Using the Knowbot Operating Environment in a Wide-Area Network

Jeremy Hylton and Guido van Rossum
Corporation for National Research Initiatives
1895 Preston White Dr., Reston VA 22091

Abstract. Mobile agents can optimize their communication patterns to reduce bandwidth and latency and can adapt to changes in network service. We report on use of the Knowbot Operating Environment to support mobile agents in a wide-area network. Experiments with an application that monitors Web pages for changes show that a mobile program can outperform its stationary counterpart. The performance benefits are obtained by moving clients within the network to reduce the costs of wide-area network connections.

1. Introduction

This paper reports on our experience with a mobile agent system and with applications that use it to optimize wide-area network communications. A mobile agent system allows programs to move among nodes in a distributed system, carrying the programs code and its current state. Applications can exploit this mobility by, for example, moving client code closer to a server to reduce network latency or bandwidth.

There is growing interest in mobile agents, and several other systems that support mobility have been reported on [1,3,7,14,20,23]. The Knowbot Operating Environment is a system for mobile agents developed at CNRI. In this paper, we report on experiments conducted across the Internet over the past year.

We have experimented with a prototype application that uses Knowbot Programs to track updates to World-Wide Web pages. One significant result of our work is demonstrating an application that benefits from mobility. Improving performance — moving computation to a different location to reduce the overall cost of communication — is an important motivations for mobile programs.

In the implementation of the current Knowbot system we chose to emphasize ease of development at the price of some degradation in performance. (The current service station kernel, for example, is implemented in Python, a high-level, interpreted language.) Nevertheless, the system performs well enough to demonstrate applications using mobile code that perform better than their stationary counterparts.

2. Knowbot system architecture

The Knowbot Operating Environment consists of individual service stations that host mobile programs and components that connect the service stations and the users that launch mobile programs. This section reviews the important features of the KOE. The current Knowbot system builds on earlier work at CNRI, including a system developed for the (U.S.) National Library of Medicine. The initial architectural framework is presented by Kahn and Cerf [9]. Current work builds on and extends these ideas.

The Knowbot system is implemented as a runtime layer on top of a Unix operating system. It also relies on CORBA-style distributed objects, currently implemented using the ILU system developed at Xerox PARC [7]. These two characteristics reflect significant design decisions. First, we made portability a top priority. Second, we have provided high-level, RPC-based communication as part of the infrastructure for mobile agents.

A Knowbot Program (KP) can control its location with the `migrate` operation and can duplicate itself with the `clone` operation. A KP migrates with its complete source code (omitting standard libraries available at every service station), its current runtime state, and a suitcase for storing additional data. A KP's implementation centers around a distinguished class. When a KP arrives at a service station, an object of this class is instantiated and its `__main__` method is invoked. The runtime state is migrated by serializing the object graph rooted at the KP's class instance. (The call stack is not captured.) The suitcase provides a second way to transport data; it is a container for data that is accessible via a filesystem-like interface. The suitcase can be easily accessed after the KP terminates.

Knowbot Programs are executed at service stations, hosts running the Knowbot Operating System software [5]. The service stations primary purpose is to allow KPs to run without violating the integrity of the hosts system. The service station kernel also accepts KP migrations, performs process managements, and manages a namespace for inter-KP communication. The service station can be extended to provide new services using plugins.

Each Knowbot Programs runs in its own interpreter process. Initially, the interpreter executes a trusted program called the supervisor. The supervisor sets up a restricted execution environment for the KP and bootstraps the KP. The KP calls back into the supervisor in order to gain access to system and network resources. The supervisor implements a padded cell security model [11, 12] that restricts the KP's access to the host system. Some operations are removed from the KP's environment. Others are wrapped in trusted code that enforces specific security policies.

