Transportable Agents

Keith D. Kotay and David Kotz

Department of Computer Science
Dartmouth College
Hanover, NH 03755-3510
kotay@cs.dartmouth.edu
dfk@cs.dartmouth.edu

November 10, 1994

Abstract

As network information resources grow in size, it is often most efficient to process queries and updates at the site where the data is located. This processing can be accomplished by using a traditional client-server network interface, which constrains the client to the set of queries supported by the server, or requires the server to send all data to the client for processing. The former is inflexible; the latter is inefficient. *Transportable agents*, which support the movement of the client computation to the location of the remote resource, have the potential to be more flexible and more efficient. Transportable agents are capable of suspending their execution, transporting themselves to another host on a network, and resuming execution from the point at which they were suspended. Transportable agents consume fewer network resources and can support systems that do not have permanent network connections, such as mobile computers and personal digital assistants. We describe a prototype transportable-agent implementation that facilitates research in this area. Agents are written in a script language that supports agent relocation, and the language is processed at each host by an agent interpreter. Electronic mail and UNIX remote shell (rsh) are the two current transport mechanisms and we plan to explore others. We present a technical-report searching agent as a demonstration of the capabilities of our prototype implementation.

1 Introduction

As network information resources continue to grow in size, query-processing efficiency becomes more of a concern. If information agents are going to process remote information resources, they have two choices: remain local to the agent's home machine and process the data remotely via a traditional client-server network interface, or travel to the remote site where the data is located and process the data there. The traditional client-server network interface constrains the client to the set of queries provided by the server, or requires the server to send all data to the client. The former is inflexible; the latter is inefficient. Permitting agents to travel to remote sites to process information resources, on the other hand, is faster and consumes fewer network resources when the interaction between a relocatable agent and the information resource is local rather than over a network. Furthermore, the ability to travel allows agents to support systems that do not have permanent network connections, such as mobile computers and personal digital

assistants. It is for these reasons that *transportable agents*, which have the ability to travel throughout a network to remote sites, will play an important role in the area of information processing and retrieval.

To research the issues related to agent travel, we have created a prototype implementation that supports transportable agents. Transportable agents are capable of suspending their execution, transporting themselves to another host on a network, and resuming execution from the point at which they were suspended. Agents are coded in an agent scripting language, which is interpreted by the current host and which may request the relocation of the agent to another host. We use electronic mail and UNIX remote shell (rsh) as transport mechanisms: agents transport both their script and a representation of their current execution state to another host.

In the next section we provide some background; in Section 3 we discuss the prototype implementation. In Section 4 we describe several applications to demonstration of the capabilities of our transportable agents. Finally, we discuss our conclusions and future work in sections 5 and 6.

2 Background

Although little has been published on transportable agents, much work has been done concerning the general concept of remote computation. Remote Procedure Call (RPC) [BN84] was an early form of remote client-server processing. Although this mechanism did not allow for the remote processing of arbitrary client code, it did allow for specific services to be performed remotely. Later work extended RPC to allow for more general forms of remote processing.

Falcone [Fal87] discusses a heterogeneous distributed system environment in which a programming language is used to provide the remote service interface as an alternative to RPC calls. In this system clients and servers communicate by programming each other by sending scripts written in the Network Command Language (NCL) created for this system. NCL is based on the LISP programming language and a NCL server essentially executes what resembles a LISP read-eval-print loop to process client programs.

Stamos and Gifford [SG90] introduces the concept of Remote Evaluation (REV) in which servers are viewed as programmable processors, increasing their flexibility and decreasing the amount of communication necessary to accomplish a given task. REV uses stubs in much the same way as RPC, although the messages sent from machine to machine may contain source code to be executed remotely. In the prototype implementation, the remote programs are a compressed form of list structure executed by a dialect of LISP. Remote procedures must be self-contained in that the only external references allowed are to procedures explicitly supported at the remote processing site.

Stoyenko [Sto94] describes SUPRA-RPC, an extension to RPC which allows subprograms to be passed as RPC parameters. The SUPRA-RPC subprograms are allowed to access variables and procedures external to the passed subprogram. These external references are implemented by out-of-scope access stubs and callback-handling stubs. The SUPRA-RPC run-time support system manages a run-time symbol table to keep track of referencing environments and out-of-scope objects. The SUPRA-RPC system has been implemented on Sun UNIX workstations and supports the C, C++, and CommonLisp languages.

The Telescript™ technology introduced by General Magic, Inc. in 1994 was the first commercial implementation of transportable agents. [Whi94] describes the advantages of transportable agents, outlines their implementation, and presents a number of potential applications for transportable agents. A key issue discussed is that of *safety:* how to provide for the execution of transportable agents without compromising the security of the host on which the agent is executing and the security of the network on which agents travel. Safety precautions implemented by Telescript™ include *interpretation*, *credentials*, and *permits*. *Interpretation* means that agents are processed by an interpreter that allows run-time checking of agent actions. *Credentials* allow agents and their owners to be identified, reducing the likelihood of anonymous

virus-like agents. *Permits* define restrictions on an agents capabilities such as lifetime, maximum size, and incurred cost. For example, an agent may be created with a 1-hour lifetime, a 100k-byte maximum size, and a $10.00 spending limit.

