Mobile agents in distributed information retrieval

Brian Brewington, Robert Gray, Katsuhiro Moizumi,
David Kotz, George Cybenko and Daniela Rus

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

firstname.lastname@dartmouth.edu

Abstract

A mobile agent is an executing program that can migrate during execution from machine to machine in a heterogeneous network. On each machine, the agent interacts with stationary service agents and other resources to accomplish its task. Mobile agents are particularly attractive in distributed information-retrieval applications. By moving to the location of an information resource, the agent can search the resource locally, eliminating the transfer of intermediate results across the network and reducing end-to-end latency. In this chapter, we first discuss the strengths of mobile agents, and argue that although none of these strengths are unique to mobile agents, no competing technique shares all of them. Next, after surveying several representative mobile-agent systems, we examine one specific information-retrieval application, searching distributed collections of technical reports, and consider how mobile agents can be used to implement this application efficiently and easily. Then we spend the bulk of the chapter describing two planning services that allow mobile agents to deal with dynamic network environments and information resources: (1) planning algorithms that let an agent choose the best migration path through the network, given its current task and the current network conditions, and (2) planning algorithms that tell an agent how to observe a changing set of documents in a way that detects changes as soon as possible while minimizing overhead. Finally, we consider the types of errors that can occur when information from multiple sources is merged and filtered, and argue that the structure of a mobile-agent application determines the extent to which these errors affect the final result.

1 Introduction

A mobile agent is an executing program that can migrate during execution from machine to machine in a heterogeneous network. In other words, the agent can suspend its execution, migrate to another machine, and then resume execution on the new machine from the point at which it left off. On each machine, the agent interacts with stationary agents and other resources to accomplish its task.

Mobile agents have several advantages in distributed information-retrieval applications. By migrating to an information resource, an agent can invoke resource operations *locally*, eliminating the network transfer of intermediate data. By migrating to the other side of an unreliable network link, an agent can continue executing even if the network link goes down, making mobile agents particularly attractive in mobile-computing environments. Most importantly, an agent can choose different migration strategies depending on its task and the current network conditions, and then change its strategies as network conditions change. Complex, efficient and robust behaviors can be realized with surprisingly little code.

Although each of these advantages is a reasonable argument for mobile agents, none of them are unique to mobile agents, and, in fact, any specific application can be implemented just as efficiently and robustly with more traditional techniques. Different applications require *different* traditional techniques, however, and

Copyright 1999 Springer-Verlag. In "Intelligent Information Agents", ISBN: 3-540-65112-8.
THIS COPY IS THE AUTHORS' PRE-PUBLICATION VERSION;
it may differ slightly from the official published version.

many applications require a combination of techniques. In short, the true strength of mobile agents is not that they make new distributed applications possible, but rather that they allow a wide range of distributed applications to be implemented efficiently, robustly and easily within a single, general framework.

In this chapter, we first motivate mobile agents in detail, comparing mobile agents with traditional client/server techniques and other mobile-code systems, and survey several existing mobile-agent systems. Then we consider a specific information-retrieval application, searching distributed collections of technical reports, and how this application can be implemented easily using our own mobile-agent system, D'Agents. Our mobile-agent implementation performs better than (or as well as) a more traditional RPC implementation when the query is complex or network conditions are poor, but worse when the query is simple and network conditions are good. Complex queries and slow networks allow inefficiencies in the core D'Agents and other mobile-agent systems to be amortized over a longer execution or data-transfer time. These inefficiencies, which are intrinsic to the early stages of mobile-agent development, primarily cause large migration and communication overheads.[1] Fortunately, solutions to many of the inefficiencies already exist in high-performance servers and recent mobile-agent work. Once these solutions are integrated into existing mobile-agent systems, mobile agents will perform competitively in a much wider range of network environments.

Improving the performance of the core system does not address all of an agent's needs. In particular, an effective mobile agent is one that can choose dynamically all aspects of its behavior, i.e., how many agents to send out, where to send them, whether those agents should migrate or remain stationary, whether those agents should send out children, and so on. The agent must have access to a wealth of network, machine and resource information, and a corresponding toolbox of planning algorithms, so that it can choose the most effective migration strategy for its task and the current network conditions. Therefore, a mobile-agent system must provide an extensive sensing and planning infrastructure.

In this chapter, we describe several simple directory and network-sensing services in the context of the technical-report application. Then we present initial work on two more complex planning services: (1) a set of planning algorithms that allow an agent or a small group of cooperating agents to identify the best migration path through a network, and (2) a set of planning algorithms that tell an agent how to observe a changing set of documents (specifically the pages available on the World Wide Web) in a way that detects changes as soon as possible while minimizing overhead. In the second case, the current planning algorithms are oriented towards a stationary agent that has moved to some attractive proxy site and is now observing the documents from across the network. We consider, however, how the algorithms can be extended to an agent that migrates continuously or sends out child agents.

Section 2 explores the motivation behind mobile agents in more detail. Section 3 surveys nine representative mobile-agent systems, and briefly mentions other mobile-agent systems. Section 4 describes the technical-report application and analyzes its performance. Finally, Section 5 discusses the two planning services.

2 Motivation

Mobile agents have several strengths. First, by migrating to the location of a needed resource, an agent can interact with the resource without transmitting intermediate data across the network, conserving bandwidth and reducing latencies. Similarly, by migrating to the location of a user, an agent can respond to user actions rapidly. In either case, the agent can continue its interaction with the resource or user even if network connections go down temporarily. These features make mobile agents particularly attractive in mobile-computing applications, which often must deal with low-bandwidth, high-latency, and unreliable network links.

Second, mobile agents allow traditional clients and servers to offload work to each other, and to *change* who offloads to whom according to the capabilities and current loads of the client, server and network. Similarly, mobile agents allow an application to dynamically deploy its components to arbitrary network sites, and to

[1] Migration overhead is the time on the source machine to pack up an agent's current state and send the state to the target machine, plus the time on the target machine to authenticate the incoming agent, start up an appropriate execution environment, and restore the state.

re-deploy those components in response to changing network conditions.

Finally, most distributed applications fit naturally into the mobile-agent model, since a mobile agent can migrate sequentially through a set of machines, send out a wave of child agents to visit machines in parallel, remain stationary and interact with resources remotely, or any combination of these three extremes. Complex, efficient and robust behaviors can be realized with surprisingly little code. In addition, our own experience with undergraduate programmers at Dartmouth suggests that mobile agents are easier to understand than many other distributed-computing paradigms.

Although each of these strengths is a reasonable argument for mobile agents, it is important to realize that none of these strengths are unique to mobile agents [CGH+95]. Any specific application can be implemented just as efficiently with other techniques. These other techniques include message passing, remote procedure calls (RPC) [BN84], remote object-method invocation (as in Java RMI [WRW96] or CORBA [BN95]), queued RPC [JdT+95] (in which RPC calls are queued for later invocation if the network connection is down), remote evaluation [Fal87, SG90, Sto94] (which extends RPC by allowing the client to send the procedure code to the server, rather than just the parameters for an existing procedure), process migration [DO91, LS92], stored procedures (such as [BP88], where SQL procedures can be uploaded into a relational database for later invocation), Java applets [CW97] and servlets [Cha96] (which respectively are Java programs that are downloaded by a Web browser or uploaded into a Web server), automatic installation facilities, application-specific query languages, and application-specific proxies within the permanent network. None of these other techniques, however, share all of the strengths of mobile agents.

Messaging passing and remote invocation. In contrast to message passing and remote invocation, mobile code (including mobile agents) allows an application to conserve bandwidth and reduce latency *even if* an information resource provides low-level operations, simply because the mobile code can be sent to the network location of the resource. The mobile code can invoke as many low-level server operations as needed to perform its task without transferring any intermediate data across the network. Moreover, the mobile code can continue its task even if the network link between the client and server machines goes down. The code has been sent to the *other side* of the link, and will not need the link again until it is ready to send back a "final" result. The resource provider can implement a single high-level operation that performs each client's desired task in its entirety. Implementing these high-level operations, however, becomes an intractable programming task as the number of distinct clients increases. In addition, it discourages modern software engineering, since the server becomes a collection of complex, specialized routines, rather than simple, general primitives.

Process migration. Typically, process-migration systems do not allow the processes to choose when and where they migrate. Instead, most are designed to *transparently* move processes from one machine to another to balance load. In addition, although some process-migration systems allow the processes to migrate across heterogeneous machines [BVW95], these facilities still are intended for "closed" environments, where security is less of a concern. Mobile agents, on the other hand, can move when and where they want, according to their own application-specific criteria. For example, although mobile agents can move solely to obtain CPU cycles, most mobile agents will move to colocate themselves with specific information resources. In addition, nearly all mobile-agent systems have been designed from the ground up to be both platform-independent and secure in open environments.

Remote evaluation, stored procedures, applets and servlets. Mobile agents are much more flexible than these other forms of mobile code. First, a mobile agent can move from a client to server or from a server to client. Most other forms of mobile code allow code transfer in a single direction only. Second, a mobile agent can move at times of its own choosing. Java applets, in particular, are downloaded onto a client machine only when a human user visits an associated Web page. Third, a mobile agent can move as many times as desired. For example, if a server is implemented as a mobile agent, it can continuously move from one network location to another to minimize the average latency between itself and its current clients [RASS97]. Conversely, a client agent can migrate sequentially through some set of machines, accessing some resource on each. For example, if a client agent needs to query one database to determine which query it should run against a second database, it can migrate to the first database, run the first query, analyze the query results to determine the second query, throw out the analysis code to make itself smaller, migrate directly to the second database, run the second query, and carry just the final result back to its home machine. Most implementations of remote evaluation and stored procedures, along with all Web browsers and servers that support applets and servlets, do not allow the mobile code to spawn additional mobile code onto *different* machines, making any form of sequential migration impossible. Instead, the client machine must interact with each resource in turn.

Finally, a mobile agent can spawn off child agents no matter where it is in the network. For example, a mobile agent can move to a dynamically selected proxy site, send out child agents to search some distributed data collection in parallel, and then merge and filter the search results on the proxy site before carrying just the final result back to the client. As with sequential migration, most implementations of the other mobile-code techniques do not support such behavior.

Application-specific solutions. Finally, in contrast to application-specific solutions, such as specialized query languages and dedicated proxies pre-installed at specific network locations, mobile agents are distinguished by both their flexibility and their ease of implementation. An application can send its own proxy to an arbitrarily selected network location, and can move that proxy as network conditions change. In addition, a server simply can make its operations visible to visiting mobile agents, rather than implementing higher-level operations or some application-specific language to minimize network traffic.

Summary. In short, an application must use one or more of these other techniques to realize the same behavior that mobile agents allow, and different applications must use different techniques. The true strength of mobile agents is that a wide range of distributed applications can be implemented efficiently, easily and robustly within the same, general framework, and these applications can exhibit extremely flexible behavior in the face of changing network conditions. As we show in Section 4, mobile-agent systems are not efficient enough yet to be competitive with the other techniques in every situation. However, the potential for mobile agents is clear, and mobile-agent researchers now share a common, realizable goal: a mobile-agent system in which (1) inter-agent communication is as fast as traditional RPC, (2) migration of code is only a small factor slower than an RPC call that transfers an equivalent amount of data, (3) computation-intensive agents execute no more than twice as slowly as natively compiled code, and (4) a wide range of network-status information is available to agents for use in their decision-making process. In such a system, migration would be advantageous even if the task involved only a few operations at each information resource, and a mobile agent could use its knowledge of the task, the needed information resources and the current network conditions to decide whether to migrate or remain stationary. In other words, mobile agents would perform no worse than equivalent solutions implemented with the other techniques, and would often perform much better.

3 Survey of mobile-agent systems

In this section, we examine nine representative mobile-agent systems, and then briefly discuss their similarities and differences.