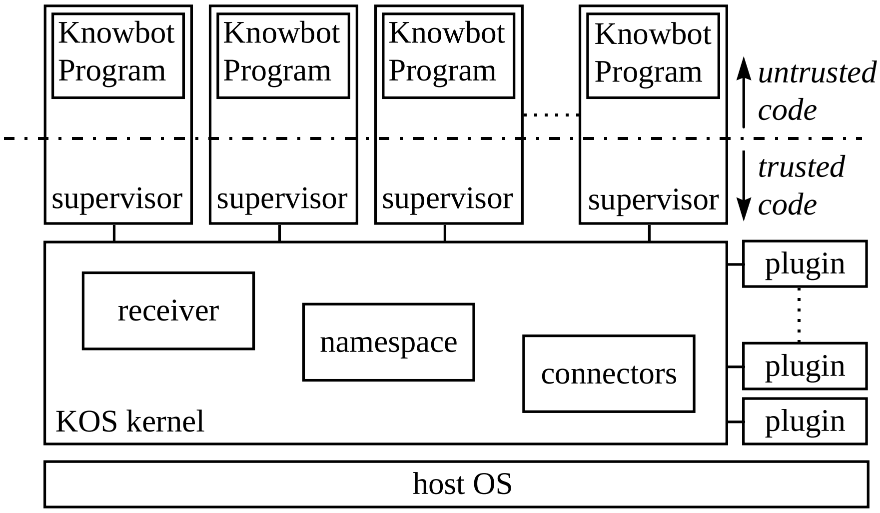

Fig. 1: Knowbot Service Station architecture

In addition to the local runtime environment, the KOE namespace and reporting stations interconnect service stations, Knowbot Programs, and users. The namespace is distributed, hierarchical, and typed; it contains entries for all publicly available objects, including each service station and object implements by KPs. The top levels of the namespace are replicated, based on a design by Lampson [5].

Reporting stations provide a mechanism for tracking a KP as it migrates from host to host and for retrieving the KP or objects in its suitcase after it exits. Every KP must contain a reference to its reporting station; typically, the reporting station is part of the tool used to launch the KP. When a service station receives a new KP, it sends a message to the reporting station. When a KP invokes the clone operation, permission is requested from the reporting station.

3. Application design and performance

We have used the Knowbot system to build an application that checks Web pages for changes. The application exploits mobility to make more efficient use of network resources and outperforms a stationary program that performs that same checks.

The basic idea behind the application is that client-server communication can be optimized by dynamic placement of client code within the network. The importance of mobility is that decisions about where to position code can be deferred until runtime. This allows the application to adapt to network conditions as it runs. Although our initial work involves service stations at the edges of the network, it is a natural extension to consider service stations operating within the network.

Our work on this application has two motivations. First, we wanted to gain experience with the Knowbot system in a wide-area setting. Second, the specific application is an example of the more general activity of monitoring distributed resources that is common to many applications.

The fact that many different applications use the same basic communication pattern is an important argument for mobile code. While we could develop a client-server system that supported the specific Web checking application, mobile code provides the basic infrastructure for many different applications. As Stamos and Gifford [19] argue, the flexibility afforded to application programmers to divide computation between server and client is an important benefit of mobile code.

The rest of this section describes the specific application in detail and presents the results of running this application in our experimental environment. We discuss other applications that might benefit from the same techniques.

3.1 Checking Web pages for changes

The Knowbot Program checks Web pages for changes by loading the page, computing a checksum, and comparing the checksum to the value obtained on the previous attempt. Instead of making HTTP requests for each page from a single location, the KP makes requests from several service stations; it attempts to make the request from the service station closest to the host that servers the page.

The Knowbot checker carries a record for each page that includes the URL for the page, an MD5 checksum for the page, the service station to load it from, and other metadata. (Statistics about how long it takes to retrieve the page are stored in this record.)

The Knowbot checker improves performance by loading each page from the service station that is "closest" to the page's Web server. As a result, the HTTP request is performed across a shorter, faster network connection that involves fewer routers and fewer congested links than loading the page from the user's machine. The KP also conserves bandwidth because it only carries the MD5 checksum of the page from the service station back to the users machine. The KP migrates from service station to service station in arbitrary order. The user submits it to an initial station, where it checks all the pages that are marked for loading from that station. When it has finished, it picks a random next station.