3 Implementation

The objective of our implementation was to create a prototype environment in which transportable agent issues can be explored, with minimal development time. The implementation environment currently consists of numerous UNIX workstations at Dartmouth College and at Duke University. The UNIX workstations include DEC Alpha, DEC Mips, Silicon Graphics Indy, Sun 3, and Sun 4 architectures.

An overview of the prototype implementation is in Figure 1. An *agent script* (written by a programmer) is submitted to an *agent generator* program, which produces an interpretable script. An *interpretable script* contains the text of the agent script plus information, such as variable values, that represents the current execution state of the script. The interpretable script is then submitted to an *agent interpreter*, which processes the statements of the script. If a **moveto** statement is encountered, the agent interpreter packages the agent for transport. This produces another interpretable script with updated state information. This interpretable script is then sent via the transport mechanism to another host for execution. The process repeats itself until the end of the script is reached. Normally, an agent will transport itself back to its original host so it can output any information it has collected.

3.1 Script Language

We considered several existing script languages as candidates for this implementation. However, because their interpreters and compilers are not generally designed to support the suspension of program execution and the extraction of execution state information, a large amount of work would be necessary to adapt them to support transportable agents. Therefore, in the interest of reducing development time, we decided to design a specific language for this prototype. Ultimately, we plan to adopt or design a more appropriate transportable-agent language.

The script language was designed to be parsed easily, so it includes many syntactic restrictions (for example, whitespace between all tokens). It supports many features of the AWK [AKW88] language, such as dynamic types, associative arrays, and regular expressions.

Most of the operators are identical to the operators of AWK. We added a '&' operator for string concatenation, an array-size operator '| array |', and an array-append operator '$=' that appends a new element to the end of an array. For example, if | A | is 3, the statement A $= X is equivalent to the statement A[4] = X. This feature, coupled with the '**foreach** variable **in** array' statement that supports looping over all indices of an array, allows arrays to be used somewhat like lists.

The language also supports traditional imperative language features such as a conditional **if-else** statement and a **while** looping statement. We added an **output** statement for writing data to a file, a **system** statement for executing external programs, a **moveto** statement for transporting the agent to another host, and a **wait** statement for delaying program execution for a specified time interval.

Figure 2 shows a sample agent script that travels to several machines to run "who", and Figure 3 shows the output of an execution of the script. This example is trivial but serves to demonstrate many features of the agent script language and also demonstrates the ability of agents to transport themselves. Note the use of variables with a leading underscore such as '_OriginalHostName' and '_CurrentHostName'. These are system variables, which the agent can examine but not change. Also note the check performed after the **moveto** statement to verify that the value of the '_CurrentHostName' variable is equal to the destination host of the **moveto** statement. It is possible for the **moveto** command to fail due to network problems such

as a host being down. Instead of exiting, the interpreter will continue to execute the script, allowing detection of the failure and error recovery actions to be taken.

3.2 Script Interpreter

We used an interpreter to process agent scripts, a choice that improved safety and heterogeneity. Safety was improved because the interpreter could check every agent action at run time. If an agent consisted of some form of executable code, run-time checking would be minimal or nonexistent. An interpreter also allowed the same script to be processed on many different machine architectures, thereby increasing heterogeneity. Again, if an agent consisted of some form of executable code, it would only be able to run on a single architecture.

We implemented the script interpreter in AWK because of its support for rapid prototyping and text processing. As a result, the scripting language inherited many of the features of the AWK language, such as associative arrays. We used GNU AWK to process the interpreter code.

One safety issue that our prototype partially addressed was access to external programs. To limit access to external programs, the interpreter had a list of external programs that agents were permitted to invoke via the **system** statement. When the **system** statement was used in a script, the interpreter verified that the requested program was in the list of allowable external programs. If the external program was not allowed an error resulted, terminating the agent. Agents could examine the system variable '_SystemCommand', which was an array containing the list of valid external programs. As shown in the example script in Figure 2, it is good practice for the agent to query the '_SystemCommand' array to verify that the desired external program is indeed available on the current host.

3.3 Transport Mechanisms

We considered three possible of transport mechanisms for our transportable agents: electronic mail, UNIX remote shell (rsh), and TCP/IP. We have implemented working versions of both the electronic mail and rsh transport mechanisms. Electronic mail is well suited to networks with a large number of hosts, such as the Internet, and for networks that are not configured to process all their mail on a single mail-server workstation. rsh is best suited to smaller networks because it requires each agent-processing host to be listed in the .rhosts file. TCP/IP is the most general transport mechanism; it is usable on any network which runs the TCP/IP protocol.

