Integrating Complex Data Access Methods into the Mosaic/WWW Environment

Bhavna Chhabra (*B.S. student, CS Dept., U. Colorado - Boulder*)
Darren Hardy (*Professional Research Assist., CS Dept., U. Colorado - Boulder*)
Allan Hundhausen (*B.S. student, CS Dept., U. Colorado - Boulder*
David Merkel (*B.S. graduate, CS Dept., U. Colorado - Boulder* [1])
Johk Noble (*B.S. student, CS Dept., U. Colorado - Boulder*)
Michael F. Schwartz (*Associate Professor, CS Dept., U. Colorado - Boulder*)

September 14, 1994

Abstract

This paper describes object-oriented extensions to the Harvest information discovery and access system that allow users to define type hierarchies and associated access methods for Web objects. These extensions enable Mosaic and the Web to represent, manipulate and display arbitrarily complex data in application-specific ways. Types and access methods are extensible; new ones can be defined and exported from Web servers and imported into Mosaic on demand, without changes to Mosaic and HTML for each new data type. This paper describes our design and prototype system implementation, including an application we built using the system.

[1] Merkel is currently a Leutenant in the U.S. Air Force, stationed at Maxwell Air Force Base.

1 Introduction

The World Wide Web is currently used primarily for sharing text and graphics; it provides little support for applications with semantically deeper data or more complex processing requirements. For example, Mosaic cannot easily be used to share and manipulate scientific data. At the server side, a user could create a forms-based interface to a scientific database, but this approach limits interactions to a sequence of selections and textual inputs. Alternatively, a user can define a MIME [1] type for the data and install some data type-specific software to handle the data at the client end. However, this approach is not dynamically extensible. Each new type of data requires modifying the browser's configuration files and installing local software, and requires general consensus on the names of data types (e.g., postscript, jpeg, etc).

The obvious solution to this problem is to provide *Object-Oriented (OO)* support in the Web, so that data are explicitly typed and associated with software *methods* that can operate on them. If designed properly, an OO interface to the Web could allow arbitrarily complex interactions and dynamic type extensibility without requiring modifications to the Web client or server software for each new data type. Object orientation can provide other advantages as well, including implementation detail hiding and easier code reuse.

While many OO systems have been built in the past, most were designed to provide sophisticated support for various language features (such as polymorphism [4]) or system features (such as mobility or dynamic linking [5]). Such functionality comes at the price of a complex and restrictive runtime environment. In contrast, one of the reasons the Web has become so successful is that it operates in an open environment available on many different platforms, and requires little cooperation between information providers and users.

In this paper we focus on providing OO support for legacy data. The goal is to allow such "dusty deck" data and their associated processing software to be deployed and used on the Web with a minimum of programming and administrative effort. Ideally, it should be possible to export data simply by registering the data, data type, and access methods in an Internet server, and then advertising a pointer the data on the Web. Users should be able to access the data simply by specifying the URL.

This focus on legacy data influenced our design decisions in a variety of ways. First, it would not have made sense to build a Web interface to an OO database management system for holding the data, since legacy data and code typically reside in a file system or pre-existing (non-OO) database. Second, for the present we ignored many problems of OO design. For example, we do not attempt to enforce encapsulation through barriers that prevent underlying data from being accessed outside of the method interfaces. The current prototype is a first step towards providing object oriented support in the Harvest information discovery and access system [3]. In the future we will will extend this system to provide more sophisticated OO support.

2 Design

The design for the Harvest Object System (HOS) specifies an object model and a typing scheme, each of which we discuss below. The *implementation* includes more support to deal with common special cases (e.g., to support programs implemented using the UNIX standard I/O paradigm), and is discussed in the Section 3.

2.1 Object Model

Many OO systems are complicated by the need to marshal language-level types into and out of network standard representations. Doing so typically requires a stub compiler, an intermediate representation language, data type conversion/interpretation code, and optimizations to achieve acceptable performance.

HOS currently supports a simpler model: Individual methods are programs (as opposed to procedures), with inputs and outputs limited to those easily expressible at this level (e.g., string-valued command line parameters). In time we may expand the system to provide more sophisticated support (e.g., interoperating with CORBA [6]).

