Towards Interoperability in Digital Libraries
Overview and Selected Highlights of the Stanford Digital Library Project

Andreas Paepcke, Steve B. Cousins, Hector Garcia-Molina, Scott W. Hassan, Steven P. Ketchpel, Martin Röscheisen, Terry Winograd

Stanford University

Abstract

We outline the five main research thrusts of the Stanford Digital Library project, and we describe technical details for two specific efforts that have been realized in prototype implementations. First, we describe how we employ distributed object technology to cope with interoperability among emerging digital library services. In particular, we describe how we use CORBA objects as wrappers to handle differences in service interaction models, and we sketch an information access protocol that takes advantage of distributed object environments. The second effort we cover is *InterPay,* a framework and protocol for payment among autonomous services. The framework addresses interoperability problems in online payment by cleanly separating information access protocols , payment and charging policies, and the actual mechanics of individual financial transactions.

Descriptors: Digital libraries, distributed objects, CORBA, ILU, client-server architecture, interoperability, protocol transformation, Z39.50, electronic payment, electronic commerce, *InterPay*, *InfoBus*, annotations, SOAPs, third-party ratings.

Introduction

Information repositories are just one of many services emerging for digital libraries of the future. Other services include automated news summarization, topic trend analysis across newspaper repositories, or copyright-related facilities. Traditional library services such as archiving and collection building continue to be relevant in the digital medium as well. Archiving, for example, remains an issue in the digital world because of problems with 'dangling' hyperlinks and because of storage media obsolescence.

This emerging distributed collection of services carries an enormous potential for helping users in their information intensive tasks. It could also turn into a confusing, frustrating annoyance because programmers and end users need to learn too many different interfaces, cannot find the resources they need or are confronted with the bewildering details of for-pay services that were previously only accessible to professional librarians.

We begin by sketching the overall approach taken by the Stanford Digital Library project to help with these issues. Then we focus on the problem of interoperability, which is particularly important because digital library services are developing rapidly with standardization lagging far behind. In particular, we describe how we use CORBA-based distributed objects [1] to implement information access and payment protocols. The protocols provide the interface uniformity necessary for interoperability, while leaving implementors a large amount of leeway to optimize performance and to provide choices in service performance profiles.

The Stanford Digital Library Project: Five Areas of Contribution

Figure 1 shows how the development of our recently started digital library testbed is driven along five thrusts. The prototype services and protocols developed by the thrusts will be demonstrated on a testbed comprising a variety of computing literature sources, including ones at Knight-Ridder's Dialog, MIT Press, ACM, the Web, and the Stanford University Libraries.

Research on information interfaces is geared towards easing interaction with information of diverse formats and with digital library services present-

Figure 1: Stanford's Approach to Digital Library Development

ing varying interaction models. Work in this thrust also explores uses of digital libraries as places for users to communicate about documents.

For example, we have built the prototype of a wide-area annotation service [2, 3]. It allows users to annotate pages on the World-Wide Web without modifying the original documents. Annotations are organized into sets, each with its own permission facility. Annotation sets may be located on servers other than the ones housing the documents the annotations are associated with. Users may choose to view documents with no annotations, or with annotations from any of the sets they have permission to access. The many uses of this facility include independent product reviews and document content ratings: users can view the ratings produced by the organization they happen to trust and rely on for guidance.

The second thrust of the project is concerned with technologies for locating appropriate library services and information relevant to user tasks. For example, we have prototyped GlOSS, a service which efficiently maintains enough meta-information about a set of repositories that it can point users to the sources most promising for a particular query [4]. The SIFT service is a prototype that explores efficient algorithms for matching large numbers of user interest profiles with large numbers of documents [5]. Other efforts address the problem of query integration across multiple services.

Technologies supporting the evolving economic aspects of digital libraries are at the core of the third project thrust. Our SCAM and COPS efforts develop algorithms and a prototype for the efficient comparison of a text document against a large number of reference documents to detect partial overlap [6]. This service can be used to protect authors against illegal use of their intellectual property. Another effort in this third thrust is the development of an architecture to manage interaction with the many emerging payment schemes. We will describe this effort in more detail below.

The fourth thrust is developing models and a supporting infrastructure for the interaction with documents and services. These models form the basis for the protocols and architecture of our testbed. They include the models for meta-information about documents and repositories, to be used to search and visualize results. They also include protocols for the effective use of client-server models when potentially large amounts of information need to be moved among sites. The access protocol described below is part of this effort.