3.3.1 Electronic Mail

Figure 4 outlines the operation of the electronic mail transport mechanism. After an interpretable script was created in response to a **moveto** statement, it was mailed to the agent processor at the destination host. Upon receipt of the mail text, the `.forward` file on the destination host vectored it to a mail processing shell script. This script established the proper environment for processing the incoming agent (creating a unique working directory for the agent and its files), and then invoked a mail-processing AWK program. This AWK program removed the mail header, verified that it was an agent script, and wrote the interpretable script to a file. This file was then used as input to the script interpreter.

Electronic mail has several advantages as a transport medium. It is a well-established service that is present on almost all networks. It also is a reliable service that has various internal mechanisms (such as queuing and retries) to cope with delivery problems, such as the destination machine being temporarily unavailable. Furthermore, electronic mail, at least in the UNIX environment, has the flexibility to process transportable agents. One limitation of electronic mail is its use of a text-only format, which makes it inconvenient to transmit binary data. This is not an issue in the current implementation, but could be in the

future. Another limitation of electronic mail is its relatively poor, unpredictable performance; delivery time can vary greatly. Also, detection of an electronic mail transport failure in response to a **moveto** command is unlikely because most electronic mail errors, such as addressing errors, are not detected by the mail program at the time when the mail is sent. Therefore, the agent on the current host will terminate assuming that the transport was successful, resulting in a lost agent. A possible solution to this problem would be to have the agent on the current host wait for an acknowledgement that the transport was successful, at which point it could correctly terminate.

3.3.2 UNIX Remote Shell

Electronic mail is inadequate as a transport mechanism on networks that have been configured to have a single workstation as a common mail-server for the network. In this case, all electronic mail is processed on the common mail-server even though the mail was sent to a specific machine on the network. This prevents agents from actually executing on the various machines in the network. The rsh transport mechanism, as outlined in Figure 5, overcomes this problem. An interpretable script, created in response to a **moveto** statement, was used as standard input to a transport shell script which was invoked by the rsh command. The transport shell script created the working directory for the agent and its files, output its standard input to the interpretable agent file, and then invoked the agent interpreter as a background process.

Using UNIX remote shell as a transport mechanism requires that the .rhosts file in the agent account have entries for each host that is allowed to send transportable agents. This is necessary to circumvent the password requirement when using rsh. Each entry consists of the host name followed by the agent account name. This guarantees that only the agent account has access to the host without using a password. A disadvantage of rsh is that it does not possess the reliability features of electronic mail. However, unlike electronic mail, rsh failure is immediately detectable by the agent itself if, after a **moveto** command, the value of the _CurrentHostName variable is compared to the host name given as the **moveto** argument. If they do not match, then the move was unsuccessful and the agent can retry the move or attempt to transport itself to a different host.

3.3.3 TCP/IP

Another alternative to electronic mail is TCP/IP. TCP/IP has many advantages over electronic mail including greater performance and the ability to transfer binary data easily. One drawback of TCP/IP is the loss of reliability features such as queuing and retries. Another is that TCP/IP is not as widely available as electronic mail. The ability to transfer binary data also is an advantage of TCP/IP over the UNIX remote shell transport mechanism. TCP/IP would also be faster than rsh, but would require a new authentication protocol. We plan to explore TCP/IP and other transport mechanisms in the future. We do note, however, that the agent does not have to know (or care) how it is transported, as long as it arrives. Indeed, different transport mechanisms may be used for each **moveto** executed by an agent.

4 Results

We wrote several agent programs to demonstrate the capabilities of our prototype implementation. Figure 6 shows the script for a full-text technical-report searching agent which used the electronic mail transport mechanism. This agent visited various hosts and invoked the 'trsearch' external program, which performed a keyword search of technical reports available at that host. The output of the 'trsearch' program is a list of World Wide Web Universal Resource Locators (URLs) of technical reports that contained the search keyword. This list was stored in an array for later processing. After all hosts were

visited, the agent returned to its home location and prepared an HTML-format document that contained links to the matching technical reports. This HTML document could then be viewed within Mosaic, which allowed the user to retrieve and display the desired technical reports. Figure 7 shows the output of the technical-report searching program when displayed by Mosaic.

The technical-report searching agent took approximately seven minutes to search both the Dartmouth College and Duke University computer-science technical reports. Most of this time was spent decompressing and searching the technical reports. Our implementation used a simple 'grep' search on the Postscript file itself. Although this was not an ideal search method, it was adequate for the purpose of testing transportable-agent behavior. In any case, improvements in the search method would occur within the 'trsearch' external program; the agent script would remain the same.