To perform effectively, the Knowbot checker must choose a good service station to load each page from, i.e. a service station that is close to the Web pages server. We sketch two alternatives here, one that we have implemented and another we are currently working on. The current implementation uses an adaptive policy: Initially, each page is assigned to a random service station. Each time the KP runs, a few pages are loaded both from their assigned stations and from an alternate station. If the alternate station is faster than the currently assigned one (based on that station's average performance), the page is assigned to the alternate server.

Ideally, the assignment of pages to service stations would be performed while the KP runs, based on information about the current network topology and performance. This information could be determined by actively monitoring the end-to-end links that KP might use, e.g. the Komodo distributed resource monitor [15], or by computing it at the router level, where link state protocols (OSPF) and path vector protocols (BGP) already provide some useful information.

3.2 Experiments and performance

We conducted two experiments with the Knowbot checker using two service stations located on either coast of the United States. The first experiment compared the time to load Web pages from a stationary program to the time to load the pages using the KP checker. The second experiment measured the overhead the Knowbot system adds to the user code.

Both experiments used 97 URLs selected from one of the authors' bookmarks files. The URLs selected were all within the continental United States; the geographic distribution was relatively even.

We relaxed the security restriction on KP opening sockets because we found the overhead of the HTTP plugin was too high. We discuss this change and the tradeoffs involved below (section 4). The relaxed security model, however, is still prevents the KP from reading or writing files on the local filesystem.

3.2.1 Comparing stationary and mobile checkers

In the second experiment, we compared the performance of the KP checker with the same code running as a standard, stationary Python program. A trace-driven analysis shows that the KP outperforms the stationary application in a majority of cases.

We chose to use a trace-driven analysis so that we can compare a variety of policies for assigning pages to service stations. Our informal use of actual KPs supports this conclusion.

We collected data for this experiment by running a stationary version of the checker at each of the service stations. Each program logged how long each HTTP request took. We ran the programs 108 times at each site between 1 p.m. and 4 p.m. EST on a weekday.

Using these traces we can consider the performance of several different policies for assigning Web pages to service stations. Based on the policy and the overhead measurements from the previous experiment, we compute the time for a KP to check all the pages.

The expected performance of three policies versus the two stationary versions is summarized in Table 1. The first policy assign each page to the service station with the lowest average response time for the first 20 trials, and is labeled first-20. The second policy bases assignments on the lowest response time average over all trials. The last policy represents the optimal assignment for each individual trial; it represents how well the KP checker could do with perfect information about future trials.

Application	**Average Time (sec.)**
stationary, CNRI	145.32
stationary, West Coast	95.64
KP-20	81.91
KP-all	66.96
KP-optimal	33.52

Table 1: Time to load 97 URLs with stationary and KP checkers The second column shows the total time to load the URLs averaged over 108 samples. (Time does not include system overhead.)

Because the average time has a high variance, we also consider the winner of each individual trial. In Table 2, we count the trials won by the stationary and mobile versions of the program, using either the first-20 or all policies.

One limitation of these measurements is that network delay isn't the only factor that affects overall application performance. The load on the Web server, in particular, can have a significant effect. As a result, a request from a distant client may complete before a request from a nearby one.

Policy	**CNRI best**	**West Coast best**	**KP best**	**total**
KP-20	15	27	66	108
KP-all	9	18	81	108

Table 2: How often did the KP outperform that stationary program? For two policies, we show the outcome of 108 trials comparing the speed a the KP checker with a stationary program.

The trials show that some simple policies can achieve good performance. We plan to experiment with better assignment policies, including policies that change during execution, in the future.

3.2.2 Measuring system overhead

The second experiment measured the total execution time for the KP, including the costs of assembling the KPs transmission format, the overhead for migration and state management at the service stations, and the actual running of the KP code. We made two sets of measurements, a series of micro-benchmarks and an end-to-end measurement.

Micro-benchmarks were performed by profiling the kernel and KP supervisor. Table 3 shows a breakdown of the overhead based on profiling the supervisor and kernel. All measurements in this section were made on an UltraSparc 2 with 200MHz SuperSPARC processor and 128 MB of memory; times reported here are averaged over 100 samples.

