Knowbot programming: System support for mobile agents

Jeremy Hylton, Ken Manheimer, Fred L. Drake Jr.,
Barry Warsaw, Roger Masse, Guido van Rossum

Corporation for National Research Initiatives
1895 Preston White Dr., Reston, VA 22091
knowboteers@cnri.reston.va.us

Abstract

Knowbot Programs are mobile agents intended for use in widely distributed systems like the Internet. We describe our experiences implementing security, process migration, and inter-process communication in a prototype system implemented using the object-oriented programming language Python. This infrastructure supports applications that are composed of multiple, autonomous agents that can migrate to use network resources more efficiently.

1. Introduction

A Knowbot® Program is a combination of data and a thread of control that can move among nodes in a distributed system. The Knowbot Operating System provides a runtime environment for these programs which includes security mechanisms, support for migration, and facilities for communication between Knowbot Programs.

Knowbot Programs enable an agent-based programming style that is well-suited for autonomous and network-efficient applications. Agents are autonomous, able to continue operation even when disconnected from their source, and can migrate closer to data or to other programs they interact with in order to conserve network bandwidth.

Our work is based on the Knowbot framework, introduced by Kahn and Cerf [14], for a mobile software component to the national information infrastructure. Our experimental system explores some aspects of the Kahn-Cerf framework.

Appears in Proceedings of the Fifth International Workshop on Object Orientation in Operating Systems (IWOOOS '96), Seatle, Wash., Oct. 1996.

This paper reviews our experience building a prototype system for supporting Knowbot Programs and reviews some of the underlying services provided by our Knowbot Operating System. It assumes a distributed object framework for communicating with other parts of the system.

Our current implementation uses Python, an object oriented scripting language [21], and ILU, a multi-language object interface system developed at Xerox PARC [10]. We use ILU to provide an object-oriented RPC mechanism for communication between objects. The KOS architecture, however, is language- and transport- neutral.

2. Overview of system services

A Knowbot Program (KP) is code with well-defined entry point and state. The Knowbot Operating System (KOS) is a runtime environment that provides underlying services enabling KPs to migrate and to interact with other programs. The underlying services fall into three major categories: (1) a safe runtime environment, (2) migration and state management, and (3) communication among KPs. Each of these underlying services is described in greater detail below; they are briefly summarized here:

- Security. The prototype system provides several safeguards to prevent the KOS from being damaged by a KP. The KP process is divided between a supervisor, which runs trusted code provided by the KOS, and the KP user code, which is untrusted. The user code runs in a restricted execution environment, which mediates access to unsafe operations. The supervisor performs all restricted operations, like RPC calls, on behalf of the user code.
- Migration. A KP can move between distributed KOSes using two primitives, `migrate`, which

moves the current program, and `clone`, which creates a copy of the current program at a new location. To migrate, the KOS creates a package including the KP's source code, the current state of the program, and a "suitcase" containing application-specific data.

- Connectors. Connectors are a thin veneer over an object-oriented RPC mechanism that regulates the creation and publication of new objects. All connectors are named and strongly typed. A KP can lookup a particular connector by name or request a group of connectors that provide the same interface.

Our use of connectors and ILU offers language independence for the Knowbot runtime environment. Any language that can support migration, has an ILU binding, and a safe way to restrict access to unsafe operations can be used to write Knowbot Programs.

The rest of the paper describes our experience implementing Knowbot Programs and the underlying services.

3. Security

There are several levels of security needed in a mobile agent system; they include providing secure transport for KPs between KOSes, protecting the KP from tampering by the KOS, and protecting the KOS from malicious KPs. Our current implementation addresses only the last issue – executing the KP in safe environment.

Security in the KOS is based on the strict separation of responsibilities between trusted and untrusted parts of the system. Untrusted user code runs in a restricted environment that is created for it by trusted supervisor code.

The restricted environment is indistinguishable to the untrusted code running within it, with the exception that various potentially unsafe operations are inaccessible. There are many potential unsafe operations – creating network connections, modifying files on the local disk, or communicating with other KPs executing at the same node. The trusted code removes some operations altogether and creates wrappers around other operations that enforce security policies.

For example, the supervisor may provide an `open` operation that allows read and write operations only in particular directories. The `open` operation exported to user code would call into the supervisor, where safety checks could be made before making the actual system call.

The KOS security model also guarantees type-safe access to distributed objects by disabling access to an object's instance variables and by performing runtime type-checking on all method calls. The trusted code creates a "bastion" object that only allows calls to specific instance methods. (The Thor object-oriented database [16] provides similar type-safe interaction using static type checking and encapsulation.)