The UNIX remote shell transport mechanism was used for agents designed to operate on 38 UNIX workstations of the Dartmouth Computer Science Department network. This network has been configured to have a single workstation as a common mail-server, which precludes the use of electronic mail as a transport mechanism. A user activity monitoring agent was written that traversed the network for a given period of time, recording any recent activity by a specified user. In a typical test run, this agent transported itself 2289 times in approximately 11 hours of monitoring. A variation of this agent is being written which can alert a user that another user has returned to his or her office by detecting the other user's activity on a host in the network and then returning to the originating user's workstation and writing a message on its screen.

A key transportable-agent issue is the ability of agents to navigate within a network and locate resources, be they information resources or other agents. To model an agent-based distributed information searching system, we designed an automated agent generator was designed to produce a specified number of agents, each of which was designed to seek some given information and advertise some other information that they were willing to provide. The information that each agent had and the information that each wanted were assigned at random by the agent generation program. In a test run, 12 agents were created which traversed the network randomly advertising the information they had by writing it into a file at each host, along with information identifying the agent. The agents would then remain at that host for 2.5 minutes searching any files written by other agents for information they were seeking. If a match was found, an agent recorded the identity of the other agent. Figure 8 shows the results of approximately two hours of searching. Further tests are needed to investigate the utility of random agent exploration and to develop more efficient strategies for agent searching.

5 Summary

Our prototype transportable-agent implementation has successfully demonstrated the basic features of a transportable-agent system. Agents can suspend their execution, transfer themselves across a network to another host, and then resume execution where they left off. Although our script language is minimal, the basic features needed to implement transportability are present. We discuss the benefits of using an interpreter to execute the agent code, including heterogeneous agent processing and the ability to enforce security measures such as execution of external programs. We used electronic mail and the UNIX remote shell command as transport mechanisms. We implemented several agents to demonstrate that real applications can be created with this prototype implementation. We anticipate that our prototype implementation will continue to be an effective vehicle for transportable-agent research.

6 Future Work

There are many avenues for future work, some of which we are exploring.

- Incorporating a mechanism for interagent communication into our transportable agent system.
- Replacing our current scripting language with a standard scripting language. Currently Tcl is our leading choice.
- Adding an agent reproduction mechanism that would allow agents to spawn child agents to perform subtasks.
- Implementing various security techniques such as credential verification, file access restriction, and permits.
- Improving the error handling mechanisms.
- Investigating agent navigation issues such as accessing common replicated servers on the Internet and exploration methods used to search for resources.
- Developing more agent applications.

Acknowledgements

Many thanks to Owen Astrachan, Wayne Cripps, Peter Schmidt, and Ralph Valentino who assisted in the development of our transportable agent system. We also thank Daniela Rus for suggesting applications for our transportable agents.

References

[AKW88] Aho, A. V., Kernighan, B. W., and Weinberger, P. J. *The AWK Programming Language*. New York: Addison-Wesley, 1988.

[BN84] Birrell, A. D., and Nelson, B. J. "Implementing remote procedure calls." *ACM Transactions on Computer Systems*, vol. 2, no. 1 (Feb. 1984), pp. 39-59.

[Fal87] Falcone, J. R. "A programmable interface language for heterogeneous distributed systems." *ACM Transactions on Computer Systems*, vol. 5, no. 4 (Nov. 1987), pp. 330-351.

[SG90] Stamos, J. W., and Gifford, D. K. "Remote Evaluation." *ACM Transactions on Programming Languages and Systems*, vol. 12, no. 4 (Oct. 1990), pp. 537-565.

[Sto94] Stoyenko, A. D. "SUPRA-RPC: SUbprogram PaRAmeters in Remote Procedure Calls." *Software–Practice and Experience*, vol. 24, no. 1 (Jan. 1994), pp. 27-49.

[Whi94] White, J. E. "Telescript Technology: The Foundation for the Electronic Marketplace." *General Magic White Paper*, 1994.

