D'Agents: Applications and Performance of a Mobile-Agent System

Robert S. Gray, George Cybenko, David Kotz, Ronald A. Peterson and Daniela Rus

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

firstname.lastname@dartmouth.edu

November 28, 2001

Abstract

D'Agents is a general-purpose mobile-agent system that has been used in several information-retrieval applications. In this paper, we first examine one such application, operational support for military field personnel, where D'Agents greatly simplifies the task of providing efficient, application-specific access to remote information resources. After describing the application, we discuss the key differences between D'Agents and most other mobile-agent systems, notably its support for strong mobility and multiple agent languages. Finally, we derive a small, simple application that is representative of many information-retrieval tasks, including those in the example application, and use this application to compare the scalability of mobile agents and traditional client/server approaches. The results confirm and quantify the usefulness of mobile code, and perhaps more importantly, confirm that intuition about when to use mobile code is usually correct. Although significant additional experiments are needed to fully characterize the complex mobile-agent performance space, the results here help answer the basic question of when mobile agents should be considered at all, particularly for information-retrieval applications.

1 Introduction

One of the most attractive applications for mobile agents is distributed information retrieval, 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 service, and perform a *custom* retrieval task local to the service. Only the final results are transmitted back to the portable device. Source data and intermediate results are not transmitted, even though the information service had no prior knowledge of the client or its specific task. 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.

If an information service does not directly support a client's task, mobile agents allow efficient service access without requiring the service developer to *add* new operations to the service software. By moving the code to the data, a mobile agent can reduce the latency of individual steps, avoid network transmission of intermediate data, continue even in the presence of network disconnections, and thus complete the overall task much faster than a traditional client/server solution.

THIS COPY IS THE AUTHORS' PRE-PUBLICATION VERSION; it may differ slightly from the official published version.
Software: Practice and Experience, May 2002. Invited paper.
doi:10.1002/spe.449.

A mobile agent will not always perform better than a client/server solution. For example, if the agent code is larger than the total intermediate data, the mobile agent must perform worse, since it will transfer more bytes across the network than the client/server solution. Similarly, if the network is fast enough, the agent might do worse even if the code is smaller, since mobile agents are typically written in an interpreted language for portability and security reasons. With a fast and reliable network, interpreting the agent on the server might be slower than transmitting the intermediate data to the client. As network speed and reliability drops, however, or data sizes increase, the picture changes considerably.

An important goal of our research at Dartmouth has been to characterize the mobile-agent performance space. When are mobile agents a better choice for information-retrieval applications than traditional client/server techniques? To support this research, we developed the D'Agents mobile-agent system. D'Agents, which was once known as Agent Tcl, was first used internally in 1994 and released publicly in 1995. Since then, D'Agents has been downloaded several thousand times, and used in numerous external research projects. The most recent version of D'Agents, D'Agents 2.1, allows the programmer to write mobile agents in three different programming languages, Tcl, Java and Scheme, supports strong mobility for Tcl and Java, provides security mechanisms to protect machines against malicious agents, and includes a high-performance, multi-threaded agent server. Multiple languages allow the agent programmer to select the most appropriate language for the task at hand, and strong mobility appears easier for the agent programmer to understand and use than weak mobility [Whi98]. At the same time, strong mobility does require more work from the system developer, since more of an agent's state must be captured for the agent to migrate to a new machine.

Using D'Agents, we have implemented several information-retrieval applications, ranging from searching three-dimensional drawings of mechanical parts for a needed part [CBC97] to supporting the operational needs of a platoon of soldiers [Gra00]. In this paper, we first describe the operational-support application, where mobile agents perform a range of simple and complex information-retrieval tasks, both for the soldiers themselves and for the planners and analysts at mission headquarters. After discussing this application in Section 2, we describe the architecture and implementation of the D'Agents system in Section 3, comparing D'Agents with other mobile-agent systems.

Finally, in Section 4, we derive a simple application that is representative of the information-retrieval tasks in the soldier application, as well as in many other applications. In the application, a client machine searches a single remote information service in one of two ways, namely, sending a mobile agent or making traditional client/server invocations across the network. Although this application is simple and involves at most a single agent migration from client to server machine, the decision of whether to make an initial migration is the first and most important decision in most mobile-agent applications. Can the client code efficiently interact with the service from its home machine, or should it send some part of itself to a more attractive network location? A performance analysis of this simplest case is an essential step toward allowing mobile agents to make the most effective migration decisions.

Our earlier paper [GCKR01] evaluates the performance of a single agent that migrates from one host to another host. In this paper, we look beyond these initial performance characterizations and examine the *scalability* of the mobile-agent and client/server solutions, comparing query completion times as more and more client machines access the information service simultaneously. In Section 4, we focus on network bandwidth, the pass ratio, and the number of client machines, where the pass ratio is the percentage of data that passes an application-specific filter after some initial and more general query. Other parameters, such as network reliability and latency, client and server processing speeds, data size, task granularity, and choice of agent programming language, are equally important, but left for future papers. As such, the scalability results in Section 4 describe an important but partial section of the performance space.

As we will see in Section 4, mobile agents scale well. Moreover, the performance curves are straightforward, and can be described easily with simple mathematical equations. As long as the server CPU is not overloaded, the query completion times scale linearly according to the amount of data transferred over

the network, which in turn is proportional to the number of client machines making simultaneous queries. Although intuition would allow us to derive the same equations, it is important to see that a real mobile-agent system does in fact correspond to our expectations, particularly given the number of overheads that are present in all mobile-agent systems. Such overheads range from starting up a new execution environment for an incoming agent to cleaning up the execution environment once the agent has finished. Experimentally confirming that mobile-agent systems conform to expectations allows the designer to confidently analyze situations for which experiments are difficult or impossible, and thus, to make effective design decisions.

2 An information-retrieval application

In this section we describe an application of mobile agents to gathering and distributing data in field operations. The specific scenario we developed concerns soldiers in a peace-keeping mission but the structure of the application is generally applicable to any field mission that involves multiple players and the need to access and distribute data across players.

This application, the largest D'Agents application, allows small teams of soldiers to communicate with each other and with their mission headquarters via wireless computers [Gra00]. The application was developed as part of the Active Communications (ActComm) project,[1] a Multi-disciplinary University Research Initiative (MURI) funded by the Department of Defense and administered by the Air Force Office of Scientific Research. The participants in the ActComm project are Dartmouth College, Harvard University, Rensselaer Polytechnic Institute, the University of Illinois, Lockheed Martin and ALPHATECH, each of which provided technology components for the implementation.

The application has been tested under several different military scenarios, with Dartmouth College students playing the role of soldiers. The most recent scenario was an urban peace-keeping mission in which a team of sixteen student "soldiers" surrounded, observed and seized a building on the Dartmouth College campus in late 1999. This scenario begins when an intelligence team at headquarters intercepts one or more phone calls and determines that a terrorist group is likely to meet in a particular building. Headquarters dispatches a team of soldiers to the building, and the soldiers take up observation posts (probably hidden) around the building. The soldiers wait for the suspected terrorists, and if the terrorists come to the building, the soldiers will seize the building and arrest the terrorists. Such a scenario is common in peace-keeping efforts.

Each soldier has a portable computing device with a GPS unit and a wireless network card (2 Mbps WaveLAN 802.11 in our experiments). One or more soldiers serve as gateways between the wireless cloud and the main military network. Routing data between the main military network and a particular soldier is a matter of routing that data to and from the gateway soldiers. In real life, these gateway soldiers might have satellite transmission equipment to establish an uplink with headquarters. In our tests runs on the Dartmouth campus, however, a satellite was not available to us, so we simulated satellite connections with wireless cards set to a different transmission frequency.

The application involves several technology components that can work with mobile agents, including wireless ad-hoc routing, persistent queries, and the like. The APRL ad-hoc routing system [KK98] from the EECS group at Harvard, for example, provides 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 themselves play two roles within the application. First, mobile agents carry special-purpose interface code onto the soldiers' devices. When the soldier sees a possible suspect enter the building, the soldier sends a description of that suspect to headquarters. Headquarters searches available databases

[1] `http://actcomm.thayer.dartmouth.edu`

to identify exactly who the suspect might be. Once one or more candidates are 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, is the person 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, a different country, or the United Nations. 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.

Second, and more importantly for this paper, mobile agents handle all information-retrieval tasks. The intelligence team at headquarters and the soldiers in the field can perform several kinds of queries against different information services. These services include databases of intercepted phone calls, physical descriptions, affiliations, and aliases of known terrorists, and military and public news articles that relate to the country in which the peace-keeping force finds itself. The news database, for example, provides an interface that allows a client to (1) perform relatively complex keyword queries, much like any Internet search engine, and (2) retrieve the full text of any article. This interface, however, does not exactly match all needs. An intelligence analyst might want to retrieve only those articles of a certain length, for example, or retrieve only (automatically constructed) article summaries.

By sending a small mobile agent to the news database, a client application can accomplish either of these tasks without transferring unwanted data across the network. Since mission headquarters might have only a slow and unreliable connection to the main network and the databases that it contains, minimizing bandwidth usage, as well as the number of distinct interface operations performed across the network, is critical. Without mobile agents, the same minimization can be achieved only if each database provides a single high-level operation for each retrieval task. Even in a military environment, where the information services and their clients are nominally under the control of a single entity, it is unlikely that such a one-to-one mapping between service operations and client tasks can be achieved. Each service and client is likely under the control of a different programming group within an extremely large military organization. By designing their services to accept mobile agents, service developers allow client developers to efficiently perform *unpredicted* retrieval tasks.

The most complex retrieval task in the ActComm application is searching the phone-call database for phone calls that appear to mention suspicious persons or activity. In the current implementation, an analyst specifies a terrorist group of interest. The client application first queries the database of aliases and affiliations to identify any name that might be mentioned in a phone call between one terrorist and another. Then the client application examines the text transcript (as well as associated annotations) of each intercepted phone call, and flags for the analyst those phone calls that score highly according to some evaluation function.

The most interesting aspect of this retrieval task is the persistence of the query. New intercepted phone calls are added to the database continuously, and the client application must apply its evaluation function to each phone call, at least for the duration of the mission at hand. In addition, the needs of the analyst (and thus the client application itself) determine the exact evaluation function, and it is unlikely that the phone-call database will support all variations. Thus, unless evaluation code can be sent to the phone-call database or to some nearby proxy machine, significant bandwidth can be wasted sending back calls that the evaluation function will reject. In fact, under some combinations of network conditions, it might be advantageous to send a mobile agent to the affiliation database, and then have the agent migrate from the affiliation database to the phone-call database. Although analyzing such multi-hop scenarios will be an important part of future work, we focus on the single-hop or remote-evaluation case later in this paper. No matter how many hops an agent makes, it must still make the first decision of whether to leave its home machine at all. Can the

agent efficiently access the two databases from its home machine, or should the agent move into the main network?

3 D'Agents

The mobile-agent system that we use to explore this question is our own system, D'Agents. We developed D'Agents to support distributed information-retrieval applications, including the soldier application described above. Our main design goal was to support multiple agent languages, so that the programmer could choose the most appropriate language, and our main implementation goal was to optimize performance, so that we could more accurately explore the potential performance advantages of the mobile-agent paradigm. The current version of D'Agents supports three languages (Tcl, Java, and Scheme), provides strong mobility (in Tcl and Java), uses encryption for authentication and privacy of moving agents, enforces limits on resource usage, and is based around a high-performance, multi-threaded agent server.

In this section, we describe the D'Agents system, and compare it with several other representative systems. Before we proceed with the description and comparison, we note two things about the other systems that we discuss. First, we know of two systems with the name "Messengers." In this paper, we refer to the system from the University of Geneva. Second, the μCODE system is not, strictly speaking, a mobile-agent system. μCODE is actually a modular set of components for developing systems based on mobile code. A mobile-agent abstraction has been built on top of μCODE, and it is that higher-level abstraction that we consider here.