3.1 Representative mobile-agent systems

3.1.1 Multiple-language systems

Ara. Ara[2] [PS97, Pei98] supports agents written in Tcl and C/C++. The C/C++ agents are compiled into an efficient interpreted bytecode called MACE; this bytecode, rather than the C/C++ code itself, is sent from machine to machine. For both Tcl and MACE, Ara provides a `go` instruction, which automatically captures the complete state of the agent, transfers the state to the target machine, and resumes agent execution at the exact point of the `go`. Ara also allows the agent to *checkpoint* its current internal state at any time during its execution. Unlike other multiple-language systems, the entire Ara system is multi-threaded; the agent server and both the Tcl and MACE interpreters run inside a single Unix process. Although this approach complicates the implementation, it has significant performance advantages, since there is little interpreter

[2] `http://www.uni-kl.de/AG-Nehmer/Ara/`

startup or communication overhead. When a new agent arrives, it simply begins execution in a new thread, and when one agent wants to communicate with another, it simply transfers the message structure to the target agent, rather than having to use inter-process communication. Nearly all Java-only systems are also multi-threaded, and see the same performance advantages.

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

D'Agents. D'Agents[3] [GKCR98], which was once known as Agent Tcl, supports agents written in Tcl, Java and Scheme, as well as *stationary* agents written in C and C++. Like Ara, D'Agents provides a `go` instruction (Tcl and Java only), and automatically captures and restores the complete state of a migrating agent. Unlike Ara, only the D'Agent server is multi-threaded; each agent is executed in a separate process, which simplifies the implementation considerably, but adds the overhead of inter-process communication. The D'Agent server uses public-key cryptography to authenticate the identity of an incoming agent's owner. Stationary *resource-manager* agents assign access rights to the agent based on this authentication and the administrator's preferences, and language-specific enforcement modules enforce the access rights, either preventing a violation from occurring (e.g., file-system access) or terminating the agent when a violation occurs (e.g., total CPU time). Each resource manager is associated with a specific resource such as the file system. The resources managers can be as complex as desired, but the default managers simply associate a list of access rights with each owner. Unlike Ara, most resource managers are not consulted when the agent arrives, but instead only when the agent (1) attempts to access the corresponding resource or (2) explicitly requests a specific access right. At that point, however, the resource manager forwards all relevant access rights to the enforcement module, and D'Agents behaves in the same way as Ara, enforcing the access rights with short wrapper functions around the resource access functions.

Current work on D'Agents falls into four broad categories: (1) scalability, (2) network-sensing and planning services, which allow an agent to choose the best migration strategy given the current network conditions; (3) market-based resource control, where agents are given a finite supply of currency from their owner's own finite supply and must spend the currency to access needed resources [BKR98]; and (4) support for mobile-computing environments, where applications must deal with low-bandwidth, high-latency and unreliable network links [KGN+98]. Some scalability issues are discussed in the next section, where we analyze the performance of a distributed retrieval application running on top of the D'Agents system. Network-sensing and planning is discussed in section 5, where we examine some services necessary for a distributed information-retrieval to make efficient use of available network resources.

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

Tacoma. Tacoma[4] [JSvR98a, JSvR98b] supports agents written in C, C++, ML, Perl, Python, Scheme and Visual Basic. Unlike Ara and D'Agents, Tacoma does not provide automatic state-capture facilities. Instead, when an agent wants to migrate to a new machine, it creates a *folder* into which it packs its code and any desired state information. The folder is sent to the new machine, which starts up the necessary execution environment and then calls a known entry point within the agent's code to resume agent execution. Although this approach places the burden of state capture squarely onto the *agent* programmer, it also allows the rapid integration of new languages into the Tacoma system, since existing interpreters and virtual machines can

[3] `http://www.cs.dartmouth.edu/~agent/`
[4] `http://www.tacoma.cs.uit.no:8080/TACOMA/`

be used without modification. Tacoma is used most notably in StormCast, which is a distributed weather-monitoring system, and the Tacoma Image Server, which is a retrieval system for satellite images [JSvR98b].

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

3.1.2 Java-based systems

Aglets. Aglets[5] [LO98, LC96] was one of the first Java-based systems. Like all commercial systems, including Concordia [WPW+97, WPW98], Jumping Beans [AA98], and Voyager [OBJ97], Aglets does not capture an agent's thread (or control) state during migration, since thread capture requires modifications to the standard Java virtual machine. In other words, thread capture means that the system could be used only with one specific virtual machine, significantly reducing market acceptance.[6] Thus, rather than providing the `go` primitive of D'Agents and Ara, Aglets and the other commercial systems instead use variants of the Tacoma model, where agent execution is restarted from a known entry point after each migration. In particular, Aglets uses an event-driven model. When an agent wants to migrate, it calls the `dispatch` method. The Aglets system calls the agent's `onDispatching` method, which performs application-specific cleanup, kills the agent's threads, serializes the agent's code and object state, and sends the code and object state to the new machine. On the new machine, the system calls the agent's `onArrival` method, which performs application-specific initialization, and then calls the agent's `run` method to restart agent execution.

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

Concordia. Concordia[7] [WPW+97, WPW98] is a Java-based mobile-agent system that has a strong focus on security and reliability. Like most other mobile-Java agent systems, they move the agent objects code and data, but not thread state, from one machine to another. Like many other systems, Concordia agents are bundled with an *itinerary* of places to visit, which can be adjusted by the agent while en route.[8] Agents, events, and messages can be queued, if the remote site is not currently reachable. Agents are carefully saved to a persistent store, before departing a site and after arriving at a new site, to avoid agent loss in the event of a machine crash. Agents are protected from tampering through encryption while they are in transmission or stored on disk; agent hosts are protected from malicious agents through cryptographic authentication of the agent's owner, and access control lists that guard each resource.

Jumping Beans. Jumping Beans[9] [AA98] is a Java-based framework for mobile agents. Computers wishing to host mobile agents run a Jumping Beans *agency,* which is associated with some Jumping Beans

[5] `http://www.trl.ibm.co.jp/aglets/`

[6] D'Agents, which does use a modified Java virtual machine to capture thread state, is a research system and is under no such market constraints.

[7] `http://www.concordia.mea.com/`

[8] 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.

[9] `http://www.JumpingBeans.com/`

domain. Each domain has a central server, which authenticates the agencies joining the domain. Mobile agents move from agency to agency, and agents can send messages to other agents; both mechanisms are implemented by passing through the server. Thus the server becomes a central point for tracking, managing, and authenticating agents. It also becomes a central point of failure or a performance bottleneck, although they intend to develop scalable servers to run on parallel machines. Another approach to scalability is to create many small domains, each with its own server. In the current version, agents cannot migrate between domains, but they intend to support that capability in future versions. Security and reliability appear to be important concerns of their system; public-key cryptography is used to authenticate agencies to the server, and *vice versa*; access-control lists are used to control an agent's access to resources, based on the permissions given to the agent's owning user.

Although they claim to move all agent code, data, and state, it is not clear from their documentation whether they actually move thread state, as in Agent Java. They require that the agent be a serializable object, so it seems likely that they implement the weaker form of mobility common to other Java-based agent systems.

3.1.3 Other systems

Messengers. The Messenger[10] project uses mobile code to build flexible distributed systems, not specifically mobile-agent systems [TDM+94, DMTH95, Muh98]. In their system, computers run a minimal Messenger Operating System (MOS), which has just a few services. MOS can send and receive *messengers,* which are small packets of data and code written in their programming language `MO`. MOS can interpret `MO` programs, which may access one of their two bulletin-board services: the *global dictionary*, which allows data exchange between messengers, and the *service dictionary*, which is a searchable listing of messengers that offer services to other messengers. Ultimately, most services, including all distributed services, are offered by static and mobile messengers. In one case, they allow the messengers to carry native UNIX code, which is installed and executed on MOS; system calls are reflected back to the interpreted `MO` code, allowing fast execution of critical routines, while maintaining the flexibility of mobile code [TMN97].

Obliq. Obliq [Car95, BN97] is an interpreted, lexically scoped, object-oriented language. An Obliq object is a collection of named fields that contain methods, aliases, and values. An object can be created at a remote site, cloned onto a remote site, or migrated with a combination of cloning and redirection. Implementing mobile agents on top of these mobile objects is straightforward. An agent consists of a user-defined procedure that takes a *briefcase* as its argument; the briefcase contains the Obliq objects that the procedure needs to perform its task. The agent migrates by sending its procedure and current briefcase to the target machine, which invokes the procedure to resume agent execution.

Visual Obliq[11] [BC95] builds on top of Obliq's migration capabilities. Visual Obliq is an interactive application builder that includes (1) a visual programming environment for laying out graphical user interfaces, and (2) an agent server that allows Visual Obliq applications to migrate from machine to machine. When the application migrates, the state of its graphical interface is captured automatically, and recreated exactly on the new machine. Obliq does not address security issues. Visual Obliq does provide access control, namely, user-specified access checks associated with all "dangerous" Obliq commands, but does not have authentication or encryption mechanisms. Typically, therefore, the access checks will simply ask the user whether the agent should be allowed to perform the given action.

Telescript. Telescript[12] [Whi94b, Whi94a, Whi97], developed at General Magic, Inc., was the first commercial mobile-agent system, and the inspiration for many of the recent mobile-agent systems. In Telescript, each network site runs a server that maintains one or more virtual *places*. An incoming agent specifies which of the places it wants to enter. The place authenticates the identity of the agent's owner by examining the agent's cryptographic credentials, and then assigns a set of access rights or *permits* to the agent. One permit,

[10] `http://www.ics.uci.edu/~bic/messengers/`
[11] `http://www.cc.gatech.edu/gvu/people/Phd/Krishna/VO/VOHome.html`
[12] `http://www.genmagic.com/technology/mobile_agent.html`

for example, might specify a maximum agent lifetime, while another might specify a maximum amount of disk usage. An agent that attempts to violate its permits is terminated immediately [Whi94b]. In addition to maintaining the places and enforcing the security constraints, the server continuously writes the internal state of executing agents to non-volatile store, so that the agents can be restored after a node failure.

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

3.2 Similarities and differences

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

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

Mobile-agent systems generally provide one of two kinds of migration: (1) `go`, which captures an agent's object state, code, and control state, allowing it to continue execution from the exact point at which it left off; and (2) *entry point*, which captures only the agent's object state and code, and then calls a known entry point inside its code to restart the agent on the new machine. The `go` model is more convenient for the end programmer, but more work for the system developer since routines to capture control state must be added to existing interpreters. All commercial Java-based systems use entry-point migration, since market concerns demand that these systems run on top of unmodified Java virtual machines. Research systems use both both migration techniques.

Finally, existing mobile-agent systems focus on protecting an *individual* machine against malicious agents. Aside from encrypting an agent in transit and allowing an agent to authenticate the destination machine before migrating, most existing systems do not provide any protection for the agent or for a group of machines that is *not* under single administrative control.

Other differences exist among the mobile-agent systems, such as the granularity of their communication mechanisms, whether they are built on top of or can interact with CORBA, and whether they conform to the emerging mobile-agent standards. Despite these differences, however, all of the systems discussed above (with the exception of Messengers, which is a lighter-weight mobile-agent system) are intended for the same applications, such as workflow, network management, and automated software installation. All of the systems are suitable for distributed information retrieval, and the decision of which one to use must be based on the desired implementation language, the needed level of security, and the needed performance.

4 Application: The technical-report searcher

Mobile agents are commonly used in distributed information-retrieval applications. By sending agents to proxy sites and to the information sources themselves, the applications can avoid the transfer of intermediate data, can continue with the retrieval task even if the network link with the client machine goes down, and can merge and filter the results from the individual document collections inside the network, rather than pulling all the results back to the client machine. In addition, many retrieval tasks require the application to simply invoke a sequence of server operations with only a modest amount of "glue" code to decide which server operation should be invoked next. Since such tasks are bound by the execution time of the server operations, rather than the execution time of the glue code, a mobile agent can perform well even when implemented in one of the interpreted languages that are found in most existing mobile-agent systems. In this section, we consider such a retrieval application, namely the retrieval of documents from a distributed collection of technical reports. This application is representative of many other distributed locating, gathering and organizing applications, and results from its study are applicable to other applications with similar structure.