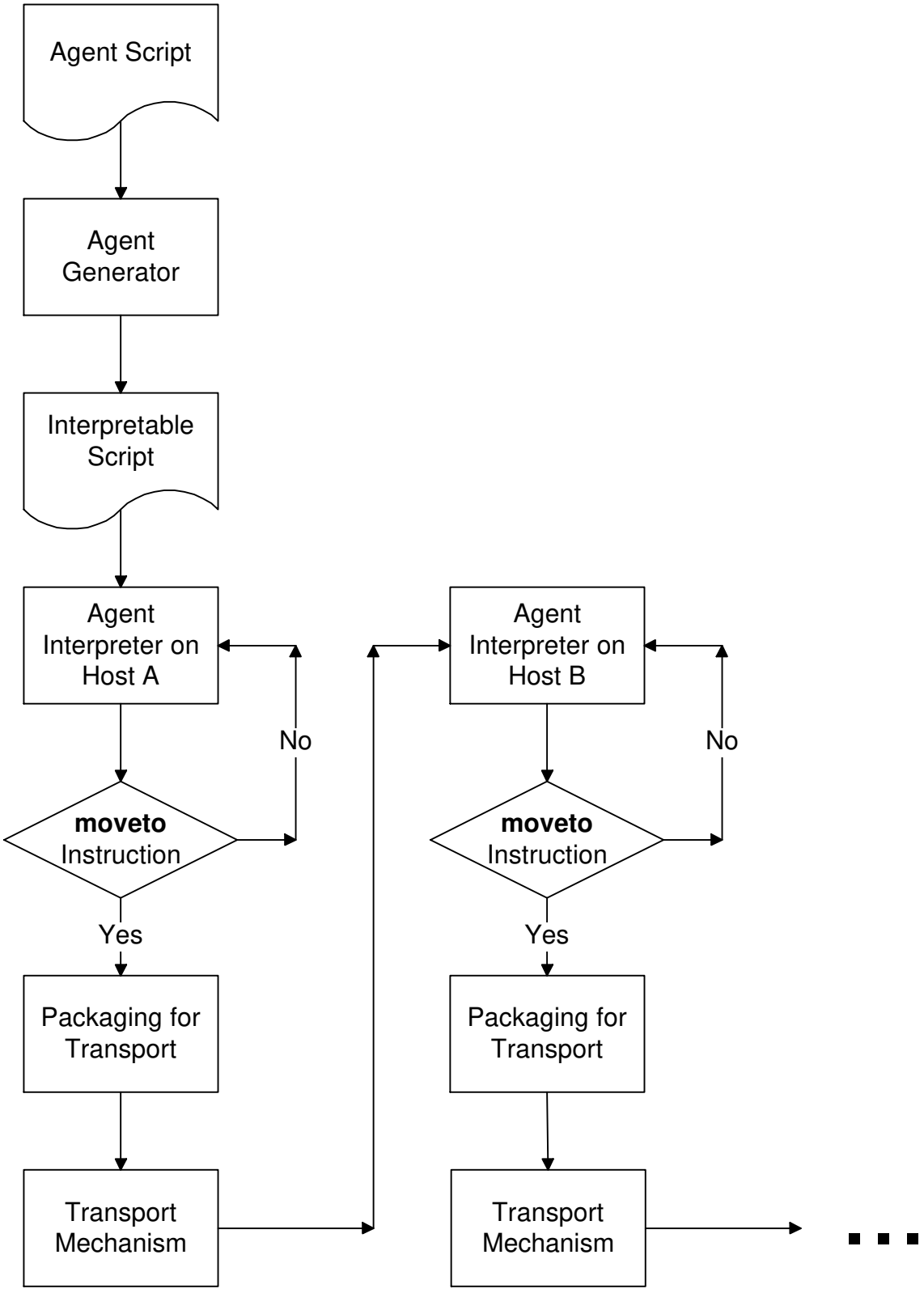

Figure 1. Overview of the prototype implementation.

```
# who.agent
# Performs a 'who' on multiple hosts and outputs
#    the results to 'who.output'

Host["a"] = "agent@grafton.dartmouth.edu"    # add host to Host array
Host["b"] = "agent@coos.dartmouth.edu"       # add host to Host array
Host["c"] = "agent@everest.cs.dartmouth.edu" # add host to Host array
k = 0

foreach i in Host
   moveto Host[i]
   if _CurrentHostName == Host[i] # are we where we want to be?
      k = k + 1
      WhoList[k] = _CurrentHostName # add current host to output array
      if "who" in _SystemCommand    # verify who is available
         Who = system "who"         # invoke the who command,
                                    # system returns output of who
                                    # as an array
         foreach j in Who
            k = k + 1
            WhoList[k] = "   " & Who[j] # add who line to output array
         endforeach
      else
         k = k + 1
         WhoList[k] = "   who unavailable" # add error message to output
      endif
   else
      WhoList[k] = "   moveto host " & Host[i] & " failed"
   endif
endforeach

moveto "agent@" & _OriginalHostName # return to original host
if _CurrentHostName == _OriginalHostName # are we where we want to be?
   i = 1
   while i <= k
      if i == 1
         output > who.output WhoList[i] # write line of output array to
                                        # who.output, overwriting any
                                        # previous file
      else
         output >> who.output WhoList[i] # append line of output array to
                                         # who.output
      endif
      i = i + 1
   endwhile
endif
```