Tables 1 and 2 summarize the comparison of mobile-agent systems, and include citations to appropriate literature.[2]

3.1 Architecture

The basic architecture of the D'Agents system is shown in Figure 1. The architecture has five levels: transport mechanisms, a server that runs on each machine and accepts mobile agents, a shared C++ library that implements the D'Agents agent functionality, an execution environment for each supported agent language, and the agents themselves. Each execution environment includes the interpreter or virtual machine for the agent's language, stub routines to allow the agent to invoke the functions in the shared library, a state-capture module to support agent mobility, and a security module to enforce resource limits.

As in all mobile-agent systems, the key component of the five-layer D'Agents architecture is the server that runs on each host. The server receives incoming agents, authenticates the identity of the agent owners, assigns resource limits to the agents, unpacks the components, code, date, and execution state, of the agents, injects the agents into the appropriate execution environments, and manages the agents until they depart for another host.

In concert with the shared library, the server provides communication, migration and bookkeeping services to the agents resident on its host. In contrast to some mobile-agent systems, these services are low level. For example, an agent addresses another agent by specifying the target agent's current machine and its unique integer id on that machine. Location-independent addressing is provided through dedicated directory agents that other agents can use or ignore as their needs dictate. Making the core services low level was a conscious design decision, since many mobile agents require only simply functionality, and should not be penalized with the overhead of unnecessarily complex routines.

In the current implementation, the server is a single multi-threaded process that executes each Tcl and Scheme agent inside its own Unix process, but executes multiple Java agents as threads inside a single Java

[2]Another useful source of information is available in the Mobile Agents List on the web at `http://mole.informatik.uni-stuttgart.de/mal/preview/preview.html`.

Table 1: Features of many mobile-agent systems.

System	Languages	Thread migration	Data migration	Code migration	Multithreaded agents?
Aglets [LO98]	Java	weak, fixed entry	Java serialization	sender push or on-demand from server	no
Ara [PS97, Pei98]	C, C++, Tcl	strong	custom	sender push	no
Bond [BJP+00]	Java, Python, Jess	weak, ?	custom	sender push?	yes?
Concordia [WPW+97, WPW98]	Java	weak, itinerary	Java serialization in various ways	sender push or on-demand from sender	yes
D'Agents [GKCR98, BMcI+00, KGN+97]	Java, Tcl, Scheme	strong	custom	sender push	no
FarGo [HBSG99]	Java	weak, ?	Java serialization		
Grasshopper [BM98]	Java	weak, ?		on-demand from sender?	yes
Jumping Beans [AA98]	Java	weak; itinerary	Java serialization?	sender push	yes
Messengers [TDM+94, TMN97, DMTH95, Muh98]	M0	weak	manual	sender push	no
Mole [BHRS98, SR99]	Java	weak, fixed entry	Java serialization	on-demand from code server	yes
μCODE [Pic98a, Pic98b]	Java	weak, ?		many ways	possible
NOMADS [SBB+00]	Java (custom JVM)	strong	custom JVM	sender push	yes
[Visual] Obliq [Car95, BN97, BC95]	Obliq	weak	push desired objects	push desired procedure	
Tacoma [JvRS95, JSvR98]	v1.2: C+others. V2.0: C only	weak, ?	manual	sender push	yes
Telescript [Whi94b, Whi94a, Whi97]	Telescript	strong	custom	sender push	no
Voyager [OBJ97]	Java	weak	Java serialization		yes

Table 2: Features of many mobile-agent systems.

System	Persistence?	Communication	Authentication	Access control	Resource control
Aglets [LO98]	yes	send object as message, via proxy	programmer signs code, user signs data	ACL based on sigs?	
Ara [PS97, Pei98]	yes	free-form via named rendezvous points	programmer signs code, user signs data	ACL based on sigs?	allowances based on sigs
Bond [BJP+00]	no?	KQML, XML messages			
Concordia [WPW+97, WPW98]	yes	send object as event, publish/subscribe	signed by user	ACL based on user	no
D'Agents [GKCR98, BMcI+00, KGN+97]	no	string messages	agent signed by user and sending machine	ACL based on user	limits on user; market-based control
FarGo [HBSG99]		method calls or events			
Grasshopper [BM98]		send object; various protocols	SSL, X.509		
Jumping Beans [AA98]	yes	CORBA method calls	central server authenticates all	ACL based on user	limits in ACL
Messengers [TDM+94, TMN97, DMTH95, Muh98]		local: shared memory and bulletin board; remote: none			market-based
Mole [BHRS98, SR99]	sophisticated rollbacks	send object; message, RPC, or publish/subscribe			no
μCODE [Pic98a, Pic98b]	no	send object	no	no	no
NOMADS [SBB+00]	no	raw messages	accounts and passwords	Java security, per-account policy file	mechanisms for limiting usage
[Visual] Obliq [Car95, BN97, BC95]		method invocation	no crypto	ACL	no
Tacoma [JvRS95, JSvR98]		raw message	relies on OS	relies on OS	no
Telescript [Whi94b, Whi94a, Whi97]	yes	method invocation or events	agent signed by user	"permits" assigned based on user	"permits" assigned based on user
Voyager [OBJ97]	yes	method call via proxy, or publish/subscribe			

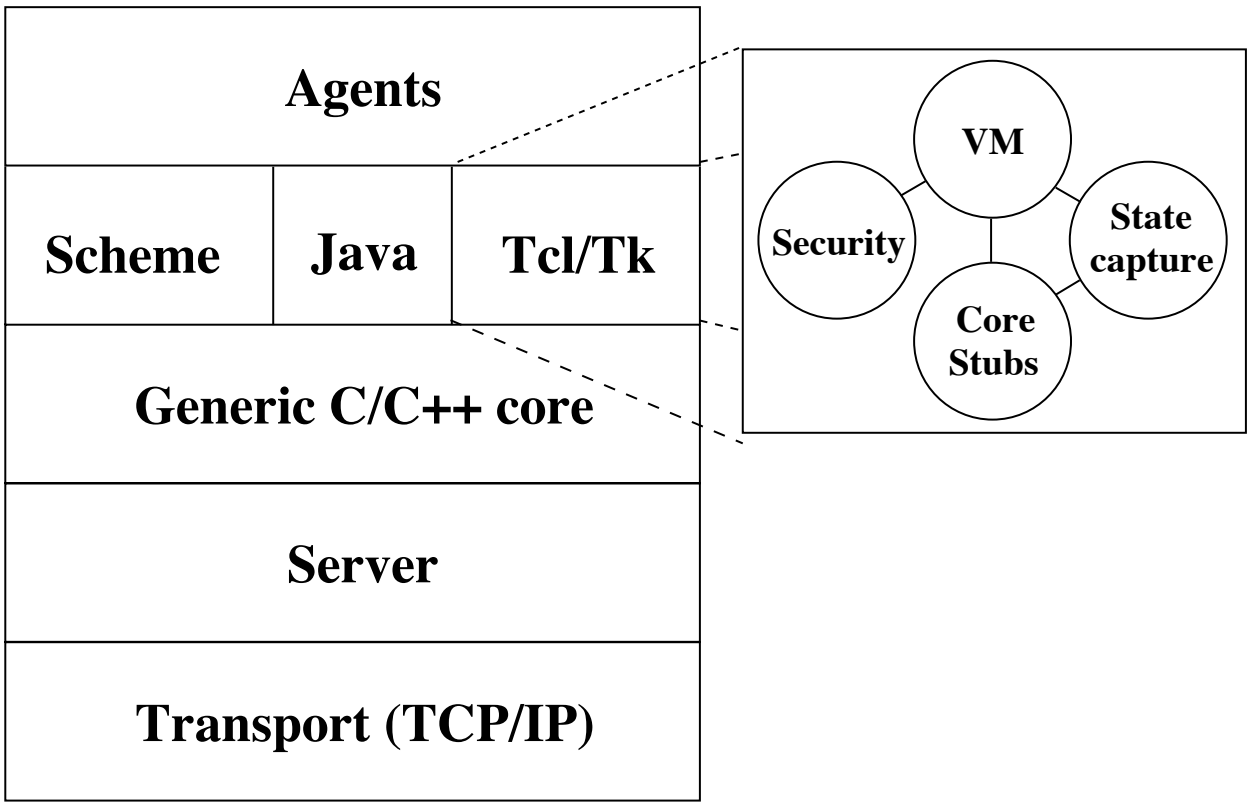

Figure 1: The D'Agents architecture.

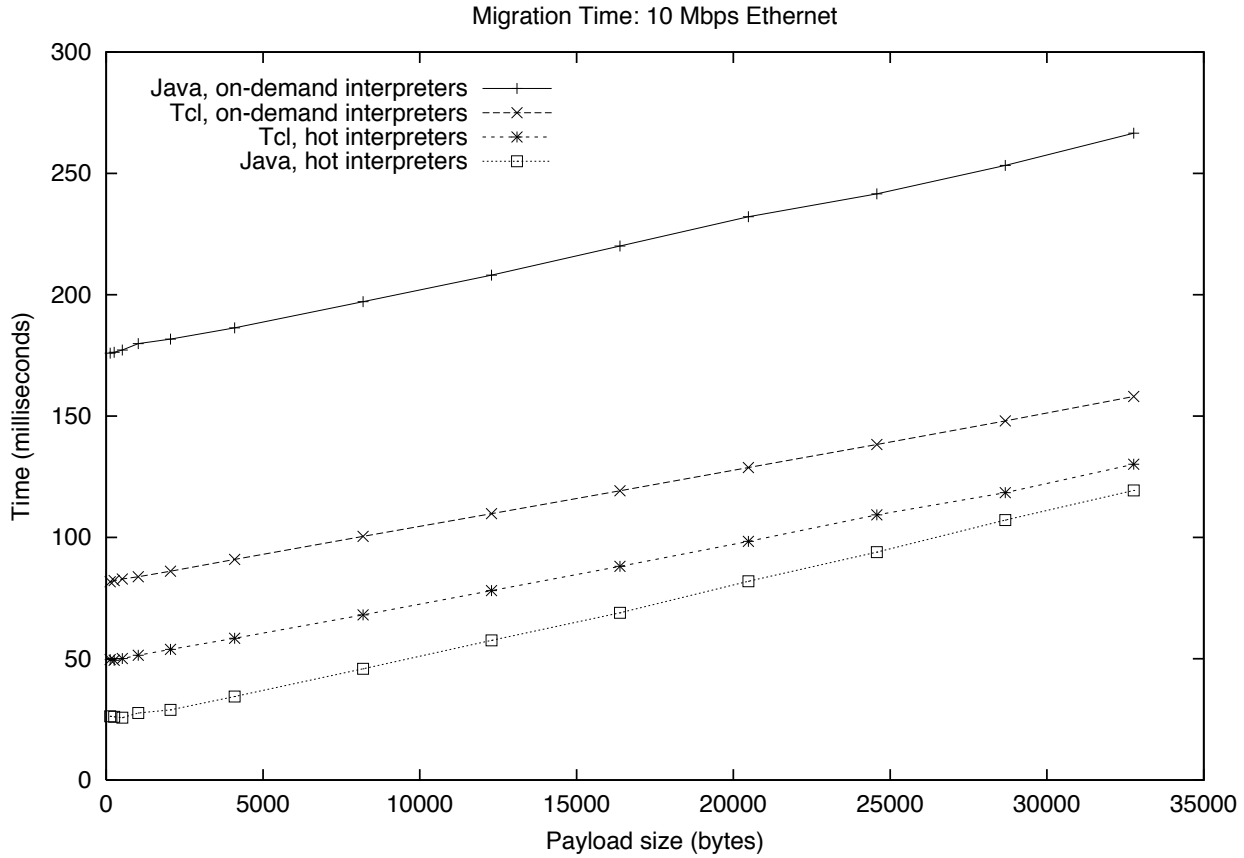

Figure 2: Performance of on-demand versus "hot" interpreters.

process. More specifically, the server starts a small number of Java virtual machines, and executes multiple agents inside each. Executing multiple agents inside a single Java process is essential for system scalability due to the large memory footprint of the Sun Java Virtual Machine 1.0, while dividing the agents among multiple Java processes avoids the problems that the Sun JVM 1.0 encounters as the number of active threads increases. The approach of running each Tcl and Scheme agent inside its own process was adopted solely for implementation simplicity, since our Tcl and Scheme interpreters have relatively small memory footprints, but no inherent thread support.