4.1 Description

Figure 1 shows the structure of the technical-report application, which was implemented on top of our mobile-agent system, D'Agents. The technical reports themselves come from the Department of Computer Science at Dartmouth College. We distributed the reports across multiple Dartmouth machines, each of which is running the D'Agents system and the Smart system. The Smart system is a successful statistical information-retrieval system that uses the vector-space model to measure the textual similarity between documents [Sal91]. The Smart system on each machine is "wrapped" inside a stationary agent, which is labeled as *Smart IR agent* in the figure. This stationary agent provides a three-function interface to the Smart system: (1) run a textual query and obtain a list of relevant documents, (2) obtain the full text of a document, and (3) obtain pair-wise similarity scores for every pair of documents in a list of documents. The pair-wise similarity scores are used to construct different graphical representations of the query results.

When each stationary Smart agent starts execution, it registers with a virtual *yellow pages* [RGK97]. The yellow pages are a simple, distributed, hierarchical directory service. When the Smart agent registers with the yellow pages, it provides its location (i.e., its identifier within the D'Agents namespace) and a set of keywords that describe its service (i.e., *smart*, *technical-reports*, *text*). A client agent searches for a service by sending a keyword query to the yellow pages. The yellow pages return the locations of all services whose keyword lists match the keyword query (more specifically, all services whose keyword lists are a superset of the keyword query). A forthcoming version of the yellow pages will allow the client agents to search by interface definition as well [NCK96].

The main application agent is a GUI that runs on the user's machine. This GUI is shown at the top of Figure 1. The GUI first lets the user enter a free-text query and optionally select specific document collections from a list of known document collections. Once the GUI has the query, it spawns a mobile agent onto the local machine. This mobile agent first consults one or more local network-sensing agents [RGK97, Moi98], which keep track of the network connection between the user's machine and the rest of the network. These network-sensing agents know what type of network hardware is in the machine, the maximum bandwidth of that hardware, an uptime/downtime history of the network link, and the current observed latency and bandwidth of the network link. The uptime/downtime history is used to calculate an approximate *reliability factor*, i.e., the probability that the network connection will go down at some point in the next few minutes.

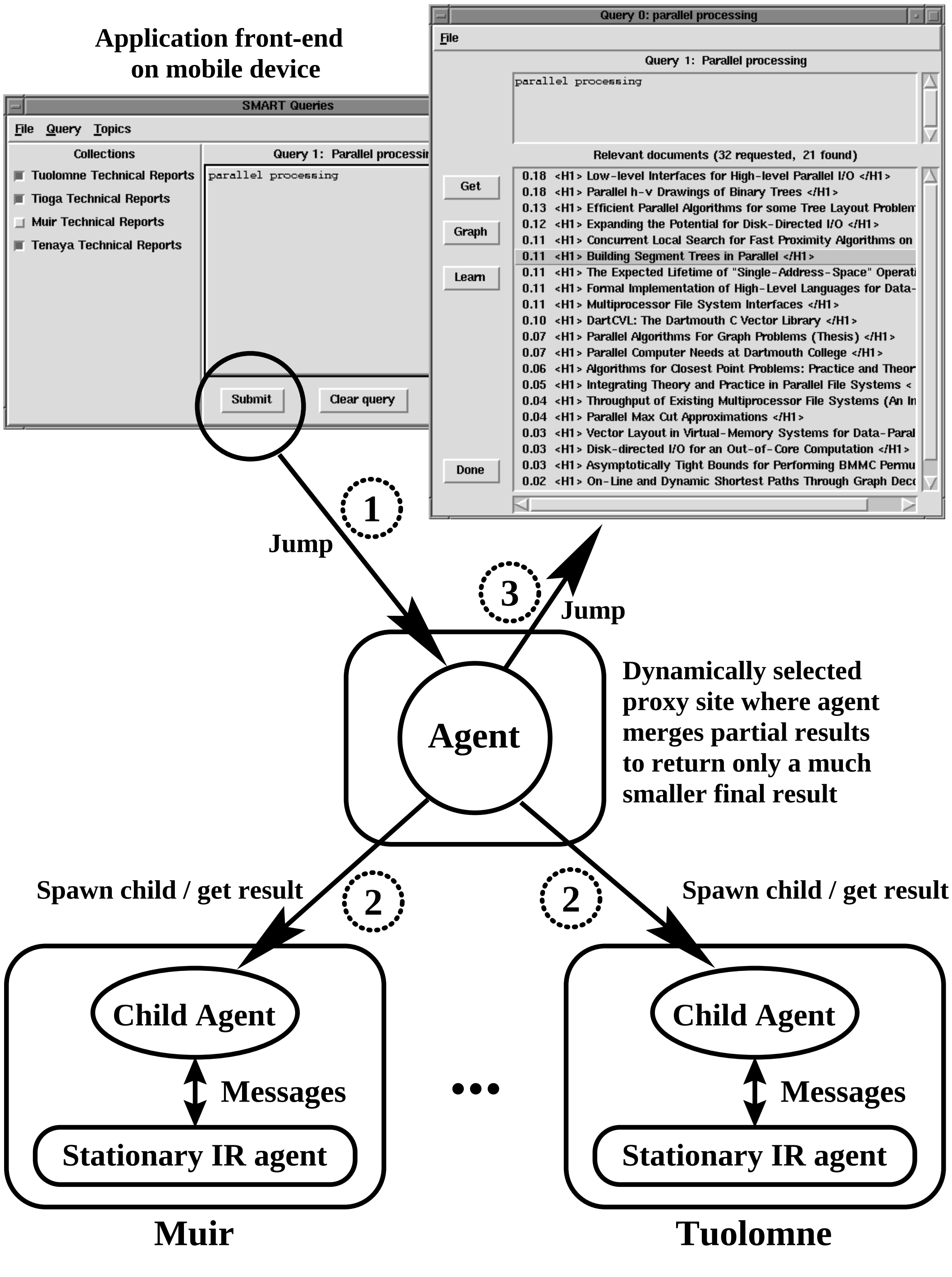

Figure 1: An example application. Here a mobile agent is searching a distributed collection of technical reports. The agent first decides whether to move to a dynamically selected proxy site. Then it decides whether to spawn child agents or simply interact with the individual document collections from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

Our reliability factor is quite simple—it is just the percentage of time that the network connection has been down during the past n hours—but is sufficient for our purposes.

After consulting the network-sensing agents, it makes its most important decision. If the network connection between the user's machine and the network is reliable and has high bandwidth, the agent stays on the user's machine. If the connection is unreliable or has low bandwidth, the agent jumps to a *proxy site* within the permanent network. This proxy site is shown in the middle of Figure 1. With our current reliability and bandwidth thresholds, the agent typically will remain on the user's machine if the machine is a workstation with a 10 Mbit/s Ethernet link or a laptop with a 1 Mbit/s wireless Ethernet link. The agent will jump to a proxy site if the user's machine is a laptop with a modem link. The proxy site is dynamically selected by the agent. In the current system, the selection process is quite simple—there is a designated proxy site for each laptop and for some subnetworks. The agent will go to the proxy associated with the subnetwork to which the laptop is attached, or to the laptop-specific proxy if there is no known subnetwork proxy. Currently, the proxy sites are hardcoded, but eventually, they will be listed in the yellow pages along with other services. Then an agent can search for the closest proxy site (to its current location or to the document collections), the closest proxy site owned by its owner's Internet Service Provider (ISP), the fastest proxy site, etc.

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

When the main agent receives the results from each child agent, it merges and filters those results, returns to the user's machine with just the final list of documents, and hands this list to the GUI. Although the behavior exhibited by this agent is complex, it is actually quite easy to implement and involves only about 50 lines of Tcl code. In particular, the decisions whether to use a proxy site and create children, although admittedly simplistic in our current implementation, involve little more than two *if* statements that check the information returned from the network sensors and the yellow pages. It is hard to imagine any other technique that would allow us to provide an equally flexible solution with the same small amount of work. More importantly, once some inefficiencies in the D'Agents implementation are addressed (and as long as the agent carefully chooses when and where to migrate), its performance should be comparable to or better than that of any other technique, regardless of the current network conditions and without any application-specific support at the collection or proxy sites. Indeed, the collection owners merely had to provide an agent wrapper that made the existing Smart operations visible to visiting agents, and the proxy site did not need to do anything at all (aside from running the agent system itself). As we will see in the next section, the critical inefficiencies involve communication and migration overhead. Due to the communication overhead, the agent performs worse than traditional client/server techniques if it chooses to remain stationary. Similarly, due to the migration overhead, the agent is better off migrating only when network conditions are poor, or when each query requires a large number of operations per collection. Fortunately, techniques to reduce these overheads already exist.

4.2 Analysis

A full performance analysis of the technical-report searcher is beyond the scope of the chapter. Here we consider only the case where the user's machine has a reliable, high-bandwidth network connection, specifically, a wired 10 Mb/s Ethernet link. Under these network conditions, the searcher will *not* use a proxy site, but still must decide whether to make cross-network calls or send out child agents. To understand the searcher's performance under these conditions, we ran a series of experiments, some involving traditional RPC-based clients and servers, others involving test agents. Each experiment was performed on the same two machines, two 200 MHz Pentium II laptops,[13] which were connected with a dedicated 10 Mb/s Ethernet link. The traditional clients and servers were written in C/C++, and since the technical-report searcher was written in Tcl, the test agents were written in Tcl also.

Base performance. First, we consider the base performance of the core D'Agents system. Figure 2a shows the results of three performance experiments. The first experiment, the results of which are shown as the *RPC* line in Figure 2a, measures the time needed for a client on one laptop to make a Sun RPC call into a server on the second laptop. The total size of the arguments to the RPC call was 256 bytes; the result size varied from 512 bytes to 8192 bytes. The server did no work aside from immediately responding to the client with a dummy result of the desired size. This first experiment does not involve the agent system in any way, but instead is used to place the D'Agent performance numbers in context.

The second and third experiments do involve the agent system. In the second experiment, an agent on the first laptop sent a request message to an agent on the second laptop, and the agent on the second laptop sent back a response. The *Agent (send/receive)* line in Figure 2a shows the total round trip time as a function of response size; the request size was again 256 bytes. In addition, as in the RPC experiment, the "server" agent did no work aside from immediately sending back a dummy response message. Finally, in the third experiment, an agent on the first laptop *sent a child agent* to the second laptop; the child agent then sent back a response message to its parent. The *Agent (submit/receive)* line in Figure 2 shows the total time from agent submission to result reception. The size of the child agent was 256 bytes, and the child agent did no work aside from immediately sending back a dummy result.

Unlike previous performance results for D'Agents [Gra97], these agent experiments were measured with the new multi-threaded version of the D'Agents server, which eliminates significant interprocess communication. In addition, the new server maintains a pool of "hot" interpreters; it starts up a set of interpreter processes at boot time, and then hands incoming agents off to the first free interpreter in that set. In addition, an interpreter process does not die when an agent finishes, but instead stays alive to execute the next incoming agent. Although this approach still runs each agent in a separate process, it eliminates nearly all of the interpreter-startup overhead.[14] Also, the agent experiments were performed with encryption turned off. The agents would perform significantly worse with encryption turned on, but so would an equivalently secure version of RPC. Turning encryption off is reasonable, since many document collections would not care about the identity of the agent's owner.