The one additional complexity defined in our object model is the ability to invoke methods either remotely (as network servers) or locally (by first fetching the code from the remote site). This distinction shows up in our design because there are strong reasons for choosing one style or the other for different applications. Local methods are primarily useful for performance or scaling reasons – by loading the code locally, it can make fine-grained access to a display, and can reduce load on remote servers. Local methods can also be useful when it is necessary to interact with a local device for which no cross-network protocol exists – e.g., unlike the case with UNIX displays (which can use the X-windows protocol), UNIX audio devices require a special-purpose piece of software to be run locally, to copy bytes from the network to the audio device. In contrast, remote methods are useful when the code needs to access resources available at a particular location (e.g., a physical device or a database that is too large to ship across the network), or when the code is proprietary and hence must remain at the remote site.

Executing remotely-defined method code on a local machine raises security and architecture-dependence issues, which we address simplistically at present: method code is annotated with architecture and operating system information, and we allow users to designate a local machine on which to run untrusted remote code – limiting exposure to security problems. We chose these simple approaches because several R&D groups (e.g., Sun Microsystems, General Magic, and CommerceNet) are working on promising solutions to both the architecture-independence and security problems. We plan to incorporate their support as it becomes available.

2.2 Object Typing

In an OO system, data must be typed in order to know what methods can legally be invoked. For example, it makes sense to query the volume of a cube, but a line would not support this operation. Supporting this involves providing a type name space, as well as an Interface Description Language (IDL) to specify the inputs and outputs for each method. Because our methods are programs rather than individual procedures, the IDL is relatively simple - it basically registers command line parameters, plus information needed for the local vs. remote invocation style described above.

We use a simple naming structure for Harvest object typing. The user first picks a locally meaningful type name (such as "SeismicWaveTimeSeries"), and then registers this name, a URL pointing to the data, and a set of URLs pointing to associated method code in a local type/method registry. This registry is accessed at object invocation time by the Harvest Object Protocol (described in Section 3). We opted away from requiring a global typing scheme, both because global naming is a large separate problem, and because we believe in many cases it will be expedient to allow users to define locally meaningful types without first having to be allocated a piece of a global name space from a global naming authority. If/when a Uniform Resource Naming (URN) [10] system becomes

available, we can switch the typing scheme to make use of URNs. In the interim, if users want to coordinate parts of the type space beyond the confines of a local registry, they will be able to do so using the Harvest brokering [3] architecture. For this purpose, users will build type registry Brokers, and then cascade some of the type definitions to other downstream Brokers as desired. At present we do not have support for this implemented in HOS, so all methods are locally defined for now.

While it would have been possible to use MIME for the HOS typing system, we chose not to do so for two reasons. First, MIME is primarily intended as a compound document encoding standard, where the usual arrangement is for the processing software to be widely known and available locally (e.g., the audio player for parts of a MIME mail message are built into MIME-enabled mail user agents). In contrast, this sort of semantic knowledge is expected to be associated with particular data objects in an OO system – hence, while MIME might be made to work for our purposes, doing so would really stretch its conceptual orientation. Second, we wanted to create an entirely separate name space for HOS objects, so that there was no potential for conflict with the growing body of MIME types. Stated another way, HOS types are assumed to be locally meaningful unless arranged into cascading brokers, while MIME types are assumed to be globally meaningful unless a particular branch of the MIME name space is used.

3 Implementation

HOS is implemented in ANSI C, except for the user interface of the Object Browser, which was written in Tcl/Tk. At present HOS runs only on UNIX.

The HOS implementation consists of four components:

1. A Harvest Object Browser (HOB), which is client code spawned from Mosaic, allowing users to interact with remote objects;
2. The Harvest Object Protocol (HOP), used for retrieving IDL information, moving object code and data, and invoking objects;
3. A Runtime System, which resides in HOB and implements the client side of HOP; and
4. A distributed type and method registration system, which holds the IDL specifications.

HOB is about 350 lines of code, while HOP is about 1,300 lines of code. The Runtime System is about 800 lines of code. The type registration system is about 300 lines of code.

We discuss each of these components below.

3.1 Harvest Object Browser and Runtime System

We modified Mosaic to invoke the Harvest Object Browser (HOB) as a child process when it encounters a URL containing the "hop://" access method identifier. When the HOB first starts running, it issues a call to the Runtime System to contact a type and method lookup server running at a well-known port on the machine that exports the objects. It then creates a pop-up window listing the methods available for the invoked object, and makes further Runtime System calls to invoke remote objects for each method selected by the user. The result objects are retained as

intermediate outputs, each with an attached HOB window that allows the user to invoke methods on the result objects. Users can save the intermediate objects if they wish, and invoke them later. Otherwise, when a window is destroyed, the intermediate object is deleted.