The server does not wait for an incoming agent before spawning an interpreter process for that agent, since interpreter startup is a large overhead that should not be part of the critical migration path. Instead, the server spawns a set of "hot" interpreters when it first starts execution, and simply assigns incoming agents to the first free hot interpreter. The server starts and stops hot interpeters as the number of resident agents changes, so that there are always some free interpreters ready for incoming agents, but not an unnecessarily large number. Figure 2 shows the performance advantage of "hot" versus on-demand interpreters. The y-axis is the time in milliseconds for an agent to migrate to a remote machine and then return to its home machine, and the x-axis is the size of the agent's payload or application-specific code. Each agent includes an additional two to four kilobytes of identification information and generic migration code. As one would expect, Java agents exhibit a dramatic performance improvement with hot interpreters due to the Java VM's large startup overhead.

The D'Agents approach of multi-threaded server, but separate interpreter processes for the agents, allows both high performance and straightforward support for multiple languages. We could achieve even

greater efficiency by running each agent as a thread inside the server itself, as do many other mobile-agent systems (particularly those based on Java). In a multiple-language system, however, the cost of this efficiency is significant implementation complexity, and we do not feel that the efficiency improvements justify the implementation effort. On the other hand, despite the complexity, one group has successfully taken this approach—Ara supports multiple languages, but runs every agent as a thread inside a single server process.

D'Agents and most other mobile-agent systems uses a peer-to-peer server architecture in which agents move directly from one machine to another. In contrast, Jumping Beans uses a centralized server architecture in which agents must pass through a central server on their way 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.

3.2 Mobility

D'Agents supports strong mobility, since our experience with undergraduate programmers suggests that strong mobility is easier than weak mobility for the *agent* programmer to understand and use [Whi98]. At the same time, however, strong mobility requires significant effort from the *system* programmer. The system must capture enough state information to restore the agent in exactly the same control state on its new machine, and in fact, we had to modify off-the-shelf Tcl and Java interpreters to include the necessary state-capture routines. Although modified interpreters pose no problems for an academic research group, they would hurt the market acceptance of commercial systems, and nearly all commercial systems support only weak mobility.

Whether a system supports weak or strong mobility, it is not always clear precisely which resources should migrate with the agent. For example, what if an agent has opened a file on one machine, and decides to migrate to a new machine without closing the file? A complete discussion of such issues extends beyond the scope of this paper, but a more thorough analysis can be found in Fuggetta et al. [FPV98]. In the paragraphs that follow, we consider only the major components of an agent's state, and compare D'Agents migration with the migration in other mobile-agent systems.

Thread. Strong-mobility systems capture and migrate thread state, which includes the procedure-call stack and local variables, while weak-mobility systems do not. In D'Agents, we support strong mobility for Tcl and Java with significant modifications to the respective interpreters. Similarly, the NOMADS system has a custom Java interpreter, and the Telescript system has a custom language and interpreter. Other systems pre-process the Java source code or bytecodes [FÖ98, SSY00, TRV+00], while Ara compiles C++ programs into bytecodes, and interprets those bytecodes inside a virtual machine designed to capture thread state.

Systems with weak mobility vary widely in the way that agents request to be moved. In one common approach, a "go" statement causes the agent's code and data to be moved to a given machine, and a predetermined procedure is called (or method invoked). Often, this procedure or method ("main()" or "run()") is the same one used when the agent was born; the programmer must use the data state to determine what to do each time the procedure is called. Another common approach is to associate an *itinerary* with the agent; the itinerary lists the sequence of hosts to visit, and the method to invoke on each host. 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. Although the lines between itinerary and non-itinerary systems are often blurred, itinerary systems usually are distinguished by a naturally event-driven programming style. For example, although Aglets does invoke the same method at each stop, the programmer can optionally supply additional methods that are automatically invoked before departure and after arrival, as well as in response to many other system events.

FarGo explicitly separates the programming of agent migration from the core of an agent's behavior. The agent programmer specifies the "layout policy" using an API from within the Java program, or using their custom scripting language. Layout changes are reactions to events, generated by FarGo hosts or by other FarGo agents. One possible reaction is to push an agent to another host.

Multiple threads. Although in most systems an agent is a single thread of execution, some systems allow agents to create and use multiple threads. When an agent has more than one thread, however, it becomes important to carefully define the semantics of the mobility ("go") instruction: when one thread calls "go", do all threads move, or just the caller? If only the calling thread, what happens to the other threads? Which data is moved along with the thread(s)? From the literature it is not clear what happens in most systems. We suspect that the default behavior, as in D'Agents, is for the calling thread to migrate and the others to be left behind. With some programmer effort, data representing the state of other threads could be gathered and transferred, then at the destination new threads could be created with the captured state. NOMADS has the capability to capture and transfer the state of all threads.

Data migration. Every mobile-agent system must provide a mechanism to transfer data, such as global variables or heap data. In D'Agents, as in most systems with strong mobility, all variables and heap data are transferred transparently with the agent. Mechanisms vary widely in systems with weak mobility. Typical Java-based systems use Java's serialization capability to transfer an `Agent` object and all data reachable from that object. Messengers do not actually "jump"; they create new messengers (with specified code and data) on a specified target machine. In Obliq, similarly, a procedure and a "briefcase" of specified objects are pushed to the target machine.

One difficult issue is to identify which data should move with the agent. In D'Agents, Agent Tcl moves all data, and Agent Java moves all data reachable from the `Agent` object and thread stack. FarGo has five explicit types of references between their migratable units ("complets"): Link, Pull, Duplicate, Stamp, and Bi-directional Pull. By choosing the type of reference, the migration of one complet might pull along a referenced complet, pull along a copy of a referenced complet, and so forth.

Code migration. Every mobile-agent system must provide a mechanism to transfer the code for the agent, although approaches vary widely. The sending host could push all necessary code to the destination host along with the agent's data, or it could simply send the data and expect the destination to specifically request code modules that it needs. The former approach guarantees that the agent has everything it needs on arrival (important in systems hoping to support disconnected computing); the latter approach reduces transfer latency and may be especially useful if not all code is needed at every host, or if code modules might be cached at each host. In some systems, the destination asks the sending host for code, in other systems, the destination asks a third-party "code server" for code. The use of a code server allows the sending host to drop out after sending the data to the destination.

In D'Agents, we push the agent's entire code base from the source machine to the target machine as part of the agent's state image. We have no plans to support on-demand code fetching from previous machines, since this approach weakens the mobile-agent abstraction: the mobile agent becomes dependent on a machine that it has chosen to leave, a dependency that is particularly undesirable in our mobile-computing and wireless-network applications. We do not have, but hope to add, support for code caching, because it is essential for the highest performance.

In some languages it may be difficult to know precisely which code modules will be needed at the destination host. Some mobile-agent systems conservatively send all code along with the agent. Most Java-based systems send classes visible through introspection, that is, those that are used by members of the agent's objects. Unfortunately this approach is not complete, if an agent uses classes that are not mentioned

in the object's members. Concordia ships much of the code with the agent, along with a URL for the agent's home server where additional classes may be obtained if necessary. μCODE examines the bytecode for each class as it is sent, to determine which other classes need to be sent. These two systems can thus push a portion of the code to the destination, rather than all or none as in the above alternatives. The Sumatra group developed an interesting compiler technique to determine exactly which Java classes would need to migrate [AS97].

3.3 Languages

A single programming language is the focus for most mobile-agent systems. Java is used in most commercial systems and in many research systems, popular because of its portable virtual machine, support for object serialization, rudimentary support for the execution of untrusted code, and penetration in the marketplace. Other systems, like Obliq, have added support for object mobility to an existing language. Still others, such as Messengers and Telescript, developed a language specifically to support the mobile-agent programming paradigm.

Ara, Bond, D'Agents, and Tacoma 1.2 all support multiple programming languages. (The beta release of Tacoma 2.0 supports only C but other languages can be added easily.) We believe that no one language is ideal for all programming tasks, so systems that allow the programmer to choose a language, or even mix languages, is inherently more flexible. Thus, D'Agents allows the programmer to write mobile agents in Tcl, Java, or Scheme. Adding a new language to D'Agents is straightforward, albeit time consuming if one wants to support strong mobility for the new language or enforce the full set of D'Agent security constraints.

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. D'Agents is no exception as all three of its languages choices, Tcl, Java, and Scheme, are interpreted.

3.4 Persistence

One concern of mobile-agent programmers is the fate of their mobile agent should its host machine crash. In most mobile-agent systems, including D'Agents, the agent is simply lost. In Telescript, however, 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. Concordia writes agents to stable storage before and after each jump, to avoid agent loss in the event of a machine crash. It also permits a checkpoint at any method boundary. Aglets, Ara, Jumping Beans, and Voyager provide mechanisms for agents to save their state (checkpoint) at any time. Considering that mobile-agent systems must be able to capture an agent's state for transfer to another host, it is relatively easy to support checkpointing.

Tacoma has a few features that an agent might use to achieve persistence. Tacoma allows agents to create folders inside *cabinets*, which are on disk and survive past the lifetime of a particular agent. Tacoma examines a special "system" cabinet at boot time, and launches new agents from any folders found there. Thus, an agent could persist across system crashes and reboots by saving its state into a folder in the system cabinet. That approach requires quite a bit of work from the programmer, however.

Mole provides perhaps the most sophisticated persistence mechanism. It uses logging to allow application-initiated rollback of prior actions, in support of transaction-oriented applications such as e-commerce and system management [SR99].

3.5 Communication

Mobile agents must be able to communicate with other agents, and, in some cases, with other non-agent services and resources. There are at least four issues in any inter-agent communication system: how to

identify the other party, what interface to use for sending and receiving data, how to pass data from one party to the other, and how to maintain communication with a moving agent.

The D'Agents servers provide a per-host namespace for agent communication. Each agent is identified with a tuple, the network address of the machine on which the agent currently resides, and a unique integer that the machine's server assigns to the agent when it first arrives. This integer is unique on that particular machine, but is not unique globally. In addition, an agent can optionally register a symbolic string name, also unique only on its current machine. Within this namespace, D'Agent agents can use three low-level communication mechanisms. With message passing, agents exchange arbitrary string or binary data with standard *send* and *receive* primitives— D'Agents does not impose any syntactic or semantic constraints on these messages. With streams, an agent establishes a direct connection with a target agent and then sends string or binary messages across the connection. With events, an agent sends events to another agent, which catches and processes the events with registered event handlers.

We chose to support only low-level communication mechanisms in the core D'Agents system, since many applications know the locations of their component agents through application-specific means. In addition, many applications require only simple data exchange. Such applications do not need complex lookup services, delivery services, or message formats, and should not be forced to suffer the associated communication overheads. For those agents that do need higher-level services, D'Agents provides those services through libraries or stationary service agents. For example, to support a few of our information-retrieval applications, we developed a simple directory service ("yellow pages") [GKN+97], so that an agent could find a needed service agent without knowing the service agent's current location. This directory service is implemented as a collection of stationary cooperating agents.

Other mobile-agent systems employ a wide range of communication approaches. Most systems use one of four approaches: 1) passing messages containing strings or arbitrary data; 2) passing messages containing serialized objects; 3) invoking methods on objects in the other agent, or 4) publishing events to some sort of channel.

Aglets, Grasshopper, μCODE, and NOMADS send serialized objects as a message to another agent. NOMADS can also send raw chunks of data if desired. Telescript agents "meet" other local agents, after which they invoke methods on each others' objects; within a site, they can also communicate by sending events.

