Mobile agents: Motivations and state-of-the-art systems

Robert Gray, David Kotz, George Cybenko and Daniela Rus

Thayer School of Engineering / Department of Computer Science
Dartmouth College
Hanover, New Hampshire 03755

firstname.lastname@dartmouth.edu

April 19, 2000

Abstract

A mobile agent is an executing program that can migrate, at times of its own choosing, from machine to machine in a heterogeneous network. On each machine, the agent interacts with stationary service agents and other resources to accomplish its task. In this chapter, we first make the case for mobile agents, discussing six strengths of mobile agents and the applications that benefit from these strengths. Although none of these strengths are unique to mobile agents, no competing technique shares all six. In other words, a mobile-agent system provides a single general framework in which a wide range of distributed applications can be implemented efficiently and easily. We then present a representative cross-section of current mobile-agent systems.

1 Introduction

A *mobile agent* is an executing program that can migrate, at times of its own choosing, from machine to machine in a heterogeneous network. On each machine, the agent interacts with stationary service agents and other resources to accomplish its task. Although numerous mobile-agent systems now exist, mobile agents are not widely used in production applications. There are three reasons for this. First, agent mobility is most useful in highly dynamic, mobile-computing environments. Such environments are just now becoming common with the recent dramatic increases in the number and power of portable computing devices. Second, most existing mobile-agent systems are research prototypes. No single system has yet implemented all of the features needed for robust and efficient operation. Finally and most importantly, there are no applications that *require* mobile agents. Any specific application can be realized just as efficiently with more traditional techniques.

When a *set* of applications is considered as a group, however, the advantage of mobile agents becomes clear—they provide a single general framework in which a wide range of distributed applications can be implemented easily, efficiently and robustly. Without mobile agents, many applications require combinations of more traditional implementation techniques, and more importantly, different applications require *different* combinations of techniques.

In this chapter, we make the case for mobile agents, drawing on experience with our own mobile-agent system, D'Agents.[1] To make our case, we examine some of the applications that we have built with D'Agents, and compare the performance of D'Agents with traditional client-server architectures.

Section 2.1 presents two motivating applications, while Section 2.2 outlines six strengths of mobile agents. We argue that although none of these strengths are unique to mobile agents (and some are not fully realized in

[1] `http://agent.cs.dartmouth.edu/. D'Agents was once known as` *Agent Tcl*.

any *current* mobile-agent system), no competing technique shares all six. Once we have made this argument, Section 3 presents a representative cross-section of current mobile-agent systems.

2 Motivation

Mobile agents are a new technology, so it is worth considering in depth their strengths and the situations in which they can be used effectively. In this section, we first present two motivating applications, where a mobile agent, possibly originating on a portable device that has only an unreliable connection to the main network, searches distributed databases for information relevant to a user query. Then, with these two applications as a starting point, we discuss six individual strengths of mobile agents. Finally, we argue that although these same strengths can be realized with combinations of more traditional distributed-computing techniques, no competing technique shares all six strengths. Thus, the true strength of mobile agents is that they provide a *single* infrastructure in which a wide range of distributed applications can be implemented easily, efficiently and robustly.

2.1 Motivating applications

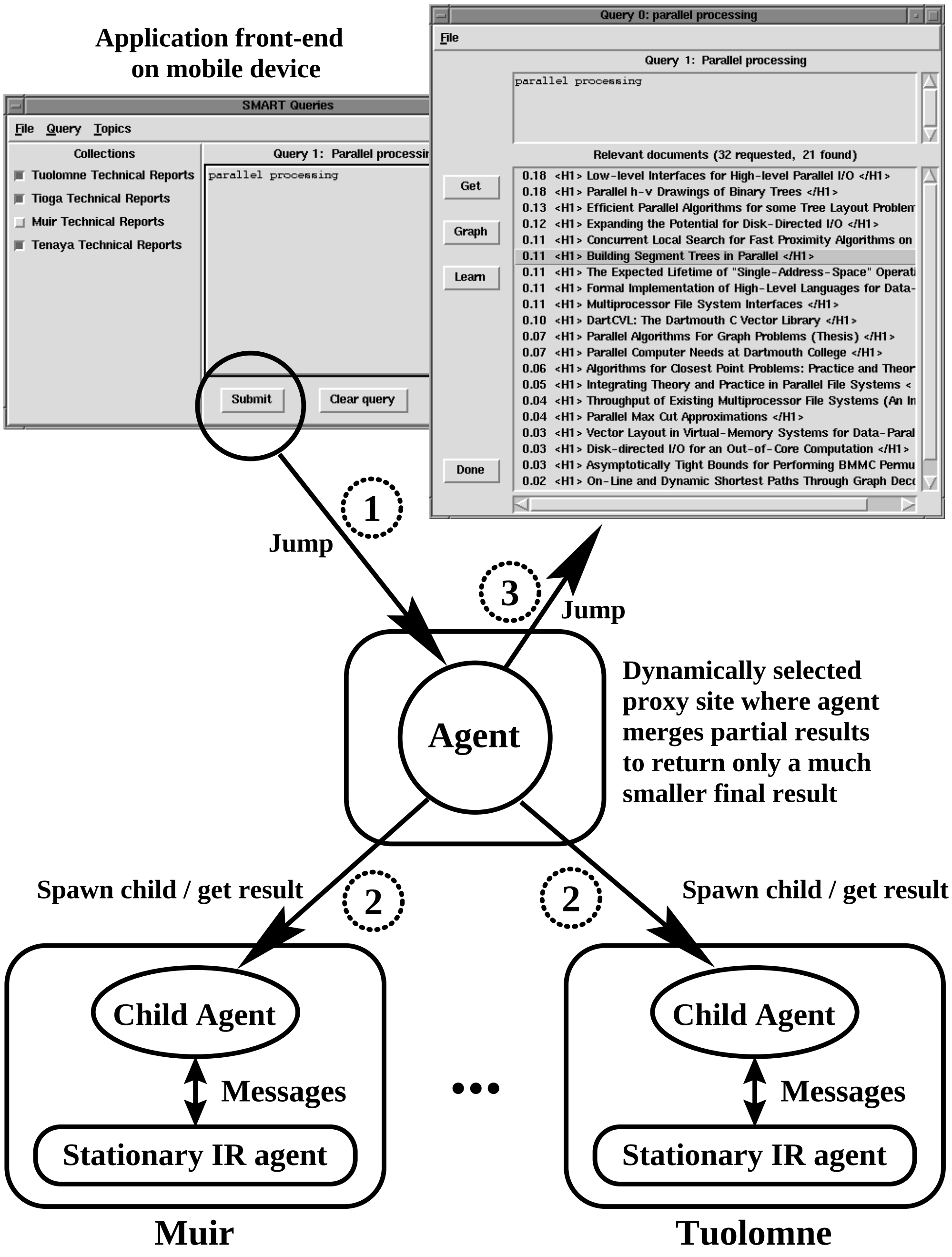
One of the most common applications for mobile agents is distributed information processing, particularly in mobile-computing scenarios where users have portable computing devices with only intermittent, low-bandwidth connections to the main network. A mobile agent can leave the portable device, move to the network location of a needed information resource, and then perform a *custom* retrieval task locally to the resource. Only the final results are transmitted back to the portable device, even though the information resource had no prior knowledge of the client or its specific task. Source data and intermediate results do not need to be transmitted to the portable device. Moreover, the mobile agent can continue the retrieval task even if the network link to the portable device goes down. Once the link comes back up, the agent sends back its results.

We have implemented several information-processing applications on top of the D'Agents system. In this subsection, we present our current implementation of two applications. The first application is a simple information-retrieval application, where a mobile agent finds technical reports relevant to a user's query. The second application is a more complex counter-terrorism application, where mobile agents support the information needs of a platoon of soldiers.

2.1.1 An information-retrieval application

The first application searches the technical reports available at the Department of Computer Science at Dartmouth College. The technical reports are distributed on several Dartmouth machines to approximate a more realistic scenario in which the application would search multiple collections at multiple institutions. Each machine runs the D'Agents system and a basic information-retrieval system. This system, which we will call the *IR system*, is wrapped inside a stationary D'Agents agent. This stationary agent provides a four-function interface to the IR system: (1) run a free-text query and obtain a list of relevant documents, (2) obtain the full text of each document, (3) organize the retrieved documents into (automatically generated) categories, and (4) summarize each category with a few distinctive keywords.

Figure 1 shows the structure of the technical-report application. The stationary interface agents, which appear at the bottom of the figure, register with the *yellow pages* [RGK97] when they begin execution. The yellow pages, which are not shown in the figure, are a simple distributed directory service. When the interface agent registers, it provides its location (i.e., its identifier within the D'Agents namespace) and a keyword description of its service (i.e., *technical-reports* and *free-text-queries*). A client agent searches for a service by sending a keyword query to the yellow pages. The yellow pages respond with the locations of all services whose keyword lists match the keyword query (more specifically, all services whose keyword lists are a superset of the keyword query). A forthcoming version of the yellow pages will allow the client agents to search by interface definition as well [NCK96].



```
parallel processing
```

```
Query 1:  Parallel processing
parallel processing
Relevant documents (32 requested,  21 found)
0.18  <H1> Low-level Interfaces for High-level Parallel I/O </H1>
0.18  <H1> Parallel h-v Drawings of Binary Trees </H1>
0.13  <H1> Efficient Parallel Algorithms for some Tree Layout Problem
0.12  <H1> Expanding the Potential for Disk-Directed I/O </H1>
0.11  <H1> Concurrent Local Search for Fast Proximity Algorithms on
0.11  <H1> Building Segment Trees in Parallel </H1>
0.11  <H1> The Expected Lifetime of "Single-Address-Space" Operati
0.11  <H1> Formal Implementation of High-Level Languages for Data-
0.11  <H1> Multiprocessor File System Interfaces </H1>
0.10  <H1> DartCVL: The Dartmouth C Vector Library </H1>
0.07  <H1> Parallel Algorithms For Graph Problems (Thesis) </H1>
0.07  <H1> Parallel Computer Needs at Dartmouth College </H1>
0.06  <H1> Algorithms for Closest Point Problems: Practice and Theor
0.05  <H1> Integrating Theory and Practice in Parallel File Systems <
0.04  <H1> Throughput of Existing Multiprocessor File Systems (An In
0.04  <H1> Parallel Max Cut Approximations </H1>
0.03  <H1> Vector Layout in Virtual-Memory Systems for Data-Paral
0.03  <H1> Disk-directed I/O for an Out-of-Core Computation </H1>
0.03  <H1> Asymptotically Tight Bounds for Performing BMMC Permu
0.02  <H1> On-Line and Dynamic Shortest Paths Through Graph Dec
```

Figure 1: An example application. Here a mobile agent is searching a distributed collection of technical reports. The agent first decides whether to move to a dynamically selected proxy site. Then it decides whether to spawn child agents or simply interact with the individual document collections from across the network. *muir* and *tuolomne* are two machines at Dartmouth. Note that the yellow pages, which the agent uses to discover the locations of the document collection, are not shown in this figure.

The main application is a GUI that runs on the user's machine. This GUI, which is shown at the top of Figure 1, lets the user enter a free-text query and optionally select specific document collections from a list of known document collections. The GUI then launches a small mobile agent onto the local machine. The agent's task is to find the available interface agents and retrieve the relevant technical reports.

The agent can go about the retrieval task in several ways. It makes two main decisions after consulting with a simple set of *network sensors* [RGK97]. First, if the network connection between the user's machine and the network is reliable and has high bandwidth, the agent stays on the user's machine. If the connection is unreliable or has low bandwidth, the agent jumps to a *proxy* site within the permanent network. This proxy site is shown in the middle of Figure 1. With our current reliability and bandwidth thresholds, the agent typically will remain on the user's machine if the machine is a workstation with a 10 Mb/s Ethernet link. The agent will jump to a proxy site if the user's machine is a laptop with a wireless Ethernet or modem link. The agent dynamically selects the proxy site. In the current system, the selection process is quite simple—there is a designated proxy site for each laptop and for some subnetworks. The agent will go to the proxy associated with the subnetwork to which the laptop is attached, or to the laptop-specific proxy if there is no subnetwork proxy. Currently, the proxy sites are hard coded, but eventually, they will be listed in the yellow pages along with other services. Then an agent can search for the closest proxy site (to its current location or to the document collections), the closest proxy site owned by its owner's Internet Service Provider (ISP), the fastest proxy site, and so on.

Once the agent has decided whether to move to a proxy site, it consults the yellow pages to find the available document collections. Once it has found the available collections (and the associated interface agents), the agent makes its second decision. If the query requires only a few operations per document collection, the agent simply makes RPC-like calls across the network (using the D'Agent communication mechanisms as described in [NCK96]). If the query requires several operations per document collection, or if the operations involve large amounts of intermediate data, the agent sends out child agents that travel to the document collections and perform the query operations locally, avoiding the transfer of intermediate data. In our case, the number of operations per document collection depends on whether the user wants to see a graphical representation of the query results (one additional operation per collection), whether the user wants to retrieve the document texts immediately or at a later time (one additional operation per document), and whether the user has specified alternative queries to try if the main query does not produce enough relevant documents (one additional operation per alternative query). The size of the intermediate data depends on the average size of the documents in each collection and the average number of relevant documents per query. Since the average document size and average number of relevant documents per query is nearly the same for all of our document collections, our current agent makes its decision based solely on the number and type of the required query operations. Later, once our yellow pages accept interface descriptions, we will allow each interface agent to annotate its description with the expected result size for each operation (and to update those annotations based on its observations of its own behavior).