Module	Operation	CPU Time
kernel	receiving the KP, access control	64 ms
supervisor	loading source code	877 ms
supervisor	pickling state	86 ms
supervisor	migration	10 ms (+ network)
supervisor	execute user code	566 ms

Table 3: Sources of overhead in the supervisor and kernel. The times reported are elapsed wall clock time and CPU time including user and system time. Execution time of user code is included for comparison.

The most significant source of overhead is the cost of loading the source code for the KP. Currently, loading the code takes more CPU time than actually executing the agent. The KP carries two general purpose modules, one for page to service station assignment and another for performing HTTP requests. (The latter is nearly 700 lines of source code.)

There are two sources of overhead that can be eliminated at this step. One is an implementation problem: The source code is stored in the kernel process that receives migrating KPs and copied to the supervisor process on a per-module basis. The modules should be loaded at once to reduce the IPC cost, which accounts for approximately half the cost of loading the source code. The second is compiling it to Python bytecodes. Using the standard Python interpreter, we measured the time to load the KP's code from source and from bytecodes. Loading and compiling the source took 435 milliseconds, while loading the compiled bytecodes took only 160 milliseconds. Transporting bytecodes, however, raises security concerns, which could be addressed by a solution like the Java bytecode verifier.

We made a rough measurement of end-to-end performance by comparing the total time for the KP to execute, from initial submission until results were returned, to the time spent executing at each service station. The difference between the two (wall clock) times is overhead — time spent in transit in the network, time spent waiting for the service station to start the KP, etc.

The overhead measurements varied widely, primarily as the result of variations in network performance between the two service stations. The average overhead was 14.13 sec. and the standard deviation was 8.78 sec. For comparison, we used ping and ftp to measure basic network performance between the service stations. The latency between the two sites is about 110 ms, but the packet loss rate is high; losses between 10 and 25 percent are not uncommon. The time to ftp a 50 KB file also varies substantially; on average transmission takes 3.49 sec., but the standard deviation is 4.39 sec.

3.3 Discussion and related work

Many proposed mobile agent applications involve programs that migrate to a server and perform some work there. Our work extends this idea by considering agents that run at intermediate nodes that lie between the end user and the server. As a result, we can benefit from mobile agents without deploying service stations at every server; a few strategically placed servers will suffice. For example, an ISPs network or a corporate intranet might position service stations at gateways to other networks and reduce traffic on its internal network.

The goal of dynamic client placement is to minimize the bandwidth or latency of communication. More generally, mobile agents can be used to make efficient use of network resources and to adapt to changing network conditions. Some examples of resource-aware adaptation that are enabled by running agents at intermediate nodes include:

- Congestion. If a link becomes congested or fails, the KP can move the client to a location that is connected to the server over a different path. In the case of a congested link, moving the client not only offers better performance for the application; it also reduces traffic on the congested link. Recent research [13] suggests that variability in routing behavior and link congestion and failures are not uncommon.
- Bandwidth. In the Web checking example, the KP conserves bandwidth by using a checksum to test whether a page has changed. The KP can be run near the Web server, so the page (which is large) is transferred a short distance, and only the checksum is transferred back to the KP's orginiator.

The Sumatra mobile agent system [1] has been used to experiment with resource- or network-aware mobile programs [15, 16]. One Sumatra application runs a specialized search algorithm over the contents of a remote database; it decides whether or not to migrate to that database based on the estimated cost of copying the data versus migrating the program. The key difference with our work is that the Sumatra agents only run at end systems.

The idea of running agents at inside the network is closely related to recent work on active networks [21, 22]. An active network allows users to inject programs into the nodes of the network, effectively interposing computation between network endpoints. Current work on active networks is primarily at the level of switches or routers that operate on packets containing code. The Knowbot infrastructure, however, interposes computation at a higher level of abstraction.

Mobile computers (as opposed to mobile programs) see much greater variation in the quality of network connections. Mobile systems must cope with periods of disconnection and low bandwidth, and techniques similar to ours have been developed. For example, Rover [8] uses relocatable dynamic objects to move parts of an application from the client to the server.