Figure 2. "who" agent.

```
grafton.dartmouth.edu
   operator    tty0        Aug 18 16:05
coos.dartmouth.edu
   operator console Aug 21 04:09
   anne     ttyp0   Aug 22 19:40        (tn-sn-271.dartmo)
   lindell  ttyp2   Aug 22 17:00        (tn-sn-213.dartmo)
   pete     ttyp3   Aug 21 12:47        (ncd:0.0)
   seiler   ttyp4   Aug 22 19:30        (tn-sn-46.dartmou)
   daveruba ttyp5   Aug 22 20:14        (128.231.154.3)
   dpease   ttyp6   Aug 22 20:15        (tn-sn-123.dartmo)
   kotay    ttyp7   Aug 22 20:21        (windsor)
   kyrie    ttyp8   Aug 22 20:22        (exodus.valpo.edu)
   pete     ttyp9   Aug 22 17:58        (tn-sn-171.dartmo)
   jcollins ttypa   Aug 22 19:50        (tn-sn-253.dartmo)
   rajendra ttypb   Aug 22 18:58        (antares)
   hoyler   ttype   Aug 22 09:08        (wolf359)
   hoyler   ttypf   Aug 22 09:08        (wolf359)
   pete     ttyq0   Aug 21 14:31        (ncd:0.0)
   crow     ttyq1   Aug 21 21:40        (tn-sn-81.dartmou)
   darcey   ttyq4   Aug 22 12:28        (WRJVA2.DHMC.DART)
   victor   ttyq7   Aug 22 09:31        (matisse)
   pete     ttyq9   Aug 22 14:53        (ncd:0.0)
   doconnor ttyqa   Aug 22 09:43        (intermedia)
   pete     ttyra   Aug 22 12:07        (ncd:0.0)
   davidg   ttys3   Aug 22 14:15        (polvadera)
everest.cs.dartmouth.edu
   samr     ttyp0   Aug 13 10:44        (ragged.dartmouth)
   wbc      ttyp1   Aug 19 09:51        (tyler.dartmouth.)
   mtaylor  ttyp2   Aug 19 14:04        (silicon.dartmout)
```

Figure 3. "who" agent output.

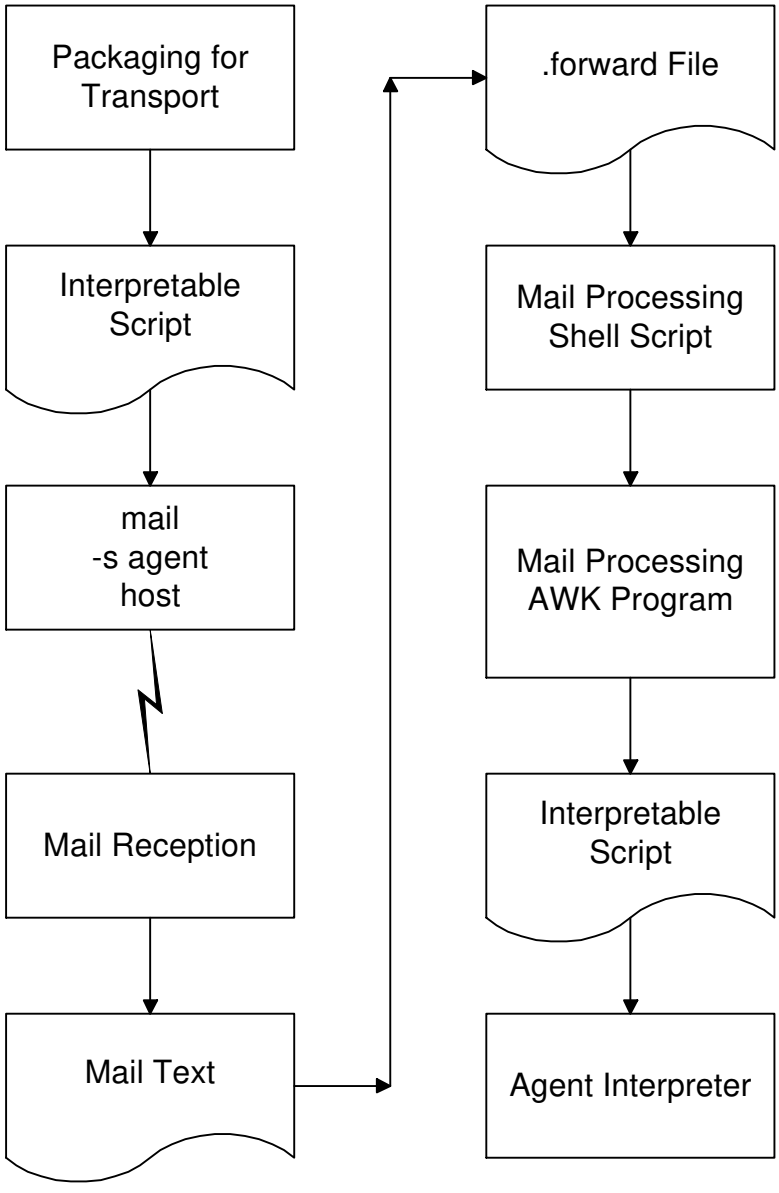

Figure 4. Overview of the electronic mail transport mechanism.