The base performance numbers illustrate two important points: (1) inter-agent communication involves significantly more overhead than RPC, and (2) migration involves even more overhead than inter-agent communication. Fortunately, this overhead comes from several clear sources. First, the agents are written in Tcl, which is a relatively slow scripting language. D'Agents, however, includes two faster languages, Java and Scheme. In addition, the newest version of the Tcl interpreter, which has not been integrated into D'Agents yet, uses on-the-fly compilation (into virtual machine bytecodes) and is more than ten times faster than previous Tcl interpreters. Second, even though the multi-threaded server uses a pool of hot interpreters,

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

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

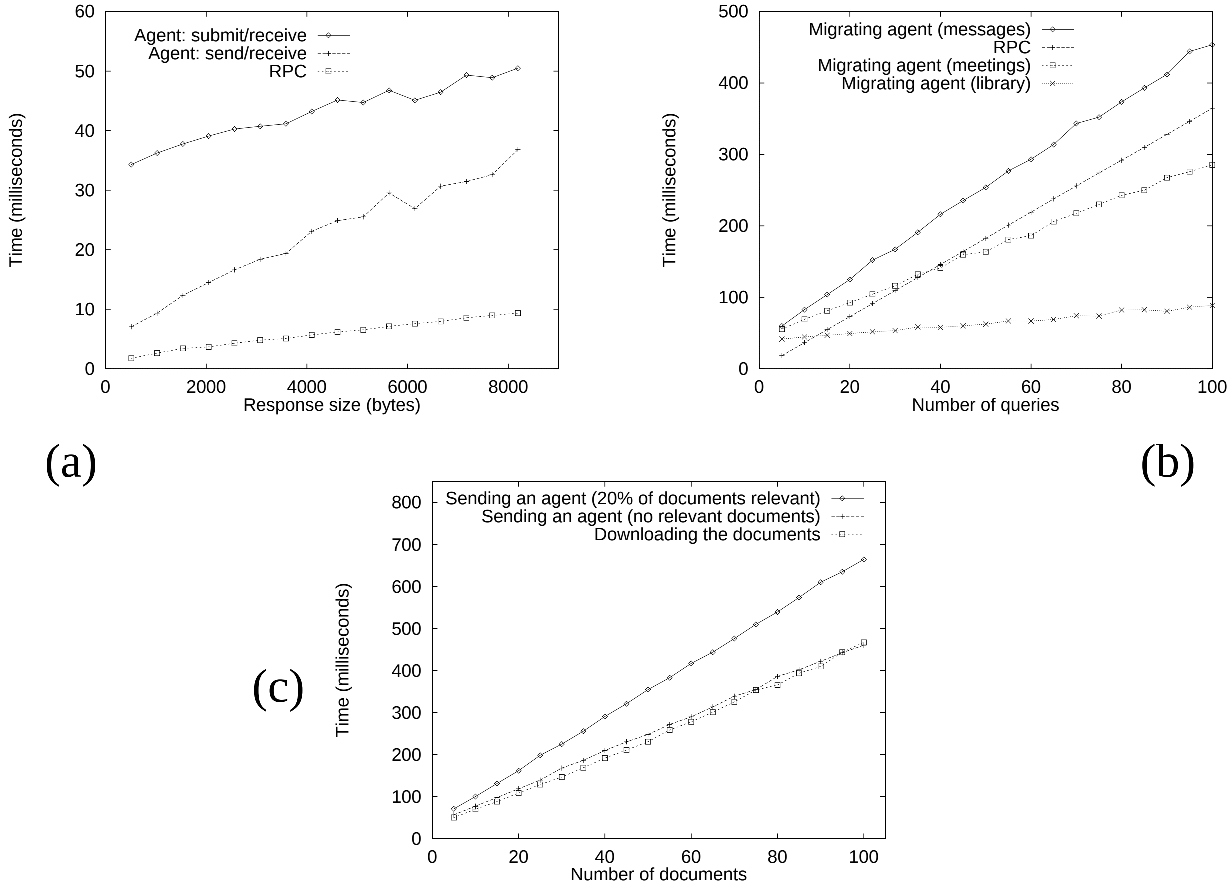

Figure 2: **(a)** Base performance of D'Agents. **(b)** Retrieval time as a function of the number of queries per document collection. **(c)** Retrieval time as a function of the number of relevant documents per query. Each data point is the average of either 100 (agent experiments) or 1000 (RPC experiments) trials. These graphs are explained in the chapter text.

each Tcl interpreter must still execute several hundred lines of Tcl code to re-initialize itself before executing an incoming agent. This re-initialization adds nearly ten milliseconds to the migration time. Some of the re-initialization can be eliminated, and the rest can be made much faster, either by switching to the newest Tcl interpreter or re-implementing the initialization code in C.

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

Performance when the searcher must perform multiple queries. With the base performance results in mind, it becomes straightforward to understand the results for the actual technical-report searcher. Figure 2b considers the case where the searcher needs to perform multiple queries against a single document collection. Perhaps, for example, the user has specified several alternative queries that should be tried if the main query does not retrieve the desired number of relevant documents. The *RPC* line shows the time needed for a traditional client (written in C) to run the queries from across the network. The query size was 256 bytes, and the result size for *each* query was 2048 bytes, which is consistent with the data volumes observed when we perform queries against our current document collections. In addition, the server did not actually perform the query, but instead sent back a dummy result. This approach allows us to run more iterations of each test. Moreover, it will not change the relative performances shown in the figure since the same query backend would be used in all four cases. In other words, we have removed exactly that portion of the code that is identical in all four cases, namely, the shared C library that actually runs the queries through the Smart system.

The remaining three cases shown in Figure 2b involve agents rather than RPC. In each case, an agent was sent to the location of the document collection, where it ran the queries locally to the collection and sent back only the final document list. The size of each test agent was $1024 + 256 + 128n$ bytes, where n is the number of queries that will be performed; the 1024 is approximately the size of the *real* agent's code, the 256 the size of the main query, and the 128 the size of the additional code and data that is needed to represent and perform the alternative queries. The final result size was 2048 bytes. In the first agent case, *Migrating agent (messages)*, the agent used D'Agent *messages* to communicate with the stationary Smart agent. In the second case, *Migrating agent (meetings)*, the agent first established a direct inter-process connection[15] or *meeting* with the stationary agent; the queries and results were sent across the meeting. Finally, in the third case, the agent loaded the Smart library itself and simply invoked the query procedures directly.

The agent that loads the Smart library itself performs quite well. As can be seen, even though the agent was written in Tcl, and the network link between the laptops was relatively fast, the agent performed better than RPC when it needed to invoke more than a dozen queries, and performed much better as the number of queries increases. Once the migration and communication overhead is reduced as discussed above, it should be competitive for even just five queries. This suggests that a useful abstraction will be services that appear to be stationary agents, but, in fact, are provided through libraries. D'Agents includes an RPC-like mechanism, which allows agents to invoke each other's procedures. This mechanism would provide a natural way of making a library appear as a stationary agent, since the client agent could make the same procedure invocations in both cases; only the hidden implementation of the stubs would be different.

On the other hand, when the client agent had to communicate with an actual stationary agent, it did worse than RPC unless it performed more than forty queries with meetings, and always did worse with

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

messages. There are several reasons why the agent performed worse, all of which were considered above. Most importantly, the overhead of inter-agent communication is large even when the agents are *on the same machine.* When an agent sends a message to another local agent, the message first is sent to the server process (over a pipe), and then sent to the interpreter process that is executing the recipient agent (over another pipe). The response message follows the reverse path back to the sending agent. Thus, each message is sent twice, from agent to server, and then from server to agent. The overhead of the double transmission is larger than the overhead of making RPC calls across the good network link. This can be seen clearly in the *Migrating agent (messages)* line of Figure 2b, which has a larger slope than the *RPC* line. It also comes into play, however, when the agents are using meetings, since establishing a meeting requires the exchange of two messages, a meeting request and a meeting acceptance. One possible solution is to allow direct inter-process communication between agents, even when the agents have not established a meeting. For example, each interpreter process could have a Unix domain socket for accepting messages from other local agents. Since the agents are local to each other, the security concerns are significantly less than if each process has a *network* socket for accepting messages from *remote* agents.

Performance when the searcher must examine the document texts. Figure 2c shows a set of experiments where the Smart search operations do not match the application's needs exactly. Here the application must perform a query to get a list of potentially relevant documents, and then examine the text of each document to decide which documents are actually relevant. The *Downloading the documents* line shows how long it takes for a client (written in C) to open a TCP connection to the document server (also written in C) and send the query, plus the time needed for the server to send back the *full text* of all potentially relevant documents. The two *Sending an agent* lines shows how long it takes to send an agent to the document collection, and then for that agent to perform the query locally, examine the text of the documents, and send back all relevant documents. Each agent was 1024 bytes, the query was 256 bytes, and each document was 4096 bytes; the agent performed the query by loading a library and directly invoking a procedure (i.e., the fastest case considered in the previous set of experiments); the query procedure returned a dummy result of the appropriate size; the agent opened the document files directly to examine their full text[16]; and the agent decided that a document was relevant if its text contained a particular two-word substring.

The difference between the two *Sending an agent* lines is that in the first case, twenty percent of the examined documents are actually relevant, while in the second case, none of the documents are actually relevant. As can be seen, when there are no relevant documents, the agent does slightly worse than the downloading solution, and when twenty percent of the documents are relevant, the agent does significantly worse. All of the implementation issues considered above are influencing these results. Three factors, however, have the most impact. First, the time to read the document files from disk, which must be done in both the client/server and agent cases, takes nearly half of the total time (even though the document files were in the file cache for all but the first run). Second, Tcl is slow enough that it takes nearly as long to perform the substring search with a Tcl agent as to send the entire document text across our good network link. Third, and most importantly, the inefficiencies in the inter-agent communication mechanisms hurt the overall performance more and more as the number of relevant documents increases. In the worst case, 80 kilobytes of document text are sent inside an agent message. Clearly, it is necessary to implement a more efficient means of "streaming" data from an agent on one machine to an agent on another. At the same time, it is worthwhile to note that if a separate network connection must be established for each *downloaded* document (as with some Web servers), the agent solution performs far better than the document-downloading solution [RGK97].

Summary. Taken together, these results mean that the technical-report searcher agent generally will take longer to complete its query than the corresponding client/server solution, since (1) inter-agent communication across machines is slower than RPC, (2) inter-agent communication on the same machine is also slow, making migration less useful, and (3) the searcher and all its agents are written in Tcl. At the same time, even with 10 Mb/s links and slow Tcl agents, the searcher does outperform the client/server solution in several cases, particularly when the document collections provide their search operations as a loadable library.

[16] It is reasonable to assume that the query procedure or agent would include the filesystem location of the document files in the result list.

Moreover, as network bandwidth, reliability, latency or load become worse, the technical-report searcher will have better and better performance relative to the client/server solution [Gra97], since it transmits less data across the network and requires fewer network-communication steps.

In addition, the performance bottlenecks in the current D'Agents system are easy to identify, and several solutions exist. In fact, in some mobile-agent systems, particularly the Java-based systems where each agent executes in its own *thread*, some of the D'Agent bottlenecks have been eliminated already. For D'Agents itself, a speedup of at least two can be realized with moderate implementation effort, without abandoning Tcl as an agent language or resorting to multi-threaded interpreters. For the many applications where even the newest version of the Tcl interpreter is not fast enough, D'Agents already includes two faster execution environments, a Scheme interpreter and a Java virtual machine. Most other agent systems have similar, faster environments, and securely executing agents less than twice as slowly as the corresponding natively-compiled code (through on-the-fly compilation and other techniques) is a realizable goal [LSW95].

In short, although current mobile-agent implementations, such as D'Agents, do not offer better performance than competing solutions in as many cases as desired, these systems are far from their maximum possible performance. As the implementations improve, mobile agents will become more and more attractive for distributed information retrieval. Finally, it is important to note that all of the experiments above involved dummy query operations, whereas the real Smart system does significant work per query. The overhead of the current D'Agents system becomes much more reasonable when considered against the Smart retrieval times, For this and other retrieval tasks, the flexibility of D'Agents makes up for the performance penalties.