When the main agent receives the results from each child agent, it merges and filters those results, returns to the user's machine with just the final list of documents, and hands this list back to the GUI.

2.1.2 A counter-terrorism application

A more complex application is a counter-terrorism application,[2] which was developed as part of a Multi-University Research Initiative (MURI) funded by the Department of Defense,[3] and demonstrated in late 1999. The basic scenario is shown in Figure 2. An intelligence team at headquarters has intercepted one or more phone calls and determined that a terrorist group is likely to meet in a particular building. Headquarters dispatches a team of soldiers to the building. The soldiers take up observation posts (probably hidden) around the building, and wait for the suspected terrorists. If the terrorists come to the building, the soldiers will secure the building and arrest the terrorists. Such a scenario is common in peace-keeping missions, where there is no overt military resistance to the peace-keeping force, but instead acts of terrorism aimed at slowing down or stopping the process of rebuilding the country's institutions.

[2] `http://actcomm.dartmouth.edu/`

[3] AFOSR/DoD contract F49620-97-1-03821

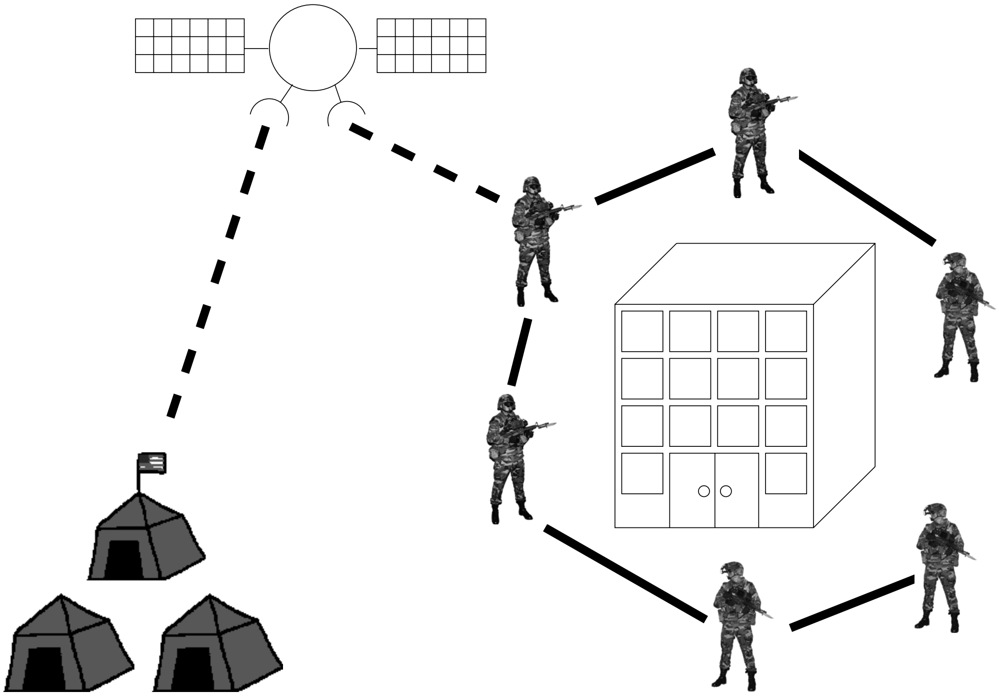


Figure 2: A counter-terrorism application. A platoon of soldiers has established observation posts around a building where suspected terrorist activity is taking place. Mobile agents are used to dynamically deploy scenario-specific code onto the soldiers' devices, and to perform queries against available information resources.

Each soldier has a portable computing device with a GPS unit and a wireless Ethernet network card. One or more soldiers serve as gateways between the wireless cloud and the main military network—these soldiers have additional transmission equipment to establish a satellite connection back to headquarters. Of course, in our demonstration application, a satellite was not available, so the satellite connection was simulated with a wireless Ethernet connection on a different frequency. Routing data to or from a particular soldier then is a matter of routing that data to or from the gateway soldiers. The APRL ad-hoc routing system from the EECS group at Harvard, which is a member of the MURI project, provides the basic routing functionality. APRL continually broadcasts "ping" packets to determine which devices are within transmission range of each other. As the soldiers and thus their devices change position, APRL updates the network routing tables so that packets are routed through as many devices as necessary to reach the gateways.

Mobile agents play several roles within this application. First, the intelligence team at headquarters and the soldiers in the field perform several kinds of queries against different information sources. These information sources include databases of intercepted phone calls, physical descriptions of known terrorists, and military and public news articles that relate to the country in which the peace-keeping force finds itself. Just as in the case of the technical-report application, the mobile agent can move to the location of the information resource, and perform its task without sending back the source or intermediate data. Without mobile agents, the same bandwidth efficiency could be achieved only if each information resource provided a single high-level operation for each retrieval task. Even in a military environment, where the information resources and their clients are nominally under the control of a single entity, it is unlikely that such a one-to-map mapping between resource operations and client tasks can be achieved. Each resource and client is likely under the control of a different programming group within an extremely large military organization.

Moreover, the mobile agent can continue a multi-step query even if the network link to the client machine goes down. This is particularly important for queries launched from the soldier machines, since the soldiers continually move relative to each other, and network disconnections occur frequently. It is also important, however, for queries launched from headquarters, since headquarters itself might be located at some field location, and might have a slow or unreliable link to the rest of the military system.

Mobile agents are also used to carry special-purpose interface code onto the soldiers' devices. For example, when the soldier sees a possible suspect enter the building, the soldier sends an observation of that suspect back to headquarters. Headquarters searches available databases to try to identify exactly who that suspect is. Once one or more possible candidates have been identified, headquarters sends pictures of the candidates, along with the *code to display the pictures*, to the soldier who made the initial observation. The soldier interacts with this dynamically-deployed code to examine the pictures and identify which picture, if any, shows the suspect that she actually saw. Of course, the code to display the pictures and accept the soldier's confirmation could be pre-installed on the soldier's device. In a peace-keeping situation, however, a group of soldiers might find themselves involved in a new, unique type of mission, under the direction of a headquarters from a different branch of the military or even a different country. The soldier might never have needed to select a suspect from a photo lineup before. By dynamically deploying the photo-lineup code to the soldier's device, when and if needed, the military avoids the need to engage in a costly and administratively complex software-installation phase before each mission.

2.2 Reasons for mobile agents

The two applications highlight six strengths of mobile agents.

2.2.1 Conservation of bandwidth

In the traditional client/server model [Lew95], the server provides a fixed set of operations that a client invokes from across the network. If the server does not provide a single operation that matches the client task exactly, either the client must invoke a sequence of server operations, or the server developer must add a new operation to the server. The first option brings intermediate data across the network on every operation, potentially wasting a significant amount of network bandwidth, especially if the intermediate data is not useful beyond the end of the client task. The second option becomes an intractable programming burden as the number of distinct clients increases. In addition, it discourages modern software engineering, since the server becomes a collection of complex, specialized routines, rather than simple, general primitives.

A mobile agent, on the other hand, does not waste bandwidth even if the server provides only low-level operations, simply because the agent migrates *to the server* [Whi94b]. The agent performs the necessary sequence of operations locally, and returns just the final result to the client. Agents that do more work avoid more intermediate messages and conserve more bandwidth. Figures 3a and 3b illustrate this bandwidth conservation by showing the bandwidth usage for two simplified versions of the technical-report application above.

In both figures, a single mobile agent is sent from a front-end application on one machine to query a database of technical reports on a second machine. Since the agent is querying only a single database, it does not use a proxy site. The database provides two operations: (1) obtain a list of documents relevant to a given query, and (2) obtain the text of a particular document for further inspection. Each query is simply a list of keywords; the database uses the standard vector-space model to determine a relevance score for each document [Sal91], and then returns a list of the most relevant documents to the agent.[4]

In Figure 3a, the agent performs a series of keyword queries to get the desired list of relevant documents. For example, suppose that the agent starts with an extremely specific query, but has an automated technique for making the query more general (e.g., dropping out the most specific keywords, or replacing keywords with synonyms of those keywords). The agent performs keyword queries against the database, making the query more general each time, until it has found a user-specified number of documents whose relevance scores are above a user-specified threshold. Figure 3a shows, as a function of the number of queries, the bandwidth

[4] The agent specifies a relevance threshold, and the database returns those documents whose relevance scores are above the threshold.

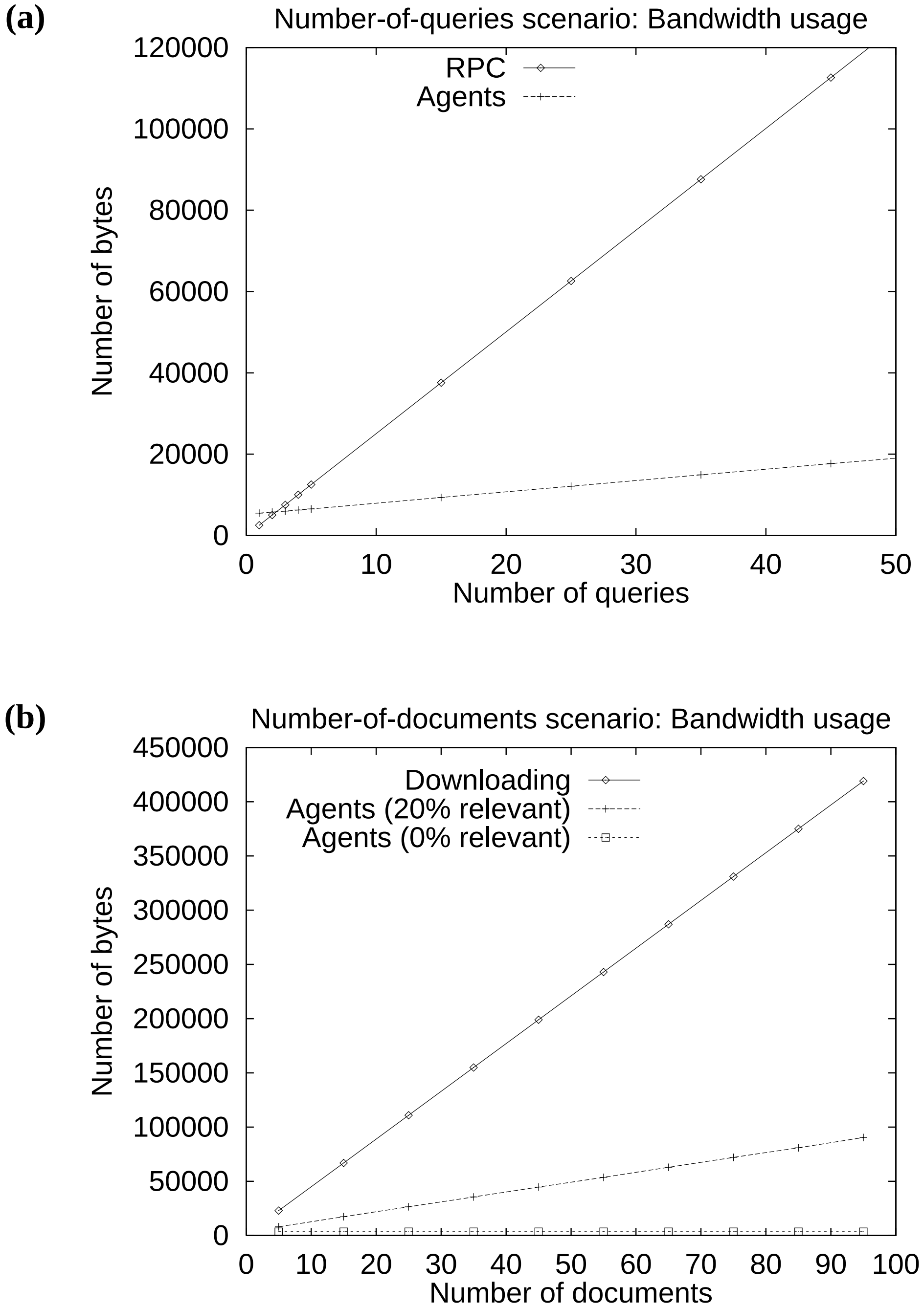


Figure 3: **(a)** Bandwidth usage of mobile agents versus Remote Procedure Call (RPC) when the application must perform multiple query operations to get the final list of relevant documents. **(b)** Bandwidth usage of agents versus downloading when the application must make one initial query and then inspect the *text* of each retrieved document to determine if it is actually relevant.

usage of such an agent versus the bandwidth usage of an equivalent client/server application (using Remote Procedure Call) that performs the queries from across the network.