The fifth thrust, finally, examines how agent technology can be employed to help operations throughout the system. We use very simple agent technology to help monitor online payment transactions. More substantial agent technologies are being used to retrieve information from the World-Wide Web based on user interest profiles that are successively refined [7].

We now focus on the problem of interoperability, presenting our architectural design and use of technologies.

Digital Libraries Thrive on Distributed Object Technology

How do different clients and service providers interact in a Digital Library? In an *ideal world*, clients and services would be created independently, on the basis of implementation choices the respective consumers and providers deemed appropriate. Then each would plug their different components into a “virtual (software) bus” that would take care of all the protocol-level interoperability issues. Within this “information bus” (*InfoBus*), library services would transparently accomplish tasks such as format translation, the brokering of required services or financial transaction support. If all the services conformed to one standard, this vision could easily be realized. Unfortunately, such convergence has not occurred even in the long-standing area of information retrieval. An overly simple solution would call for cross-translations among all *n* standards. This would be a formidable $O(n^2)$ effort. In this section we discuss how distributed object technology may help achieve the long-term goal of an "InfoBus" without the need for all participants to agree on a single standard mode of interaction, and with an $O(n)$ effort in translation machinery.

Interoperating Across Protocol Domains

To understand how this vision might be achieved, we start with a very simple example. Figure 2 shows three protocol domains. The top is an exam-

Figure 2: Interoperating Across Protocol Domains

ple for a local network used by a provider of new information services. This might be a company, university, or an individual wishing to use a Digital Library, and, maybe, to add their own services. The service interaction protocols used in this domain are under local control.

The Telnet protocol in the second domain allows clients to log into remote machines. The third is the Web's HTTP protocol; servers running Z39.50 [8] would be another example. All three domains in the figure are populated with services accessible through the respective protocols. Knight-Ridder's Dialog Information service is an example for a telnet-based information provider. The WebCrawler is an example for an indexing service available via HTTP. It indexes documents on the World-Wide Web and returns their URLs in response to queries.

For the purpose of illustration, we will use the Dialog and WebCrawler information repositories as our example below because information repositories are the best-known kind of digital library service. We anticipate that many services will eventually conform to some of the emerging standards, such as HTTP, Z39.50 or SQL, or to new ones yet to be developed. We use Dialog's current minimally standardized human-oriented teletype interface to illustrate the breadth of diversity that remains today.

The multisearch program in the local domain of Figure 2 illustrates why interoperability is a base requirement for the development of Digital Library services. It accepts a query and multiplexes it to several information sources. The copy detection service shown in the figure accepts documents and checks them for substantial overlap with a database of other documents. The multisearch program uses it to eliminate near-duplicates. Without an interoperability infrastructure, the multisearch program would be very cumbersome to write. The programmer would have to learn the interaction models and search languages of both Dialog and the WebCrawler. To avoid this, two translators are needed to link the local domain to the two remote ones.

Exploring translators

For illustration, Figure 3 shows a very simplified view of interactions with both Dialog and the WebCrawler. The Dialog service presents a teletype interface with which a human user is intended to interact through a Telnet session. The user is led through a standard login sequence (**Please logon:**). This is usually followed by the user selecting one of the many databases offered through Dialog (**begin 245**). After some queries through a proprietary query language (**select Library/ti**) and the examination of the results, the session is terminated (**logout**). Figure 3 shows one possible abstraction of this process. The abstraction combines several operations. An **open session** operation is followed by **open database**, **search**, and **quit**. For a full-

Figure 3: Unification of Simplified Service Interaction Models

scale system, this abstraction would, of course, be more elaborate. Parts of the Z39.50 protocol or variants of other related resources could, for instance, be used [9].

Now we look at the lower part of Figure 3. It shows an interaction with the WebCrawler. Its model is at the surface very different from that of Dialog. The user finds the service's home page, clicks at the appropriate place to open a search form, fills it out, views the results and eventually leaves the service's home page. As shown in Figure 3, the same abstraction can be used for this interaction model as for Dialog. If a programmatic interface could be created which presented this common abstraction, the job of writing the multisearch program would be significantly easier. It turns out that object technology is ideally suited for this purpose.