A widely deployed mobile agent system will require stronger security measures than our prototype. For example, The KOS should be able to identify the owner of the KP and verify its integrity, based on a digital signature or encryption. Agent Tcl [8] uses PGP encryption for authentication and protection. When an agent is created it is signed or encrypted by its owner and submitted to a server; when an agent moves between two servers, the originating server encrypts the agent. The Agent Tcl system assumes that each server trusts the others (and their public keys).

4. Migration

Knowbot Programs control their location using two related operations, `migrate` and `clone`. A KP calls `migrate` with the name of the destination KOS; the supervisor interrupts the KP, captures its current state in persistent form, and sends it to the specified KOS where execution resumes. The `clone` operation is the same as `migrate`, except that the `clone` call returns and execution continues at the original KOS.

Knowbot Programs are transported between KOS nodes as MIME documents. The MIME representation includes the program's source code, a pickled version of its running state, its "suitcase" (which holds data files created by the KP), and metadata that describes how it should be handled by the KOS. The metadata includes the KP's origin, the name of the module that contains the KP entry point, and instructions for handling exceptions and errors.

To support migration, the KOS must be able to stop a running KP, serialize its state, and restart the KP at another node based on that state. In our current Python implementation, a KP always resumes execution at a single entry point – its `__main__` method. In the future, we intend to support true stack mobility, which would allow a migrating KP to resume execution at any point in the program, preserving its current call stack.

The KP's state includes all data stored within the KP object instance and references to other objects existing within the restricted KP environment, including connectors. Objects in the supervisor are not considered part of this state.

In Python, the KP's state is captured using an extended version of the pickle library, which generates a machine-independent representation of complex objects. Starting with a root object, that object and any object it holds a reference to are added to the pickle.

The KP pickler supports custom pickling operations for objects. In the case of connectors, a reference to the server's object and the type of the object are placed in the pickle, and the unpickling method re-establishes the connection with the server. (The current implementation does not address the reverse problem – moving the KP without invalidating connectors to services it provides. Shapiro et al. [18], however, describe a solution using a chain of references that point from the node where an object resided to the node it migrated to.)

KPs also have access to a transported file system, or suitcase, to carry data independently of the pickled program state. The suitcase holds application-created data that isn't stored as an instance variable of the KP object, e.g. a log of KOSes visited or the results of a remote search. For convenience, the suitcase acts like a hierarchical file system. The suitcase offers two significant advantages to applications:

- Files in the suitcase can be accessed without running the KP. Thus, an application that uses a KP to perform a remote operation can retrieve the results without incurring the overhead of starting a Python interpreter and the KP.

- The suitcase gives better performance to applications that create custom data representations. For example, a KP that indexes Web pages might write its index in binary form directly to the suitcase and later transfer the index directly to a search service.

The Tacoma system [11] provides a similar facility for creating and carrying files – a "briefcase" that holds one or more "folders." Tacoma also allows an agent to store folders at the server, so that it can store site-specific data for later use.

5. Connectors

Independently-running processes, including KPs and the KOS kernel, communicate with each other using connectors. Connectors are layered on top of ILU objects, adding mechanisms for creating objects and sharing references to them.

Connectors preserve the integrity of the restricted execution environment, which could be compromised by offering lower-level access to object RPC mechanisms. A client KP uses connectors to request a service, specifying a name and a type, and the KP supervisor creates a client-side surrogate object that communicates with the process offering the service.

Programs offering services publish their services using the connection broker, which binds connectors to instances of class objects. The services class instance is bound to a symbolic name and an interface type registered with the KOS.

Knowbot programs define their own class objects and interface types using interface definition language, which supports a large subset of CORBA functionality. KPs communicate with each other using connectors to these well-defined interfaces. For example, a KP that searched a remote database would migrate to the KOS managing the database and request a connector for the database's search interface.

Clients can request a connector for a known service by specifying the service's name and type. There are several other basic properties of connectors:

- Clients can also look for all connectors of a particular interface type.

- Connectors are first-class; they can be be delivered by other objects.

- Clients can carry connectors from one station with them as they travel to others, and maintain contact with the services they represent.

This connector architecture enables creation of add-on directory, or "trader," services that track connectors based on more specific properties. A directory service could be implemented by a KP that exports a directory interface to clients.

6. Applications of Knowbot technology

An example of a complete Knowbot Program written in Python is shown in Figure 1. The KP searches up to 20 random KOSes looking for services that implement the `Search.Boolean` interface, storing a list of those services in its suitcase. The code in Figure 1 shows a class definition for the KP that has four instance methods. The `__main__` method, invoked when the KP arrives at a new KOS, receives a bastion KOS object as its second argument; this object provides access to KOS services like connector lookup and migration.