The agent was written in Tcl and ran inside our mobile-agent system, D'Agents [Gra96]. The database was wrapped inside a stationary Tcl agent for interaction with the agent, and inside a simple C++ server for interaction with the traditional client. The traditional client was written in C++ and used Sun RPC calls (over UDP) to interact with the C++ server. In the RPC case, the size of each query was 256 bytes, and the size of each query result was 2048 bytes. In the agent case, the size of the agent was $1024 + 256 + 128n$ bytes, where n is the number of queries that will be performed, the 1024 the size of the agent's code, the 256 the size of the initial query, and the 128 the size of the additional code and data that is needed to construct and perform each of the more general queries.

One kilobyte of agent code may seem small. This code size is consistent with what was observed in the actual technical-report application, however. Several factors explain the small size. First, the agent's task involves three distinct steps: contact the stationary Tcl agent to perform the initial query, check the number of retrieved documents, and then construct and perform a new query if needed. Since Tcl is a scripting language, and has a powerful set of string- and list-processing commands, each of these three steps can be accomplished with an extremely small amount of code. Second, the agent performs *only* these three steps. The main application remains on the client machine, since only the retrieval subtask is performed on the database's machine. When a large application sends out a much smaller agent to perform a specific subtask, as in this case, the agent is often called a *minion* of the main application [BGH$^+$99]. Third, comments and unnecessary whitespace are removed from the agent before the agent is sent to the database's machine, since neither the comments nor the whitespace are needed to execute the agent. This "stripping" of the agent makes the agent even smaller. Fourth, the kilobyte of code does not include the agent header, which contains information about the agent's origin and owner. This header is approximately one additional kilobyte. Since this header is constant size across all the agents in our experiments, we will not consider it again. The reader should remember, however, that this header is being transmitted to the database's machine, and that it is included in the bandwidth data points. Finally, and perhaps most importantly, even if the agent were significantly larger, such as if the agent were doing a more complex filtering step, the agent would still outperform the traditional client/server approach in most cases. For example, if the agent were twenty times larger, twenty kilobytes of code instead of one kilobyte, the agent would still outperform the traditional client/server approach in all cases in Figure 3b, although the crossover point in Figure 3a would increase from two to eight queries.

The Figure 3a experiments were performed on two Linux laptop computers connected with a dedicated 10 Mb/s Ethernet link that went through only a single hub.[5] The security features of D'Agents were turned off, so both the agents and the RPC calls were unencrypted.[6] The actual query operation in the database interface was replaced with a stub that just returns a hard-coded document list, since whether the query is actually performed has no bearing on the results. Finally, only one agent or RPC call was active at a time—there was no competition for the network link, and thus there was no extra bandwidth usage due to re-transmissions. Although this is a network situation that would not occur in real life, it does not affect the relative results. For all cases where a single agent uses the network less than a single RPC client, a group of those agents would use the network less than an equivalent group of clients. The agents would produce fewer network collisions and fewer re-transmissions, and each individual agent would still be generating less traffic.

Figure 3b shows the results for similar experiments where either (1) an agent makes a single query against the database, inspects the document *texts* to decide which documents are truly relevant, and sends only the relevant documents back to the front-end application, or (2) a traditional client makes the initial query and then downloads all the retrieved documents to inspect them locally. The x-axis is the number of documents retrieved by the initial query, i.e., the number of documents whose texts are inspected to determine their

[5] More specifically, the laptops were identical, each with a 200 MHz Pentium II processor, 64 MB of main memory, 128 MB of swap space, and 1.7 GB of disk space. The operating system on each laptop was Slackware Linux, kernel version 2.0.30. All C/C++ code used in the experiments (including the D'Agent interpreters and servers) was compiled with GNU gcc version 2.7.2.3 with an optimization level of 2.

[6] The relative difference in the bandwidth usage should be approximately the same if we compared the secure version of D'Agents against a secure version of RPC. We have not yet confirmed this experimentally, however.

actual relevance. The experimental setup was identical to the first case, except that the traditional client made a single TCP/IP connection to the C++ server (rather than RPC calls) and downloaded all of the document texts across that one connection. The size of each document was 4096 bytes. In both cases, once the agent or client obtained the initial list of documents via the keyword query, it determined which documents were actually relevant by looking for a particular two-word phrase.[7] As above, this subphrase search can be implemented compactly with the string-processing capabilities of Tcl. Documents that did not contain the two-word phrase were considered irrelevant and were not sent back to the front-end application. The figure shows two cases for the agent approach, one where none of the documents contained the two-word phrase, and one in which twenty percent of the documents contained the two-word phrase. Since the agent sends back only the relevant documents, these two cases have significantly different bandwidth usage.

As one would expect, in both Figures 3a and 3b, the agent uses far less bandwidth than the equivalent client/server solution as the number of queries or documents increases, simply because the agent is transferring neither intermediate results nor irrelevant results. Similar bandwidth conservation can be seen in other agent applications. For example, StormCast [JJS$^+$97], which was built on top of the Tacoma mobile-agent system [JSvR98a], is a distributed weather-monitoring system in which (1) the data volumes are so immense as to make data movement *impractical*, and (2) the operations that different StormCast users want to perform against the data change rapidly over time. Mobile agents allow new monitoring and retrieval operations to be rapidly constructed and rapidly deployed to the sensor and data locations.

Of course, in some applications, the agent's code will be larger than any intermediate results, and the agent will actually consume more bandwidth than the equivalent client/server solution. Thus, the goal of many mobile-agent researchers is to provide sufficient introspection and communication capabilities so that an agent can estimate its potential bandwidth usage, and then decide whether to move to the location of the data, or remain stationary and interact with the data from across the network (as if it were a traditional client) [Moi98].

2.2.2 Reduction in total completion time

As we saw in the previous section, a mobile agent often uses much less bandwidth than a corresponding RPC-based client. Except for the currently rare case in which the end user is paying for network access on a per-byte basis, however, the end user is only interested in conserving bandwidth if it leads to a reduction in the query completion time. In this section, we will show that agents can complete the queries faster than the RPC-based clients, since the agents avoid the time-consuming transmission of intermediate results. At the same time, however, the agents complete the queries faster only if the queries involve enough database operations for the transmission time-savings to make up for the migration overhead. In the current implementation of D'Agents, the required number of database operations is quite large for medium-speed or faster networks (such as a 10 Mb/s Ethernet), but quite small and reasonable for slower networks.

Base performance. Before looking at the query completion times, it is worth considering the base performance of the D'Agents system. Figure 4 shows the results of three experiments, all of which were run on the same two laptops and under the same conditions as the experiments that were discussed in the previous section. The first experiment, the results of which are shown as the *RPC* line in Figure 4, measures the time needed for a client on one laptop to make a Sun RPC call (over UDP) into a server on the second laptop. The total size of the arguments to the RPC call was 256 bytes; the result size varied from 512 bytes to 8192 bytes. The server responds to each client request immediately with a dummy result of the appropriate size. This first experiment does not involve the agent system in any way, but instead is used to place the D'Agent performance numbers in context.

The second and third experiments do involve the agent system. In the second experiment, an agent on the first laptop sent a request message to an agent on the second laptop; the agent on the second laptop then sent back a response. The *Agent messages* line in Figure 4 shows the total round trip time as a function of response size. As in the RPC experiment, the request size was 256 bytes, and the "server" agent simply answered each request with a dummy response of 512 to 8196 bytes. Finally, in the third experiment, an agent

[7] The database does not provide a query operation that takes the adjacency of words into account.

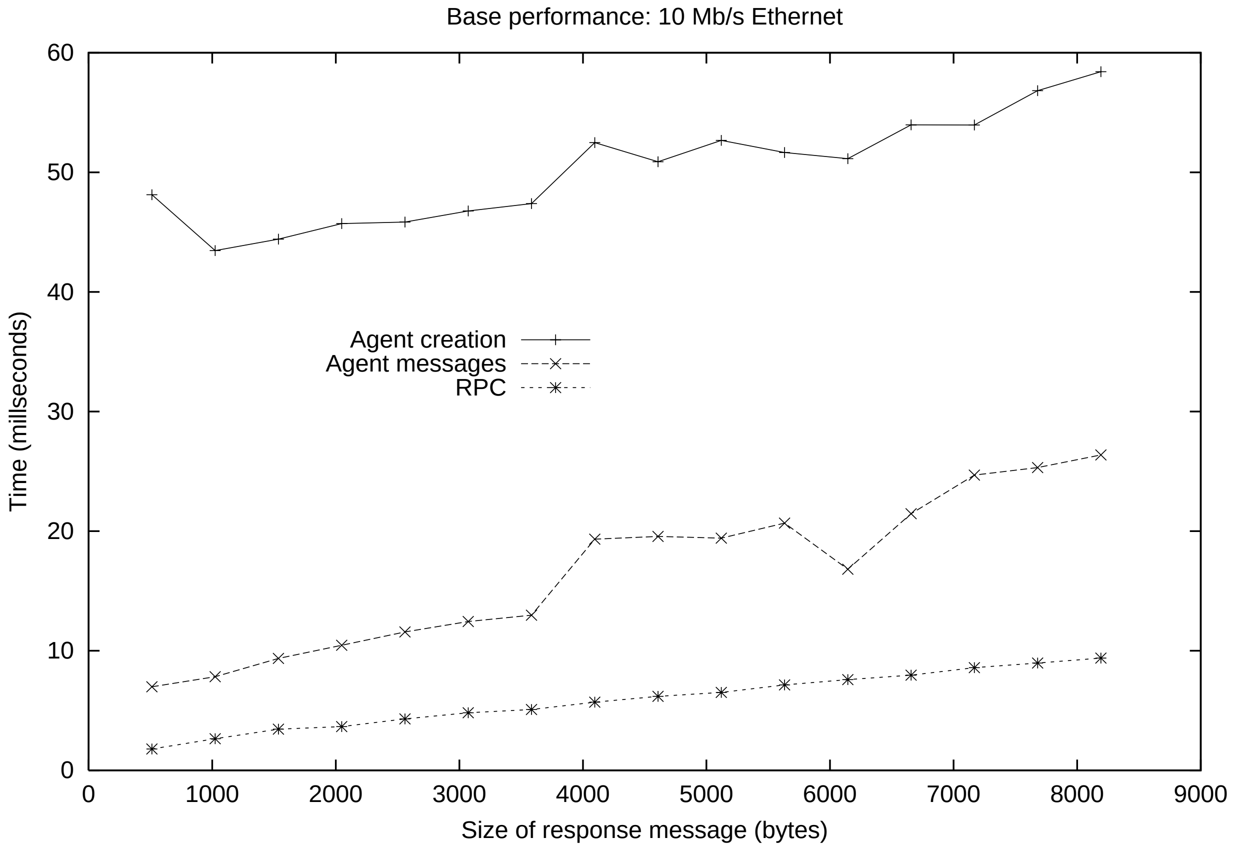


Figure 4: Base performance of the D'Agents system in a 10 Mb/s Ethernet network. Sending a message from one agent to another agent on a different machine (**Agent messages**) is significantly slower than a traditional RPC call (**RPC**) due to the overhead of establishing a TCP/IP connection. In turn, sending an *agent* to a different machine (**Agent creation**) is significantly slower than sending a message due to the overhead of injecting the agent into an appropriate execution environment. These two overheads, which are similar in slower network environments, play a large role in the outcome of the other experiments. Fortunately, there are several straightforward ways to reduce the overheads.

on the first laptop sent a *child agent* to the second laptop; the child agent then sent back a dummy response message to its parent. The *Agent creation* line in Figure 4 shows the total time from agent submission to result reception. The size of the child agent was 256 bytes, and the size of its response message was 512 to 8196 bytes.

Unlike previous performance results for D'Agents [Gra97], all the agent experiments were performed with the new multi-threaded version of the D'Agents server, which eliminates significant interprocess communication and startup overhead. The new server maintains a pool of "hot" interpreters. It starts up a set of interpreter processes at boot time, and then hands incoming agents off to the first free interpreter in that set. An interpreter process does not die when an agent finishes, but instead stays alive to execute the next incoming agent. Although this approach still runs each agent in a separate process, it eliminates nearly all of the interpreter-startup overhead.[8]

As before, the agent experiments were performed with encryption turned off. The agents would perform significantly worse with encryption turned on, but so would an equivalently secure version of RPC. Turning encryption off is reasonable, since many document collections would not care about the identity of the agent's owner.

The base performance numbers illustrate two important points: (1) inter-agent communication involves significantly more overhead than RPC, and (2) migration involves even more overhead than inter-agent communication. Fortunately, this overhead comes from several clear sources. First, the agents are written in Tcl, which is a relatively slow scripting language. D'Agents, however, now includes two faster languages, Java and Scheme, which could be used in place of Tcl.[9] In addition, the newest version of the Tcl interpreter,[10] which has not been integrated into D'Agents, uses on-the-fly compilation (into virtual-machine bytecodes) and is two to ten times faster than previous Tcl interpreters. Second, even though the multi-threaded server uses a pool of hot interpreters, each Tcl interpreter must still execute several hundred lines of Tcl code to re-initialize itself before executing an incoming agent. This re-initialization adds nearly ten milliseconds to the migration time. Some of the re-initialization can be eliminated, and the rest can be made much faster, either by switching to the newest Tcl interpreter or re-implementing the initialization code in C.