Dynamic distillation [2] is another technique that has been used to adapt to variations in connectivity. Distillation works by placing a filter at the mobile hosts base station. When the user loads Web pages, for example, the filter can take several measures to reduce the amount of traffic to the mobile host when the connection is bad. The filter can compress data, reduce color images to grayscale, or eliminate images entirely.

Knowbot Programs can be used to move computing from the mobile client to an arbitrary location in the network, not just the base station. For example, if a mobile Web browser is filtering images, it could move the distillation process closer to the Web server and avoid sending data across the Internet only to discard it at the base station.

4. Tradeoffs in implementing security policies

This section discusses some tradeoffs between the use of Python's restricted execution environment and service station plugins to implement security policies. The original implementation favored plugins, but experiments with the KP Web checker showed that performance suffered substantially.

The basic philosophy for KP security is to remove all access to system resources — to place it in a padded cell — and then selectively give access back. (This is basically the principle of least privilege [18].) The language interpreter performs automatic memory management, for example, that prevents the KP from having direct access to virtual memory. We use the supervisor and service station plugins to give access back to the KP. The supervisor adds special versions of selected system calls to the KPs environment. A plugin provides services that are performed by the trusted code of the plugin.

There are several tradeoffs involved in choosing the supervisor or a plugin as the mechanism for providing a new operation to the KP. Using a plugin is fairly expensive, because the plugin runs in a separate protection domain and incurs additional operating system overhead on each call. The supervisor runs in the same protection domain and avoids this overhead. On the other hand, the plugin is more easily configured by a system administrator. Individual plugin modules can be enabled or disabled easily, but adding or removing functions from the supervisor involves editing the source code. A single plugin can also provide service for KPs written in many different languages; each languages supervisor might need to be changed to add a new feature.

Consider the HTTP access required by the Knowbot Web checker. We built a plugin that performed HTTP requests on behalf of its clients. The plugin provides a high-level interface: The KP makes a request for a specific URL; the plugin retrieves it and returns the Web server's response. The plugin, however, added a substantial overhead to each call. Including socket access in the restricted execution environment eliminated this overhead.

Another consideration that affects the choice between supervisor and plugin is the kind of abstraction that each provides and the kind of access policy that can be expressed. For HTTP requests, we want to prevent a KP running at a service station in domain foo.bar from accessing Web servers that are internal to that domain (and potentially behind its firewall). The Web server often restricts access based on the client's IP address, but that would be the service station's IP address in this case.

The HTTP plugin provided a convenient way to insert appropriate access control. The plugin verifies that the URL request by the KP does not refer to an internal server (based on a simple access control file similar to a Web servers access.conf file). It would be more difficult to enforce the same policy in the supervisor. At the socket level it is hard to tell what kind of server the KP is connecting to or what resource is being requested. The internal Web server might run on the same machine as a public server, but use a different port; or the same server might provide access to public and private documents. The plugin is better suited to enforcing this access control policy because the type of object it operates on (URLs) is the same as the type used for access control.

5. Acknowledgments

The Knowbot Operating Environment was implemented by a team including of Fred L. Drake Jr., Ken Manheimer, Roger Masse, Barry Warsaw, and the authors. Bob Kahn provided detailed comments on drafts of this paper. Discussions with David Ely, Ted Strollo, Barry Warsaw, and Dick Binder provided some of the initial motivation for the bookmark checking application. The anonymous referees made many helpful comments. Funding for this work was provided by the Defense Advanced Research Projects Agency (DARPA) under grant MDA972-95-1-0003.