Messengers is truly different from all the others. Two messengers can communicate via a sort of shared memory if they both know the necessary address. For less intimate communication, there are also two bulletin-board services: the *global dictionary*, which allows data exchange between messengers, and the *service dictionary*, which is a browsable listing of messengers that offer services to other messengers. Remote communication is not possible.

A few systems attempt to allow communication with a traveling agent, through the use of a stationary proxy object. Correspondents send messages (or method invocations, or events) to the proxy, and the proxy forwards the communication on to the current location of the traveling agent. Aglets and Voyager use this approach. It is difficult to ensure reliable delivery to a moving agent, however, although μCODE does so [MP99].

Several systems provide higher-level communication abstractions. Ara uses "named rendezvous points" for free-form message passing. Concordia, Mole, and Voyager support a "publish/subscribe" model for events, in which anyone can publish an event to a given topic, and all subscribers can receive that event.

In short, approaches to communication vary widely, as widely as in any distributed system. The only aspect that is truly related to *mobility* is the difficulty in communicating with a moving agent.

3.6 Security

Security is a critical issue in mobile-agent systems, but also the most time consuming to discuss. Since D'Agents security is thoroughly described in two earlier papers [Gra96, GKCR98], and does not play a key role in our scalability analysis, we provide only a brief discussion here.

Like most mobile-agent systems, D'Agents protects a machine from malicious agents, but, aside from encrypting an agent in transit, does not protect an agent against malicious machines or other attackers [GKCR98]. The agent's owner or sending machine digitally signs and encrypts the agent; and the receiving machine decrypts the agent, verifies the signature, accepts or rejects the agent based on its signature, assigns access restrictions to the agent, then executes the agent in a secure execution environment that enforces the restrictions. Other mobile-agent systems primarily differ in which principals can sign and encrypt all or part of the agent, in the type and granularity of resource restrictions that can be enforced, and in whether an agent can authenticate a server before migrating to it.

Authentication. In some systems, such as Aglets and Ara, the programmer (or manufacturer) signs the code, and the user signs the data and parameters. Then hosts decide how much trust to assign to the agent based on both factors. In Jumping Beans, a central server coordinates all activity. The server authenticates all agent "places" that wish to participate, and they in turn authenticate the server. The server also authenticates all agents and users.

Access control. D'Agents uses access-control lists, keyed on the owner of the agent, to decide which resources each agent can access. Resource-manager agents maintain the access-control lists. Whenever an agent attempts a "dangerous" action such as opening a file or network connection, the security-enforcement module inside the agent's interpreter consults the resource managers. This approach cleanly separates the security policy and mechanism.

Visual Obliq includes 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 human user whether the agent should be allowed to perform the given action. Tacoma relies on the underlying operating system's native access-control mechanisms, but the Tacoma group has explored several interesting fault-tolerance and security mechanisms. The mechanisms include the use of cooperating agents to search replicated databases in parallel and then securely vote on a final result [MvRSS96], and the use of security automata (state machines) and software fault isolation to specify and enforce a machine's security policy [Sch97].

Resource control. D'Agents decides whether an agent can access a particular resource, and in some straightforward cases such as CPU time, does impose a limit on how much of that resource can be used in *total*. In general, however, D'Agents does not impose limits on total resource consumption, and in particular, does not impose limits on the resource consumption per unit time. For example, a D'Agents agent can use CPU time as quickly as it desires, until the point at which it reaches its total limit. Most other mobile-agent systems have the same limitations, mostly because the mechanisms for efficient and effective fine-grained control over the agents' usage of CPU, disk, and network are rarely available in the language run-time system. Mobile-agent systems primarily differ in exactly which total consumption limits can be assigned and enforced. In Ara, for example, an agent can be assigned an arbitrary limit on its total memory consumption, whereas a D'Agents agent can consume memory up to the limit that Unix imposes on the interpreter process.

NOMADS has the most extensive set of resource-control mechanisms of all Java-based systems, made possible by their choice to develop their own JVM. They control usage of CPU cycles, network bandwidth, and disk usage.

3.7 Summary

The preceding overview of D'Agents, and comparison with other mobile-agent systems, touches on only some of the more important features of a mobile-agent system. Again, Tables 1 and 2 summarize our comparison effort, and include citations to appropriate literature. The D'Agents system is explained in more detail in several earlier papers [RGK97, KGN+97, GKCR97, Gra97, CCMG98, GKCR98, BGM+99, GCKR01, KCG+02].

Despite the differences described above, and other small differences, all of the systems discussed above (with the exception of Messengers and μCODE, which are lighter-weight mobile-code systems) are intended for the same applications, such as information retrieval, 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. In the rest of this paper, we will use D'Agents to explore the scalability of mobile-agent, information-retrieval applications.

4 D'Agents Scalability

Previous studies [GCKR01] indicate that mobile agents can provide a significant performance benefit, particularly when a service does not directly support the client's task. These results, however, were for a scenario that involved a single agent and an unloaded service. Given that mobile agents are typically CPU bound, while the corresponding client/server implementations are often network bound, it is important to understand how the two solutions scale relative to each other as the service load increases. The scalability can be better or worse depending on the network speed, the client and server computing power, and many other factors. In this section, we present scalability results for mobile agents in an information-retrieval task where more and more clients generate simultaneous requests. These scalability results are a snapshot from an ongoing and much larger scalability study.

As we discuss the performance of the mobile-agent approach, it is important to remember our underlying assumption—the service developers were unable to anticipate and directly support the needs of all clients, and instead elected to provide a general but low-level interface. If the service developers had anticipated a specific client's needs, they could have implemented an operation to support exactly that client task, and invoking this builtin operation from across the network would always be faster than sending a mobile agent. In information-retrieval applications, however, predicting the needs of future clients or extending a service as new clients are developed can be both difficult and time-consuming. Mobile agents are a way to achieve some measure of efficiency, even when the information-retrieval task requires a sequence of low-level service operations.

Under this assumption, we show below that mobile agents often outperform traditional client/server solutions, particularly in low-bandwidth conditions, and scale well as the number of clients increases. When comparing mobile-agent and client/server solutions, we focus on network bandwidth, the pass ratio, and the number of client machines, where the pass ratio is the percentage of data that passes an application-specific filter after some initial and more general query. Other factors, such as network reliability and latency, client and server processing speeds, data size, task granularity, and choice of agent programming language, are equally important, but are left for future papers. Thus the experiments in this paper describe an important but partial section of the information-retrieval performance space, and significant additional experiments are needed to characterize the rest. Our ultimate goal is to develop predictive performance models for information-retrieval applications, along with experimental methods to rapidly characterize application behavior in new network domains. The experiments here, along with the modeling work in [KCG+02], are first steps toward this goal.

Several other research groups have modeled or experimentally analyzed mobile-agent performance. On the modeling side, Straßer and Schwehm [SS97] compare Remote Procedure Calls (RPC) and mobile agents in data-processing applications; Küpper and Park [KP98] compare mobile-agent and client/server implementations of a telecommunications signaling application; Picco, Fuggetta and Vigna [Pic98b, FPV98] compare code-on-demand, remote-evaluation, mobile-agent and client/server implementations of a network-management application; and Puliafito et al. [PRS99] use Petri nets to compare remote evaluation, mobile agents and client/server solutions for data-processing applications. With an experimental approach, Samaras et al. [SDSL99] compare different strategies for accessing a Web server, and Johansen [Joh98] compares mobile-agent and client/server implementations of an image-processing application. Of this work, Johansen's image-processing application is the closest to the application we measure below. Few, if any, papers examine the scalability of the system as the number of clients increases.

4.1 Our experiments

We implemented client/server and mobile-agent versions of a simple information-retrieval application and used query completion times as the performance metric for comparing them. The application is a simplified version of the information-retrieval tasks that the soldiers in the field perform against the military databases in Section 2. Specifically, the application allows a user to retrieve a set of "interesting" documents from a document server. The server's interface allows traditional keyword queries, but the application narrows the set of documents returned from a keyword query by searching for specific phrases in the documents.

There are several things to note about this application. First, it is tempting to argue that the retrieval and filtering operations are *too* simple. After all, any modern Internet search engine, such as Google[3] and Excite[4], allows the user to search Web pages for phrases as well as individual keywords. With such search engines, there is no need for the application to perform a separate filtering step to identify which documents contain a particular phrase. Our intention, however, was not to measure the performance of a domain-specific retrieval application, but instead to measure the performance of a simplified application that (1) allowed straightforward control over parameters such as data size, filtering ratios, and so on, and (2) was *representative* of many domain-specific applications. Our simplified application fills both roles quite well, and in particular, is an effective stand-in for any application that uses a service's built-in operations to obtain a set of candidate documents, but then performs a single pass over the candidate documents, either to reduce the number of candidates or the size of each candidate. For the military news database, for example, the client application used built-in service operations to perform standard keyword and phrase searches, but then automatically constructed summaries of the candidate documents. For the phone database, the client application used built-in operations to retrieve phone calls that were made at the right time of day and to or from the right phone number, but then further weighted the documents according to the topics and names mentioned in each one.

Second, the agents in the mobile-agent version of the application make only a single migration from client to server machine (and back again). The decision of whether to make a single initial migration, however, is the first and most important decision in any mobile-agent application. Can the client code efficiently interact with the service from its home machine, or should it send some part of itself to a more attractive network location? Analyzing the scalability of the single-migration case is an essential step toward an understanding of mobile-agent performance.

Finally, as we will see, the performance curves turn out to be straightforward, and can be described easily with simple mathematical equations. In fact, as long as the agent system itself is not overloaded, the query completion times scale (roughly) linearly according to the amount of data transferred over the network, which in turn is proportional to the number of client machines making simultaneous queries.

[3] http://www.google.com

[4] http://www.excite.com

Although this result conforms to our intuition, it is not at all obvious that our intuition would have been correct even in this simplest case, simply due to the wide range of performance overheads that are present in all mobile-agent systems. Such overheads range from setting up an execution environment for an incoming agent to interpreting the Java bytecodes that make up the agent. It is important to verify that a real mobile-agent system does meet our scalability expectations, so that application designers can confidently analyze situations for which experiments are difficult or impossible.

The agent version of our application uses a parent agent to launch from one to twenty identical agents, each to a separate workstation. Each agent acts as a client in an information-retrieval task in which it jumps to a central server, does a single initial query against a document collection to get a list of candidate documents, then searches the text of the candidate documents for a particular substring, and then returns to its home machine with the matching documents. The time it takes for the processing and jumps of each agent is calculated and reported to the parent agent where they are averaged and summarized as a performance measurement. By varying the number of client agents from one to twenty a curve can be constructed showing how mobile-agent performance scales with number of clients. By using one workstation per client agent, we ensure that we are measuring the scalability of the server and network, rather than of the client machine. Figure 3 shows a Java agent that embodies much of the functionality of our application. It is a good example of a D'Agents Java agent that uses the `jump` method to move between machines.

Since our goal is to explore the performance of *mobile* code, the mobile-agent version always sends an agent to the document collection, rather than interacting with the document collection from across the network when network conditions are particularly good. The results of our performance experiments would provide important information, however, to actual applications that need to decide whether to use a proxy site and whether to spawn child agents.

In the client/server version of the application, the client opens a TCP/IP connection over which it sends queries, using a simple protocol, to a multithreaded server (implemented as multiple independent server tasks for simplicity). The server performs this initial query to get a list of candidate documents and then sends all the resulting documents back to the client. The client then does a substring search on the documents to select the final results. The clients and servers keep track of the connection duration (including data transmission times) and substring search times as a performance measure.

We ran our experiments on twenty-one identical Linux computers.[5] The one to twenty client machines were connected to the one document-server machine through a 100 Mbps switch and hub, a hardware bandwidth manager set to 7 or 10 Mb/s,[6] or a machine running the DummyNet bandwidth manager,[7] depending on the desired network bandwidth. Thus the client machines share the given bandwidth. The authentication features of D'Agents were turned off, so both the agents and the client/server messages were unencrypted and unsigned.