Finally, whereas the RPC client and server use UDP, every communication between machines in the agent system involves a TCP connection. Thus, there is one TCP connection for the request message or migrating agent, and a second connection for the response. Moreover, all communication goes through the agent servers, which, although necessary in the case of a migrating agent, is not necessary when one agent is simply sending a message to another. Possible implementation changes include using UDP for some agent communication, increasing the speed with which the agent server forwards incoming messages to the correct interpreter processes, allowing the agent servers to "cache" open connections to machines with which they are communicating heavily, and possibly even associating unique network addresses with stationary service agents so that client agents can communicate with those service agents directly. Unfortunately, this last change complicates the security implementation, since the agent server will no longer be the single point at which the system needs to authenticate incoming messages.

Performance for multiple queries. Figures 5a, 5b and 5c shows the completion times for the scenario where the searcher performs multiple queries against the single document collection, but does not examine the document texts. The bandwidth usage for this same scenario was shown in Figure 3a. Figure 5a shows the completion times when the laptops were connected with a dedicated 10 Mb/s wired Ethernet; Figure 5b shows the completion times when the laptops were connected with a dedicated 1 Mb/s wireless Ethernet (Digital/Cabletron RoamAbout cards); and Figure 5c shows the completion times when the laptops were connected with a dedicated 100 Kb/s radio modem link (Metricom Richochet cards). Aside from performing the experiments in each of the three different network environments, the experimental conditions were identical to those of Figure 3a.

[8] It does not eliminate all of the overhead since each interpreter process is only allowed to handle a certain number of agents before it terminates and is replaced with a new process. In addition, even though the interpreter process remains active from one agent to another, there is still some initialization and cleanup that must be done for each agent.

[9] We finished adding full support for Scheme and Java as this chapter was going to press. Future publications will report on the performance of Java and Scheme agents.

[10] Tcl 8.2 from Scriptics (`http://www.scriptics.com/`)

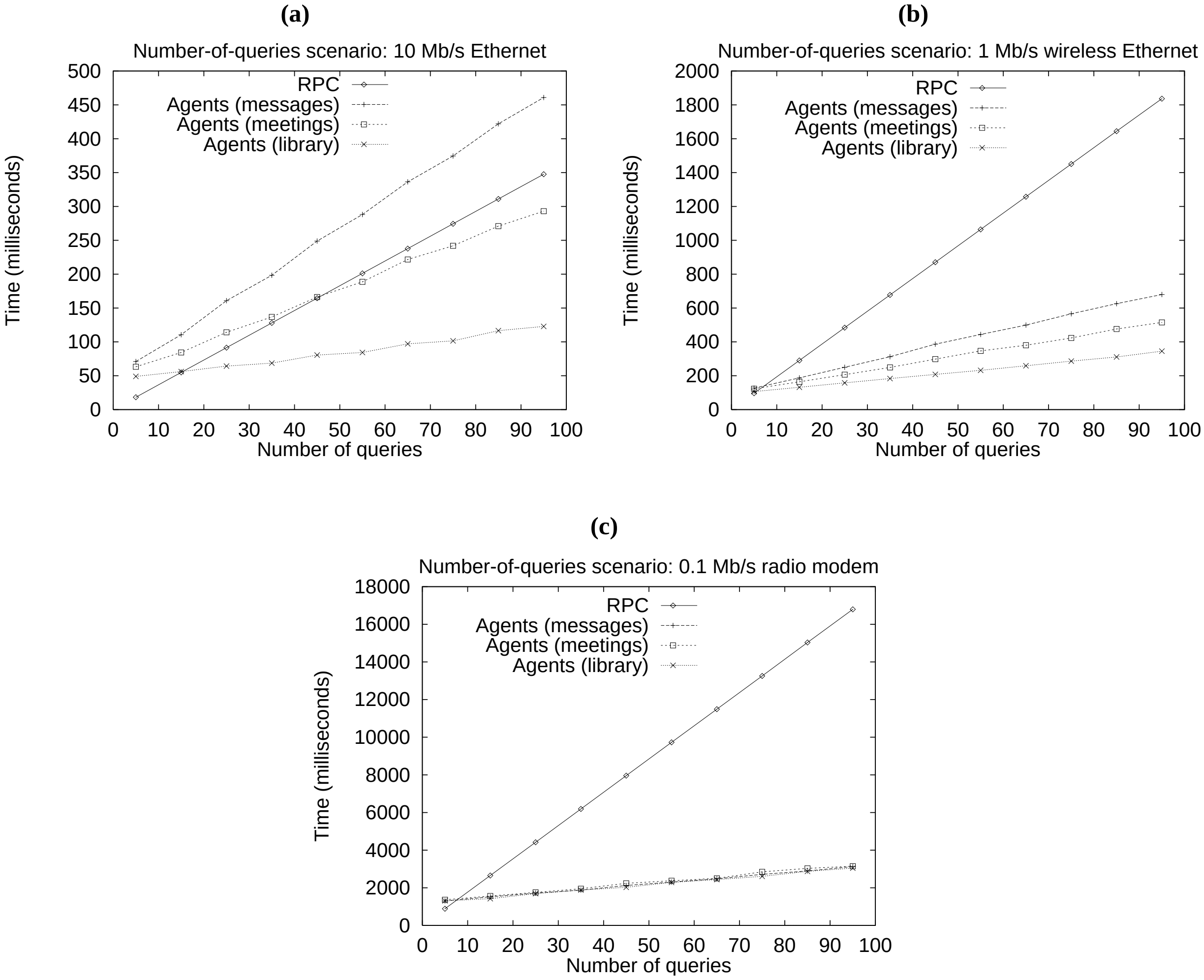


Figure 5: Total completion time when an agent performs a sequence of queries to get a final list of relevant documents. In the agent case, the application sends an agent to the location of the document collection. The agent performs the query and sends back only the final list of relevant documents. In the RPC case, the application performs the query operations from across the network, transferring intermediate results at each step. Results are shown for three different network environments: **(a)** 10 Mb/s Ethernet, **(b)** 1 Mb/s wireless Ethernet, and **(c)** 0.1 Mb/s radio modem.

In the RoamAbout and Richochet cases, we disconnected all other RoamAbout and Richochet cards in the building to ensure a "dedicated" network link. In addition, the two laptops were placed next to each other to minimize any packet loss due to environmental noise or obstructions. As before, such a network environment would not occur in real life. As the figures clearly show, however, agent performance relative to RPC improves dramatically as the speed of the network decreases, simply because (in most cases) the agent transmits a smaller number of bytes than the RPC approach, and transmits these bytes in a smaller number of distinct transmission steps.[11] As bandwidth drops or latency increases, the reduced network usage leads to better relative completion times. This is true no matter whether the bandwidth or latency penalties arise from a change in network hardware, an increase in network contention, or an increase in network distance between the machines. In other words, by using a contention-free local network, we create the *worst* case for agent performance.

The *RPC* curve in each of the three figures shows the time needed for the traditional client to run the queries from across the network. As before, the query size was 256 bytes, the result size for *each* query was 2048 bytes, and the server did not actually perform the query, but instead sent back a dummy result. Sending back a dummy result allows us to run more iterations of each test within the same amount of time. Moreover, it will not change the *differences* between the completion times shown in the figures since the same query backend would be used in all cases. In other words, we have removed exactly that portion of the code that is identical in all cases, namely, the shared library that actually runs the queries through the vector-space retrieval system.

The remaining three curves in Figures 5a, 5b, and 5c show the agent results. As before, the size of each agent was $1024 + 256 + 128n$ bytes (where n is the number of queries), and the final result size was 2048 bytes. Each of the three curves corresponds to a different way of interacting with the document database. In the first curve, *Agents (messages)*, the agent used D'Agent *messages* to communicate with a stationary agent that serves as a wrapper around the document database. In the second curve, *Agents (meetings)*, the agent first established a direct inter-process connection[12] or *meeting* with the stationary agent; the queries and results were sent across the meeting. Finally, in the third curve, *Agents (library)*, the agent loaded the interface library itself and simply invoked the query procedures directly.

Figures 5b and 5c show that, for the 1 Mb/s wireless Ethernet and 100 Kb/s radio-modem networks, the agent finishes faster than the RPC client as long as the query involves more than six or seven operations. The time savings come entirely from the fact that the agents do not transfer any intermediate results across the network. In fact, the time savings are large enough that the agent's choice of communication strategy once it is on the database machine (messages, meetings or a builtin library) does not have a significant effect.

On the other hand, Figure 5 shows that, for the 10 Mb/s wired Ethernet, the agent's choice of communication strategy leads to significantly different results. When the agent loads the interface library itself and invokes the query procedures directly, it completes the query faster than the RPC solution if the query requires more than fifteen operations, and does significantly better as the number of operations increases further. If the migration overhead is reduced, it should be competitive for even just five queries.[13] On the other hand, when the agent interacts with a stationary agent using *meetings*, it does not outperform the RPC client unless the query involves more than forty-five operations, and when it interacts with the stationary agent using *messages*, it never outperforms the RPC client.

There is one primary reason why the agent performs worse than the RPC client in the 10 Mb/s network. The overhead of inter-agent communication is large even when the agents are *on the same machine.* When an agent sends a message to another local agent, the message first is sent to the server process (over a pipe), and then sent to the interpreter process that is executing the recipient agent (over another pipe). The response message follows the reverse path back to the sending agent. Thus, each message is sent twice, from agent to

[11] A transmission step here is a request to the remote service followed by the response.

[12] In the current implementation, this connection is a Unix-domain Berkeley socket, which is not the most efficient connection possible, but is easy to port from one version of Unix to another.

[13] The good performance of loading the interface library directly suggests that a useful abstraction will be services that appear to be stationary agents, but, in fact, are provided through libraries. D'Agents includes an RPC-like mechanism, which allows agents to invoke each other's procedures. This mechanism would provide a natural way of making a library appear as a stationary agent, since the client agent could make the same procedure invocations in both cases; only the hidden implementation of the stubs would be different.

server, and then from server to agent. The overhead of the double transmission is larger than the overhead of making RPC calls across the (relatively) fast network link. This can be seen clearly in the *Agents (messages)* line of Figure 5b, which has a larger slope than the *RPC* line. The overhead also affects meetings, since establishing a meeting requires the exchange of two messages, a meeting request and a meeting acceptance. One possible solution is to allow direct inter-process communication between agents, even when the agents have not established a meeting. For example, each interpreter process could have a Unix domain socket for accepting messages from other local agents. Since the agents are local to each other, the security concerns are significantly less than if each process has a *network* socket for accepting messages from *remote* agents.

It is also important to note that not all mobile-agent systems suffer from the same messaging overhead when the agents are on the same machine. For example, both Ara [PS97] and Voyager [OBJ97] run each agent as a separate thread inside a *single* process, and can simply transfer a message data structure from the sending agent's thread to the recipient agent's thread. In systems such as Ara and Voyager, we would expect performance near that of the direct-invocation approach (i..e, the directly loaded interface library) for all communication options.

Performance when the searcher examines the document texts. Figure 6 shows the completion times for the scenario where the searcher performs a single query, and then examines the document texts to decide which ones are actually relevant. The bandwidth usage for this same scenario was shown in Figure 3b. As with the multiple-query scenario, aside from performing the experiments in each of three different network environments, the experimental conditions were identical to those of Figure 3b.

The results are similar to those for the multiple-query scenario. The agent does better than a traditional client in the two slower networks. In the fastest network, however, the agent does slightly worse than the traditional client when none of the documents turn out to be relevant, and does significantly worse when twenty percent of the documents turn out to be relevant. All of the implementation issues considered above are influencing these results. Three factors have the most impact. First, the time to read the document files from disk, which must be done in both the agent and client cases, takes nearly half of the total time (even though the document files were in the file cache for all but the first run). Second, Tcl is slow enough that it takes nearly as long to perform the substring search with a Tcl agent as to send the entire document text across our (relatively) fast network link. Third, and most importantly, the inefficiencies in the inter-agent communication mechanisms hurt the overall performance more and more as the number of relevant documents increases. In the worst case, 80 kilobytes of document text are sent inside an agent message. Clearly, it is necessary to provide a more efficient mechanism for "streaming" data from an agent on one machine to an agent on another. At the same time, it is worthwhile to note that if the client/server application required a separate network connection for each downloaded document (as with some Web servers), the agent solution would perform far better than the downloading solution [RGK97].

Summary. The experimental results in this section show that the technical-report agent and other similar retrieval agents generally will perform better than the corresponding client/server solution in slow networks, but worse in a relatively fast network (unless the agent performs a large number of separate retrieval operations). Other recent studies, such as [PRS99], have produced the same basic results. For D'Agents, there are several reasons for the poor performance in faster networks: (1) inter-agent communication across machines is slower than RPC, (2) inter-agent communication on the same machine is slower than RPC across a fast network, and (3) the agents are written in Tcl. At the same time, even in the faster network, the agent solution does outperform the client/server solution in several cases, particularly when the document collections provide their search operations as a loadable library. Moreover, the performance bottlenecks in D'Agents are easy to identify, and several solutions exist.