Objects for interface unification

Polymorphism in object-oriented programming systems can be used to present a unified interface like the abstraction of Figure 3 for different services. A *library service proxy* object (LSP) is created for each kind of service. The client invokes each interface element, **open-session**, **open-database**, etc., by means of a method call on one of the LSPs. The implementation of each method performs the appropriate operation on the corresponding service. For example, the **open-session** method for a Dialog LSP starts a Telnet session and logs into the Dialog service. The implementation of the same method for the WebCrawler LSP instead contacts an HTTP demon with the proper URL.

Figure 4a shows how library service proxies can be used as the building blocks for the translators in Figure 2. The translator 'clouds' are filled with

Figure 4: (a)Service Proxy Objects Implement Translation
(b)Programmers Experience the Illusion of an *InfoBus*

library service proxies, each of which represents one service. A common interface thus makes two very different services accessible from the local domain. The effect of this arrangement for a programmer of digital library services is shown in Figure 4b. The proxy objects and their polymorphic implementations act as a wrapper which provides the programmer of the multisearch program with the beginnings of an *InfoBus* abstraction.

The *URL-Getter* object in Figure 4a offers a pure bridge functionality which can suffice when the convenience of a full LSP is not required. It represents the low end of a graceful translation service degradation spectrum enabled by this approach.

Requirements for information flow

Our example has illustrated how object technology can help provide extensible interfaces for information access. However, for each method a proxy or service provides, there are several important “information flow” issues that need to be resolved. To be specific, consider the **search** method discussed earlier. Some services may implement a single interaction model: the client calls the **search** method once (including a query as a parameter) and waits; when the server has assembled the result set, it returns the complete answer. On the other hand, a system that delivers information piecemeal may be preferable: here the user would quickly begin to receive a steady stream of information that slowly builds up, rather than seeing the complete set after a longer wait. This produces the perception of faster response time and allows users to overlap their work with the ongoing retrieval. (An example of this can be observed in some Web browsers when pictures are being loaded. The picture appears first in coarse granularity and is refined slowly as more information arrives.)

Since we cannot dictate how clients and services wish to operate, we would like the **search** method on library service proxies to be as general-purpose as possible. A client that wishes to wait for complete results should be able to do that. If the information service (or its proxy) can give piecemeal information, and the client can handle it, then the **search** method should support that too.

There are other dimensions along which we would like to have flexibility:

- Services may wish to cache result sets of searches for possible future use. The client may in addition or instead wish to cache some of the information. Our search method should cover all these possible modes of interaction.
- It should be possible to instantiate and materialize the objects in a result set (e.g., documents) at various points in time and at various locations. For instance, a pre-fetching strategy may materialize documents at the

client side before their contents are requested. An on-demand scheme would wait until an application program asks for the contents of a given document.

- The method should also allow related processing tasks to be off-loaded to other machines, including the client computer.
- If we are operating across a slow link, we should have the ability to minimize the number of message exchanges.

Existing information access protocols typically do not provide flexibility across all these dimensions. As an example, consider Z39.50, one of the best-known protocols used for information access. The protocol requires that result sets for searches are maintained at the server side and are delivered to clients on request. Thus, Z39.50 makes and fixes particular choices for the first two dimensions listed above. For interoperability, we would prefer a protocol that does not fix these choices, allowing for example a provider to asynchronously and incrementally push information and associated management responsibility to the client. In the next section we will sketch a protocol that uses the distributed object infrastructure to provide the desired flexibility.

Before describing the protocol, we briefly discuss object instantiation and materialization. Instantiation is the act of creating the empty object. Materialization is the act of filling it with information from the provider. When and where these activities occur can impact efficiency. This is an aspect of protocol design that arises specifically in a distributed object environment as we described it because in these systems documents are generally packaged into objects as well. The alternative would be to maintain documents as strings. One advantage of the object approach is that document structure which is often painstakingly provided by repositories, can more easily be preserved and accessed when documents are turned into objects before the client program accesses them. Methods on document objects, such as **title**, **author** or **abstract**, for instance, can be used to extract the corresponding document pieces. The implementations for those methods encapsulate the respective work, such as necessary searches for tagged fields. For example, in the case of SGML documents, client programs do not need to contain code for parsing out pieces of marked up text.