6. References

1. Anurag Acharya, Mudumbai Ranganathan, and Joel Saltz. Sumatra: A language for resource-aware mobile programs. In *Mobile Object Systems: Towards the Programmable Internet* (Lecture Notes in Computer Science No. 1222.), pages 111-130. Springer-Verlag, April 1997.
2. Armando Fox, Steven D. Gribble, Eric A. Brewer, Elan Amir. Adapting to Network and Client Variability via On- Demand Dynamic Distillation. In Proceedings Seventh International Conference on Architectural Support for Progrogramming Languages and Operating Systems (ASPLOS- VII), Cambridge, Ma., October 1996.
3. Robert S. Gray. Agent Tcl: A flexible and secure mobile agent system. In Proceedings of the Fourth Annual Tcl/Tk Workshop, pages 9-23, Monterey, Cal., July 1996.
4. Christian Huitema. *Routing in the Internet.* Englewood Cliffs, NJ: Prentice-Hall, 1995.
5. Jeremy Hylton, Ken Manheimer, Fred L. Drake, Jr., Barry Warsaw, Roger Masse, and Guido van Rossum. Knowbot programming: System support for mobile agents. In Proceedings of the Fifth International Workshop on Object Orientation in Operating Systems, pages 8-13, Seattle, Wash., October 1996.
6. B. Janssen, D. Severson, and M. Spreitzer. ILU 1.8 Reference Manual. Xerox Corp., 1995. Available via Inter- Language Unification Web page.
7. Dag Johansen, Robbert van Renesse, and Fred B. Schneider. Operating system support for mobile agents. In Proceedings of the 5th IEEE Workshop on Hot Topics in Operating Systems, pages 42-45, Orcas Island, Wash., May 1994.
8. A. D. Joseph, A. F. deLespinasse, J. A. Tauber, D. K. Gifford, and M. F. Kaashoek. Rover: A toolkit for mobile information access. In Proceedings of the 15th ACM Symposium on Operating Systems Principles, pages 156-171, December 1995.
9. Robert E. Kahn and Vinton G. Cerf. *The Digital Library Project*, *volume I: The world of Knowbots*. Unpublished manuscript, Corporation for National Research Initiatives, Reston, Va., March 1988.
10. Butler Lampson. Designing a Global Name Service. Proceedings of the 6th ACM Symposium on Principles of Distributed Computing, pages 1-10, Calgary, 1986.

11. Jacob Levy and John K. Ousterhout. Safe Tcl: A Toolbox for Constructing Electronic Meeting Places. In Proceedings of the First USENIX Workshop on Electronic Commerce, New York, July 1995.
12. John K. Ousterhout, Jacob Y. Levy, Brent B. Welch. The Safe-Tcl Security Model. Unpublished draft, 1997.
13. Vern Paxson. End-to-End Routing Behavior in the Internet. ACM SIGCOMM ‘96, Stanford, CA, August 1996.
14. Holger Peine and Torsten Stolpmann. The architecture of the Ara platform for mobile agents. In Proceedings of the First International Workshop on Mobile Agents, Berlin, Germany, April 1997.
15. Mudumbai Ranganathan, Anurag Acharya, and Joel Saltz. Distributed resource monitors for mobile objects. In Proceedings of the Fifth International Workshop on Object Orientation in Operating Systems, pages 19-23, Seattle, Wa., October 1996.
16. Mudumbai Ranganathan, Anurag Acharya, Shamik Sharma, and Joel Saltz. Network-aware mobile programs. In Proceedings of the USENIX 1997 Annual Technical Conference, Anaheim, Cal., January 1997.
17. Guido van Rossum. Python tutorial. Technical Report CS-R9526, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, May 1995. Revised version available from the Corp. for National Research Initiatives, 1996.
18. Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. Proceedings of the IEEE, 63(9):1278-1308, September 1975.
19. James W. Stamos and David K. Gifford. Remote evaluation. *ACM Transactions on Programming Languages and Systems*, 12(4):537-565, October 1990.
20. Markus Straßer, Joachim Baumann, and Fritz Hohl. Mole — a Java based mobile agent system. In Proceedings of the 2nd ECOOP Workshop on Mobile Object Systems, pages 28-35, Linz, Austria, July 1996.
21. David L. Tennenhouse, Jonathan M. Smith, W. David Sincoskie, David J. Wetherall, and Gary J. Minden. A survey of active network research. *IEEE Communications*, 35(1):80-86, January 1997.
22. David L. Tennenhouse and David J. Wetherall. Towards an active network architecture. *Computer Communication Review*, 26(2), April 1996
23. James E. White. Telescript technology: Mobile agents. General Magic White Paper, 1996.