One key area of future work is to experimentally compare the agent and traditional client/server solutions when multiple agents are searching the database at the same time. Of course, our current agents are written in the scripting language Tcl, so the load on the server will be significantly higher than if the server had provided a high-level operation that directly supported the client's task. The situation, however, is not as bad as it first appears. For example, in the technical-report application, a significant portion of the server's time is spent doing the vector-space calculations that determine which documents are relevant to the given keyword query. These calculations are done regardless of whether we adopt an agent or traditional client/server solution. Thus, the increase in server load is only the time needed to execute the Tcl wrapper

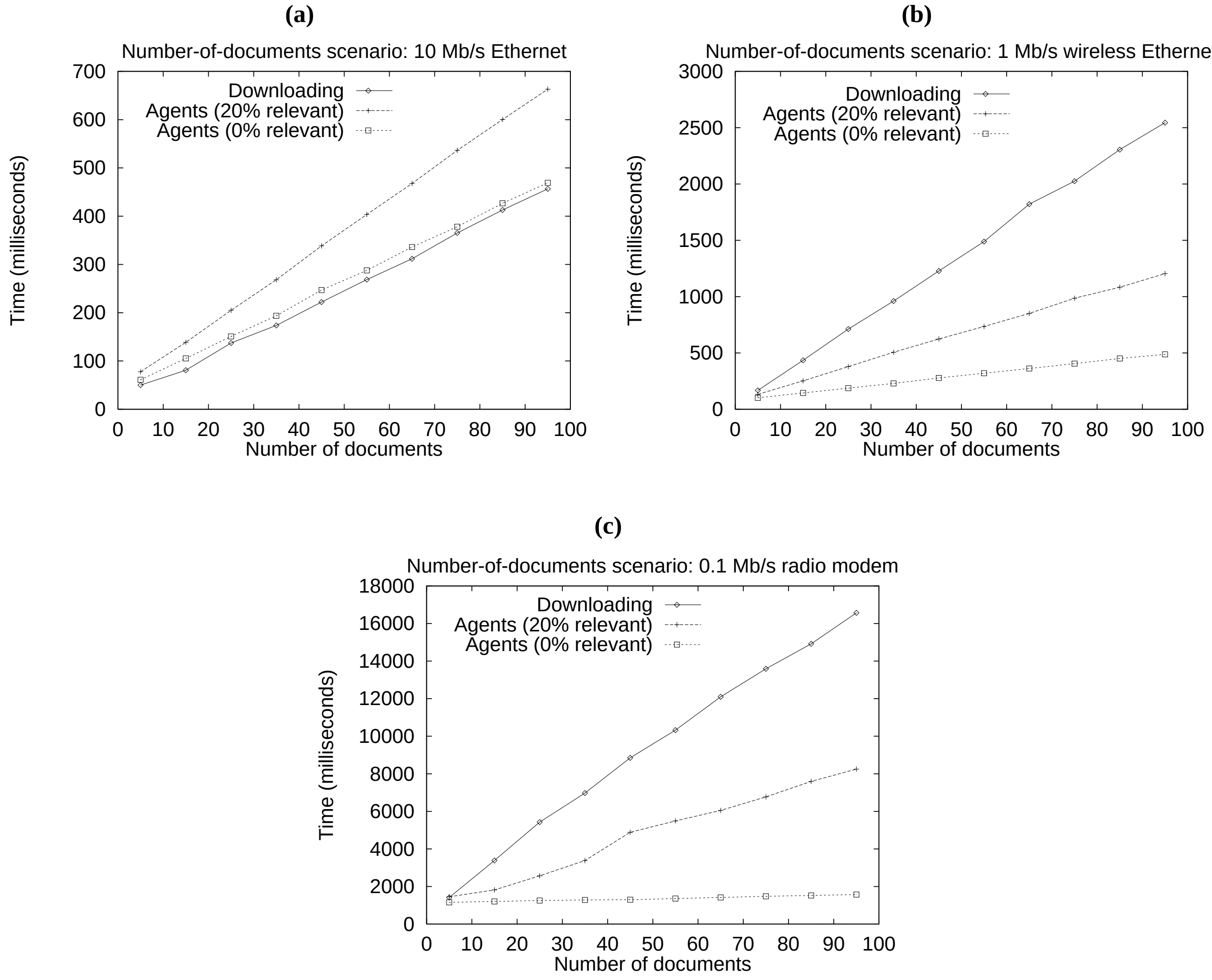


Figure 6: Total completion time when an application makes an initial query and then inspects the texts of the documents to determine which ones are actually relevant. In the agent case, the application sends an agent to the location of the document collection. The agent performs the initial query, inspects the document texts, and sends back only the relevant documents. In the downloading case, the application makes the same initial query, but then downloads the texts of *all* the retrieved documents so that it can inspect the texts locally. Results are shown for three different network environments: **(a)** 10 Mb/s Ethernet, **(b)** 1 Mb/s wireless Ethernet, and **(c)** 0.1 Mb/s radio modem.

code that invokes the vector-space functions and makes some simple decisions after each invocation, e.g., whether or not to do another keyword query to get more documents. Moreover, Tcl is at the slow end of the spectrum of agent languages. In fact, if the agent is written in a language such as Java, it is possible to securely execute the agent code less than twice as slowly as the corresponding natively-compiled code (through on-the-fly compilation and other techniques) [LSW95]. If such a capability were available in our agent system, a server that accepted agents would see a load only a small factor higher than a server that provided a high-level (native) operation for each distinct client task. In many cases, this would be a small price to pay for (1) the increased flexibility that the service provider could offer to its clients and (2) the reduced programming burden on the service developers.

2.2.3 Reduction in latency

The primary goal in the two information-retrieval applications above is to reduce the total completion time, namely, the time from when the user submits her query to when she receives the results. In the traditional client-server approach, the time that it takes each RPC call to actually "cross the wire" does not matter, since this latency is small relative to the processing time for the query and the transmission time for the query results. In other words, the agent approach is advantageous because it eliminates the transmission of intermediate results, not because it eliminates the invocation latencies.

There are applications, however, in which the requests, the results, and the amount of processing per request are all small. In such cases, the invocation latencies can dominate the overall completion time. The best example of such an application is one that must react quickly to some event by sending out new status or control information. In such cases, if the reactive component is implemented as a mobile agent, it can move closer to the producer of the events, the consumer of the status or control information, or both. In addition, the agent can move again if the set of producers and consumers, or the current network conditions, change, continually placing itself at the best network location. Thus, the status and control information will reach the consumers much faster than if the reactive component was installed at some fixed network location. In situations where one administrator has access to all the machines (e.g., an Intranet application) and can pre-install the reactive component at every network location, the mobile agent approach is still attractive. It eliminates the need for pre-installation, and provides a seamless way to transfer the reactive control from one location to another. The agent simply moves.

The simplest example is any application that needs to interact with a remote user through a graphical user interface. By sending even just its interface component to the user's location, the application can respond much more rapidly to user actions. In fact, faster interaction with a user is one of the main motivations behind Java applets, which are programs that are embedded inside World Wide Web (WWW) pages [CH97]. When a user accesses the page through their browser, the applet is downloaded and executed on their machine, where it typically presents some complex interface for interacting with back-end databases and other Web resources.

A more complex example is the chat server that was implemented on top of the Sumatra mobile-agent system [RASS97, RAS96]. A chat server simply allows multiple users to "chat" online; each user types whatever they want, and what they type is displayed on the screens of all the other users. The Sumatra chat server is written as a mobile Java agent. It monitors network latencies, and continually moves to the network position that minimizes the average latency between itself and its clients. When the Sumatra chat server and a corresponding fixed-location server were tested in a local-area network (using Internet traces to introduce Internet-sized transmission delays), the maximum latency between the moving Sumatra server and any of its clients was two to four times smaller than between the fixed-location server and its clients [RASS97].

2.2.4 Disconnected operation and mobile computing

A mobile computer often disconnects from the network, and when it reconnects, it might find itself with a radically different network connection in terms of bandwidth, latency or reliability. The simple ability of agents to migrate to the other side of a network link is a critical advantage, since the agent can avoid extensive use of a low-bandwidth or high-latency link, and can continue its task even if the link goes down. In fully wireless networks where the computers are continuously moving relative to each other, such as in the counter-terrorism application above, mobile agents are even more advantageous. The network connections appear and disappear extremely rapidly, and when a computer does have a connection, it can not count on that connection remaining up for any length of any time. Encapsulating the complete task as a mobile agent and moving it off of (or onto) the mobile computer as a single unit is a better choice than trying to perform the task one step at a time during brief periods of connection.

In other words, traditional client-server network applications either require a continuous connection between the client and server, or at best, are forced to stall between steps while the connection is down. Moreover, the applications can not easily adapt to network links of different bandwidths and latencies. For example, when faced with a lower bandwidth link, a traditional client can not send its own compression algorithm to some proxy site (to compress the stream of data coming back from a server). A mobile agent, on the other hand, moves some of the client code to a server or proxy machine, or moves some of the server code to the client. The client and server then conduct their dialog through their mobile-agent representatives. When the network link goes down, the dialog can continue, and the agent can return just the final result when the link goes back up. When the connection is up, but of unsuitably low-bandwidth, an agent can filter and compress the data stream going from the server to the client (or vice versa). In addition, an agent can carry interface code onto the client so that it can interact in a timely fashion with a user, or can carry time-critical control code onto a server.

The ability of a mobile agent to operate disconnected from the client machine is a key feature in both of the motivating applications above, In the technical-report application, suppose the user enters the free-text query on a laptop that has a transient network connection. The application passes the query to a mobile agent, which leaves the laptop and moves to a proxy site. The agent then decides whether to search the relevant repositories sequentially, by migrating through them in order, or in parallel, by spawning one child agent per site. The search process can last some time, especially if the query is imprecise and relevance feedback (from the program, not the user) is necessary to refine the query after an initial failure. Because the agent can operate disconnected from the client machine, neither the user nor her machine need to be available during the search.

In the counter-terrorism application, the soldiers in the field launched queries in exactly the same manner. The query agents left the mobile computers to return later with just the final results. Moreover, any query in the counter-terrorism application could be marked as *persistent*. A persistent query is a query that remains in effect for a specified period of time, and is applied to each new document that is inserted into some dynamically changing collection of documents. For example, a query about the country in which the soldiers are stationed might be applied to every news article coming across a military news feed. If an incoming news article matches the query, it is sent to the soldier who launched the query. Mobile agents are a natural way to implement persistent queries. An agent is dispatched to the location of the information source to monitor its contents in some application-specific way. The agent rejects irrelevant incoming documents on its own, and queues up just the relevant documents for transmission to the client. Since the duration of the monitoring task is likely to be long, the ability to operate disconnected from the client machine makes the persistent query much more efficient. Many irrelevant documents can be rejected while the client machine is not even connected. Of course, the information repository must provide some notification mechanism so that agents will only poll the repository if some interesting change *might* have occurred. Polling continuously would be too inefficient in most cases.

2.2.5 Load balancing

Load-balancing problems arise in a distributed computing system when there is an unequal work allocation at the different processing elements in the system. Performance typically improves if the work can be equitably distributed among the elements. Load balancing is a large and active area of research, and many load-balancing techniques exist, each with the goal of improving performance by partitioning a distributed application into components and distributing those components across multiple processors [SHK95].

Load balancing can be done either statically or dynamically. Load balancing is static if the work allocation is based on an *a priori* analysis of the computation or input dataset. Once the work is allocated, typically before execution begins, it is not reallocated during the execution life cycle of the computation. Most load balancing done in parallel and distributed computing today is static. Load balancing is dynamic if the distributed workload is reallocated during execution.

Mobile agents easily support dynamic load balancing with three key features: they can move from one platform to another, they can move across *heterogeneous* platforms, and they carry all application-specific code with them, rather than requiring pre-installation of that code on the destination machine. In fact, mobile agents support the most general and effective form of dynamic load balancing: executing processes, implemented as mobile agents, can migrate under external or internal control to any agent-capable computing node within the distributed system.

Under the mobile-agent approach, the components of a distributed computation that are candidates for load balancing (and hence migration) are "wrapped" as mobile agents. Machines that are potential targets for migration run an agent server, just as all load-balancing systems require pre-installation of some appropriate infrastructure. The mobile agents can start execution anywhere within the network, and in response to control signals based on measurements of the distributed load, migrate to new machines. Since mobile-agent migration allows runtime determination of the target machine, the target machine can be determined an arbitrary, external load-balancing module. Many different dynamic load-balancing control algorithms have been proposed [SHK95], and any of these algorithms could be used as the control logic inside the load-balancing module, allowing the agent's code to be independent of the actual load-balancing scheme.

Most existing load-balancing systems are not as flexible as mobile agents, since they lack one of the three key features. They do not allow an executing process to move (using data migration instead), they support only homogeneous platforms, or they require that application code be pre-installed on each machine. On the other hand, they typically allow the migration of compiled code, which no current mobile-agent systems support. Notable recent examples include Sprite [DO91] and Condor [LS92], both of which allow runtime migration of an entire process within a cluster of homogeneous machines; Emerald [JLHB88, SJ95], which supports object migration within a *heterogeneous* cluster, if the object code is pre-installed; Legion [GWtLT97], which provides runtime migration within a *heterogeneous* cluster, again if the code is pre-installed; and several Java-based systems [AAB98, RN98], which distribute parallel program fragments submitted by one programmer to machines volunteered by others, but do not migrate a program once it starts execution.