This presents a clean interface to programmers, but it raises the question of when and where these document objects are instantiated and materialized. A simple-minded protocol would have the library service proxy instantiate and materialize document objects for all the documents contained in a result set. Clients would reach these documents through remote method calls. This would be wasteful because users often throw away the results of their queries until they have narrowed their search sufficiently, and because local method calls are cheaper than remote ones. A protocol that takes advantage of the capabilities of an object-based architecture should allow implementations to determine when a document is needed, to shift the corresponding raw information to the site where it will be used most, and then to 'cast' it into an object. The protocols being developed by the Stanford project attempt to provide this flexibility. The following section sketches a simplified version to illustrate some of the issues and our approach to them.

Sketch of an example protocol

We now describe a protocol that provides a uniform search interface, while preserving flexibility for implementors. This protocol has been developed in cooperation with researchers at the Universities of Illinois and Michigan. Several variants have been implemented in our testbed, and it will be initially used to exchange information between those universities and Stanford.

Figure 5 shows how we present the process of querying to programmers of clients. The process includes three steps: the programmer creates a *query object* that contains the query string and any other search specification details[1].The second step is to create a local *result collection object*, specifying the query object and the intended service proxy (indicated by the dashed lines in the Figure). The client program's subsequent interactions are with this result collection, as if the result collection was immediately filled with document objects. For example, the client may invoke the **howMany** method to find out how many documents are in the result, or the

1. The query string could be of a form native to one source, or it could be of a more standard form that is later translated to a native form. We are not concerned with this aspect of interoperability in this example.

Figure 5: Clients Program to a Very Simple Interface

getDoc method to fetch a particular document. When these methods are called, the Result Collection may or may not have the necessary information, so the client calls may be blocked.

Before or after the client tries to get its information, the Result Collection gets the necessary information from the proxy on the server side (LSP). The protocol for this is illustrated in Figure 6, and consists of the following four steps. These steps are described in more detail below. We use the term *client collection* to refer to the Result Collection object on the client side; the server side may choose to create a *server collection* object to assist in the processing.

1. Client collection asynchronously requests query execution.
2. Service asynchronously delivers document references, either as they arrive, or all in one method call.
3. Client collection repeatedly requests more document references (optional).
4. Client collection asynchronously requests document contents, using the references of steps 2/3. (optional). If necessary, object documents are instantiated on the server or client side.

In step 1, the client collection initiates the query in an asynchronous LSP method invocation, passing its own object identifier as the return address

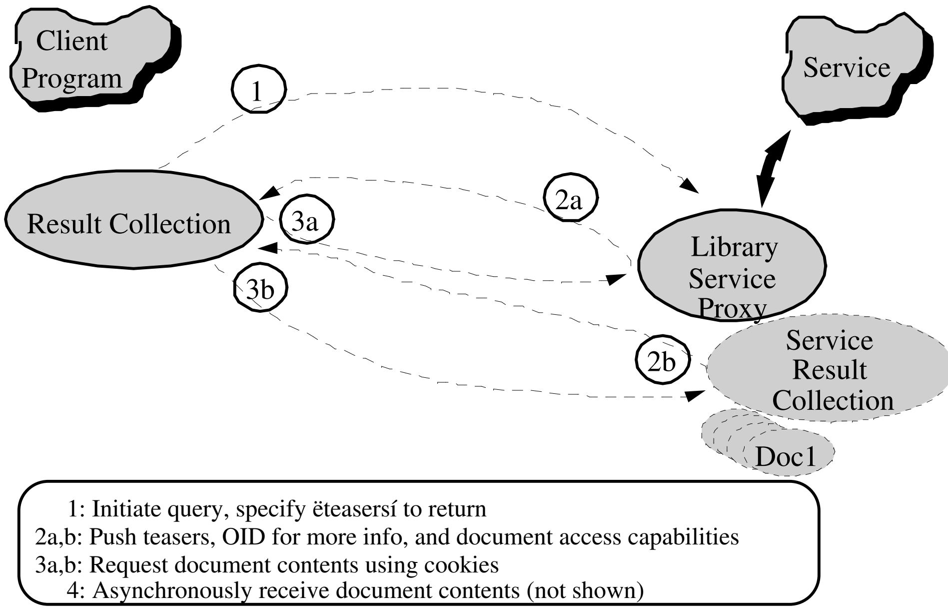

Figure 6: Moving Information

for the query results. It also indicates how many result documents it wants to be able to access initially. As in the Z39.50 protocol, the LSP may be requested to return selected “teaser” fields from some number of the resulting documents. For example, the title and author fields, or the costs of the first 20 documents might be requested initially to help the user decide which documents to request. This allows earliest-possible delivery of some useful information, without having to transfer the entire document bodies. In contrast to Z39.50, the client does not need to wait for the server to complete its result collection because of the call’s asychronicity.