5 Planning

5.1 Planning a route

In the example of the technical-report searcher, we launched a mobile information-retrieval agent to each destination machine, and we assumed that each dispatched agent could definitely find the information it was tasked to send. A more general class of information-retrieval problem anticipates the possibility that an agent may not be able to find its desired information at a destination machine. Additionally, we may want to use less network resources by sending fewer agents than the number of possible destination machines. In this case, there is a need for planning that decides the best sequence (itinerary) of machines to be visited by each agent so that the desired information can be found in minimum time. In this section, we will discuss these planning problems, along with their solutions and some limited experimental results.

An itinerary determined by planning will be based on three things: a list of machines where an agent may be able to find its desired information, the uncertainty in the quality of the data available on those machines, and the current network conditions. The list of machines and document uncertainty are provided by a more advanced *yellow pages* service than that used in the technical report searcher. The uncertainty degree is defined to be the probability that an agent can successfully find information at each of those machines. Last, the network conditions include information regarding connectivity of links, operability of machines in the network, latency and available bandwidth of links. These statistics are collected by a *network-sensing module*.

5.1.1 Architecture of the Mobile Agent Planning System

The architecture of our planning system for mobile agents is depicted in Figure 3. The planning system consists of three main components: a planning module, a network-sensing module and a yellow-pages module. In our system, when a mobile agent is tasked with searching for information, it consults with the planning module first. The planning module then asks a yellow-pages module for possible locations where the mobile agent might find this desired information.

Although the current implementation of the yellow-page service does not have a function to measure this probability of success, we assume that this probability is measurable. For example, the probability might

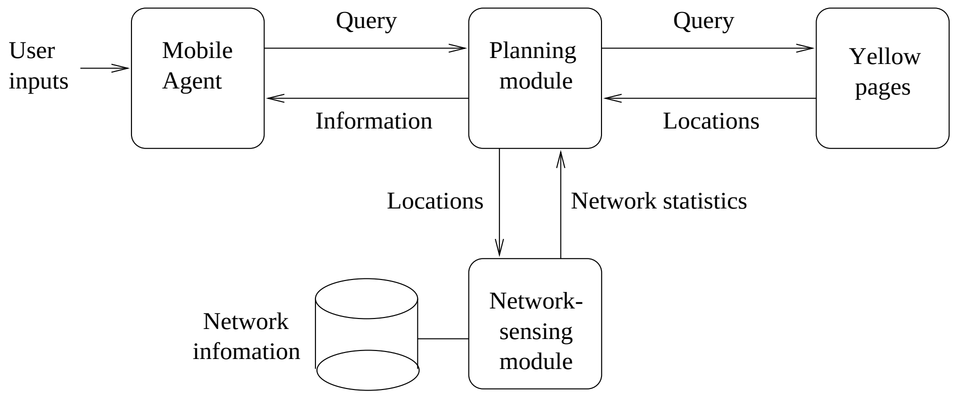

Figure 3: The architecture of the planning system

be as simple as the ratio of data cached at a proxy server to the full amount of data available at the actual server.

After obtaining the list of machines and their corresponding probabilities of success, the planning module passes the list to the network-sensing module, which returns the latencies and bandwidths between the machines and their current CPU loads. The network-sensing module keeps track of these statistics by probing the network at fixed intervals.

As soon as the network statistics are returned to the planning module, the sequence in which agents are to visit machines (to minimize total expected execution time) is calculated from the network statistics and probabilities of success. The calculation is done using the algorithms and theorems described in the following subsection.

5.1.2 Traveling Agent Problems

The planning problem can formulated as deciding the sequence of machines to visit to minimize the total expected time until the desired information is found. We name the planning problem the *Traveling Agent Problem* (TAP) due to the analogy with the Traveling Salesman Problem [GJ79]. Formally, the Traveling Agent Problem is defined as follows:

> *The Traveling Agent Problem* – There are $n+1$ sites, s_i with $0 \leq i \leq n$. Each site has a known probability, $0 \leq p_i \leq 1$, of being able to successfully complete the agent's task, and a time $t_i > 0$, required for the agent to attempt the task at s_i regardless of whether it is successful. These probabilities are independent of each other. Travel times or latencies for the agent to move between sites are also known and given by $l_{ij} \geq 0$ for moving between site i and site j. When the agent's task has been successfully completed at some site, the agent must return to the site from which it started (i.e., site 0). For site 0, $p_0 = t_0 = 0$. The *Traveling Agent Problem* is to minimize the expected time to successfully complete the task.

A solution to the Traveling Agent Problem consists of specifying the order in which to visit the sites, namely a permutation $< i_1, i_2, ..., i_n >$ of 1 through n. Such a permutation will be called a *tour* in keeping with the tradition for such problems.

The expected time to complete the task or visit all sites in failure, for a tour $T =< i_1, i_2, ..., i_n >$ is

$$C_T = l_{0i_1} + t_{i_1} + p_{i_1} l_{i_1 0} + \sum_{k=2}^{n} \left\{ (\prod_{j=1}^{j=k-1} (1 - p_{i_j})) \right\} (l_{i_{k-1} i_k} + t_{i_k} + p_{i_k} l_{i_k 0}) + \prod_{j=1}^{n} (1 - p_j) l_{n0}. \qquad (1)$$

This formula can be understood as follows. The first site, s_{i_1}, on the tour is always visited and requires travel time l_{0i_1} to be reached. Upon arrival, time t_{i_1} must be spent there regardless of success. With probability p_{i_1}

Case	# of agents	Latency	Probability	Computation Time	Complexity
1	Single	Variable	Variable	Variable	NP-Complete
2	Single	Constant	Variable	Variable	Sorting (P)
3	Single	Constant in the same subnetwork	Variable	Variable	Dynamic Programming
4	Multiple	Constant	Variable	Variable	NP-Complete
5	Multiple	Constant	Constant (0)	Variable	Partitioning (PP)
6	Multiple	Constant	Constant ($>$ 0.5)	Variable	Sorting (P)
7	Multiple	Constant	Variable	Constant	Sorting (P)

Table 1: Variation of Traveling Agent Problems

the task is successfully completed in which case the agent can return to site 0 with time cost $l_{i_1 0}$. However, with probability $(1-p_{i_1})$ there was failure and the agent proceeds to site i_2. The contribution to the expected time to moving from site i_1 to site i_2 and succeeding there is

$$(1 - p_{i_1})(l_{i_1 i_2} + t_{i_2} + p_{i_2} l_{i_2 0}).$$

Here the factor $(1 - p_{i_1})$ is the probability of failing at site i_1. Similarly, the contribution to the expected time due to moving from site i_2 to site i_3 and succeeding there is

$$(1 - p_{i_1})(1 - p_{i_2})(l_{i_2 i_3} + t_{i_3} + p_{i_3} l_{i_3 0}).$$

Here the $(1-p_{i_1})(1-p_{i_2})$ term is the probability of failing at both sites i_1 and i_2. In general, the contribution to the expected time due to site i_k is

$$\text{(probability of failure at the first } k-1 \text{ sites)} \times \text{(expected time for success at site } i_k).$$

Adding all these contributions together gives us the summation in (1). Finally, the last term in (1) arises when failure occurs at all nodes and we must return to the originating site. We have used independence of the various probabilities here. Not surprisingly, this problem is NP-complete [Moi98].

5.1.3 Variation of Traveling Agent Problems

Because of its NP-complete complexity, some simplifying assumptions have to be employed so as to more easily obtain optimal solutions for the Traveling Agent Problem. There are several variations of Traveling Agent Problems depending upon the assumptions employed. These assumptions are made regarding the four entities of Traveling Agent Problems, (1) the number of mobile agents, (2) the network latencies, (3) probabilities of success and (4) the task computation time at each machine. Table 1 shows the complexity of each of the Traveling Agent Problems when these assumptions are employed.

We present only the single-agent cases in this section. Please see [Moi98] for a thorough discussion of the multiple-agent cases.

The complexity of the single Traveling Agent Problem can be reduced when latencies between nodes are assumed to be equal. For example, if the processing time at each node is extremely large (compared to the latency between the nodes), differences among the latencies could be ignored, or even taken to be zero. Alternately, if no information about internodal latencies is known, we might assume all of them to be constant. The constant latency assumption is reasonable in the case of a single subnetwork as well.

Theorem 1 *Under the assumption that the all the latencies are constant, the TAP can be solved in polynomial time. The optimal solution for the TAP is attained if the nodes are visited in decreasing order of* $p_i/(t_i + l)$.

Refer to [Moi98] for the proof. The proof uses an interchange argument commonly used in finance and economics [Ber87], in which we determine the relative merit of exchanging the order of visiting two machines. The criteria for exchange is precisely that the machine with a larger value of $p_i/(t_i+l)$ be visited first. When all necessary exchanges have been made, a sorted list of $p_i/(t_i+l)$ results.

Many more complicated situations can be modeled by variable latencies that are constant within subnetworks and across subnetworks. Specifically, consider the case of two subnetworks separated by a great distance (say, one in Japan and one in the US). Latencies between any two nodes within the same subnetwork are treated as constant, as are latencies across the two subnetworks. That is, for sites in Japan, latencies are a constant, l_J, and in the USA they are l_U. Latencies between two nodes, one in Japan and one in the USA, are known to be a third constant, l_{JU}. Formally, we define the Two Subnetwork Traveling Agent Problem (TSTAP) as follows.

> *Two Subnetwork Traveling Agent Problem* — The relevant sites belong to two subnetworks, S_1 and S_2. Sites in S_i are s_{i_j} where $1 \leq j \leq n_i$. n_i is the number of sites in subnetwork S_i. There are three latencies: $L_1, L_2, L_{12} \geq 0$. For $s_{1_j} \in S_1, s_{2_k} \in S_2$, $l_{1_j 2_k} = l_{2_k 1_j} = L_{12}$ while for $s_{1_j}, s_{1_k} \in S_1$, we have $l_{1_j 1_k} = l_{1_k 1_j} = L_1$. Similarly, for $s_{2_j}, s_{2_k} \in S_2$, we have $l_{2_j 2_k} = l_{2_k 2_j} = L_2$. Probabilities, $p_{i_j} > 0$ are nonzero and independent as before. Computation times $t_{i_j} \geq 0$ are arbitrary but nonnegative. The home site, s_0 can be in a third sub-network. Latencies between s_0 and sites in S_i are L_{0i}. We assume that $L_{0i}, L_{12} \geq L_i$. That is, latencies within a subnetwork are smaller than latencies across networks and to the home sites.

Theorem 2 *The Two Subnetwork Traveling Agent Problem (TSTAP) can be solved in polynomial time using the algorithm in Theorem 1 and dynamic programming.*

Outline of algorithm – The algorithm in Theorem 2 consists of two steps. The first step is to sort machines within the same subnetwork in decreasing order of $p_i/(t_i+l)$, which can be accomplished in $n_i \log n_i$ steps. This sorted ordering is used in the second step, where a dynamic programming algorithm is used to compute the optimal solution. Actions taken in the dynamic programming are either to stay in the same subnetwork and migrate the next unvisited machine there, or to migrate the next unvisited machine in other subnetwork. Even though the problem is stochastic, it can be solved by a deterministic dynamic programming algorithm in roughly $O((n_1+1)(n_2+1))$ steps [Moi98].

5.1.4 Experimental results

Next, we show the result of an experiment where a single mobile information-retrieval agent must search for information in the network with the assistance of the planning (module) agent.

In the experiment, information-retrieval agents were launched, and the time each agent takes was measured. The task of the agent is to open a certain text file (the size is 234KB) in a text database on a machine, and parse the file to determine if the file satisfies a given query. The outcome on a machine is determined by a random number generator, so that the probability of success is the same as that given by the directory service agent. Note that the result of parsing the text while looking for a given query does not affect the success of the task, which is in fact decided by the random generator. If the search at a machine is successful, the information retrieval agent returns to the home machine where it was launched. Otherwise, it migrates to the next unvisited machine.

