure call—it is reasonably easy to obtain the content of the Tcl execution stack, but there is no straightforward way of using this information to re-initialize a Tcl interpreter. While this is not completely out of the question, support for this would mean patching Tcl [18], and for the time being it seems more important to be able to use the standard Tcl interpreters that are available on many machines. Besides, using a patched Tcl interpreter would make it more difficult to track the ongoing development of Tcl itself.

```
# Time server

# Register ‘clock/1.0’ service with agent server
set myId [agent attrib Local-ID]
agent put server:info-$myId clock/1.0

# Obtain and service requests
set sub [agent subscribe $myId:req-*]
while 1 {
    set item [agent fetch $sub]
    agent dget $item   ;# content of request irrelevant
    set yourId [string range $item [expr [string last - $item] + 1 end]
    agent put $myId:repl-$yourId [clock format [clock seconds]]
}
```

Figure 3: A simple “time server” agent

```
# Time client

# Go to ‘yourserver’, unless already there
if {![info exists arrived] || $arrived != 1} {
    set arrived 1
    agent moveto yourserver
}

# Find an agent that supports the ‘clock/1.0’ service
# and query it
set clockId [agent find -info clock/1.0]
if {$clockId != ""} {
    set date [agent request $clockId {}]
}
```

Figure 4: A “time client” agent

When the agent server executes an agent written in Tcl, it starts a Tcl shell running a program called `launch-tcl`. This program creates a “safe” sub-interpreter, into which it restores the agent state before loading and running the actual agent code. The agent can access the runtime environment through the `agent` command, which offers various subcommands related to agent mobility, communication and general management (see table 1). The `launch-tcl` program is approximately 700 lines long.

Figures 3 and 4 show two example agents, a server for a simple protocol and a matching client. The server agent in figure 3 would be started on an agent server as a “stationary” agent; it registers with the agent server and advertises that it supports a protocol called `clock/1.0`. (The client will use the `agent find` convenience function to locate a server for this protocol; see below.) Then it goes into an endless loop waiting for requests according to the simple protocol outlined in section 3.5. This version of the server waits synchronously for requests using `agent fetch`; it would be equally possible to put most of the loop body into a callback function and register this via a command like `agent subscribe $myId:req-* timeCallback`. This would allow the agent to do something sensible while no requests are pending. The `agent fetch` call returns the name of the new item. The server agent then parses to obtain the local ID of its client, which it needs to return the reply according to the protocol.

The client agent in figure 4 tries to find a `clock/1.0` server on the agent server `yourserver`. We assume that the agent is started on another agent server with empty state (the default). This means that the global variable `arrived` is undefined, so we set `arrived` to 1 (for future reference) and try to move to `yourserver` using an `agent move` command. Once there, `arrived` will exist in the agent's state, and the first block of code will be skipped. The agent proceeds to locate the time server agent in `yourserver`'s directory of running agents, and if this succeeds it makes its request for the current date and time, using the `agent request` convenience function.

From the Tcl point of view, an interesting issue is implementing asynchronous notifications for the `agent subscribe` callback interface. Usually, with event-based Tcl applications such as those utilizing the Tk toolkit, some setup is performed in Tcl code before the program enters an event loop to begin processing events arising from the GUI, files, sockets and so on. In the case of mobile agents, we need to execute the agent code while still processing (mainly) file events from the connection to the agent server "in the background". This should be transparent to the agent programmer, so an approach that would require an agent to be sprinkled with calls to the Tcl `update` procedure would be inconvenient and error-prone. Instead, a small extension is installed which uses a `Tcl_CreateTrace` procedure to process the event loop every so often in parallel to the actual agent execution. This results in a bit of overhead; there is a tradeoff between the timeliness of notifications and the slow-down of the agent in general which can be controlled by deciding how often the event loop should be checked—every single Tcl command, every 10 or 100 commands and so on.

4.3 Security

An important issue with mobile agents is how to prevent untrusted code—the agents—from causing damage to a host system (e. g., by formatting the hard disk) or obtaining information that they are not entitled to look at (such as the password file). In *ffMAIN*, it is the responsibility of the runtime environments to prevent such antics.