When the Runtime System encounters an IDL specification for a method to be invoked locally, it retrieves the code and data using a library of URL access software (allowing the code to be passed via FTP, for example), saves them into local temporary files, and then executes the code using the input/output arrangement specified in the IDL. If the IDL specifies a remote method, the Runtime System establishes a TCP connection to the remote server, sends the object data, creates an intermediate object, reads the results into the intermediate object.

The Runtime System wraps each intermediate object in an IDL structure, so that it is self-describing when needed in future computations. This approach allows intermediate objects to reside on a workstation that is not running a type and method lookup server. Because we wanted to avoid forcing method code to be aware of the IDL wrapping structure, the Runtime System automatically wraps intermediate results in an IDL structure, and strips the wrappers before passing the the code to each method.

3.2 Harvest Object Protocol

We developed our own protocol for HOS (rather than using HTTP) for three reasons. First, we wanted to allow HOS objects to be recognized easily from within Mosaic (based on the "hop://" prefix). Without an explicit syntax, we would need Mosaic to try to contact a remote HOP registry each time the user clicked on a Web pointer, impacting performance and increasing network load. Second, we wanted to retain the flexibility to modify the protocol as needed. Third, we wanted to avoid the performance overhead of the current HTTP implementation, which requires separate connections for each data transfer. We felt this would be important, since performance problems have been one of the important "show-stoppers" for OO systems in the past. HOP provides a way to keep the connection open to the HOP server and to stream object and meta-data together. If HTTP resolves this problem in the future (e.g., see [8]), we may switch to using HTTP.

The HOP client/server mechanism is fashioned after the FTP scheme of using separate data and control connections. When a client connects to a HOP server, the server first forks a process to allow multiple concurrent connections to this server host. The child process then takes over the connection, and creates a second connection back to the client. The client uses the initial connection to send commands to the server, and the server sends replies and error messages to the client over this connection. The second connection is used only for data. When the client requests some form of data (e.g., an entire object or a list of methods), the HOP server responds over the data connection.

Commands are sent to the HOP server as a string of the form, "COMMAND DATA", where the data consist of a list of blank-delimited parameter strings. There is a reply for each command, either telling the client some data is coming, or that an error occurred. Data are sent to the client in a series of typed, byte-count delimited blocks of up to 64 KB each, with a final zero-sized block to indicate the end of data.

An example command might look something like "GETOBJ hop://rd.cs.colorado.edu/ñoblej-/some.obj SOME-ARCH". This command asks the HOP server to send the object data in the file ñoblej/some.obj along with its meta data, including a list of methods for the specified machine architecture. If no architecture is specified, a list of all methods available on all architectures is

sent. The HOP server returns a result to the client after each command, saying that the command can be completed or that an error occurred. The connection is kept open until the client closes it or the server times out after a given period.

3.3 Type/Method Registry

The HOS design does not specify how the type and method lookup server is to be implemented – it could use typing information available from the underlying operating system if available (e.g., in the Macintosh OS), a registry of typing information built explicitly to support HOS, or even a heuristic typing scheme, such as that used by Essence [7]. [1] All that is required is that the lookup server use the Harvest Object Protocol (HOP), and be able to return the object's type, data, and any methods that are globally associated with the appropriate type.

While the type/method registry implementation is not specified as part of HOP, at present we provide a GDBM-based implementation of the registry. This database holds type information for each local object (keyed on the object's URL), and a method list template for each type (keyed on type name), which includes an IDL description for the method. In the future, we plan to convert our system to use Harvest Brokers, so that users can coordinate data typing among sites by arranging Brokers that gather from other Brokers (see Section 2). We used GDBM for the initial implementation because the Harvest Broker code was not yet complete when we began implementing HOS.

HOS represents an object's typing and interface description data using Harvest's Summary Object Interchange Format (SOIF) [2]. SOIF was originally designed as a protocol between Harvest Gatherers and Brokers. It is an attribute-value stream protocol, which delineates streams of object summaries, and allows for multiple levels of nested detail. We reused SOIF as a concrete representation for the IDL, because its structure fits the needs of an IDL well, and the Harvest system provides SOIF parsing, storage management, and transmission software.