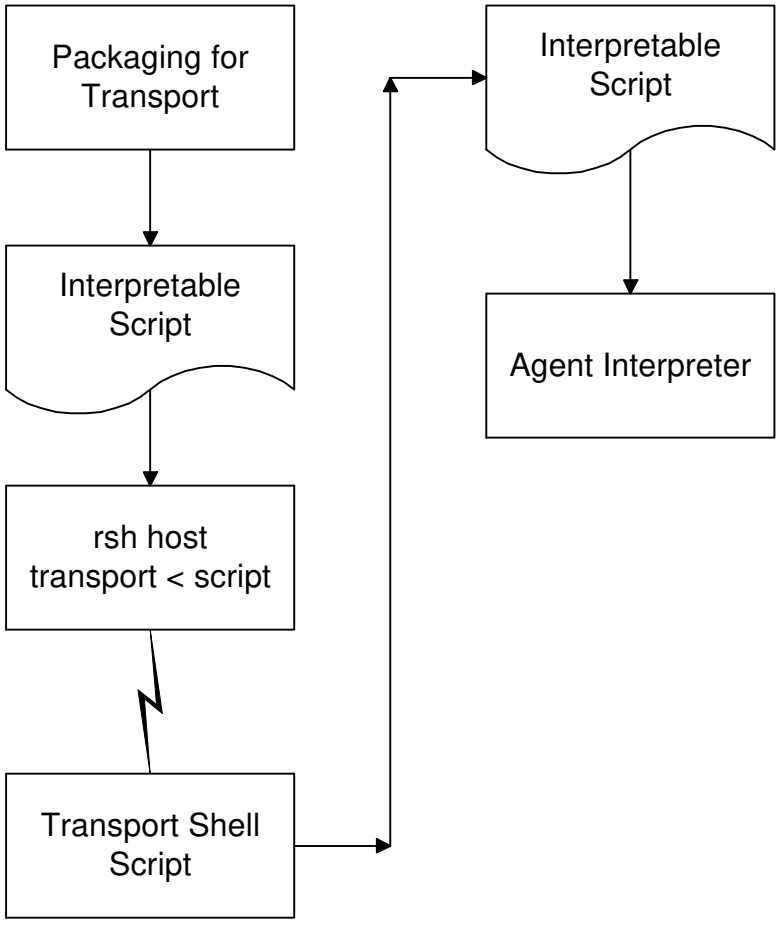

Figure 5. Overview of the UNIX remote shell (rsh) transport mechanism.

```
# trsearch.agent
# Performs a 'trsearch' on multiple hosts and outputs
#    the results to 'trsearch.html'
Host $= "agent@cs.dartmouth.edu"      # append to Host array
Host $= "agent@juliet.cs.duke.edu"    # append to Host array
Host $= "agent@grafton.dartmouth.edu" # append to Host array
Keyword = "distributed"

foreach i in Host
   moveto Host[i]
   if _CurrentHostName == Host[i] # are we where we want to be?
      TRList $= "Host: " & _CurrentHostName # append current host to match
                                            # list
      if "trsearch" in _SystemCommand       # verify trsearch is available
         TR = system "trsearch " & Keyword  # execute trsearch on Keyword,
                                            # system returns output of
                                            # trsearch as an array
         foreach j in TR                    # for each tech report that
                                            # matched
            TRList $= TR[j]                 # append to match list
         endforeach
      else
         TRList $= "trsearch unavailable" # append error message to match list
      endif
   else
      TRList $= "moveto " & Host[i] & " failed"
   endif
endforeach

moveto "agent@" & _OriginalHostName # go back to original host
if _CurrentHostName == _OriginalHostName # did we get there?
   # output results of search as an html file...
   output > trsearch.html "<HTML>"
   output >> trsearch.html "<HEAD><TITLE>Search agent results</TITLE></HEAD>"
   output >> trsearch.html "<BODY>"
   output >> trsearch.html "<H1>Results of search for '" & Keyword & "'</H1>"
   output >> trsearch.html ""
   output >> trsearch.html "<UL>"
   i = 1
   while i <= | TRList |
      if TRList[i] ~ "^Host: " # is TRList[i] a host name?
         output >> trsearch.html "<H2>" & TRList[i] & "</H2>"
      else
         if TRList[i] ~ "://" # is TRList[i] an html address?
            output >> trsearch.html "<LI>"
            output >> trsearch.html "  <A HREF=\"" & TRList[i] & "\">"
            output >> trsearch.html "    " & TRList[i]
            output >> trsearch.html "  </A></LI>"
         else # must be an error message
            output >> trsearch.html "<P>" & TRList[i] & "</P>"
         endif
      endif
         output >> trsearch.html ""
      i = i + 1
   endwhile
   output >> trsearch.html "</UL>"
   output >> trsearch.html ""
   output >> trsearch.html "</BODY>"
   output >> trsearch.html "</HTML>"
endif
```

Figure 6. Technical-report searching agent.