Since current mobile-agent systems use interpreted languages rather than compiled code, the decision to use agents for load balancing must take into account not only the cost of agent migration itself, which was discussed in Section 2.2.2, but also the speed penalties due to interpretive execution. Since agents are interpreted, a compute-intensive application will see a significant execution slowdown solely because it is implemented as a mobile agent, rather than as a "traditional" program compiled into native code. For this reason, mobile-agent systems are not yet suitable for applications in which load balancing is the primary goal, i.e., applications in which CPU time is the only resource of interest. If mobile agents are used for other reasons, however, the agent load can be balanced easily by having the agents migrate from machine to machine. For example, in the technical-report application, where we use agents to conserve bandwidth and handle disconnected client machines, we can easily direct each agent to the most lightly loaded copy of a replicated document collection, or have an agent move from one proxy site to another if the initial proxy site becomes overloaded. Moreover, the next generation of mobile-agent systems, which use the newest Java virtual machines with just-in-time compilation, will be suitable for general load balancing.

For these reasons, load balancing is a significant growth area for mobile-agent technology. Increased interest in deploying continuously running simulations and information-processing applications will require system support for dynamic load balancing that cannot be based on prior installation of the application code at all target computing sites. In addition, the network and available computing elements will change during the course of such long running computations, so maximal flexibility in dynamically distributing and reallocating pieces of such applications will be essential. The powerful code-migration infrastructure in mobile-agent systems provides such flexibility. In fact, we expect that mobile-agent and load-balancing systems will become more and more similar in the future, with optimized load-balancing algorithms and highly efficient migration techniques coming out of the load-balancing world, and just-in-time compilation for heterogeneous platforms and extremely flexible component re-deployment coming out of the mobile-code and mobile-agent worlds.

2.2.6 Dynamic deployment

As discussed above, if a traditional server does not directly support a desired client operation, the client must make multiple RPC calls into the server, transmitting intermediate results across the network on every call. Mobile agents allow an application to avoid this transmission—a new "server" operation can be implemented as a mobile agent and sent from the client to server machine, where it executes locally to the server resources and returns just the final result to the client. The agent might perform the operation just once, terminating as soon as it has finished, or it might sit in a loop, performing the operation again and again in response to further client requests. Similarly, an application that needs to interact with a user, such as the picture-viewing component of the counter-terrorism application, can send a mobile agent to the user's machine if and when interaction is required. In addition, a mobile-agent "server", such as the Sumatra chat server, can continually re-deploy itself to the network location that has the minimum average latency to its current clients.

Such uses of mobile agents represent specific examples of *dynamic deployment*, where an application dynamically installs a software component on some remote machine, and then invokes that software component as if it were part of the remote machine's pre-installed services. In the mobile-agent case, the agent essentially installs and invokes itself, since it simply migrates to the remote machine and continues its execution there.

Many other approaches to dynamic deployment exist. Common approaches such as enterprise-wide software management tools (such as Tivoli's IT Director [tiv99]) and relatively classical UNIX systems management (such as FTP-ing code to a remote machine and then running an rshell script to install and invoke it) are meant for a system administrator who needs to install an entire software package, and are not lightweight enough for a client program that simply wants to extend a server's functionality with some single operation. Other approaches are suitable for client programs, however. For example, most stored-procedure systems, particularly those inside SQL database servers, allow a client to upload code for later invocation—the code is stored at the server and can be invoked as many times as desired. On the other hand, Remote Evaluation (REV) sends the procedure code as part of every invocation [SG90]. Dynamic classes, which are an object-oriented variation of stored procedures and REV, allow a new subclass of a pre-known superclass to be dynamically inserted into an executing C++ program [HG98]. In this case, the client sends an insertion instruction to the remote C++ program; the instruction includes the location from which the program can download a shared library containing the subclass implementation.

The drawback of all these approaches is that it is often difficult (or impossible) for a dynamically-deployed component to deploy *subcomponents*, or for the component to re-deploy itself. Mobile agents, on the other hand, handle both situations with ease. In the technical-report application, for example, the application front-end sends merging and filtering code to a proxy site, which, in turn, sends query code to the database locations. In more complicated retrieval tasks, an application might need to query multiple databases sequentially, i.e., the query made against one database might depend on the results from a previous database. In this case, the query agent can migrate sequentially through the database locations, and the proxy agent can re-deploy itself to a new proxy site that is closer to the databases involved in the current subtask. In other applications, an agent might re-deploy itself according to completely different criteria, such as current network latencies and bandwidths, current machine loads, or the expected probability that a network link will go down in the near future. In all cases, an agent can deploy itself to a machine that has no previous knowledge of the agent's task or its encompassing application. This is particularly important for mobile computing, where the best proxy site or database replica might be completely different depending on the mobile computer's current location.

In short, as with load balancing, mobile agents support the most general form of dynamic deployment, where an application can distribute its components "on-the-fly" to arbitrary network sites, and those components can move at will from one site to another as conditions change. In fact, although the true strength of mobile agents is their *combination* of individual strengths, it can be argued that flexible dynamic deployment is what makes mobile agents such an effective choice for distributed applications. Mobile agents conserve bandwidth, reduce latencies and completion times, handle disconnected operation, and balance load by deploying and redeploying themselves to more attractive network locations. Moreover, any existing system that is extended to allow such arbitrary code movement (as well as communication between the moving components) will in effect become a mobile-agent system, facing the same implementation issues and applicable to the same application domains.

2.3 The big picture

Despite the six strengths above, any specific application can be realized just as efficiently without mobile agents. For example, consider the technical-report application, where a user with a mobile computer launches a query against several remote document collections. Suppose first that no form of mobile code is available. To minimize the data transferred from the document collection to the application, the server developer writes and installs a high-level query operation that matches the client's needs exactly. Similarly, to minimize the data transferred across the low-bandwidth link to the mobile computer, the application developer writes an application-specific proxy to merge and filter the results from multiple collections. Further, the developer installs the proxy at multiple sites, since the mobile computer will be in different places at different times. Finally, since the network to the mobile computer might go up and down, the developer uses queued RPC [JTK97] for communication between the application front-end and the proxies.[14]

The approach outlined above, however, involves significantly more implementation work and administrative overhead than the mobile-agent solution, where a single mobile agent moves to the dynamically selected proxy site, and sends out child agents as needed. There are other forms of mobile code that could be used instead of mobile agents, such as remote evaluation (REV) [SG90], SUPRA-RPC (SUbprogram PaRAmeters in Remote Procedure Calls) [Sto94], and Java servlets [Cha96].[15] Typically, systems that support these forms of mobile code allow "one-hop" migration only. A program is sent to and executed on a target machine, but that program can not send out sub-programs. Some systems do allow a program to send out sub-programs, but this leads to a tree-like control structure where the results flow up the tree back to the spawner of the original program. Such a control structure works well for the technical-report searcher, but is inappropriate for the Sumatra chat server [RASS97], which does not send out subprograms, but instead needs to move itself from one machine to another.

In short, different applications require different combinations of traditional techniques, and the techniques that work for one application might not work for another. Mobile agents, on the other hand, support flexible and completely dynamic code deployment to arbitrary network sites, and thus provide a single, general framework in which a wide range of applications can be implemented easily, efficiently and robustly. Most importantly, the applications can exhibit extremely flexible behavior in the face of changing network conditions, simply by re-locating one or more of their component mobile agents.

3 The State-of-the-art

The current and future potential of mobile agents has led to a flurry of recent implementation work. In this section, we first examine ten representative mobile-agent systems, and then briefly discuss their similarities and differences.

3.1 Representative mobile-agent systems

Mobile-agent systems can be categorized in several ways. Here we categorize them according to the programming languages that they support. Some systems allow agents to be written in multiple languages; many allow agents to be written in only Java, which is the most popular agent language; and others allow agents to be written in some single language other than Java.

[14] Queued RPC queues RPC requests and responses for later transmission, if the network link between the client and server has gone down temporarily.

[15] Java servlets are the opposite of Java applets. A servlet is a Java program uploaded from a client to a Web server, and then executed inside the Web server.

3.1.1 Multiple-language systems

A few systems attempt to support mobile agents outside of the context of any specific programming language, and indeed support more than one programming language.

Ara. Ara [PS97, Pei98] supports agents written in Tcl, Java and C/C++. The C/C++ agents are compiled into an efficient interpreted bytecode called MACE; this bytecode, rather than the C/C++ code itself, is sent from machine to machine. For all three languages, Tcl, Java and MACE, Ara provides a *go* instruction, which automatically captures the complete state of the agent, transfers the state to the target machine, and resumes agent execution at the exact point of the *go*. Ara also allows the agent to *checkpoint* its current internal state at any time during its execution. Unlike other multiple-language systems, the entire Ara system is multi-threaded; the agent server and the Tcl, Java and MACE interpreters run inside a single Unix process. Although this approach complicates the implementation, it has significant performance advantages, since there is little interpreter startup or communication overhead. When a new agent arrives, it simply begins execution in a new thread, and when one agent wants to communicate with another, it simply transfers the message structure to the target agent, rather than having to use inter-process communication. Nearly all Java-only systems are also multi-threaded, and see the same performance advantages.

The Ara group is finishing implementation work on their initial security mechanisms [Pei98]. An agent's code is cryptographically signed by its manufacturer (programmer); its arguments and its overall resource allowance are signed by its owner (user). Each machine has one or more virtual places, which are created by agents and have agent-specified admission functions. A migrating agent must enter a particular place. When it enters the place, the admission function rejects the agent or assigns it a set of allowances based on its cryptographic credentials. These allowances, which include such things as filesystem access and total memory, are then enforced in simple wrappers around resource-access functions.

D'Agents. Our own mobile-agent system D'Agents [GKCR98], which was once known as Agent Tcl, supports agents written in Tcl, Java and Scheme, as well as *stationary* agents written in C and C++. Like Ara, D'Agents provides a *go* instruction for each language, and automatically captures and restores the complete state of a migrating agent. Unlike Ara, only the D'Agent server is multi-threaded; each agent is executed in a separate process, which simplifies the implementation considerably, but adds the overhead of inter-process communication. The D'Agent server uses public-key cryptography to authenticate the identity of an incoming agent's owner; stationary *resource-manager* agents assign access rights to the agent based on the authentication and the administrator's preferences; and language-specific enforcement modules enforce the access rights, either preventing a violation from occurring (e.g., filesystem access) or terminating the agent when a violation occurs (e.g., total CPU time exceeded). Each resource manager is associated with a specific resource such as the filesystem. The resource managers can be as complex as desired, but the default managers simply associate a list of access rights with each owner. Unlike Ara, most resource managers are not consulted when the agent arrives, but instead only when the agent (1) attempts to access the corresponding resource or (2) explicitly requests a specific access right. At that point, however, the resource manager forwards all relevant access rights to the enforcement module, and D'Agents behaves in the same way as Ara, enforcing the access rights in short wrapper functions around the resource-access functions.

Current work on D'Agents falls into three broad categories: (1) scalability, (2) market-based resource control, where agents are given a finite supply of currency from their owner's own finite supply and must spend the currency to access needed resources [BKR98]; and (3) support for mobile-computing environments, where applications must deal with low-bandwidth, high-latency and unreliable network links [KGN$^+$98].

D'Agents has been used in several information-retrieval applications, including the technical-report searcher and 3DBase [CBC97], a system for retrieving three-dimensional drawings (CAD drawings) of mechanical parts based on their similarity to a query drawing.

Tacoma. Tacoma [JSvR98a, JSvR98b] supports agents written in C, C++, ML, Perl, Python and several other languages. Unlike Ara and D'Agents, Tacoma does not provide automatic state-capture facilities.

Instead, when an agent wants to migrate to a new machine, it creates a *folder* into which it packs its code and any desired state information. The folder is sent to the new machine, which starts up the necessary execution environment and then calls a known entry point within the agent's code to resume agent execution. Although this approach places the burden of state capture squarely onto the *agent* programmer, it also allows the rapid integration of new languages into the Tacoma system, since existing interpreters and virtual machines can be used without modification. Tacoma is used most notably in StormCast, which is a distributed weather-monitoring system, and the Tacoma Image Server, which is a retrieval system for satellite images [JSvR98b]. Both applications involve large amounts of data and rapidly changing client needs. Rather than downloading all the data for local processing, client applications implement the desired high-level search operations as mobile agents, and then send these agents to the data sites.

The public versions of Tacoma rely on the underlying operating system for security, but do provide hooks for adding a cryptographic authentication subsystem so that agents from untrusted parties can be rejected outright. The Tacoma group is adding enforcement facilities roughly equivalent to those of Ara and D'Agents. In addition, they have explored several interesting fault-tolerance and security mechanisms, such as (1) using cooperating agents to search replicated databases in parallel and then securely vote on a final result [MvRSS96], and (2) using security automata (state machines) to specify a machine's security policy and then using the automata and software fault isolation to enforce the policy [Sch97].

3.1.2 Java-based systems

Java is an increasingly popular language for developing distributed applications. Its common implementation as a virtual machine allows Java programs to run on a wide variety of platforms. Furthermore, it has built-in support for serialization, security, remote method invocation, and dynamic class loading. All of these factors make Java a natural choice for mobile agents. Several systems have been developed with Java specifically in mind [WPM99].