In response, the LSP causes execution of the query in its associated service. When it receives the results, it *may* delegate further handling of related requests to its own (server) collection object which has the same capabilities as the client collection object. In the case of a session-based service, the server collection object can continue to maintain a session with the service in anticipation of requests for documents, or it can pull documents out of the service and cache them itself (Figure 6). Note that since distributed objects may be created anywhere, this server collection object may be located on a different machine, freeing the LSP machine to handle more requests. Alternatively, the LSP can decide not to create the server collection object and continue to manage follow-up requests for documents.

Depending on whether the delivery of results was thus delegated or not, the next step, is executed by the proxy (2a), or by the server result collection object (2b). The purpose of step 2 is to deliver the number of documents found, some or all of the teasers, and document *access capabilities* (document references) that enable the client to obtain full contents of the result documents. Step 2 can be repeated many times as the proxy or server collection accumulates result information. This way the implementation can deliver access to documents before it has found all the documents requested in step 1. We explain some of the implementation details of step 2 below. The key elements are:

- Step 2 method calls are asychronous and return contact information for the client to use for requesting access to more documents than were indicated in the original request. This is how the delegation to server collection objects is accomplished.
- When the LSP or server collection returns teasers for a document, it includes the document's *access capability.* It describes how the full document can be found. Each capability is made up of one or more *access options,* each specifying one alternative way to get the document.

The server side objects send information to the client side via "callback" methods on the client collection. Each of these callbacks includes the object id of the server side object to contact for additional information. Thus, at any time, a server object (e.g., the LSP) can delegate the responsibility of future interactions with this client collection to other objects (e.g., the server collection). Similarly, each document access option received by the client collection contains the id of the server object to contact to obtain that document.

As we have stated, access capabilities may contain several options for actually getting a document. Each option contains the id of the object to contact to get the document, plus a "cookie" that identifies the document. From the point of view of the client, a cookie is simply an uninterpreted bit string that must be given to the server object from which the document is being fetched. From the server object's point of view, the cookie contains information necessary for accessing the document to deliver. For example, a cookie could be an index into a memory cache where the document was placed earlier; it could be a file name for a local file containing the document; it could be a call number in some information retrieval system, or it could be a permanent document handle as described in [10].

The reason for allowing multiple access options within a capability is that the mechanisms for getting a document may vary over time. For example, consider a search over the Dialog Information Service. While the LSP (or the server collection) maintains an open session with the service, it can refer to a particular document by an index into a Dialog-generated result set. Thus, one possible cookie for an access option would be the result set identifier and the index. However, once a session with Dialog is terminated, this access mechanism no longer works. Instead, the document's unique record identifier needs to be used as the cookie. By providing both options in the access capability, the LSP is free to serve document contents quickly while sessions with the service are open, but to close down sessions without losing the ability to deliver documents for which it handed out access capabilities. The holder of an access capability tries the easier options first. As they fail, it tries more expensive ones.

As the client's collection object receives document access capabilities, it has several choices: it can instantiate all the corresponding documents locally, or it can wait until the client program actually requests them. If it instantiates, it can fill in any teaser fields it received, but can wait to materialize the rest on-demand, or it can begin to materialize immediately in anticipation of impending demand. The decision may, for example, be made dependent on statistical user behavior, or it may be based on an evaluation of the likelihood that the remote site will crash or disconnect.

If the client result collection needs teasers and access capabilities for more documents than it initially requested in step 1, it initiates step 3, using the contact information received in step 2. The client collection has no knowledge of whether this request for additional information is handled by the proxy, by the server result collection, or by any other helper object. The result of this request is another round of step 2a/b activity which delivers the teasers and capabilities.

We have implemented an experimental version of the access protocol we have sketched, including proxy objects for Dialog, various Web information sources, Z39.50 servers, Oracle's ConText summarization tool, and others. The standard we follow for our distributed object infrastructure is CORBA [1]. Our implementation is based on Xerox PARC's Inter-Language Unification facility (ILU), a public-domain implementation of CORBA[1]. It is supported on common platforms, such as SUN, IBM RS/6000, HP, SGI, Linux and Windows 3.1/NT. Language bindings include C, C++, CommonLisp, Python, Modula3 and Java. We are using several of these vendor platforms and languages in our experiments. The easy availability of ILU helps our experimentation with the wider community.