The text of each document is exactly 4 kilobytes in size, but each retrieved document is 4.2 kilobytes, since the retrieved document includes a URL, a unique integer identifier, and other meta-information. In the agent case, each retrieved document actually takes up 4.5 kilobytes, an additional 0.3 kilobytes, due to the details of how D'Agents packages up agents for transmission.

The Java class file that contains the mobile Java agent is 3.2 kilobytes. 3.2 kilobytes of code may

[5]More specifically, each computer was a VA Linux VarStation 28 Model 2871E, which has a 450 MHz Pentium II, 512KB ECC L2 Cache, and 256MB RAM. Each computer ran Linux 2.2.14. We compiled all C/C++ code (including the agent server and interpreters) with GNU gcc version 2.91.66 (egcs-1.1.12) with an optimization level of 2.

[6]For the 7.0 and 10.0 Mbps network, the 100 Mbps hub was replaced with an ET/R1700i-BW-5 bandwidth manager, manufactured by Emerging Technologies, which can enforce preset bandwidth limits.

[7]For the 1.0, 2.0, and 3.0 Mbps network, the 100 Mbps hub was replaced with a machine running DummyNet, which is a software package that can simulate (and enforce) bandwidth limitations, network latencies, packet loss, and multi-path effects [Riz00], on full-duplex traffic between the client and server machines. For our tests, we did not introduce any packet losses or multi-path effects, and simply used DummyNet as a throttle on the underlying physical network. DummyNet is available at `http://www.iet.unipi.it/~luigi/ip_dummynet/`.

```
class QueryAgent extends AgentEntryPoint {
  public void run (Agent a) {
         // Submit the desired number of queries, sending results to parent.
    for (int queries = 0; queries < numQueries; queries++) {
           // jump to the location of the document collection
       try {
          a.jump (documentMachine, TIMEOUT_SECS);
       } catch (Exception e) {
          Message message = new Message (0, "Unable to jump to server" +
             e.getMessage());
          a.send (a.getRootId(), message, TIMEOUT_SECS);
          a.end(TIMEOUT_SECS);
       }
           // the agent is on the document-collection machine - first make
           // the initial query to get a list of candidate document ids.
       int dids[] = makeInitialQuery (query, desiredInitialDocuments);
            // calculate the number of documents that will turn out to
           // be relevant once we examine the document texts
       int relevantDocuments = (int)(desiredInitialDocuments*relevancePercent);
            // examine the document texts to determine which are relevant
       try {
           examineDocumentTexts(dids, relevantDocuments, documentDirectory);
       } catch (Exception e) {
           Message message = new Message (1, "examineDocumentTexts error: " +
              e.toString());
           a.send (a.getRootId(), message, TIMEOUT_SECS);
           a.end (TIMEOUT_SECS);
       }
          // jump back to the Client home machine
       a.jump (clientMachine, TIMEOUT_SECS);
           // send some information to the parent
       Message message = new Message (0, results);
       a.send (a.getRootId(), message, TIMEOUT_SECS);
    } // end of QUERIES loop.
    a.end (TIMEOUT_SECS);  // done
  } // end of run()
}
```

Figure 3: A simple Java agent (with some of the initialization code and method definitions removed for clarity). This agent migrates back and forth between a client and a server machine, performing an information-retrieval task on the server machine and returning the results to the client machine.

seem small, but remember that the agent's code is primarily a wrapper around the database operations. The agent performs only a modest amount of application-specific processing. The state image during migration, which includes the 3.2 kilobytes of code, is approximately 16.5 kilobytes, since (1) the state image includes variables and control state, not just code, and (2) the D'Agents format for Java state images is not particularly compact. In other words, when the agent migrates to the database machine, its size is 16.5 kilobytes, and when the agent returns to the home machine, its size is 16.5 kilobytes plus 4.5 kilobytes for each document that contained the desired substring. Of course, the bandwidth utilization would include Ethernet, IP and TCP headers, so it is larger than just the sum of the query, agent and document sizes.

The C++ client sends a 256-byte query across the TCP/IP connection, and then downloads each 4.2-kilobyte candidate document across that same connection. Again, the full bandwidth utilization would include Ethernet, IP and TCP headers.

In addition, *each client* generates one query every two seconds, and each query always produces sixty candidate documents. The client/server approach sends all sixty documents to the client. The agents filter through the documents on the server (to correctly represent the CPU time that filtering uses), but then choose an arbitrary 5% or 20% of the documents (to allow straightforward repeatability). One query every two seconds, sixty candidate documents, and a 5% or 20% pass ratio are simply illustrative points in the performance space that we have explored, although these numbers do roughly correspond to the news and phone-call retrieval tasks from our military application.

Figure 4 shows the scalability of the application in a 10 Mbps network. The x-axis is the number of client machines generating queries simultaneously, and the y-axis is the average time in milliseconds per query. We show two cases for mobile agents, one case in which 20% of the documents pass the application-specific filter, and a second case in which only 5% of the documents pass the filter. The client/server approach has identical performance in the two cases, of course, because 100% of documents are transmitted across the network regardless.

When interpreting these results, there are three important points to note about the experimental setup. First, to simplify the mobile-agent control program, there is actually only one mobile agent created per client machine. This single mobile agent migrates back and forth between the document server and the client machine, performing the query on the document machine, and waiting on the client until it is time for the next query. This approach does eliminate the small overhead of creating a new agent for each query, but since this overhead is local to each client machine, i.e., no contention with other agents, it has no significant effect on the results.

Second, the control program on each client machine *attempts* to generate one query every two seconds according to a uniform random distribution, but always waits for the previous query to complete before starting the next query. If the previous query takes two seconds or longer, the application generates the next query immediately. Thus, if the network or server becomes overloaded, we would see the overload as a drop in the query generation rate.

Finally, the routine to obtain a list of candidate documents is actually a stub routine that returns a hardcoded list of candidate documents. The real routine would use significantly more CPU time than the stub routine. This routine, however, is used exactly once per query and is exactly the same in both the client/server and mobile-agent approaches, since it is an operation built into the database service. By using the stub routine, we are simply assuming that the CPU capacity of our test machines is the CPU capacity *left over* after invoking the real routine for each query. Of course, using the real routine would change the query completion times, but by the same absolute amount in both client/server and mobile-agent cases.

With these three points in mind, Figure 4 is straightforward. The CPU on the document server never reaches maximum capacity in any of the experiments presented here,[8] and thus the behavior in Figure 4 is due to network effects.

[8] We monitored CPU load during the scalability experiments.

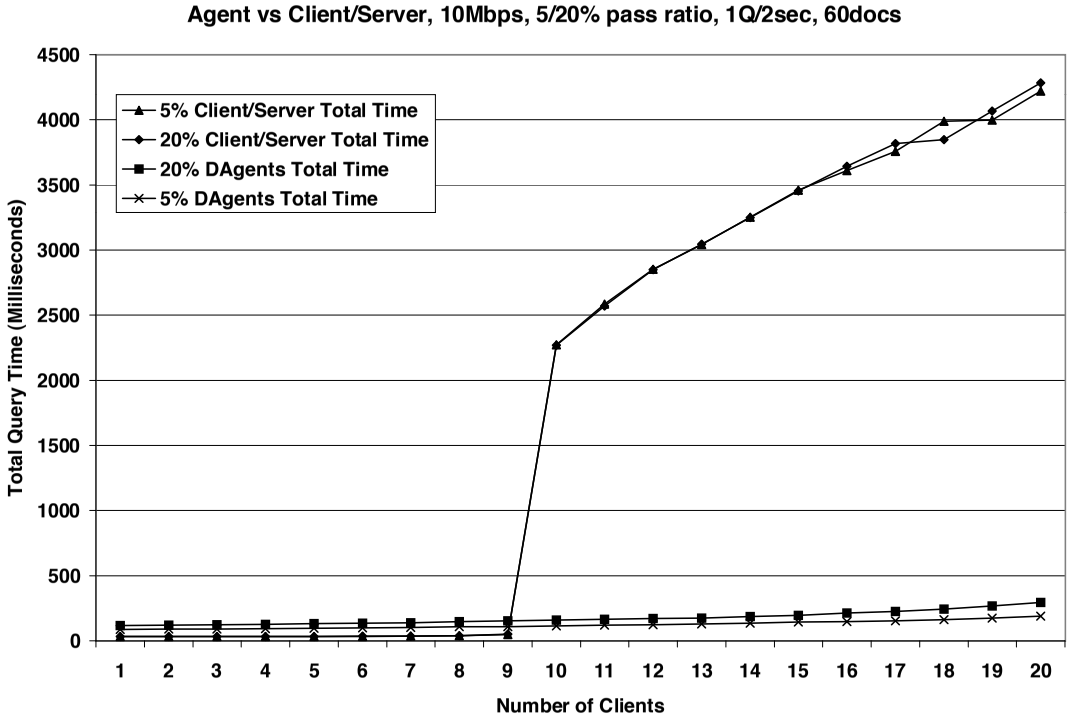

Figure 4: Scalability of the client/server and mobile-agent solutions in a 10 Mbps network. The x-axis is the number of client machines that are generating queries simultaneously, and the y-axis is the average query completion time. Each data point is the average across 100 to 500 queries per client. The number of queries was adjusted to the minimum needed to produce consistent results with a minimum of computation time.

In the client/server case the query time is dominated by the network; adding clients causes a linear increase in the network load and a linear increase in the query-completion time. With nine clients, the network has nearly reached its capacity, and at ten clients the time per query jumps significantly. With ten or more clients there is not enough network bandwidth for all of the documents.[9] Here, the query time exceeds two seconds, indicating that we have dropped below our desired query generation rate. Were the clients unable to slow their query generation rate, 10 or more clients would be impossible. Instead, as more clients arrive each client slows its query generation to match the capacity and the query times again climb linearly.

The mobile-agent case, which transmits only the documents that pass the application-specific filter, scales much better. Each query completes in less than two seconds across the entire client range.

Figure 5 shows that the situation is quite different for a 1.0 Mbps network (note the change in y-axis). The mobile-agent approach still does better than the client/server approach, but with the reduced bandwidth, the mobile-agent approach eventually uses up the entire available bandwidth as well. Once this happens, the agent completion times scale according to the total transmission time needed for the additional documents and agent code. Of course, the agents are transmitting a filtered document set, rather than every candidate document, so the mobile-agent times still grow much more slowly than the client/server times.

Figure 6 shows the completion times for multiple network speeds on a single graph, including the data from Figures 4 and 5 as well as data for a 2 Mbps network. It is reassuring to note how closely the 10 Mbps client/server curve matches the 2 Mbps agents curve, because they transmit roughly the same amount of data across the network. The slight difference between the curves reflects the overhead of transporting and executing the agents.

Figure 7 shows the *ratio* between client/server and mobile-agent completion times for different network speeds, 100, 10, 7, 3, 2 and 1 Mbps with a 20% pass ratio. A ratio of 2, for example, means that queries take twice as long with the client/server approach.

The ratio graph illustrates two important points. First, the client/server approach does better than the mobile-agent approach over a 100 Mbps network, simply because transmitting the intermediate results over the network is faster than executing the agent on the server. Second, we clearly see the mobile-agent approach reaching the network bandwidth limit in the slower networks. As the number of clients increase, the ratios peak, drop, and then level off. The peak actually occurs before we reach the maximum channel bandwidth, so competition for the network, not just overloading of the network, is playing a key role. In the 7 and 10 Mbps cases we do not see the curves level off within our test range of 20 clients. We are still experimenting with the system to determine why the 7 Mbps network has better ratios than the 10 Mbps ratios.

Figure 8 shows the ratio between client/server and mobile-agent completion times for different network speeds, 100, 10, 3, 2 and 1 Mbps with a *5%* pass ratio. This parameter affects only the agent implementation and allows them correspondingly better agent performance.