The Tcl runtime environment executes agent code in a safe Tcl interpreter, from which all "dangerous" commands have been removed. Therefore it is impossible for a standard mobile agent to open files or socket connections, among other things. However, there can be exemptions for trusted agents, which may be given access to various system resources according to their tasks. (Trust in an agent can be established on the basis of its origin—a locally-developed agent may be more trustworthy than an agent coming from `evil.guy.com`—, among other criteria.) The preferred way is to install stationary agents with suitable privileges which can then service requests from untrusted mobile agents. These requests can be vetted appropriately, and logging or quotas can be imposed as necessary.

Extra privileges can be established on the basis of an agent's *type* and *context*. These are part of an agent's set of attributes and give a rough classification of the sort of thing an agent is supposed to do. For example, an agent of type `web-searcher` could be expected to want to access local WWW resources. On the part of the agent infrastructure, this could be catered for by means of a command like `FetchLocalURL`, which would be installed in the safe interpreter for the agent's use and allow it to retrieve resources identified by local URLs. This restriction would be enforced by the command's implementation; therefore it is not necessary to let the agent access arbitrary files. In the same fashion, an agent that needs to talk to a local NNTP server could be handed a pre-opened I/O channel to that program in order to not have to be allowed to open arbitrary sockets. Of course there could also be a set of command-based abstractions such as `EnterGroup`, `GetArticle`, …, so the agent would not even need access to the NNTP socket connection.

Prior to starting any agent, the `launch-tcl` program reads a local *context file* which is executed in the trusted interpreter. Usually context files use the `interp alias` Tcl command to add commands such as `FetchLocalURL` to the safe interpreter for the use of the agent (with an implementation running in the trusted interpreter). This approach allows very fine-grained security policies.

4.4 Debugging

Debugging a program that moves from host to host can be a bit of a chore. While the level of help that *ffMAIN* has to offer for this cannot compare with modern development environments, there are still a few things that can be done to make the task easier.

One reasonably typical failure mode is for an agent to crash on some remote machine due to a runtime error. Unfortunately Tcl, due to its interpreted nature, is somewhat more prone to runtime syntax errors than other languages. What the infrastructure does in this case is to capture any error messages arising from the crash and returning them to the agent's home server together with notification of the agent's untimely demise. This makes it possible for the programmer to look at what went wrong and (hopefully) diagnose the problem.

Sometimes it is helpful to look at the communication between two agents in order to identify errors. The agent server can trace all the traffic between itself and various runtime environment modules and display the messages exchanged in order together with time stamps. Agents can also emit diagnostic messages that the server will log.

Finally, the agent server's user interface contains a facility to display, inspect and change the contents of the information space. As mentioned earlier, the information space can also be accessed from the outside using a standard HTTP browser, which makes it possible to inspect its contents even without specialized tools.

5 Lessons Learned, and Future Ideas

Generally, our experience with the TclHttpd-based agent infrastructure have been very favourable. The server is adaptable enough to accommodate the agent extensions with reasonably little extra code—as mentioned above, the extensions run to approximately 1500 lines of Tcl, whereas the much less featureful Perl-based implementation took more than twice that—, and the agent code can be added in a very modular fashion without impinging the rest of the server in any way. Another invaluable plus is the ability to load updated versions of the agent support code into TclHttpd without the need to restart the program, which makes for very quick development. The only disadvantage that is readily apparent is that, at the time of implementing the bulk of the infrastructure, the server wasn't documented very well from the point of view of an extension implementer (other than by studying the source code). In this respect the situation has improved since.

As far as writing agents is concerned, Tcl is still our language of choice (in spite of the new Java environment). The runtime environment for Tcl is fairly mature, and the Safe-Tcl security mechanism has proved both conceptually simple and easily adaptable to a wide range of security policies.

Other than the extensions already alluded to—such as the introduction of thread-based multi-agent processes, which should become much easier with the new thread-safe Tcl releases—, some of the things we are going to look at in the near future are authentication and encryption using SSL (we have implemented prototypical SSL support in TclHttpd but are looking forward to the "official" version), and experiments using CORBA (via Frank Pilhofer's TclMico extension [19]). Another challenging issue is resource management, both at the Unix level (CPU time, memory consumption) and the Tcl level. It would be interesting to see which sort of resource control could be implemented for Tcl programs, such as quotas on the number of commands executed in an interpreter or the amount of data stored in variables. This would be especially worthwhile in the context of a runtime environment executing lots of agents in individual threads, since the Unix mechanisms are too coarse-grained to cover this case.