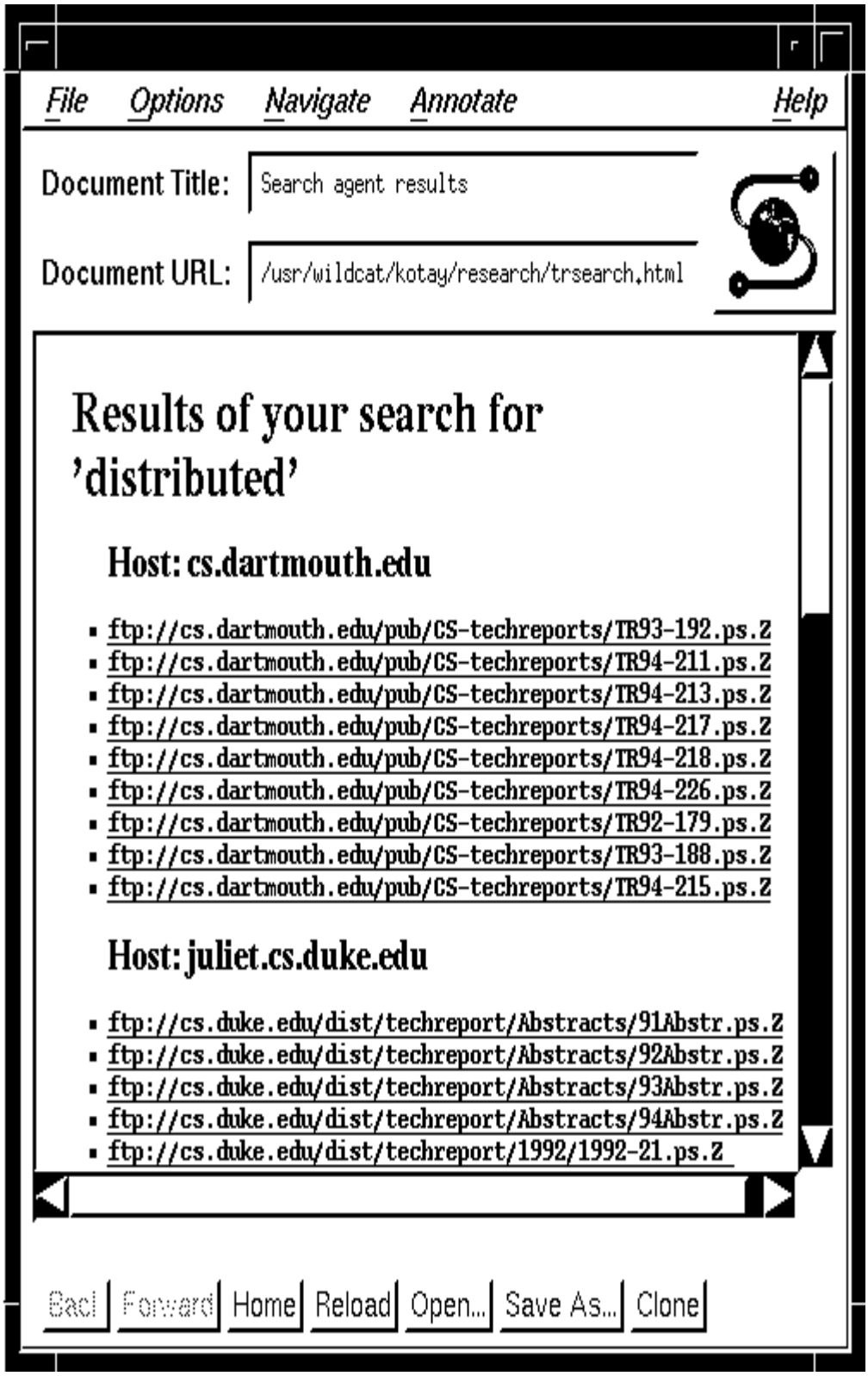

Figure 7. Output of technical-report searching agent viewed in Mosaic.

Agent	Hosts Visited	**Agents Found(# Hosts)** / Agents Not Found
A	35	B **D(2)** **E(1)** **G(2)**
B	36	**C(3)** F **K(2)**
C	33	**H(2)** **I(1)** J **L(1)**
D	32	C **F(3)** **K(1)**
E	34	**A(2)**
F	28	**A(1)**
G	26	**A(2)**
H	31	**C(2)** F **K(3)**
I	32	**A(1)**
J	34	C **F(2)** K
K	27	**H(2)** **I(3)** J **L(2)**
L	28	**C(1)** **F(3)** **K(2)**

Figure 8. Output of the distributed information searching agents. Each agent visited some number of hosts, looking for information possessed by at least one other agent. Bold letters indicate the identity of an agent which was found to possess information sought after by the searching agent. The numbers in parentheses show the number of hosts on which the agent was found. Normal letters indicate agents which possessed the sought after information but were not found. For example, agent A was searching for information possessed by agents B, D, E, and G. Agent B was not found by agent A. Agent D was found by agent A on two hosts, agent E was found on one host, and agent G was found on two hosts.