We ran experiments using seven laptop computers distributed in three subnetworks; one subnetwork contained the home site, and the other two subnetworks contained the document collections. We introduced artificial delays on network links so that the latencies between sub-networks were much larger than the latencies between machines within the same subnetwork. To use the TSTAP algorithm, we set the latencies both within and across subnetworks to be constant.

For the sake of comparison with our optimal planning algorithm, two greedy algorithms are employed, one of which is based on the probability of success and the other on the estimated computation time at each

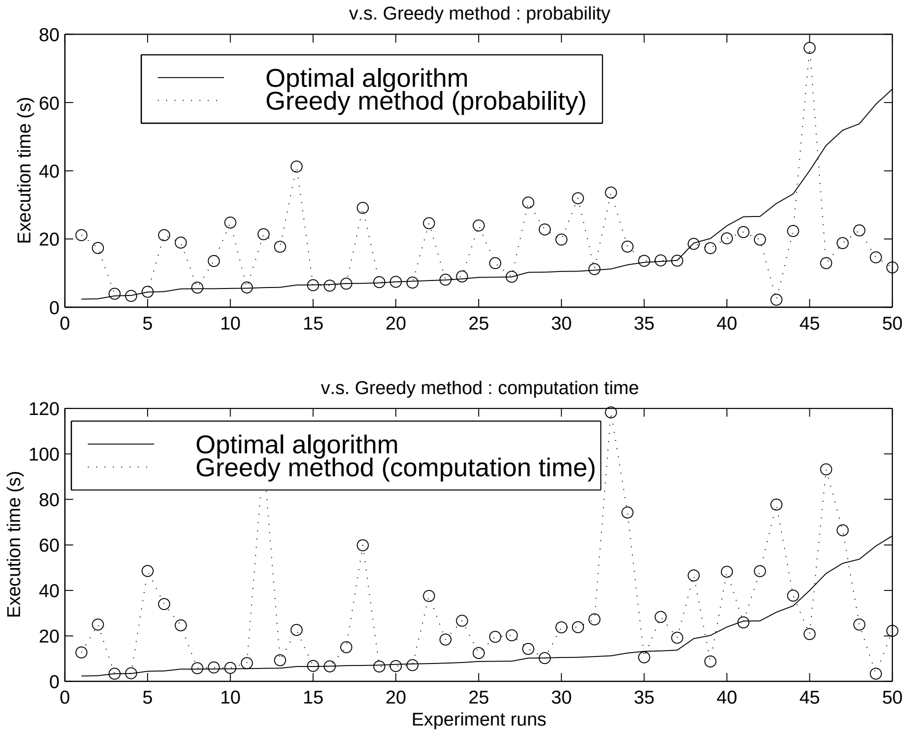

Figure 4: Performance comparison: execution time. For each experimental run, we generated a new set of probabilities, and ran both the optimal and greedy algorithms with the same set. On the two graphs, the x-axis is the run number, and the run numbers are assigned in order of increasing optimal-algorithm execution time.

machine. Note that the estimated computation time at each machine is obtained based on its current CPU load, its benchmarked CPU performance, and the estimated size of a task.

The results of the experiment are shown in Figure 4 and Table 2. The top graph shows the results for the greedy algorithm that uses success probabilities, while the bottom graph shows the results for the greedy algorithm that uses estimated computation times. As we can see in the figure, the optimal planning algorithm does not always outperform the other two methods. This is due to the stochastic nature of the planning problem. For example, an agent may find the information at the first machine even if it has a small probability of success. The optimal algorithm only guarantees the minimum *expected* time until the desired information is found, not the minimum time in all cases. Thus, due to the stochastic character of the planning problem, it is more appropriate to compare algorithms based on the average values shown in Table 2. According to the results in the table, we can see the optimal algorithm outperforms the other algorithms.

The weighted arithmetic mean and the geometric means in Table 2 are defined as follows:

	Optimal algorithm	Greedy algorithm (probability)	Greedy algorithm (computation time)
First place finishes	27	15	8
Geometric mean	1	1.31	1.79
Weighted arithmetic mean	1	1.26	1.67

Table 2: Performance comparison: TAP. The geometric and arithmetic means are defined in the text, but essentially are normalized execution times.

- Geometric mean: The geometric mean of the times for each method is

$$G_k = \left(\prod_{i=1}^{n} \frac{Time_{M_k}(i)}{Time_{M_{OPT}}(i)} \right)^{\frac{1}{n}}.$$

  where $Time_{M_k}(i)$ stands for the execution time of method M_k on its ith run, $Time_{M_{OPT}}(i)$ stands for the execution time of the optimal algorithm on its ith run, and n is the number of runs.

- Weighted arithmetic mean: The weighted arithmetic mean of the times for each method is

$$A_k = \frac{1}{n} \left(\sum_{i=1}^{n} \frac{Time_{M_k}(i)}{\sum_{j=1}^{m} Time_{M_j}(i)} \right)$$

  where $Time_{M_k}(i)$ is the execution time of method M_k on its ith run, $Time_{M_j}(i)$ is the execution time of method M_j on its ith run, n is the number of runs, and m is the number of methods.

  This value is the average percentage of the combined execution time used by method M_k.

5.2 Observation agents

The planning methods described in the previous section all relied upon information provided by the yellow pages service. This service provided the probability of success in a search for certain kinds of information at possible machines. As the information available on the network is in constant flux, the yellow pages must be kept up-to-date by adding new sites, removing old ones, and re-indexing sites that have changed. Our yellow pages index entire document sites; other yellow pages might index *particular* documents, such as World Wide Web pages. We consider the best ways to maintain such indices so as to catch changing content quickly. During the discussion, we will use the word *document* extensively, but the approaches apply equally well to document collections.

To solve the indexing problem, limited computational, network, and storage resources are devoted to scouring available collections for new documents, and also re-examining old documents to inspect them for changes. Whether done in sequence or in parallel, a search engine must always decide what document or documents to examine next. There are many questions to consider: when is the best time to re-examine a document, given knowledge of the document's history and the priority placed on having correct knowledge of its state? Indeed, how should we describe a document's state?

If resources were unlimited, the solution is simple: each and every document could be monitored as frequently as desired, watching for changes to appear. Of course, an observation does have obvious costs associated with it: a machine uses time (some network latency and some CPU cycle time) to retrieve and inspect a document, and disk space to store the results. In exchange for this cost, the search engine benefits from a more current index of previously explored documents, a more comprehensive collection (if new documents are discovered), and an accurate picture of the "dynamics" of the document in question.

An understanding of how documents change is necessary to maximize the recency of such an index. Knowing a document's change dynamics allows us to make fewer wasteful observations. When the engine must decide which document to examine next, some documents will be more likely to have changed since last inspection than others. If looking for changes, it makes sense to re-examine these documents more often than documents which exhibit greater stability. Algorithms for selecting which observation to make next can also account for what the collection may look like as a result of document checks yet to be run. That is, planning observations can take into account the likely outcome of making those observations. This gives rise to planning problems similar to those discussed in the context of the traveling agent problem (TAP). If the index is being used for user searches, then it has the most value for frequently-requested documents. It seems reasonable that resources should be preferentially allocated to the documents that are popular, fast-changing, or both.

5.2.1 Analysis and modeling

To mathematically demonstrate how this allocation should proceed, three things are essential: a representation of the document's state, the dynamics of state evolution, and a formalization of the value of perfect state knowledge.

Time-since-modification as a Markov chain If probability distributions on document ages can be determined, we can use the age to define the modification state of a document. This rests on the assumption that probability of modification is a strong function of the time since the previous modification, which is true for a large class of document changes [Bre98]. Any reasonable discretization of this time will serve our purpose, such as the number of days since the last change, for example. Using this definition of state, a probability distribution of the time intervals between changes can be viewed as a Markov chain. Given a state $s = n$ days since the last change, there are only two things that can occur next: either the state will advance to $s = n + 1$, or it will reset to $s = 0$. If we model $N + 1$ states, then the state $s = N + 1$ can be treated as "N or more days since last change." In this way, we can define a matrix of transition probabilities as

$$\mathbf{M} = \begin{bmatrix} p_{reset}(0) & 1 - p_{reset}(0) & 0 & \dots \\ p_{reset}(1) & 0 & 1 - p_{reset}(1) & \dots \\ \dots & \dots & \dots & \dots \\ p_{reset}(N) & 0 & \dots & 1 - p_{reset}(N) \end{bmatrix} \tag{2}$$

The function $p_{reset}(t)$ can be determined either from knowledge of the distribution of time intervals between changes, or from the distribution of observed ages for a particular document. Either distribution implies the conditional probability that a reset will occur in the following time interval, given that no change has occurred for t time steps. Further discussion can be found in [Bre98].

Generally, by raising $\mathbf{M}$ to an integer power k, we can find the probability (for initial age i) that the system became age j after k time steps have elapsed since the last observation:

$$P(s_{t+k} = j | s_t = i) = \left[\mathbf{M}^k\right]_{ij} \tag{3}$$

Defining a cost function. Using this model, we can define an objective function that can be optimized for the collection. The objective in a real system could be fairly complex; for our discussions, more simplistic criteria are sufficient. In this section we restrict ourselves using some simplifying assumptions. First, we consider a document collection containing d documents. Second, assume that documents can be retrieved at a rate of α documents per day, and that all document fetches have identical cost. Assume the states corresponding to the rows of matrix $\mathbf{M}_r$ denote the age in days of the rth document in the collection. Any unit of time could be used, so long as it is consistent across the collection and the rate α is expressed in the same unit. For each document r, we know that it was last observed k_r days ago to have age i_r. The probability that document r has changed during this k_r-step interval is

$$\begin{aligned} &P(change | \{i_r, \mathbf{M}_r, k_r\}) = \sum_{j \in [0, i_r + k_r - 1]} \left[\mathbf{M}_r^{k_r}\right]_{i_r j} \\ &= 1 - \left[\mathbf{M}_r^{k_r}\right]_{i_r (i_r + k_r)} \end{aligned} \tag{4}$$

Using this result, we can formulate a cost function for the collection. One simple cost function is the expected total number of documents that are incorrectly indexed (the indexed version is out-of-date), which is just the sum of the probabilities listed in (4) over the entire collection:

$$C = \sum_{r=1}^{d} \left(1 - \left[\mathbf{M}_r^{k_r}\right]_{i_r (i_r + k_r)}\right) \tag{5}$$

Greedy cost minimization. Using (5) we can find the best way to reduce the costs that we will incur in the coming day. By our assumption, we can check α documents per day. The smallest possible cost for the following day can be obtained if we fetch and re-index the α documents corresponding to the largest terms in the cost summation. These terms correspond to those documents with the largest probability of being out of date (5). For all of the α documents we fetch, the probability of them now being out of date is zero. If there is no single best choice for the α documents, then we can select α at random from the pool of "best choices." This situation occurs when applying (5) if more than α documents have probability 1 (to working precision).

"Liveness" conditions. While having the advantage of being relatively simple, greedy algorithms may force a situation in which long-term performance is not optimal. For example, in the re-indexing system, there is the possibility of never checking some subset of the documents. This is a situation the indexing system must avoid, especially if all items to be indexed are equally important. In queuing models for computer operating systems, the analogous constraint that all processes be served is termed a "liveness" condition, which would not be met if there were a subset of documents that changed so quickly that its members always contributed the largest terms in the cost function (5). Liveness can be shown to hold for (5) under some simple constraints. The proof centers on the fact that all terms in the cost summation monotonically approach unity if the documents to which they correspond go unchecked. An unchecked document's contribution to the cost will eventually exceed any threshold value $1-\delta, 0<\delta<1$. This is usually sufficient to give assurance of its inclusion in the set of largest terms. See [Bre98] for more.