More interesting applications of Knowbot technology include applications that make more efficient network bandwidth by moving computation closer to data

```
import rand                         # Python random number module
import nstools                      # helper module for using KOS namespace

class KP:

    def __init__(self):
        "Initialize KP's instance variables."
        self.maxhops = 20
        self.hopcount = 0
        self.visited = []           # list of KOSes that have been visited

    def __main__(self, kos):
        "Finds services available here, then migrates to a new KOS."
        self.find_services(kos, 'Search.Boolean')
        self.visited.append(kos.get_kos_name())
        self.hopcount = self.hopcount + 1
        if self.hopcount < self.maxhops:
            places = self.get_new_places(kos)
            if places:
                kos.migrate(rand.choice(places))

    def find_services(self, kos, service_type):
        "Save a list of available services in the suitcase"
        services = kos.list_services(service_type)
        file = kos.get_suitcase().open(kos.get_kos_name(), 'w')
        for serv in services:
            file.write(serv.name + '\n')
        file.close()

    def get_new_places(self, kos):
        "Return list of KOSes that have not been visited."
        descriptor = nstools.Lookup(kos.get_namespace(), 'world/kos')
        context = descriptor.Open('Namespace.Context')
        places = []
        for place in context.List():
            if place not in self.visited:
                places.append(place)
        return places
```

Figure 1. Example Knowbot Program

or that implement widely distributed systems on top of loosely coupled, autonomous Knowbot Programs.

One example of the network-bandwidth-conserving Knowbot Program is one that performs a search in an image database. Instead of loading each image over the network and applying some computation to it, the KP moves to the database, performs the search there, and returns with the results.

The searching example can be extended to a more general indexing Knowbot Program, where a KP moves to a database to build an index of its contents. The KOS allows multiple search services to each build their own customized index of database without copying the database's entire contents [9].

Intellectual property rights management and control of caching and replication are areas where the ability to create autonomous Knowbot Programs is valuable. A Knowbot Program can act as a courier for data for which access is restricted. The KP carries an encrypted version of the data and requires some authentication or payment to decrypt it, perhaps interacting with another KP that carries a key for decryption.

We can generalize this example to a general mechanism for providing caching and replication of objects on the World-Wide Web. We envision a proxy server that runs Knowbot Programs. A content provider interacts with a proxy server by sending a group of objects managed by a KP. The manager program could enforce access controls, perform specialized logging (hit counts), or generate dynamic pages using a database copied from the content provider. The manager also helps deal with the cache consistency program, because the manager can contain site-specific code for make decisions about freshness.

7. Related Work

An increasing number of agent-based programming systems are being described in the research literature. Support for mobility in these systems builds on earlier work on object migration.

Emerald [13] was one of the first systems to support fine-grained mobility for objects and processes, i.e. a thread executes within an object and moves with that object. The Emerald system was designed for a small-scale network of homogeneous computers, although a recent paper discusses mobility among heterogeneous computers [19].

Object migration is also of interest in mobile computing, where there is great need to reduce bandwidth requirements and cope with intermittent lack of connectivity. The Rover toolkit [12] uses relocatable dynamic objects to move computation between servers and mobile clients. However, these objects do not maintain an active thread of control as they move.

Recent work on agent technology includes several systems using high-level scripting languages like Tcl and the commercial Telescript system from General Magic.

Agent Tcl [8, 15] extends the standard Safe Tcl interpreter with facilities for migration and resource allocation. The system provides for encrypted and authenticated transport of agents and for limited control over the resources an agent can use (e.g. CPU time, disk space).

Another agent environment using Tcl is Tacoma [11], which also supports agents written in Perl, Python, and Scheme. In Tacoma, agents communicate using shared files, or "folders:" One agent places some data in a folder and issues a `meet` instruction specifying another agent. That agent begins execution with the suitcase from the first agent. All system services are structured as agents run by `meet`.

Obliq [5] is a scripting language for distributed object-oriented computing that is based on a network object [4] model. Bharat and Cardelli [3] describe several interactive applications that migrate the user interface to the user' site.

General Magic has developed a commercial agent system centered around its programming language Telescript [22]. Telescript addresses migration, security, and resource control. The system, however, exposes a complex security model to the programmer [20] and does not support programs written in more common scripting languages.

Research in safe programming languages is an important enabling technology for agent systems. The Safe-Tcl and Java languages also offer restricted environments. Sandboxing [2] is an alternative to Python's restricted execution environment.

Java has also been proposed as a language for agent programming, but the language itself does not provide necessary support services for agents. Using Java applets involve many of the same security concerns as agents [7]. Several projects have proposed to use or are using Java for agent system: Sumatra [1, 17] is an extension to Java that supports mobile programs that adapt to changing network conditions. The Open Software Foundation has proposed a middleware system written in Java [6].