Figure 3.3 gives an example of an IDL specification. The values in the curly braces are byte count field delimiters, required by SOIF.

The "TEXT@METHOD" template defines the TCP methods that map domain names to latitude/longitude coordinates, and do the scatterplot of the lat/long. The "PBM@METHOD" defines the TCP merge method, and defines the method to do the display using the "xv" viewer program. The IDL tells where the method is (URL or host:port), what type of method it is (TCP, EXT for external, etc.), what type of object the method takes (PBM, TEXT, etc.), and what type of object the method returns (PBM, TEXT, NONE, etc.).

In the future we may experiment with heuristic typing. For this purpose we have modified Mosaic to invoke the HOB if the user clicks on a hypertext link while holding the shift button. In this case, a user could retrieve remote data from some non-HOP access method (e.g., FTP), and then attempt to perform application-specific processing by first using Essence to identify its type.

3.4 Runtime Support for Common UNIX Programming Styles

While the HOS design is system independent, at present Internet server technology is dominated by UNIX systems. For this reason we provide additional implementation support to ease the task

[1] Essence attempts to recognize object types thorough a combination of naming syntax and file contents - e.g., looking at file name extensions and "magic" numbers in UNIX files.

```
TEXT@METHOD { method
Version{3}:     0.1
Type{4}:        text
Arch1{8}:       DEC-MIPS
DomainToLatLong{42}:    powell.cs.colorado.edu:9501;TCP;TEXT;TEXT;
PlotLatLong{41}:        powell.cs.colorado.edu:9502;TCP;TEXT;PBM;
Display{66}:    http://burton.cs.colorado.edu/~hundhaus/cat;TEXTDISPLAY;TEXT;NONE;
}

PBM@METHOD { method
Version{3}:     0.1
Type{3}:        PBM
Arch1{8}:       DEC-MIPS
xv{56}: http://burton.cs.colorado.edu/~hundhaus/xv;EXT;PBM;NONE;
MergeWithUSA{39}:       clark.cs.colorado.edu:9503;TCP;PBM;PBM;
}
```

Figure 1: Interface Description Language Specification

of writing methods or converting existing UNIX programs to work with HOS.

At present we provide no special support for server-based UNIX methods. Basically, the HOS Runtime System expects an interface in which you first connect to the server, send all the needed data, and receive all the needed results, as a single be improved by wrapping *mediators* [11] around client and server, an issue we may investigate in the future.

For locally executed methods the Runtime System supports two special cases:

1. Stdin $\mapsto$ stdout. This method supports the familiar UNIX pipe facility.
2. File $\mapsto$ stdout. This method supports programs that operate on a file specified on the command line, and output to stdout.

Since methods are general UNIX programs, they are free to communicate with users and the execution environment however they choose (e.g., through mouse-based interactions with the user). However, at present we provide no special support for more complex interaction styles than those discussed above, so it would probably be necessary to modify such application code to make it function as a method.

4 Application

To experiment with HOS, we implemented software to operate on Domain Naming System (DNS) host names in two stages of methods. The first stage translates host names into longitude/latitude

coordinates, using a database we assembled from the Netfind site database [9] plus Merit's Geographic Name Server [2]. It then uses the GNUplot software package to create a scatter plot of the data. The second stage merges the scatter plot with a bitmap of the United States. [3]

Figure 2 shows some sample output from the final stage of the DNS mapper application, applied to a list of 5 supercomputer centers, as listed in Table 4.