Extending the cost horizon. Having an assurance of liveness is not enough to be satisfied with the long-term performance of the indexing system. The one-step algorithm does not take into account anything other than the current probability of change for various documents. Indeed, no one-step method can take advantage of knowing the difference in change rates among documents.

Consider the following simplistic system that demonstrates how to take advantage of knowing document change rates. There are two documents, A and B, and we can check one per day. Page A changes quickly: it has a probability of 85% of having been changed today, and if unobserved, it will have a 95% chance tomorrow. If, however, we observe today, there will be a 25% chance of having been changed by the end of tomorrow. Page B changes more slowly. It has an 80% chance today, which becomes 81% tomorrow if B is unchecked today. If it is checked today, then tomorrow's probability will be 5%. Assume we can only choose one of these to observe per day, and that we wish to minimize the total number expected number of documents out-of-date over the two-day period:

$$\sum_{t=0}^{1} \kappa^t \left[Prob(A \text{ changed}, t) + Prob(B \text{ changed}, t)\right] \tag{6}$$

There are four possible strategies in this situation. We can write these as two-day sequences of observations, namely AA, AB, BA, and BB. If we observe a document, then it contributes zero cost on that day, since we consider it "up-to-date" if checked within the last day. Therefore, the cost for observing A on both days is exactly the cost of not observing B on those days, namely, $0.8 + 0.81 = 1.61$. Likewise, we can find the two-day cost for each possible sequence of observations, as shown in Table 3.

To implement a two-day cost function in practice, we need to determine the possible costs we might see on the second day for each document. Since we are assuming that observation forces the document to contribute zero cost on the day in question, the three nonzero costs we need to be able to determine are (i) the cost of not observing on the first day (same as in previous section), (ii) the cost of not observing on the second day if we did not observe on the first day either, and (iii) the cost of not observing on the second day if we did observe on the first day. Moving through these in order, we know that the cost on the first day is of the form:

Sequence	Cost	Comment
AB	$0.8 + 0.25 = 1.05$	lower first day cost; one-day algorithm would pick this one
BA	$0.85 + 0.05 = 0.90$	lower total cost; two-day algorithm would pick this one
AA	$0.8 + 0.81 = 1.61$	document B ignored
BB	$0.85 + 0.95 = 1.80$	document A ignored

Table 3: Possible costs in example two-document, one-check system

$$P_c = \sum_{j \in [0, i_r + k_r - 1]} \left[\mathbf{M}_r^{k_r}\right]_{i_r j} = 1 - \left[\mathbf{M}_r^{k_r}\right]_{i_r (i_r + k_r)} \tag{7}$$

After two days, the form is exactly the same, only the document has aged by one day, accounted for by incrementing the value of k_r. Therefore, if we choose not to observe on the second day, the cost for that day is:

$$\sum_{j \in [0, i_r + k_r]} \left[\mathbf{M}_r^{k_r+1}\right]_{i_r j} = 1 - \left[\mathbf{M}_r^{k_r+1}\right]_{i_r (i_r + k_r + 1)} \tag{8}$$

If we do choose to observe a document on the first day, we can calculate the expected second-day cost of not observing. The second day's costs will be calculated using the new value of $k_r^+ = 1$ (days since last observation) and the newly observed age i_r^+. Since we only know a distribution of possible values for i_r^+, second-day costs will necessarily consist of weighted values from the matrix $\mathbf{M}_r$. These correspond to one-day state transition probabilities in (2). If the document was observed to be in state j_r on the first day, then the probability of it being out-of-date at the end of the second day is just $p_{reset}(j_r)$, as used in (2). These probabilities are also the first column of the matrix $\mathbf{M}_r$. In our cost function, the values in this column vector will contribute in proportion to their probability of occurrence. These probabilities are obtained from the row over which we sum in (7), or the distribution of possible ages on the previous day. Therefore, the probability that the document is out-of-date at the end of the second day, given that the document was observed on the first day, is

$$\sum_{j_r = 0}^{N} \left[\mathbf{M}_r^{k_r}\right]_{i_r j_r} \mathbf{M}_{j_r 1} \tag{9}$$

If we choose not to observe on the second day, then (9) would be the cost contributed for this document. Note that this cost, being a probability, can be no greater than 1.

Even though we can now find the probabilities for a general collection, the computational situation is still rather grim. In a collection of d documents, in which we can check α per day, there are ${}_dC_\alpha^2$ possible strategies. A brute-force approach, in which we evaluate a cost for every strategy, is entirely infeasible for the collection sizes under consideration. Even a simple algorithm will be fairly intimidating for a collection containing literally millions of documents. Our current research [Bre98] includes methods by which to guarantee optimal long-term performance.

5.2.2 Accounting for unequal cost of observation

Throughout the above discussion, we have assumed that we were capable of checking α documents per day. While this may be true in the long run, there is definitely a variation in service times required for the processing of documents. Further, service times can vary dramatically even for a single document. Download and processing times are both proportional to document size, and available bandwidth depends strongly upon the time of day (e.g., one expects long service time around 4:00 PM EST). Our cost function should be modified to account for this variation both among different documents and for a single document.

Deterministic document retrieval times. We assume that all documents require some constant time to process, but that this time may not be the same for different documents. This requires us to restate our objective, since we can no longer count on a constant number of documents processed per day. Specifically, we wish to discover incorrectly indexed documents as quickly as possible.

To quantify this, we introduce some notation. We would like to determine an optimal ordering of documents to check, $S^* = \{s_1, s_2, \ldots, s_d\}$, such that the expected time t taken to find an incorrectly indexed document is minimized. Each document i has a fixed probability P_i of having been changed. In order to consider P_i constant and calculable from (4), we must create a new list S whenever the probabilities P_i have changed. Corresponding to each document, a time T_i is required for processing. The time t expected to find an incorrectly indexed document can be expressed by a probability-weighted sum of these times. If we assume some ordering S as listed above, then this time can be written:

$$t = P_{s_1}T_{s_1} + (1 - P_{s_1})\,[P_{s2}T_{s2} + (1 - P_{s_2})\,[P_{s_3}T_{s_3} + (1 - P_{s_3})\,[\ldots]]] \tag{10}$$

This is the same as the constant-latency TAP presented earlier as (1), in which the solution was to sort by decreasing order of P_i/T_i:

$$S^* = \{s_1, s_2, \ldots, s_d\}\,, \text{ where } \frac{P_{s_1}}{T_{s_1}} \geq \frac{P_{s_2}}{T_{s_2}} \geq \cdots \geq \frac{P_{s_d}}{T_{s_d}} \tag{11}$$

This result is intuitively pleasant—we have moved from obtaining some fixed amount of benefit per document, to an expected benefit per unit time. In an economic context, we think of comparing salaries being offered by different employers. In order to minimize the time taken to acquire our next unit of income, we will always wish to work for the highest salary for as long as we are allowed to do so. This corresponds to the intuitive notion that a document having $T_i = 1$ second and $P_i = 0.9$ would have a payoff rate of 0.9 changed documents observed per second, and would provide the same utility as checking a sequence of two documents both having $T_i = 0.5$ seconds and $P_i = 0.45$. These two could be checked within one second, and if their changes were independent, then the expected changed documents observed per second would also be 0.9. In this formulation, we also assume that there is no reason to prefer a correct index entry for one document over that for another; the value of a correctly indexed document is independent of the document. Correspondingly, in the TAP problem presented earlier, we assumed that there was no reason to prefer one information source over another.

Nonetheless, the earlier problem (TAP) differs from this one in some important ways. First, both the probability of "success" and the processing latency are both strong functions of time. Planning by the P_i/T_i method will only be valid for time scales on which both the probability and time spent are essentially constant. While this may be an appropriate assumption for timescales on the order of a few minutes, it is certainly not correct when used for longer scales. The tradeoff for ignoring variation in the probability and the retrieval time is that we accept that some error will develop in the ordering as time elapses. That is, it is possible that by accepting lower immediate benefit, better long-term benefits might be achieved, just as was the case in the Table 3 example. If using single-step planning, then we must reexamine the order in which we had planned to fetch documents after either the probability or the retrieval time has changed significantly.

Problems due to variation in retrieval times. Using the methods suggested above for documents with non-constant retrieval times will result in indexing preference being given to documents that are either closer to the database or smaller in size. That is, we select between two documents with identical change probability based upon either how far away they are (if of identical size) or how large they are (if at the same location). If we truly value only the expected number of current documents or some other metric that does not account for how the entire collection is treated, then this is not a problem. Intuitively, though, an index should not assign preference based strictly upon convenience of indexing. Methods are needed for removing unwanted bias against documents that are either larger or farther away.

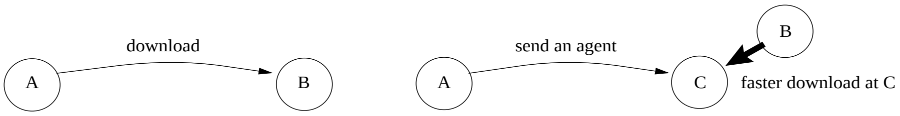

Figure 5: Remote observation

5.2.3 Reducing observation costs using mobile agents

These two biases naturally lead into two targeted solutions whereby we might make limited use of remote sites. First, it is critical that observation time be reduced. The two sources of this time are network-related delay and document size. In this section, we discuss how these difficulties can be ameliorated by moving the observer closer to the data and using encoding schemes so as to enable more frequent observation of large documents.

In order to even consider use of a remote observation post, there must be more machines available for this limited use. We may or may not have significant privileges on these machines, but even a very narrow use, such as simply making an observation from a remote machine, could be useful. Mobile agents are a means by which such limited access might be granted. If we have this access, then we can choose to make observations from machines if it benefits us to do so. By adding mobility to the observer, we give it the freedom to observe in closer proximity to the resource and the chance to perform pre-filtering on the result.

As with the technical report searcher, mobile agents are only a good choice if the additional machines at our disposal can only be utilized through an agent server; if we have full access to a machine, there is no reason not to have a search robot permanently resident on the machine. Mobile agents enable us to take advantage of situations in which one has such limited privileges on a machine. Since relatively few permissions are necessary in order to make observations and perform simple filtering, mobile agents are a viable solution.

The simplest type of remote observer might migrate to the vicinity of a document (or collection) of interest, compress the documents and send them back to the home machine, where they would be decompressed and analyzed. If the remote machines are at a great distance, this could result in a significant time savings. More complicated agents might transmit only the changes in document state in some compressed form. This would be especially appropriate for large documents that only experienced minor changes. Being able to transmit changes in state this way is a large step towards the use of "delta encoding" (analogous to MPEG) schemes for HTTP transmissions [MDFK97]. The key to making this type of encoding work is to package the agent with knowledge of the previous state of the document. Then, when a change is observed, the agent need only transmit the change in the document's index entry, not the entire document or even the entire index entry. This scheme has the desired features of reducing network traffic as well as removing bias against large or distant documents.

As packaging agents with an index entry and a lookup mechanism might produce a rather large piece of code, a better candidate for a remote observer would probably be a simpler filter. Large routines such as compression algorithms might be made available as part of standard libraries on remote machines, preventing us from having to carry compression code from machine to machine. The proxy server agent could be modified so that it would only return requested documents if they did not match a previously hashed version of the document, carried with the agent. This agent could observe these documents more frequently (being closer to the resource of interest) and then send documents back to the server (compressed, if utilities are available to the agent) only if they had changed. A typical retrieval task would entail sending an agent with instructions to look at some set of documents and return compressed versions of those that do not match a hash carried by the agent. Alternatively, the agent could simply notify the home machine that the document had definitely changed.