8. Conclusions and future work

We expect to refine and extend the current prototype of the Knowbot Operating System and make its source code available to other researchers in the coming year.

There are several unexplored aspects of Knowbot programming that will be addressed in our future work: (1) developing a broader security model for KPs that addresses access control, authentication and verification of KPs and KOSes, and resource management, (2) implementing support for KPs written in multiple languages, (3) using migration to experiment with scheduling and load balancing algorithms, and (4) instrumenting the system to study efficiency and performance. We are also developing several real-world applications to confirm our expectations about the usefulness of Knowbot programming.

9. Acknowledgments

Amy Friedlander made many helpful comments on this paper. Our work was supported by the Advanced Research Projects Agency of the United States Department of Defense under grant MDA972-95-1-0003.

References

[1] A. Acharya, M. Ranganathan, and J. Saltz. Distributed resource monitors for mobile objects. In *Proceedings of the Fifth IEEE International Workshop on Object-Orientation in Operating Systems*, Oct. 1996.

[2] A.-R. Adl-Tabatabai, G. Langdale, S. Lucco, and R. Wahbe. Efficient and language-independent mobile programs. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation (PLDI)*, pages 127–136, May 1996.

[3] K. A. Bharat and L. Cardelli. Migratory applications. In *Proceedings of the 8th Annual ACM Symposium on User Interface Software and Technology*, Nov. 1995.

[4] A. Birrell, G. Nelson, S. Owicki, and E. Wobber. Network objects. *Software–Practice and Experience*, 25(S4):87–130, Dec. 1995. Also available as DEC SRC Research Report 115.

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

[6] M. Condict, D. Milojicic, F. Reynolds, and D. Bolinger. Towards a world-wide civilization of objects. In *7th SIGOPS European Workshop*, Sept. 1996.

[7] D. Dean, E. Felten, and D. Wallach. Java security: From HotJava to Netscape and beyond. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, May 1996.

[8] R. S. Gray. Agent Tcl: A flexible and secure mobile agent system. In *Proceedings of the Fourth Annual Tcl/Tk Workshop*, pages 9–23, July 1996.

[9] J. Hylton. Creating collections with a distributed indexing infrastructure. May 1996. Position statement for the W3C Distributed Indexing/Searching Workshop.

[10] B. Janssen, D. Severson, and M. Spreitzer. *ILU 1.8 Reference Manual*. Xerox Corp., 1995. Available via Inter-Language Unification Web page at URL:ftp://ftp.parc.xerox.com/pub/ilu/ilu.html.

[11] D. Johansen, R. van Renesse, and F. B. Schneider. Operating system support for mobile agents. In *Proceedings of the 5th IEEE Workshop on Hot Topics in Operating Systems*, pages 42–45, May 1994.

[12] 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, Dec. 1995.

[13] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(2):109–133, Feb. 1988.

[14] R. E. Kahn and V. G. Cerf. The Digital Library Project, volume I: The world of Knowbots. Unpublished manuscript, Corporation for National Research Initiatives, Reston, Va., Mar. 1988.

[15] D. Kotz, R. Gray, and D. Rus. Transportable agents support worldwide applications. In *7th SIGOPS European Workshop*, Sept. 1996.

[16] B. Liskov, A. Adya, M. Castro, M. Day, S. Ghemawat, R. Gruber, U. Maheshwari, A. C. Myers, and L. Shrira. Safe and efficient shareing of persistent objects in Thor. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 318–329, June 1996.

[17] M. Ranganathan, A. Acharya, S. Sharma, and J. Saltz. Network-aware mobile programs. In *Proceedings of the USENIX 1997 Annual Technical Conference*, 1997. To appear.

[18] M. Shapiro, P. Dickman, and D. Plainfossé. Robust, distributed references and acyclic garbage collection. In *Symposium on Principles of Distributed Computing*, pages 135–146, Aug. 1992. Revised version available as Rapport de Recherche INRIA 1799 and Broadcast Technical Report no. 1.

[19] B. Steensbaard and E. Jul. Object and native code thread mobility among heterogeneous computers. In *Proceedings of the 15th ACM Symposium on Principles of Operating Systems*, pages 68–78, Dec. 1995.

[20] J. Tardo and L. Valenta. Mobile agent security and Telescript. In *Proceedings of IEEE COMPCON '96*, Feb. 1996.

[21] G. van Rossum. Python tutorial. Technical Report CS-R9526, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, May 1995.

[22] J. E. White. Telescript technology: The foundation for the electronic marketplace. Available at Web page titled "Telescript Technology: Whitepaper #1," URL:http://www.genmagic.com/Telescript/-Whitepapers/wp1/whitepaper-1.html, 1996.