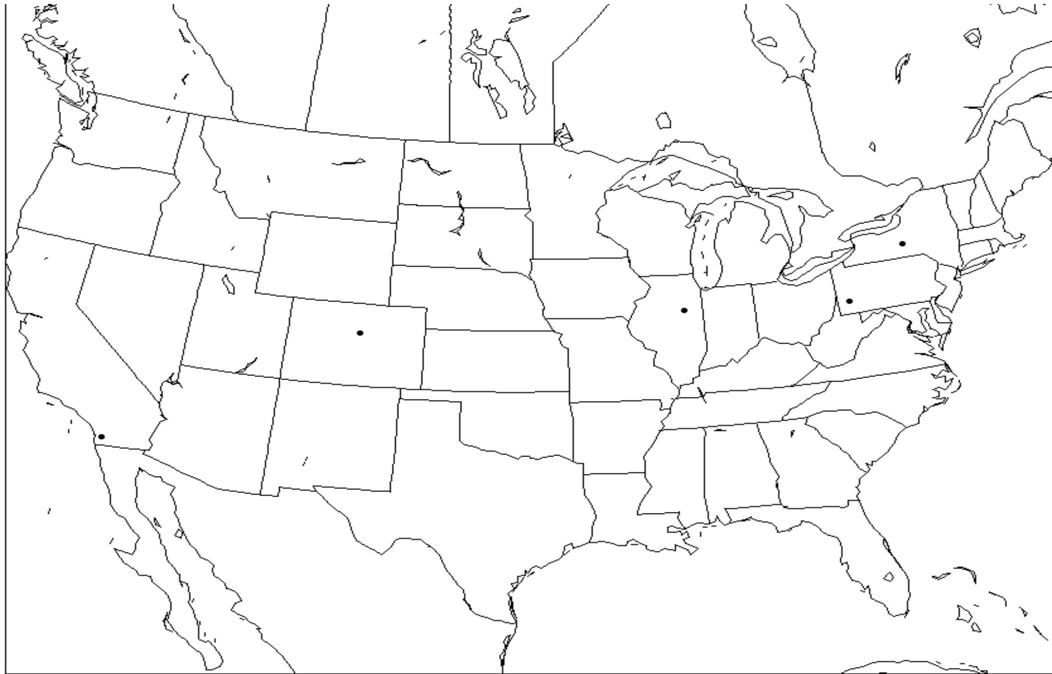

Figure 2: DNS Mapper Application Sample Output

It would have been possible to implement this DNS mapping application without OO support, using a CGI script at the server that plots a list of domain names and returns a bitmap containing the map. We chose to implement it as above to push the design issues involved with supporting

[2]The Geographic Name Server is reachable at telnet://martini.eecs.umich.edu:3000.

[3]To simplify the task of building the coordinate database the application currently supports only U.S. sites.

Domain Name	Location
ncar.ucar.edu	Boulder, Colorado
ncsa.uiuc.edu	Urbana-Champaign, Illinois
psc.edu	Pittsburgh, Pennsylvania
sdsc.edu	San Diego, California
tc.cornell.edu	Ithaca, New York

Table 1: Plotted Supercomputer Center Sites

intermediate objects, and to get ideas about user interface support for chaining together multi-stage computations. In addition to its value as a design exercise, implementing the DNS mapper as a stage of mappings allowed us to illustrate the possibilities of exposing internal data transformation states, since each is potentially useful (e.g., there may be another application that needs to translate DNS names to coordinates, but does not need the coordinates plotted; that would be possible by extending the implementation to use three stages instead of the two above). Finally, it was easier to implement the application in a series of stages, as it allowed us to reuse various pieces of existing code – a traditional OO design goal.

At present we are investigating applying HOS to a number of problems with legacy data and software, for which HOS' OO support will be critical.

5 Summary

While the WWW provides a popular environment for sharing multimedia information, it presently provides limited support for more complex data processing. To address this problem, we designed and implemented an Object-Oriented system that allows users to type data and associate method code with the data, which is automatically invoked when a user selects a hypertext link to the data. A key design goal was that it should be possible for people to deploy legacy data and associated processing software on the Web with a minimum of programming and administrative effort, while retaining the essential power of the object model. Towards this end, we defined a simple object model and typing scheme. Our implementation extends what is specified in the design so that the system can support useful functionality for the common case of UNIX applications.

The implementation consists of an object browser, an invocation protocol, a runtime system, and a type/method registration system. We implemented a two-stage application on top of this system to press on the design issues.

The source code for this system will be available from ftp://ftp.cs.colorado.edu/pub/cs/distribs-/harvest/hos.tar.Z around December 1994. We are quite interested in getting users to export their data using this system.

We have a number of plans for future work on this system, some of which were discussed in the body of this paper (e.g., porting to encapsulated execution environments as they become available). One additional area that we consider particularly interesting is providing better support so that users can chain methods code together into multistep, distributed pipelines. We have two sets of plans here. First, we would like to build a graphical interface that allows users to specify pipelines easily. Second, we want to provide support so that results can be piped directly among the processing stages, rather than requiring each intermediate result to be shipped to the invoking site and then sent to the next processing stage.