References

[1] The Apache project. See http://www.apache.org/httpd.html.

[2] T. Berners-Lee, R. Fielding, and H. Frystyk. Hypertext Transfer Protocol—HTTP/1.0. RFC 1945, Network Working Group, May 1996.

[3] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol—HTTP/1.1. RFC 2616, Network Working Group, June 1999.

[4] Robert Gray. Agent Tcl: A flexible and secure mobile agent system. In *Proc. 4th Annual Tcl/Tk Workshop*, pages 9–23, July 1996. See http://www.cs.dartmouth.edu/~agent/papers/tcl96.ps.Z.

[5] Fritz Hohl. The mobile agent list. See http://mole.informatik.uni-stuttgart.de/mal/mal.html.

[6] IKV++. Grasshopper: The agent platform. See http://www.ikv.de/products/grasshopper/.

[7] Java remote method invocation (RMI). See http://java.sun.com/products/jdk/rmi/.

[8] Dag Johansen, Robert van Renesse, and Fred B. Schneider. An introduction to the TACOMA distributed system. Computer Science Technical Report 95-23, Universitet Tromsø, Institute of Mathematical and Physical Sciences, Department of Computer Science, University of Tromsø, N-9037 Tromsø, Norway, June 1995.

[9] Don Libes. *Exploring Expect: A Tcl-based Toolkit for Automating Interactive Programs*. O'Reilly & Associates, December 1994.

[10] Anselm Lingnau and Oswald Drobnik. Making mobile agents communicate: A flexible approach. In *The First Annual Conference on Emerging Technologies and Applications in Communications (etaCOM'96)*, pages 180–183. IEEE, May 1996.

[11] Anselm Lingnau and Oswald Drobnik. Agent-user communications: Requests, results, interaction. In *Mobile Agents: Second International Workshop, MA '98*, number 1477 in Lecture Notes in Computer Science, pages 209–221. Springer, September 1998.

[12] Anselm Lingnau, Oswald Drobnik, and Peter Dömel. An HTTP-based infrastructure for mobile agents. In *Fourth International World Wide Web Conference Proceedings*, number 1 in World Wide Web Journal, pages 461–471, Sebastopol, CA, December 1995. W3C, O'Reilly and Associates.

[13] Andreas Möbs. Entwurf und Implementierung einer Laufzeitumgebung für mobile Agenten in Java. Master's thesis, Johann Wolfgang Goethe-Universität Frankfurt am Main, August 1998.

[14] Dejan Milojicic, Markus Breugst, Ingo Busse, et al. MASIF – the OMG mobile agent system interoperability facility. In *Mobile Agents: Second International Workshop, MA'98*, number 1477 in Lecture Notes in Computer Science, pages 50–67. Springer, September 1998.

[15] Mitsubishi Electric. Introducing concordia. See http://www.meitca.com/HSL/Projects/Concordia/.

[16] ObjectSpace, Inc. Voyager. See http://www.objectspace.com/products/vgrORBpro.htm.

[17] John K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, Reading, MA, 1994.

[18] Holger Peine. Ara—agents for remote action. In William R. Cockayne and Michael Zyda, editors, *Mobile Agents: Explanations and Examples*. Manning/Prentice Hall, 1997.

[19] Frank Pilhofer. TclMico: A Tcl interface to CORBA. See http://www.informatik.uni-frankfurt.de/~fp/Tcl/tclmico/.

[20] Brent Welch and Stephen Uhler. Web enabling applications. In *Proc. 5th Annual Tcl/Tk Workshop*. USENIX, July 1997.

[21] James E. White. Telescript technology: The foundation for the electronic marketplace. General Magic White Paper GM-M-TSWP1-1293-V1, General Magic, Inc., 2465 Latham Street, Mountain View, CA 94040, 1994.