Figure 9 and Figure 10 show the same ratio information as in Figure 7 and Figure 8 displayed as a two-dimensional graph of the overall performance space.

The performance ratio is largest when the client/server approach is overloading the network while the bandwidth saved in a mobile-agent approach is sufficient for the agents to avoid overloading the network. In the fastest network, the agent's bandwidth savings do not significantly affect performance, and in high-load situations on slower networks both methods struggle and the performance ratio is low.

In summary, mobile agents scale well under many network and application conditions. In situations where the service developer cannot directly support all client tasks, mobile agents can provide significantly better performance than traditional client/server approaches. Mobile agents do not provide better performance in all cases, however, and the application developer must carefully consider whether her application

[9] 10 sets of 60 candidate documents is 2700 kilobytes of application-level data. Ignoring protocol headers, the network would need to transmit 10.5 Mbps to sustain a rate of 0.5 queries per second per client.

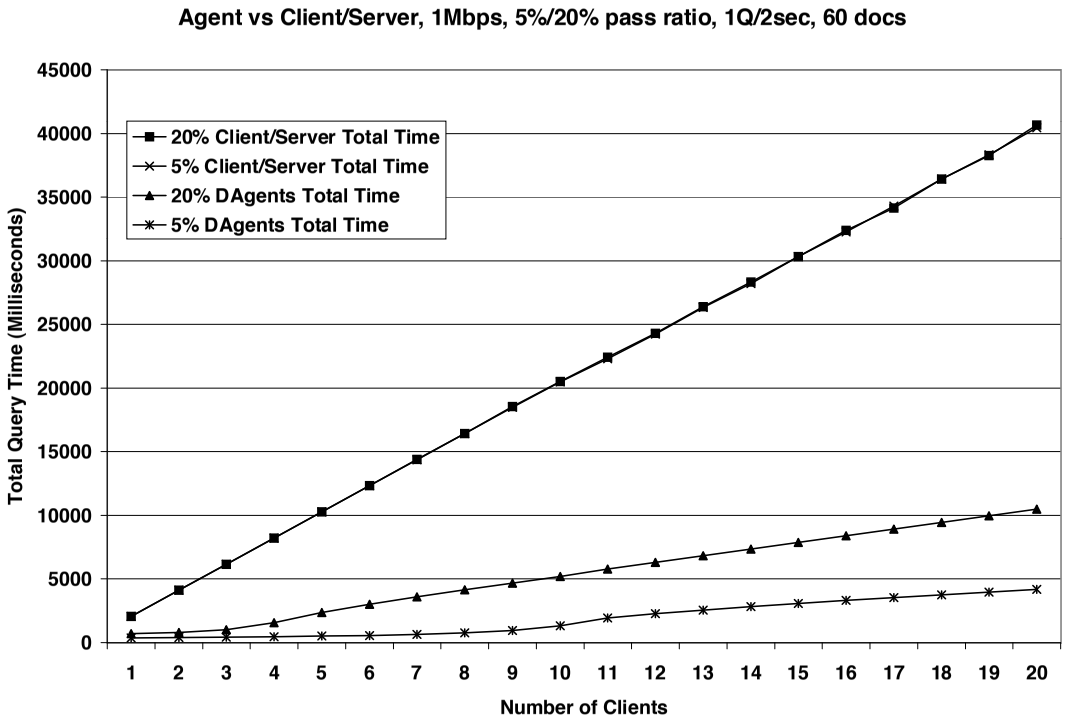

Figure 5: Scalability of the client/server and mobile-agent solutions in a 1.0 Mbps network. The experiment behind this graph is the same as in Figure 4 except that we switched from a 10 Mbps to 1.0 Mbps network. Note the values on the y-axis are an order of magnitude larger than in the previous graph. Each point is an average across 1000 queries per client.

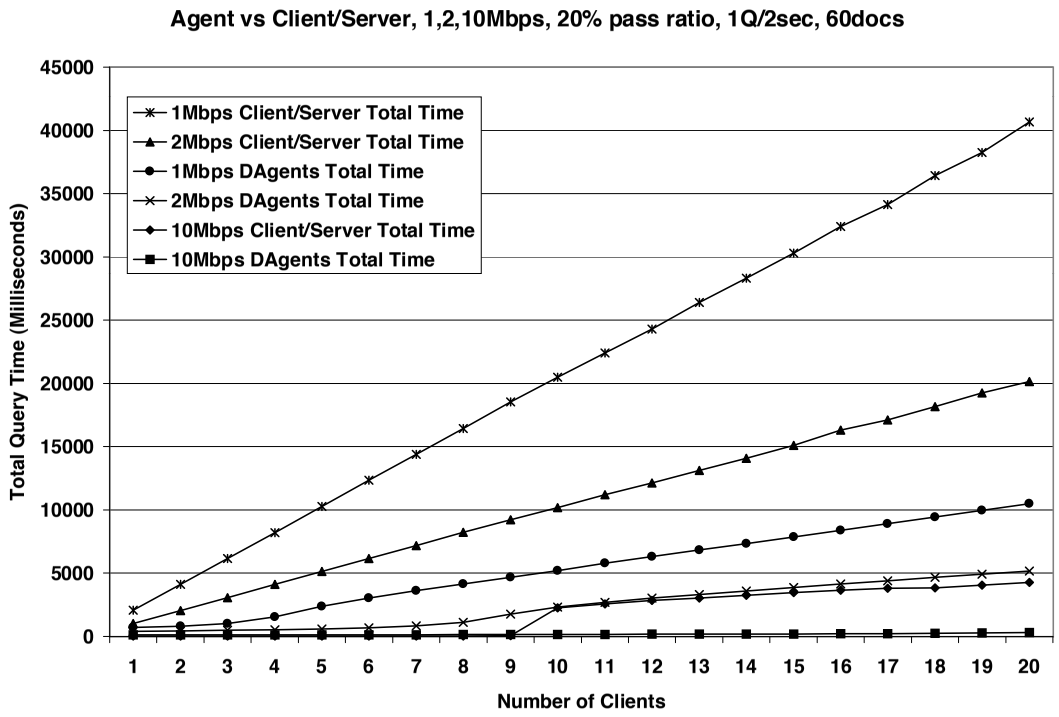

Figure 6: Scalability at different network speeds with a 20% pass ratio. This graph includes results for a 2.0 Mbps network, in addition to the 10 Mbps and 1.0 Mbps networks. Each point is an average across 1000 queries per client. The order of the entries in the key is the top-to-bottom order of the lines on the graph.

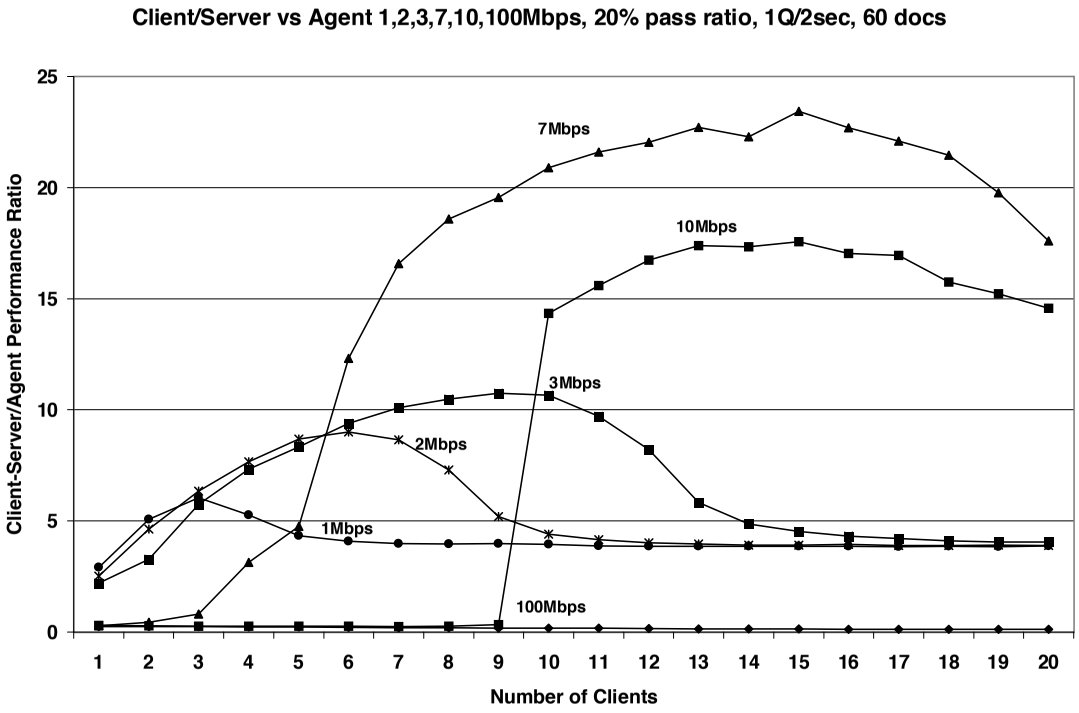

Figure 7: Ratio of client/server to mobile-agent performance. This graph provides an interesting view on system scalability. Each line shows the ratio between the client/server and mobile-agent query times for a particular network speed. A ratio of 2, for example, means that the average client/server query takes twice as long as the average mobile-agent query. The pass ratio was 20% in all cases.

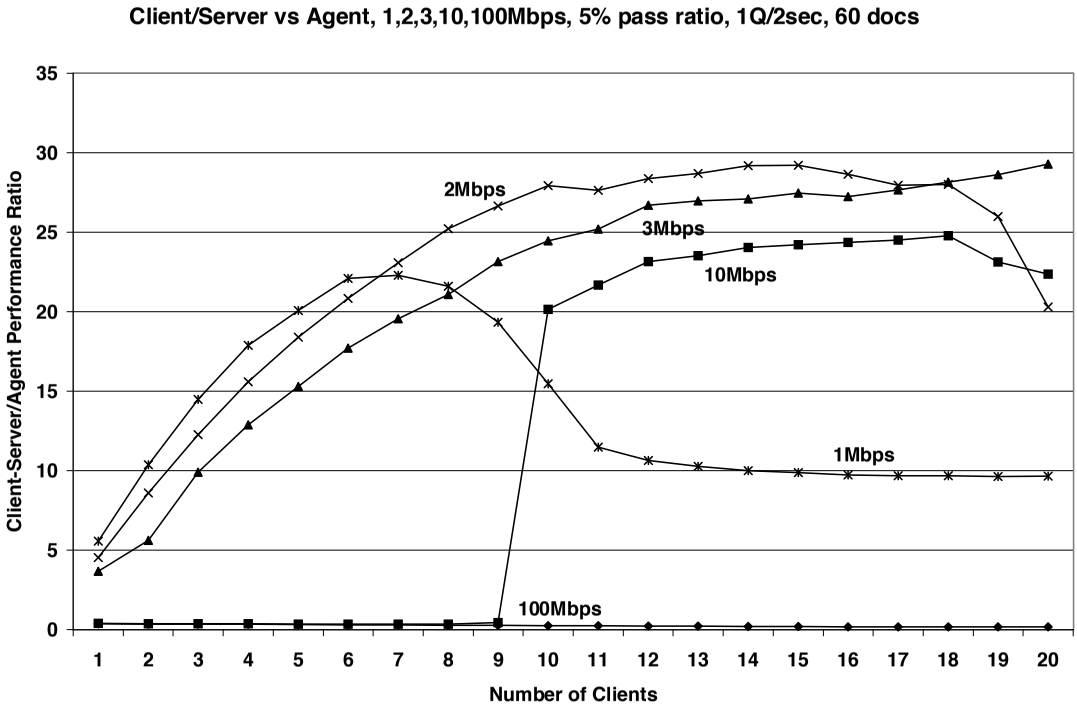

Figure 8: Ratio of client/server to mobile-agent performance. Each line shows the ratio between the client/server and mobile-agent query times for a particular network speed. The pass ratio was 5% in all cases.