Acknowledgements

This work was supported in part by the National Science Foundation under grant numbers NCR-9105372 and NCR-9204853, the Advanced Research Projects Agency under contract number DABT63-93-C-0052, and an equipment grant from Sun Microsystems' Collaborative Research Program.

The information contained in this paper does not necessarily reflect the position or the policy of the U.S. Government or other sponsors of this research. No official endorsement should be inferred.

Peter Danzig provided useful comments on an earlier draft of this paper.

References

[1] Nathaniel Borenstein and Ned Freed. MIME (Multipurpose Internet Mail Extensions) part one: Mechanisms for specifying and describing the format of internet message bodies. Technical report, September 1993.

[2] C. Mic Bowman, Peter Danzig, Darren Hardy, Udi Manber, and Michael F. Schwartz. Harvest protocol and subsystem specifications. Technical report, 1994. In preparation.

[3] C. Mic Bowman, Peter B. Danzig, Darren R. Hardy, Udi Manber, and Michael F. Schwartz. The Harvest information discovery and access system. *Second International World Wide Web Conference*, October 1994. Available from ftp://ftp.cs.colorado.edu/pub/cs/techreports-/schwartz/Harvest.Conf.ps.Z.

[4] A. Demers, L. Cardelli, and P. Wegner. Data types are values on understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985.

[5] Robert A. Gingell, M. Lee, X. T. Dang, and M. S. Weeks. Shared libraries in SunOS. *Proceedings of the USENIX Summer Conference*, pages 131–146, June 1987.

[6] Object Management Group. Common Object Request Broker: Architecture and specification. Technical report, Framingham, Massachusetts, 1991.

[7] Darren R. Hardy and Michael F. Schwartz. Customized information extraction as a basis for resource discovery. Technical report, March 1994. Submitted for publication. Available from ftp://ftp.cs.colorado.edu/pub/cs/techreports/schwartz/Essence.Jour.ps.Z.

[8] Jeffrey C. Mogul and Venkata N. Padmanabhan. Improving WWW latency. *Second International World Wide Web Conference*, October 1994.

[9] Michael F. Schwartz and Calton Pu. Applying an information gathering architecture to Netfind: A white pages tool for a changing and growing Internet. *To appear, IEEE/ACM Transactions on Networking*, 1994.

[10] K. Sollins and L. Masinter. Requirements of Uniform Resource Names. Technical report, March 1994. Available as ftp://cc.mcgill.ca/pub/Network/uri/urn-functional-requirements-sollins.masinter-00.txt.

[11] Gio Wiederhold. Mediators in the architecture of future information systems. *IEEE Computer Magazine*, 25(3):38–49, March 1992.

Author Biographies

Bhavna Chhabra is an undergraduate senior in the Computer Science Department at the University of Colorado, Boulder. She expects to graduate in May 1995. She can be reached at chhabra@cs.colorado.edu.

Darren Hardy is a Professional Research Assistant in the Computer Science Department at the University of Colorado. He received his M.S. in Computer Science from the University of Colorado, Boulder in 1993. He specializes in network resource discovery, distributed systems, and information retrieval. Hardy can be reached at hardy@cs.colorado.edu.

Allan Hundhausen is an undergraduate senior in the Computer Science Department at the University of Colorado, Boulder, and a Research Assistant on the Harvest project. He expects to graduate in May 1995. He can be reached at hundhaus@cs.colorado.edu.

David Merkel received his B.S. in Computer Science from the University of Colorado, Boulder in 1994. He is currently a Leutenant in the U.S. Air Force, stationed at Maxwell Air Force Base, Alabama. Merkel worked on the Harvest Object System while he was a student at the University of Colorado, prior to beginning active duty in the Air Force.

John Noble is an undergraduate senior in the Computer Science Department at the University of Colorado, Boulder, and a Research Assistant on the Harvest project. He expects to graduate in May 1995. He can be reached at noblej@cs.colorado.edu.

Michael Schwartz is an Associate Professor of Computer Science at the University of Colorado. He received his Ph.D in Computer Science from the University of Washington in 1987. His research focuses on international-scale networks and distributed systems. Schwartz chairs the Internet Research Task Force Research Group on Resource Discovery (IRTF-RD), which built the Harvest system. Schwartz can be reached at schwartz@cs.colorado.edu.

Contact author: Mike Schwartz (schwartz@cs.colorado.edu).