Aglets. Aglets [LO98] was one of the first Java-based systems. Like all commercial systems, including Concordia, Jumping Beans, and Voyager below, Aglets does not capture an agent's thread (or control) state during migration, since thread capture requires modifications to the standard Java virtual machine. In other words, thread capture means that the system could be used only with one specific virtual machine, significantly reducing market acceptance.[16] Thus, rather than providing the *go* primitive of D'Agents and Ara, Aglets and the other commercial systems instead use variants of the Tacoma model, where agent execution is restarted from a known entry point after each migration. In particular, Aglets uses an event-driven model. When an agent wants to migrate, it calls the *dispatch* method. The Aglets system calls the agent's *onDispatching* method, which performs application-specific cleanup, kills the agent's threads, serializes the agent's code and object state, and sends the code and object state to the new machine. On the new machine, the system calls the agent's *onArrival* method, which performs application-specific initialization, and then calls the agent's *run* method to restart agent execution.

Aglets includes a simple persistence facility, which allows an agent to write its code and object state to secondary storage and temporarily "deactivate" itself; proxies, which act as representatives for Aglets, and among other things, provide location transparency; a lookup service for finding moving Aglets; and a range of message-passing facilities for inter-agent communication. The Aglet security model is similar to both the D'Agent and Ara security models, and to the security models for the other Java-based systems below. An Aglet has both an owner and a manufacturer. When the agent enters a context (i.e., a virtual place) on a particular machine, the context assigns a set of permissions to the agent based on its authenticated owner and manufacturer. These permissions are enforced with standard Java security mechanisms, such as a custom security manager.

[16] D'Agents, which does use a modified Java virtual machine to capture thread state, is a research system and is not under such market constraints.

Concordia. Concordia [WPW$^+$97, WPW98] is a Java-based mobile-agent system that has a strong focus on security and reliability. Like most other Java-based systems, it moves an agent's object code and data, but not thread state, from one machine to another. Like many other systems, Concordia agents are bundled with an *itinerary* of places to visit, which can be adjusted by the agent while en route.[17] Agents, events, and messages can be queued if the remote site is not currently reachable. Agents are carefully saved to a persistent store, before departing a site and after arriving at a new site, to avoid agent loss in the event of a machine crash. Agents are protected from tampering through encryption while they are in transmission or stored on disk; agent hosts are protected from malicious agents through cryptographic authentication of the agent's owner, and access control lists that guard each resource.

Jumping Beans. Jumping Beans [AA98] is another Java-based framework for mobile agents. Computers wishing to host mobile agents run a Jumping Beans *agency,* which is associated with some Jumping Beans *domain.* Each domain has a central server, which authenticates the agencies joining the domain. Mobile agents move from agency to agency within the domain, and agents can send messages to other agents in the domain. Both mechanisms are implemented by passing through the domain server. Thus the server becomes a central point for tracking, managing, and authenticating agents. It also becomes a central point of failure or a performance bottleneck, although the company plans to develop scalable servers to run on parallel machines. Another approach to scalability is to create many small domains, each with its own server. In the current version, agents cannot migrate between domains, but the company plans to support cross-domain migration in future versions. Security and reliability are key aspects of Jumping Beans. Public-key cryptography is used to authenticate agencies to the server and *vice versa*, and access-control lists are used to control an agent's access to resources, based on the permissions given to the agent's owner.

Although the company claims that Jumping Beans moves all agent code, data, and state, it is not clear from their documentation whether they actually move thread state as in D'Agents. Jumping Beans requires that the agent be a serializable object, so it seems likely that it implements the weaker form of mobility common to other Java-based agent systems.

Voyager. Voyager [OBJ97] is a Java-based mobile-agent environment integrated with CORBA. It provides a convenient way to interact, somewhat transparently, with remote objects (via proxy objects called "virtual" references), and for objects to move from host to host. When an object moves, it leaves behind a forwarder object that redirects any messages to the new location. "Agent" objects, unlike other objects, may move themselves autonomously by applying the moveTo() method on themselves. Voyager moves the code and data of the agent, but not thread state, to the new location and invokes the desired method there. Voyager also allows objects and agents to be made persistent through explicit saveNow() calls, supports group communication (multicast), and a provides federated directory service. Voyager provides the same basic security mechanisms as other Java-based systems [OBJ97].

3.1.3 Other systems

Finally, there are a few systems that are focused on a single agent-programming language other than Java. Indeed, all three systems have developed a new language.

Messengers. The Messenger project uses mobile code to build flexible distributed systems, not specifically mobile-agent systems [TDM$^+$94, DMTH95, Muh98]. In their system, computers run a minimal Messenger Operating System (MOS), which has just a few services. MOS can send and receive *messengers,* which are small packets of data and code written in their programming language $M\phi$. MOS can interpret $M\phi$ programs, which may access one of their two bulletin-board services: the *global dictionary*, which allows data exchange between messengers, and the *service dictionary*, which is a browse-able listing of messengers that offer services to other messengers. Ultimately, most services, including all distributed services, are offered

[17] Aglets calls the same method at each stop on the itinerary, while Jumping Beans, Concordia and Voyager all allow the agent to specify a different method for each stop.

by static and mobile messengers. In one case, messengers can carry native UNIX code, which is installed and executed on MOS; system calls are reflected back to the interpreted $M\phi$ code, allowing fast execution of critical routines, while maintaining the flexibility of mobile code [TMN97].

Obliq. Obliq [Car95, BN97] is an interpreted, lexically scoped, object-oriented language. An Obliq object is a collection of named fields that contain methods, aliases and values. An object can be created at a remote site, cloned onto a remote site, or migrated with a combination of cloning and redirection. Implementing mobile agents on top of these mobile objects is straightforward. An agent consists of a user-defined procedure that takes a *briefcase* as its argument; the briefcase contains the Obliq objects that the procedure needs to perform its task. The agent migrates by sending its procedure and current briefcase to the target machine, which invokes the procedure to resume agent execution. Visual Obliq [BC95] builds on top of Obliq's migration capabilities. Visual Obliq is an interactive application builder that includes (1) a visual programming environment for laying out graphical user interfaces, and (2) an agent server that allows Visual Obliq applications to migrate from machine to machine. When the application migrates, the state of its graphical interface is captured automatically, and recreated exactly on the new machine. Obliq does not address security issues. Visual Obliq does provide access control, namely, user-specified access checks associated with all "dangerous" Obliq commands, but does not have authentication or encryption mechanisms. Typically, therefore, the access checks simply ask the user whether the agent should be allowed to perform the given action.

Telescript. Telescript [Whi94b, Whi94a, Whi97], developed at General Magic, Inc., was the first commercial mobile-agent system. In Telescript, each network site runs a server that maintains one or more virtual *places*. An incoming agent specifies which of the places it wants to enter. The place authenticates the identity of the agent's owner by examining the agent's cryptographic credentials, and then assigns a set of access rights or *permits* to the agent. One permit, for example, might specify a maximum agent lifetime, while another might specify a maximum amount of disk usage. An agent that attempts to violate its permits is terminated immediately [Whi94b]. In addition to maintaining the places and enforcing the security constraints, the server continuously writes the internal state of executing agents to non-volatile store, so that the agents can be restored after a node failure.

A Telescript agent is written in an imperative, object-oriented language, which is similar to both Java and C++, and is compiled into bytecodes for a virtual machine that is part of each server. As in D'Agents, a Telescript agent migrates with the *go* instruction, which captures the agent's code, data and thread state. On its new machine, the agent continues execution from the statement immediately after the *go*. A Telescript agent can communicate with other agents in two ways: (1) it can *meet* with an agent that is in the same place—the two agents receive references to each other's objects and invoke each other's methods; and (2) it can *connect* to an object in a different place—the two agents pass objects along the connection. Despite the fact that Telescript remains one of the most secure, fault-tolerant and efficient mobile-agent systems, it has been withdrawn from the market, largely because it was overwhelmed by the rapid spread of Java.

3.2 Similarities and differences

All mobile-agent systems have the same general architecture: a server on each machine accepts incoming agents, and for each agent, starts an appropriate execution environment, loads the agent's state information into the environment, and resumes agent execution. Some systems, such as the Java-only systems above, have multi-threaded servers and run each agent in a *thread* of the server process itself; other systems have multi-process servers and run each agent in a separate interpreter process; and the rest use some combination of these two extremes. D'Agents, for example, has a multi-threaded server to increase efficiency, but separate interpreter processes to simplify implementation. Jumping Beans [AA98] is of particular note since it uses a centralized server architecture (in which agents must pass through a central server on their way from one machine to another), rather than a peer-to-peer server architecture (in which agents move directly from one machine to another). Although this centralized server easily can become a performance bottleneck, it greatly simplifies security, tracking, administration and other issues, perhaps increasing initial market acceptance.

Currently, for reasons of portability and security, nearly all mobile-agent systems either interpret their languages directly, or compile their languages into bytecodes and then interpret the bytecodes. Java, which is compiled into bytecodes for the Java virtual machine, is the most popular agent language, since (1) it is portable but reasonably efficient, (2) its existing security mechanisms allow the safe execution of untrusted code, and (3) it enjoys widespread market penetration. Java is used in all commercial systems and in several research systems. Due to the recognition that agents must execute at near-native speed to be competitive with traditional techniques in certain applications, however, several researchers are experimenting with "on-the-fly" compilation [LSW95, HMPP96]. The agent initially is compiled into bytecodes, but compiled into native code on each machine that it visits, either as soon as it arrives or while it is executing. The most recent Java virtual machines use on-the-fly compilation, and the Java-only mobile-agent systems, which are not tied to a specific virtual machine, can take immediate advantage of the execution speedup.

Mobile-agent systems generally provide one of two kinds of migration: (1) *strong mobility*, where the system captures an agent's object state, code and *control state*, allowing the agent to continue execution from the exact point at which it left off; and (2) *weak mobility*, where the system captures only the agent's object state and code, and calls a known entry point inside the code to restart the agent on the new machine [FPV98]. The strong-mobility model is more convenient for the end programmer, but more work for the system developer since routines to capture control state must be added to existing interpreters. All commercial Java-based systems use weak mobility, since market concerns demand that these systems run on top of unmodified Java virtual machines, and current Java virtual machines do not support the capture of control state. Research systems use both migration techniques.

Existing mobile-agent systems focus on protecting an *individual* machine against malicious agents. Typically the agent's owner or sending machine digitally signs the agent; the receiving machine verifies the signature, accepts or rejects the agent based on its signature, assigns access restrictions to the agent based on its signature and migration history, and then executes the agent in a secure execution environment that enforces the restrictions. Aside from encrypting an agent in transit and allowing an agent to authenticate the destination machine before migrating, most existing systems do not provide any protection for the agent or for a group of machines that is *not* under single administrative control.

Other differences exist among the mobile-agent systems, such as the granularity of their communication mechanisms, whether they are built on top of or can interact with CORBA, and whether they conform to emerging mobile-agent standards such as MASIF [MBB$^+$99] and FIPA.[18] Despite these differences, however, all of the systems discussed above (with the exception of Messengers, which is a lighter-weight mobile-agent system) are intended for the same applications, such as workflow, network management, and automated software installation. All of the systems are suitable for distributed information retrieval, and the decision of which one to use must be based on the desired implementation language, the needed level of security, and the needed performance. The commercial systems, for example, have notably higher performance than most research systems, simply due to the availability of more programmer man-hours (and the importance of performance in a commercial offering).

4 Conclusion

As the two information-retrieval applications in Section 2 illustrate, mobile agents have six strengths: conservation of bandwidth, reduction in latency, reduction in total completion time, support for disconnected operation, support for loading balancing, and support for dynamic deployment. Although none of these six strengths are unique to mobile agents, no competing technique shares all six. The true strength of mobile agents is that they are a general-purpose framework for implementing distributed applications.

At the same time, current mobile-agent systems only partially realize these six strengths. In particular, due to the migration and interpretive overhead in current systems, an agent will out-perform a traditional RPC solution (in terms of total completion time) only if it invokes several operations against the remote data source, particularly if the client and server machines are connected with a medium-speed or faster

[18] FIPA specifications are available at `http://http://drogo.cselt.stet.it/fipa/`.

network (such as 10 Mb/s Ethernet). In addition, current systems do not adequately protect an agent against malicious machines, which limits an agent's migration choices. Some agents will need to act like traditional clients and access a service from across the network, rather than migrating to the service location and opening themselves to attack.

Researchers are working on several promising solutions to these two problems of performance and security, however, and now share a clear and realizable goal: a mobile-agent system that (1) adequately protects an agent from malicious machines, (2) supports agent migration that is only a small factor slower than an RPC call that transmits an equivalent amount of data, and (3) allows agents to execute nearly as quickly as if they were compiled (directly) into native machine code.[19] Even with current systems, if an agent needs to invoke several operations against a data source, it outperforms traditional solutions handily as the network speed drops. Moreover, the more qualitative strengths of mobile agents, such as their support for disconnected operation and dynamic deployment, make current systems an attractive choice for some applications even if overall completion times are worse. As the performance and security issues are addressed, mobile-agent systems will become attractive for a wider and wider range of distributed applications.