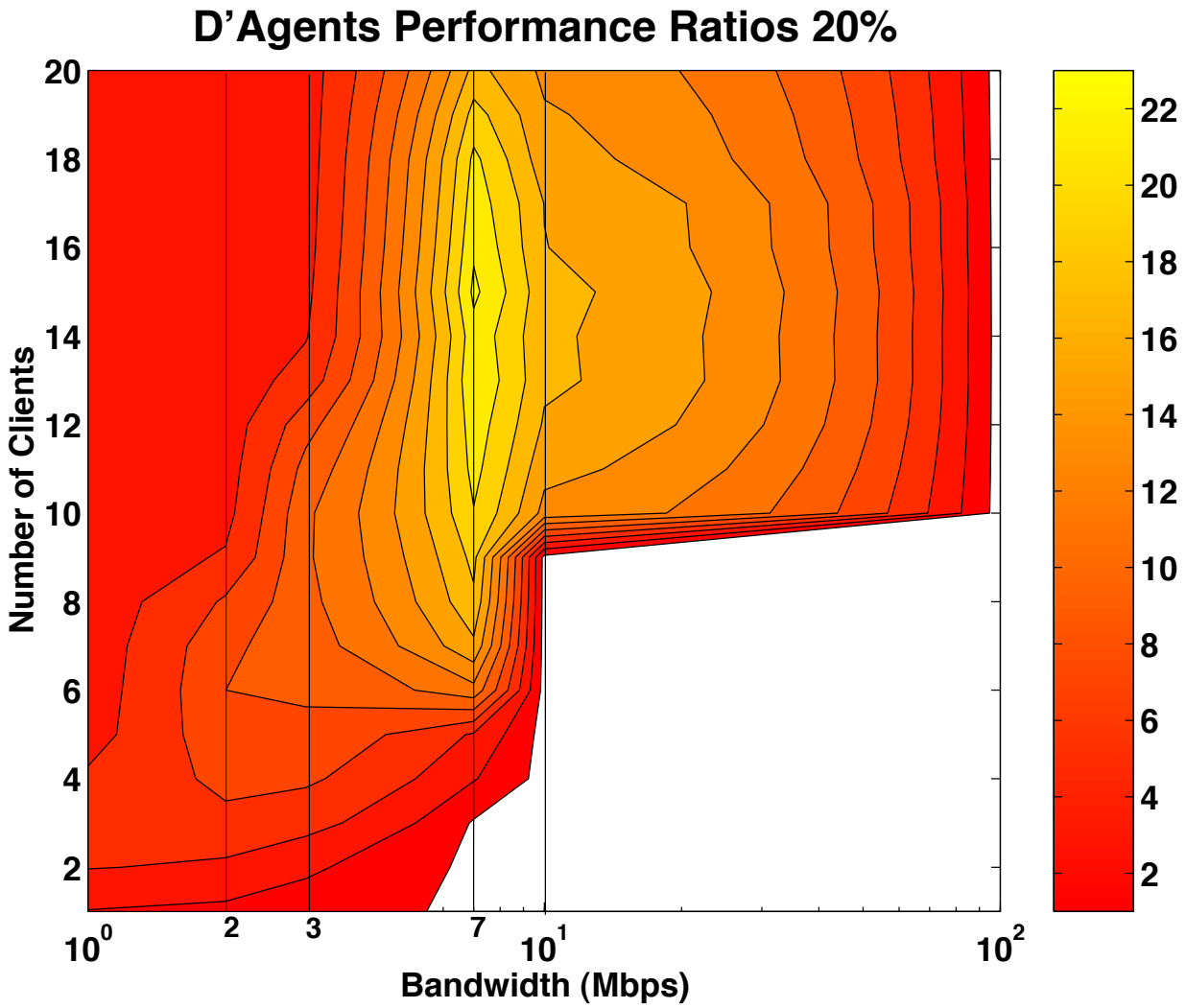

Figure 9: Ratio of client/server to mobile-agent performance. The shading represents the ratio between the client/server and mobile-agent query times for each particular network speed and number of clients. The white area is where the ratio is less than one and hence the client/server approach is better than the mobile-agent approach. Measurements were taken for 1,2,3,7,10, and 100Mbps for one to twenty clients, and interpolated at other points. The pass ratio was 20% in all cases.

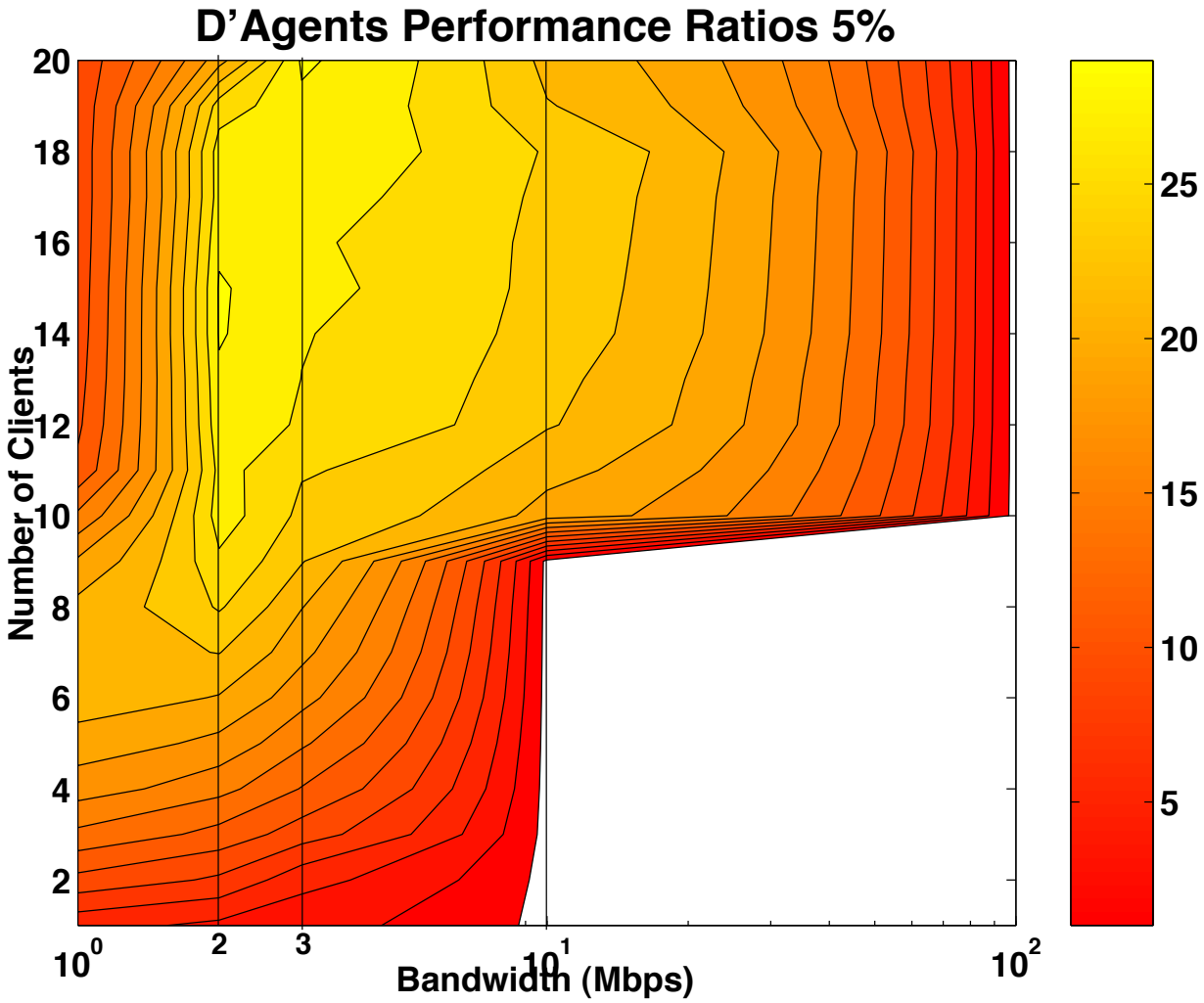

Figure 10: Ratio of client/server to mobile-agent performance. The shading represents the ratio between the client/server and mobile-agent query times for each particular network speed and number of clients. The white area is where the ratio is less than one and hence the client/server approach is better than the mobile-agent approach. Measurements were taken for 1,2,3,10, and 100Mbps for one to twenty clients, and interpolated at other points. The pass ratio was 5% in all cases.

falls in the right section of the mobile-agent performance space. Significant additional work is needed to fully map this space, and give the developer effective guidelines as to exactly when mobile agents should (and should not) be used. In earlier experiments, for example, the performance of our information-retrieval agents was significantly worse when the agents were implemented in Tcl [GCKR01]. In particular, the Tcl agent could not construct the list of relevant documents quickly enough, and thus transferring all the candidate documents across a 10 Mbps network was faster than sending the Tcl agent to the document machine. The application developer must take the efficiency of the available languages into account, or he might end up with far less than the expected performance. Our next set of experiments focuses on retrieval tasks in which the agent's analysis task is more intensive or in which the agent is written in a less efficient interpreted language, and on scenarios with more client machines and a faster query-generation rate. These ongoing experiments are essential for understanding mobile-agent performance in our larger military application, and for developing and validating a predictive model that can cover more general information-retrieval applications.

5 Lessons Learned and Conclusions

In this paper, we described the D'Agents system, which supports multiple languages, communication, security, resource control, and strong migration, and quantified its performance in distributed information-retrieval applications. Our experience with the use of D'Agents in a large application that supports soldiers in the field shows that the mobile-agent paradigm works. Mobile agents are a useful approach for many distributed applications. A mobile agent does not require a permanent connection to the computer from which it was launched. The agent does not care if the computer that launched it becomes disconnected because it can pursue its search operations independently. Moreover, agents are an efficient paradigm for information processing and transfer over wireless networks, which typically have low bandwidth and high latency. By migrating to the location of an electronic resource, an agent can access the resource locally and eliminate costly data transfers over congested networks. This reduces network traffic and improves data delivery, because it is often faster and cheaper to send a small agent to a data source than to send all the intermediate data to the requesting site.

In short, we see six key benefits of mobile agents: (1) conservation of bandwidth, (2) reduction in latency, (3) reduction in total completion time, (4) support for disconnected operation in mobile computing, (5) load balancing, and (6) dynamic code deployment. This paper focused on bandwidth conservation and reduction in completion times, and an earlier paper [GCKR01] discussed the other four reasons. Although none of these six strengths are unique to mobile agents, no competing technique shares all six. Thus, the true strength of mobile agents is that they provide a general-purpose framework in which a range of distributed applications can be implemented easily and efficiently.

First- and second-year undergraduate students developed many of the D'Agent applications even though they had never taken courses in distributed or network computing. Our experience with these undergraduates suggests that mobile agents are easier to understand than message- or RPC-based techniques, emphasizing the potential for mobile agents to be a uniform paradigm for developing distributed applications. Agents also remove the need for distributed applications to have their own control language, and multiple-language systems such as D'Agents, Tacoma [JvRS95] and Ara [PS97] demonstrate that the mobile-agent paradigm is independent of language choice.

D'Agents is an excellent prototyping tool for distributed applications, but several challenges must be overcome to fully realize the six mobile-agent benefits. Specifically, the protocol overhead of the mobile-agent paradigm is large, which has significant performance consequences. For example, the experiments described in Section 4 show that due to the migration and interpretive overhead in our system (which is the case for all current systems), an agent will outperform a traditional client/server solution (in terms of total

completion time) only if a query retrieves a sufficiently large number of documents and the agent is able to filter most of them on the server side of the connection. This observation is particularly true if the client and server machines are connected with a medium-speed or faster network (such as a 10 Mbps Ethernet). From a scalability viewpoint, even though the mobile agent approach does better than the client/server approach when there are more clients in the system, it is easy to see that scalability becomes an issue for mobile agents as well when the maximum channel bandwidth is reached. In addition, mobile agents are more likely to overload the CPU on the server machine, and we must understand whether bandwidth or the server CPU will be the limiting factor in any given application.