While mobility may be a valuable option, we need to know the relative merit of observing locally versus using a mobile agent on a remote machine. Consider three machines, A, B, and C. Machine A contains the main document index database, machine B contains a document of interest, and machine C is available for use to a mobile agent. We emphasize that this is the only mode in which machine C can be used; for whatever reason, we are not allowed to compile and install code there on a permanent basis. We wish to determine what observation scenarios favor the use of a mobile observer in this situation. To do this, we consider a comparison of the two scenarios shown in Figure 5. We wish to compare the time it takes to transfer a document directly from B to A versus the time it takes to send a remote observer to C, observe the document at B and return relevant information. Making this more concrete, the document has size S_D bytes. The link from machine i to j has latency L_{ij} seconds, and an effective transfer rate of N_{ij} seconds per byte . If observed directly from machine A, the time for k observations would be:

$$t_1 = k\,(2L_{AB} + N_{AB}S_D) \tag{12}$$

The request for the document is assumed to be small enough that the time taken to transfer it is essentially the same as the link latency. Half of the latency term is due to this request, and half is due to the response. We compare t_1 with the time t_2 taken to perform the same k observations using a mobile agent of size $S_A = \beta S_D$ transferred to machine C. Further, we assume that the agent is clever enough to compress the document to a fraction η of its original size before transmitting ηS_D bytes of index information back to A. The total time is then

$$t_2 = (L_{AC} + N_{AC}\beta S_D + C_{startup}) + k\,[(2L_{CB} + N_{CB}S_D) + \gamma\,(L_{AC} + N_{AC}\eta S_D)] \tag{13}$$

To initiate the agent on the remote machine takes a time $C_{startup}$. For simple agents, this should be the only computation time on the same order as the transfer times. Other computation times, such as compression/decompression times, are assumed negligible by comparison. Also, notice that the agent's messages to the home machine are only required for the fraction γ of the observations on which a changed document is observed. It is immediately clear that the download portion of t_2, namely the term $k\,(2L_{CB} + N_{CB}S_D)$, must be strictly less than t_1 in order to even consider the use of a remote observer. If this is the case, then the sum of the outer terms in (13) must be less than the savings in download time in order for it to be worthwhile to observe remotely. In other words, remote observation is a good option when we save more by downloading at the alternate site than we spend in sending an agent and returning results. The two times can be estimated in advance in order to determine the relative benefit of the two modes of observation.

5.2.4 Multiple tasks, filtering, and change assessment

This was a relatively simple comparison, but we state it to emphasize that agents can present advantages even in simple situations. But an observation agent can be given tasks that are arbitrarily complex. For example, it need not perform observations of only a single document, and it can be free to move to a better vantage point if this saves time. In fact, the agent will multiply its efficiency if it observes many documents that might be resident on the target machine, B. Multiple observations of multiple documents will serve to amortize costs over time. Additionally, as an agent completes observations, it can "diet" by dumping code corresponding to completed tasks, whereupon it can migrate more quickly to the next observation site.

Whether observed by an remote agent, or by a robot running on the home machine, there must be a well-defined means by which to determine whether or not a document has "changed." We have been rather slippery about avoiding explanation of what might be meant by this, so as allow for more general types of observation. For example, the yellow pages index entire document sites according to the content types available at each site, rather than indexing specific documents. Our formulation above, however, is completely general, and the description we gave of document dynamics applies equally well to collection dynamics. We simply need to use a different change-detection function.

If one defines a change in the strict sense of whether or not any bytes were altered, this may be problematic, especially if we are considering changes within an entire collection. There will be situations in which the object in question has certainly changed, but the change that occurred was insignificant. For example, if the documents are Web pages, unimportant changes include the "counter" images on some web pages, randomized advertisements chosen for display, extra whitespace in the HTML source, and anything else essentially unrelated to the page's content. Furthermore, it may be simple to assess what portion of a page's content is of interest. For example, certain robots may be tasked with looking for new links. Changed pages that do not have new links are then no longer of interest. Simple filtering tasks such as this could be carried out in order to determine if a change is of interest.

6 Conclusion

Mobile agents have the potential to be a single, general framework in which a wide range of distributed, information-retrieval applications, such as the technical-report searcher described in this chapter, can be implemented efficiently, easily and robustly. By migrating to the location of a data repository, an agent can access the repository locally and avoid the network transfer of all intermediate data, regardless of whether the server provides low- or high-level operations. By migrating to a high-powered or lightly loaded machine, an agent can gain additional CPU cycles for its computation. By migrating to the other side of an unreliable link, the agent can continue its task even if the link goes down. By migrating to the other side of a low-bandwidth or high-latency link, an agent can avoid transferring partial results and intermediate operations across that link, reducing its total completion time. Most importantly, the agent can decide dynamically how to behave – i.e., migrate sequentially through a set of machines, send out child agents, or remain stationary – according to its task, repository characteristics, machine capabilities, and current network conditions.

To make mobile agents attractive in as wide a range of applications as possible, two key issues must be addressed. First, mobile-agent systems must become more scalable. In the short term, the main scalability problem is the raw performance of the low-level agent infrastructure. Specifically, the overhead of inter-agent communication must be reduced, so that *stationary* agents can compete with traditional client/server implementations. The overhead of agent migration must be reduced, so that an agent will find migration advantageous even in the best network environments and even if it needs to invoke only a few operations per information resource. Lastly, agent execution environments must be able to run agents nearly as fast as if they were natively compiled code; then agents could be used for load balancing tasks, and the load on a "server" machine due to an agent's presence would be only a modest amount worse than if the server implemented the agent's functionality itself. Solutions to all of these implementation problems exist in both traditional high-performance servers and the mobile-agent literature, and the main task now is to identify and combine the most suitable. In the long term, more research-oriented scalability issues revolve around higher-level services, such as agent tracking, debugging and visualization.

Second and more importantly, mobile agents require a wealth of information to make reasonable decisions about when and where to migrate. Numerous support services are needed to obtain and analyze current network, machine, and repository conditions, and then make an effective plan for accomplishing the desired task. Some of these services, such as directories and network-sensing modules, have seen extensive development within other distributed-computing contexts. Much work remains, however, to make these services work well within mobile-agent systems, where software components move rapidly and continuously from one machine to another. Other services are more unique to mobile-agent systems. Such services include planning algorithms that allow a single agent or a small group of cooperating agents to identify the best migration path through the network, as well as algorithms that allow an agent application to determine how to best "observe" a changing document collection. Promising work on both these services was described in this chapter.

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 legion of graduate and undergraduate students who have worked on D'Agents, particularly Katya Pelehkov, Debbie Chyi, Pablo Stern and Ron Peterson, who implemented significant portions of the technical-report application (and associated support services) and are now implementing the next-generation version of the technical-report application; and to the editors, Matthias Klusch and Katia Sycara, for their invitation to contribute a chapter.

Biographies

Brian Brewington received his his B.S. in Engineering and Applied Science, emphasizing robotics and control systems, from the California Institute of Technology in 1995. He is currently in his fourth year of Ph.D. work at the Thayer School of Engineering at Dartmouth College, where he is focusing on problems of optimal allocation of observation resources. His other research interests include signal processing, data mining, and aspects of information theory.

Robert Gray is a Research Assistant Professor in the Thayer School of Engineering. He is the lead researcher and programmer for the D'Agents system, one of the mobile-agent systems discussed in this chapter. He is primarily interested in the performance, security and fault-tolerance of mobile agents. He received his Ph.D in computer science from Dartmouth College in 1997.

Katsuhiro Moizumi is a postdoctoral researcher in the Thayer School of Engineering. He received his Ph.D. degree in computer engineering from Dartmouth College in 1998. His research interests include planning, scheduling, Markov Decision processing, optimal control, machine learning, mobile computing, and agent systems.

David Kotz is an Associate Professor of Computer Science at Dartmouth College. He received the M.S. and Ph.D degrees in computer science from Duke University in 1989 and 1991, respectively. He received the A.B. degree in computer science and physics from Dartmouth College in 1986. He rejoined Dartmouth College in 1991 and was promoted with tenure to Associate Professor in 1997. His research interests include parallel operating systems and architecture, multiprocessor file systems, transportable agents, and parallel computing in computer-science education.

George Cybenko, Dorothy and Walter Gramm Professor of Engineering in the Thayer School of Engineering, received his B.Sc. in mathematics at the University of Toronto, and an M.A. in mathematics and Ph.D. in electrical engineering from Princeton. He has taught on the computer science faculty at Tufts University and was professor of electrical engineering and computer science at the University of Illinois, Champaign-Urbana. At Illinois, he was also a director of the university's Center for Supercomputing Research and Development. He has served as editor for several mathematics, computer and information-theory publications, and has published over fifty journal papers, book chapters and conference papers.

Daniela Rus is an Assistant Professor of Computer Science at Dartmouth College. Previously, she was a research associate and director of the Information Capture and Access project at Cornell University. She holds a Ph.D degree in computer science form Cornell University. Her research interests include distributed manipulation, three-dimensional navigation, self-reconfiguring robotics, mobile agents, and information organization. She holds an NSF Career award and a Sloan fellowship.

References

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

[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.

[Ber87] D. M. Bertsekas. *Dynamic Programming*. Prentice Hall, 1987.

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

[BN84] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[BN95] Ron Ben-Natan. *CORBA: A guide to Common Object Request Broker Architrecture*. McGraw-Hill, 1995.

[BN97] Marc H. Brown and Marc A. Najork. Distributed active objects. *Dr. Dobb's Journal*, (263):34–41, March 1997.

[BP88] Andrea J. Borr and Franco Putzolu. High performance SQL through low-level system integration. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 342–349, Chicago, Illinois, 1988. ACM Press.

[Bre98] Brian Brewington. Ph.D. thesis proposal: Optimal observation with WWW applications. Available from http://comp-engg-www.dartmouth.edu/˜brew/research/proposal.ps, 1998.

[BVW95] Matt Bishop, Mark Valence, and Leonard F. Wisniewski. Process migration for heterogeneous distributed systems. Technical Report PCS-TR95-264, Dept. of Computer Science, Dartmouth College, August 1995.

[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.

[CGH+95] David Chess, Benjamin Grosof, Colin Harrison, David Levine, Colin Parris, and Gene Tsudik. Itinerant agents for mobile computing. *IEEE Personal Communications*, 2(5):34–49, October 1995.

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

[CW97] Mary Campione and Kathy Walrath. *The Java tutorial: Object-oriented programming for the Internet*. Addison Wesley, 1997.

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

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

[Fal87] Joseph R. Falcone. A programmable interface language for heterogeneous systems. *ACM Transactions on Computer Systems*, 5(4):330–351, November 1987.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, 1979.

[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 Agent Security*, Lecture Notes in Computer Science. Springer-Verlag, 1998. To appear.

[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.

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

[JdT+95] Anthony D. Joseph, Alan F. de Lespinasse, Joshua A. Tauber, David K. Gifford, and M. Frans Kaashoek. Rover: A toolkit for mobile information access. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 156–171, Copper Mountain, Colorado, December 1995. ACM Press.

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

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

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

[LC96] Danny B. Lange and Daniel T. Chang. IBM Aglets Workbench: Programming mobile agents in Java. IBM White Paper, 1996.

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

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

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

[MDFK97] J. Mogul, F. Douglis, A. Feldmann, and B. Krishnamurthy. Potential benefits of delta-encoding and data compression for HTTP. In *Proceedings of ACM SIGCOMM'97 Conference*, pages 181–194, September 1997. Available from http://www.research.digital.com/wrl/techreports/abstracts/97.4.html.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[WPW+97] David Wong, Noemi Paciorek, Tom Walsh, Joe DiCelie, Mike Young, and Bill Peet. Concordia: An infrastructure for collaborating mobile agents. In *Proceedings of the First International Workshop on Mobile Agents (MA '97)*, pages 86–97, 1997.

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

[WRW96] Ann Wollrath, Roger Riggs, and Jim Waldo. A distributed object model for the Java system. *Computing Systems*, 9(4):265–290, Fall 1996.