Availability

The D'Agents software can be downloaded at the D'Agents Web site.[20] Most of the other systems mentioned in this chapter are also available for download— interested readers are referred to either the corresponding referenced paper(s) or the Cetus Web site.[21]

Acknowledgments

Many thanks to the Office of Naval Research (ONR), the Air Force Office of Scientific Research (AFOSR), the Department of Defense (DoD), and the Defense Advanced Research Projects Agency (DARPA) for their financial support of the D'Agents project: ONR contract N00014-95-1-1204, AFOSR/DoD contract F49620-97-1-03821, and DARPA contract F30602-98-2-0107; to the army of student programmers who have worked on D'Agents and D'Agent applications; to Brian Brewington, Daniel Bilar, Arne Grimstrup and Ron Peterson for reading and commenting on early drafts of this chapter; and to our editor, Jeffrey Bradshaw, and the anonymous reviewers for their invaluable feedback.

References

[AA98] Jumping Beans white paper. Ad Astra Engineering, Inc., September 1, 1998. See http://www.JumpingBeans.com/.

[AAB98] Yair Amir, Baruch Awerbuch, and R. Sean Borgstrom. A cost-benefit framework for online management of a metacomputing system. In *Proceedings of the First International Conference on Information and Computation Economies*, pages 140–147, Charleston, SC, October 1998. ACM Press.

[BC95] Krishna A. Bharat and Luca Cardelli. Migratory applications. In *Proceedings of the Eighth Annual ACM Symposium on User Interface Software and Technology*, November 1995.

[19] This third characteristic is particularly important since it means that a server will see only a slightly higher load than if it had provided a suite of high-level operations, i.e., if it had provided each unique agent's functionality as a unique builtin operation.

[20] `http://agent.cs.dartmouth.edu/.`

[21] `http://www.cetus-links.org/oo_mobile_agents.html`

[BGH$^+$99] J. M. Bradshaw, M. Greaves, H. Holmback, W. Jansen, T. Karygiannis, B. Silverman, N. Suri, and A. Wong. Agents for the masses? *IEEE Intelligent Systems*, pages 53–63, March/April 1999.

[BKR98] Jonathan Bredin, David Kotz, and Daniela Rus. Market-based resource control for mobile agents. In *Proceedings of the Second International Conference on Autonomous Agents*, pages 197–204. ACM Press, May 1998.

[BN97] Marc H. Brown and Marc A. Najork. Distributed active objects. *Dr. Dobb's Journal*, 22(3):34–41, March 1997.

[Car95] Luca Cardelli. A language with distributed scope. *Computing Systems*, 8(1):27–59, Winter 1995.

[CBC97] George Cybenko, Aditya Bhasin, and Kurt D. Cohen. Pattern recognition of 3D CAD objects: Towards an electronic yellow pages of mechanical parts. *Smart Engineering Systems Design*, 1:1–13, 1997.

[CH97] Gary Cornell and Cay S. Horstmann. *Core Java.* Sunsoft Press (Prentice Hall), 1997.

[Cha96] Phil Inje Chang. Inside the Java Web Server: An overview of Java Web Server 1.0, Java Servlets, and the JavaServer architecture. Sun Microsystems White Paper, Sun Microsystems, 1996.

[DMTH95] Giovanna Di Marzo, Murhimanya Muhugusa, Christian Tschudin, and Jürgen Harms. The Messenger paradigm and its implications on distributed systems. In *Proceedings of the ICC'95 Workshop on Intelligent Computer Communication*, 1995.

[DO91] Fred Douglis and John Ousterhout. Transparent process migration: Design alternatives and the Sprite implementation. *Software: Practice and Experience*, 21(8):757–785, August 1991.

[FPV98] Alfonso Fuggetta, Gian Pietro Picco, and Giovanni Vigna. Understanding Code Mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361, May 1998.

[GKCR98] Robert S. Gray, David Kotz, George Cybenko, and Daniela Rus. D'Agents: Security in a multiple-language, mobile-agent system. In Giovanni Vigna, editor, *Mobile Agents and Security*, volume 1419 of *Lecture Notes in Computer Science*, pages 154–187. Springer-Verlag, 1998.

[Gra96] Robert S. Gray. Agent Tcl: A flexible and secure mobile-agent system. In Mark Diekhans and Mark Roseman, editors, *Proceedings of the Fourth Annual Tcl/Tk Workshop (TCL '96)*, Monterey, California, July 1996.

[Gra97] Robert Gray. *Agent Tcl: A flexible and secure mobile-agent system.* PhD thesis, Dept. of Computer Science, Dartmouth College, June 1997. Available as Dartmouth Computer Science Technical Report TR98-327.

[GWtLT97] Andrew S. Grimshaw, Wm. A. Wulf, and the Legion Team. The Legion vision of a worldwide virtual computer. *Communications of the ACM*, 40(1):39–45, January 1997.

[HG98] Gisli Hjalmtysson and Robert S. Gray. Dynamic C++ classes: A lightweight mechanism to update code in a running program. In *Proceedings of the 1998 USENIX Technical Conference*, 1998.

[HMPP96] John Hartman, Udi Manber, Larry Peterson, and Todd Proebsting. Liquid software: A new paradigm for networked systems. Technical Report TR96-11, Department of Computer Science, University of Arizona, 1996.

[JJS$^+$97] Dag Johansen, Kjetil Jacobsen, Nils P. Sudmann, Kaare J. Lauvset, Kenneth P. Birman, and Werner Vogels. Using software design patterns to build distributed environmental monitoring applications. Technical Report TR97-1655, Department of Computer Science, Cornell University, 1997.

[JLHB88] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[JSvR98a] Dag Johansen, Fred B. Schneider, and Robbert van Renesse. Operating system support for mobile agents. In Dejan Milojicic, Frederick Douglis, and Richard Wheeler, editors, *Mobility, Mobile Agents and Process Migration – An Edited Collection*. Addison Wesley, 1998. Originally appeared in the *Proceedings of the 5th IEEE Workshop on Hot Topics in Operating Systems*.

[JSvR98b] Dag Johansen, Fred B. Schneider, and Robbert van Renesse. What TACOMA taught us. In Dejan Milojicic, Frederick Douglis, and Richard Wheeler, editors, *Mobility, Mobile Agents and Process Migration – An Edited Collection*. Addison Wesley, 1998.

[JTK97] Anthony D. Joseph, Joshua A. Tauber, and M. Frans Kaashoek. Mobile computing with the Rover toolkit. *IEEE Transactions on Computers: Special Issue on Mobile Computing*, 46(3):337–352, March 1997.

[KGN$^+$98] David Kotz, Robert Gray, Saurab Nog, Daniela Rus, Sumit Chawla, and George Cybenko. Mobile agents for mobile computing. In Dejan Milojicic, Fred Douglis, and Rick Wheeler, editors, *Mobility, Mobile Agents and Process Migration— An Edited Collection*. Addison Wesley, 1998.

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

[LO98] Danny B. Lange and Mitsuru Oshima. *Programming and deploying Java mobile agents with Aglets*. Addison Wesley, 1998.

[LS92] Michael Litzkow and Marvin Solomon. Supporting checkpointing and process migration outside the Unix kernel. In *Proceedings of the 1992 Winter USENIX Technical Conference*, pages 283–290, 1992.

[LSW95] Steven Lucco, Oliver Sharp, and Robert Wahbe. Omniware: A universal substrate for Web programming. *World Wide Web Journal*, (1), December 1995.

[MBB$^+$99] Dejan Milojicic, Markus Breugst, Ingo Busse, John Campbell, Stefan Covaci, Barry Friedman, Kazuya Kosaka, Danny Lange, Kouichi Ono, Mitsuru Oshima, Cynthia Tham, Sanka Virdhagriswaran, and Jim White. MASIF: The OMG Mobile Agent System Interoperability Facility. In Dejan Milojicic, Frederik Douglis, and Richard Wheeler, editors, *Readings in Agents*, chapter 14, pages 628–642. ACM Press, 1999.

[Moi98] Katsuhiro Moizumi. *The mobile agent planning problem*. PhD thesis, Thayer School of Engineering, Dartmouth College, November 1998.

[Muh98] Murhimanya Muhugusa. Implementing distributed services with mobile code: The case of the Messenger environment. In *Proceedings of the IASTED International Conference on Parallel and Distributed Systems (Euro-PDS'98)*, Austria, July 1998.

[MvRSS96] Yaron Minsky, Robbert van Renesse, Fred B. Schneider, and Scott D. Stoller. Cryptographic support for fault-tolerant distributed computing. In *Proceedings of the Seventh ACM SIGOPS European Workshop*, pages 109–114, September 1996.

[NCK96] Saurab Nog, Sumit Chawla, and David Kotz. An RPC mechanism for transportable agents. Technical Report PCS-TR96–280, Department of Computer Science, Dartmouth College, March 1996.

[OBJ97] ObjectSpace Voyager core package technical overview. ObjectSpace, Inc., December 1997. Version 1.

[Pei98] Holger Peine. Security concepts and implementations for the Ara mobile agent system. In *Proceedings of the Seventh IEEE Workshop on Enabling Technologies: Infrastructure for the Collaborative Enterprises*, Stanford University, USA, June 1998.

[PRS99] A. Puliafito, S. Riccobene, and M. Scarpa. An analytical comparison of the client-server, remote evaluation and mobile-agent paradigms. In *Proceedings of First International Symposium on Agent Systems and Applications and the Third International Symposium on Mobile Agents (ASA/MA'99)*, pages 278–292, Palm Springs, California, October 1999.

[PS97] Holger Peine and Torsten Stolpmann. The architecture of the Ara platform for mobile agents. In *Proceedings of the First International Workshop on Mobile Agents (MA '97)*, volume 1219 of *Lecture Notes in Computer Science*, Berlin, April 1997. Springer-Verlag.

[RAS96] M. Ranganathan, Anurag Acharya, and Joel Saltz. Distributed resource monitors for mobile objects. In *Proceedings of the Fifth International Workshop on Object Orientation in Operating Systems*, 1996.

[RASS97] M. Ranganathan, Anurag Acharya, Shamik Sharma, and Joel Saltz. Network-aware mobile programs. In *Proceedings of the 1997 USENIX Technical Conference*, pages 91–104, 1997.

[RGK97] Daniela Rus, Robert Gray, and David Kotz. Transportable information agents. *Journal of Intelligent Information Systems*, 9:215–238, 1997.

[RN98] Ori Regev and Noam Nisan. The POPCORN market— an online market for computational resources. In *Proceedings of the First International Conference on Information and Computation Economies*, pages 148–157, Charleston, SC, October 1998. ACM Press.

[Sal91] G. Salton. The Smart document retrieval project. In *Proceedings of the Fourteenth International ACM/SIGIR Conference on Research and Development in Information Retrieval*, 1991.

[Sch97] Fred B. Schneider. Towards fault-tolerant and secure agentry. In *Proceedings of the 11th International Workshop on Distributed Algortithms*, September 1997.

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

[SHK95] Behrooz A. Shirazi, Ali R. Hurson, and Krishna M. Kavi. *Scheduling and Load Balancing in Parallel and Distributed Systems.* IEEE Press, 1995.

[SJ95] Bjarne Steensgaard and Eric Jul. Object and native code thread mobility among heterogeneous computers. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 68–78, Copper Mountain, CO, December 1995. ACM Press.

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

[TDM$^+$94] Christian Tschudin, Giovanna Di Marzo, Murhimanya Muhugusa, Christian Tschudin, and Jürgen Harms. Messenger-based operating systems. Technical Report 90, University of Geneva, Switzerland, July 1994. Revised September 14, 1994.

[tiv99] Tivoli IT director. Tivoli Systems White Paper, Tivoli Systems, 1999.

[TMN97] Christian Tschudin, Murhimanya Muhugusa, and Guy Neuschwander. Using mobile code to control native execution of distributed UNIX. In *Proceedings of the Third ECOOP Workshop on Mobile Object Systems*, Finland, June 1997.

[Whi94a] James E. White. Mobile agents make a network an open platform for third-party developers. *IEEE Computer*, 27(11):89–90, November 1994.

[Whi94b] James E. White. Telescript technology: The foundation for the electronic marketplace. General Magic White Paper, General Magic, Inc., 1994.

[Whi97] James E. White. Mobile agents. In Jeffrey M. Bradshaw, editor, *Software Agents*, chapter 19, pages 437–472. MIT Press, 1997.

[WPM99] David Wong, Noemi Paciorek, and Dana Moore. Java-based mobile agents. *Communications of the ACM*, 42(3):92–102, March 1999.

[WPW$^+$97] David Wong, Noemi Paciorek, Tom Walsh, Joe DiCelie, Mike Young, and Bill Peet. Concordia: An infrastructure for collaborating mobile agents. In *Proceedings of the First International Workshop on Mobile Agents (MA '97)*, volume 1219 of *Lecture Notes in Computer Science*, pages 86–97. Springer-Verlag, 1997.

[WPW98] Tom Walsh, Noemi Paciorek, and David Wong. Security and reliability in Concordia. In *Proceedings of the Thirty-First Annual Hawaii International Conference on System Sciences*, volume VII, pages 44–53, January 1998.