Thus, while many difficult problems have been solved for mobile agents, many challenging research problems remain. We believe that the key research areas are scalability, security, and program verification, and our projects focus on scalability, the first of these three issues. Our future goals for D'Agents are to (1) support agent migration that is only a small factor slower than an RPC call that transmits an equivalent amount of data, (2) execute agents nearly as quickly as if they were compiled (directly) into native code, (3) integrate a high-performance encryption library, and, most importantly, (4) expand on the scalability experiments and develop general models for mobile-agent performance, particularly for information-retrieval applications. Such models will help a designer decide whether to use mobile agents at all, and will help mobile agents dynamically decide what to do given current network and machine conditions.

Availability

D'Agents software and documentation can be downloaded at the D'Agents Web site.[10] Note that only the D'Agents server and the Tcl and Scheme modules are available through the web site, since our licensing agreement with Sun Microsystems prevents us from making a completely open release of the Java module (which required modifications to the standard Java virtual machine to support strong mobility). Readers interested in the Java module should send e-mail to the first author.

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; and to the anonymous reviewers for their invaluable comments and suggestions. Finally, many thanks to the developers of other mobile-agent systems, who in their correspondence helped us to better understand the capabilities and operation of their systems: Gian Pietro Picco (μCODE), Holger Peine (Ara), Niranjan Suri (NOMADS), and David Wong (Concordia).

References

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

[AS97] Anurag Acharya and Joel Saltz. Dynamic linking for mobile programs. In Jan Vitek and Christian Tschudin, editors, *Mobile Object Systems: Towards the Programmable Internet*, volume 1222 of *Lecture Notes in Computer Science*, pages 245–262. Springer-Verlag, 1997.

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

[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.

[BGM+99] Brian Brewington, Robert Gray, Katsuhiro Moizumi, David Kotz, George Cybenko, and Daniela Rus. Mobile agents for distributed information retrieval. In Matthias Klusch, editor, *Intelligent Information Agents*, chapter 15, pages 355–395. Springer-Verlag, 1999.

[BHRS98] Joachim Baumann, Fritz Hohl, Kurt Rothermel, and Markus Straßer. Mole— concepts of a mobile agent system. *World Wide Web Journal*, 1(3):123–137, 1998.

[BJP+00] Ladislau Bölöni, Kyungkoo Jun, Krzysztof Palacz, Radu Sion, and Dan C. Marinescu. The Bond agent system and applications. In *Proceedings of the Second International Symposium on Agent Systems and Applications and Fourth International Symposium on Mobile Agents (ASA/MA2000)*, volume 1882 of *Lecture Notes in Computer Science*, pages 99–112, Zurich, Switzerland, September 2000. Springer-Verlag.

[BM98] Markus Breugst and Thomas Magedanz. Mobile agents— enabling technology for active intelligent network implementation. *IEEE Network Magazine*, 12(3):53–60, August 1998. Special Issue on Active and Programmable Networks.

[BMcI+00] Jonathan Bredin, Rajiv T. Maheswaran, Çagri Imer, Tamer Başar, David Kotz, and Daniela Rus. A game-theoretic formulation of multi-agent resource allocation. In *Proceedings of the Fourth International Conference on Autonomous Agents*, pages 349–356. ACM Press, June 2000.

[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.

[CCMG98] Wilmer Caripe, George Cybenko, Katsuhiro Moizumi, and Robert Gray. Network awareness and mobile agent systems. *IEEE Communications Magazine*, 36(7):44–49, July 1998.

[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.

[FÖ98] Stefan Fünfrocken. Transparent migration of Java-based mobile agents. In *Proceedings of Mobile Agents: 2nd International Workshop (MA'98)*, volume 1477 of *Lecture Notes in Computer Science*, pages 26–37, Stuttgart, Germany, September 1998. Springer-Verlag.

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

[GCKR01] Robert S. Gray, George Cybenko, David Kotz, and Daniela Rus. Mobile agents: Motivations and state of the art. In Jeffrey Bradshaw, editor, *Handbook of Agent Technology*. AAAI/MIT Press, 2001. Accepted for publication. Draft available as Technical Report TR2000-365, Department of Computer Science, Dartmouth College.

[GKCR97] Robert Gray, David Kotz, George Cybenko, and Daniela Rus. Agent Tcl. In William Cockayne and Michael Zyda, editors, *Mobile Agents: Explanations and Examples*, chapter 4, pages 58–95. Manning Publishing, 1997. Imprints by Manning Publishing and Prentice Hall.

[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.

[GKN+97] Robert Gray, David Kotz, Saurab Nog, Daniela Rus, and George Cybenko. Mobile agents: The next generation in distributed computing. In *Proceedings of the Second Aizu International Symposium on Parallel Algorithms and Architectures Synthesis (pAs '97)*, pages 8–24, Fukushima, Japan, March 1997. IEEE Computer Society Press.

[Gra96] Robert S. Gray. Agent Tcl: A flexible and secure mobile-agent system. In *Proceedings of the 1996 Tcl/Tk Workshop*, pages 9–23. USENIX Association, 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.

[Gra00] Robert S. Gray. Soldiers, agents and wireless networks: A report on a military application. In *Proceedings of the Fifth International Conference and Exhibition on the Practical Application of Intelligent Agents and Multi-Agents*, Manchester, England, April 2000.

[HBSG99] Ophir Holder, Israel Ben-Shaul, and Hovav Gazit. Dynamic layout of distributed applications in FarGo. In *Proceedings of the 21st International Conference on Software Engineering (ICSE'99)*, pages 163–173, Los Angeles, May 1999.

[Joh98] Dag Johansen. Mobile agent applicability. In *Proceedings of the Second International Workshop on Mobile Agents (MA '98)*, volume 1477 of *Lecture Notes in Computer Science*, pages 80–98. Springer-Verlag, 1998.

[JSvR98] 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.

[JvRS95] D. Johansen, R. van Renesse, and F.B. Schneider. An introduction to the TACOMA distributed system version 1.0. Technical Report 95-23, University of Tromsø, June 1995.

[KCG+02] David Kotz, George Cybenko, Robert S. Gray, Guofei Jiang, Ronald A. Peterson, Martin O. Hofmann, Daria A. Chacón, Kenneth R. Whitebread, and James Hendler. Performance analysis of mobile agents for filtering data streams on wireless networks. *Mobile Networks and Applications*, 7(2), March 2002. Accepted for publication.

[KGN+97] David Kotz, Robert Gray, Saurab Nog, Daniela Rus, Sumit Chawla, and George Cybenko. Agent Tcl: Targeting the needs of mobile computers. *IEEE Internet Computing*, 1(4):58–67, July/August 1997.

[KK98] Brad Karp and H. T. Kung. Dynamic neighbor discovery and loop-free, multi-hop routing for wireless, mobile networks. Draft. Available at `http://www.eecs.harvard.edu/~karp/aprl.ps`, May 1998.

[KP98] Axel Küpper and Anthony S. Park. Stationary vs. mobile user agents in future mobile telecommunications networks. In *Proc. of the Second Int'l Workshop on Mobile Agents (MA '98)*, volume 1477 of *Lecture Notes in Computer Science*, pages 112–123. Springer-Verlag, September 1998.

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

[MP99] Amy L. Murphy and Gian Pietro Picco. Reliable communication for highly mobile agents. In *Proceedings of the First International Symposium on Agent Systems and Applications and Third International Symposium on Mobile Agents (ASA/MA99)*, pages 141–150, Palm Springs, California, October 1999. IEEE Computer Society Press.

[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)*, pages 1–31, Austria, July 1998. ACTA Press.

[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.

[OBJ97] ObjectSpace Voyager core package technical overview. ObjectSpace, Inc., December 1997. Version 1.0.

[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.

[Pic98a] Gian Pietro Picco. μCODE: A Lightweight and Flexible Mobile Code Toolkit. In *Proceedings of the Second International Workshop on Mobile Agents*, volume 1477 of *Lecture Notes in Computer Science*, pages 160–171, Stuttgart, Germany, September 1998. Springer-Verlag.

[Pic98b] Gian Pietro Picco. *Understanding, Evaluating, Formalizing, and Exploiting Code Mobility*. PhD thesis, Politecnico di Torino, Italy, February 1998.

[PRS99] A. Puliafito, S. Riccobene, and M. Scarpa. An analytical comparison of the client-server, remote evaluation and mobile agents paradigms. In *Proceedings of the First International Symposium on Agent Systems and Applications and Third International Symposium on Mobile Agents (ASA/MA99)*, pages 278–292, Palm Springs, California, October 1999. IEEE Computer Society Press.

[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*, volume 1219 of *Lecture Notes in Computer Science*, pages 50–61, Berlin, Germany, April 1997. Springer-Verlag.

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

[Riz00] Luigi Rizzo. dummynet. Available at `http://www.iet.unipi.it/~luigi/ip_dummynet/`, 2000.

[SBB+00] Niranjan Suri, Jeffrey M. Bradshaw, Maggie R. Breedy, Paul T. Groth, Gregory A. Hill, and Renia Jeffers. Strong mobility and fine-grained resource control in NOMADS. In *Proceedings of the Second International Symposium on Agent Systems and Applications and Fourth International Symposium on Mobile Agents (ASA/MA2000)*, volume 1882 of *Lecture Notes in Computer Science*, pages 2–15, Zurich, Switzerland, September 2000. Springer-Verlag.

[Sch97] Fred B. Schneider. Towards fault-tolerant and secure agentry. In *Proceedings of the 11th International Workshop on Distributed Algorithms*, volume 1320 of *Lecture Notes in Computer Science*, Saarbucken, Germany, September 1997. Springer-Verlag.

[SDSL99] G. Samaras, M. Dikaiakos, C. Spyrou, and A. Liverdos. Mobile agent platforms for web-databases: A qualitative and quantitative assessment. In *Proceedings of the First International Symposium on Agent Systems and Applications and Third International Symposium on Mobile Agents (ASA/MA99)*, pages 50–64. IEEE Computer Society Press, October 1999.

[SR99] Markus Straßer and Kurt Rothermel. System mechanisms for partial rollback of mobile agent execution. Technical Report TR-1999-10, University of Stuttgart, Stuttgart, Germany, 1999.

[SS97] Markus Straßer and Markus Schwehm. A performance model for mobile agent systems. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, volume II, pages 1132–1140, Las Vegas, July 1997. CSREA.

[SSY00] Takahiro Sakamoto, Tatsurou Sekiguchi, and Akinori Yonezawa. Bytecode transformation for portable thread migration in Java. In *Proceedings of the Second International Symposium on Agent Systems and Applications and Fourth International Symposium on Mobile Agents (ASA/MA2000)*, volume 1882 of *Lecture Notes in Computer Science*, pages 16–21, Zurich, Switzerland, September 2000. Springer-Verlag.

[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.

[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.

[TRV+00] Eddy Truyen, Bert Robben, Bart Vanhaute, Tim Coninx, Wouter Joosen, and Pierre Verbaeten. Portable support for transparent thread migration in Java. In *Proceedings of the Second International Symposium on Agent Systems and Applications and Fourth International Symposium on Mobile Agents (ASA/MA2000)*, volume 1882 of *Lecture Notes in Computer Science*, pages 29–43, Zurich, Switzerland, September 2000. Springer-Verlag.

[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., Sunnyvale, California, 1994.

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

[Whi98] D. Eric White. A comparison of mobile agent migration mechanisms. Senior Honors Thesis, Dartmouth College, June 1998.

[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*, volume 1219 of *Lecture Notes in Computer Science*, pages 86–97, Berlin, Germany, April 1997. Springer-Verlag.

[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.