Our initial experience indicates that a distributed object framework, and our access protocol in particular, do give clients and servers flexibility to manage their communication and processing resources in an effective way. If we are trying to access existing services, we can write proxies for them without changing the way they operate, e.g., without changing the way they manage state information or cache documents. However, it is important to note that the example protocol sketched here only provides base-level functionality for searching over diverse information services. It thereby addresses only one of the many aspects of interoperability. In the following section we describe how we use our architecture for other digital library interoperability problems.

Fee-For-Service as an Interoperability Problem

As the number of potential customers for online information and services grows, so does the need of providers for effective means of collecting fees. Several online payment mechanisms have been suggested, and some are beginning to be deployed (e.g. [11] or **http://www.fv.com**). For the user of a digital library which includes some for-pay services, the differences in payment scheme are one more potential source of frustration. Our *InterPay* architecture is designed to ease this problem [12]. A prototype has been implemented which allows access to several services, each with a different

1. ILU is available free from Xerox PARC via **http://www.xerox.com/PARC/ilu/index.html**

payment scheme.

The Layered *InterPay* Architecture

Figure 7 shows our *InterPay* architecture. It is structured into three layers: the *services* layer, the *payment policy* layer and the *payment mechanisms* layer. The user's task-related interactions with services occur at the services layer. For information services, these include activities such as logging in, submitting a query, transmitting results, etc. The activities of the protocol described in the previous section occur at this layer.

The payment policy layer controls and enforces payment-related preferences and rules. The policies are implemented by *payment agents* on the payer side, and *collection agents* on the payee side. For example, a payment agent may enforce a policy such as "*pay charges of $1 or less without conferring with the human operator, but notify the operator when total charges exceed $30"*. On the service side, a collection agent may include rules about delayed payment for trusted clients, or limitations on the use of particular payment mechanisms for small transactions (as is customary in many stores regarding the use of credit cards).

The payment mechanisms layer comprises elements that implement the mechanics of particular payment schemes. On the payer side, these are *payment capabilities* (PCs); on the payee side, they are *collection capabilities* (CCs). Each payment capability is programmed to interact with one particular payment agency or payment scheme. Each collection capability is programmed to verify receipts or otherwise interact with one agency or scheme. New payment capabilities can easily be added to the system because all elements of *InterPay* are objects. A new payment scheme is added by implementing a payment and collection capability pair which may even be installed and removed dynamically.

Figure 7 also shows how *InterPay* components interact in a typical transaction. The transaction is executed through six major phases:

1. Set up the session and make a request
2. Initiate a charge
3. Send an invoice

Figure 7: Interactions Among *InterPay* Components

4. Validate the invoice and agree on a payment mechanism
5. Initiate the fund transfer
6. Verify the payment and complete the transaction

Step one is an interaction between the client entity and a service entity, such as the submission of a query. In the example of Figure 7 the client entity is a query interface program and the service entity is a library service proxy as described earlier. The client's payment agent is passed as one piece of information during this step. Depending on the particular service, charges might be initiated immediately, after a search is done or at the end of a session. Once the service decides to charge, it delegates this task to its collection agent (step two).

In step three, the collection agent contacts the payment agent specified during step one, sending an invoice which identifies the service and the charge and which lists the payment mechanisms acceptable to the service. In step four, the payment agent verifies the legitimacy of the charge and picks one of the payment mechanisms offered by the collection agent. During step five the payment agent delegates the mechanics of payment to the proper payment capability. The capability interacts with the respective financial service and the server-side collection capability to accomplish transfer of funds and the receipt. In case of an account-based service, the currency tendered could simply be the user's account number. Finally, the collection capability verifies payment and notifies the collection agent which in turn notifies the service proxy. The proxy may then release information to the client.

Third-Party Payment

One activity an architecture like *InterPay* needs to accommodate is payment through third parties. For example, research libraries generally have bulk discount accounts at commercial information providers. When patrons of the library's local community access these providers, they do it under the library's bulk contract, with expenses sometimes billed to the patron's department. Figure 8 sketches an example of how third-party payment is accomplished in *InterPay*.

Figure 8: Example for Third-Party Payment

Conclusion

We explained how distributed object technology helps us deal with some of the interoperability problems that arise in a digital library comprised of numerous independent services, each potentially presenting a different interface and interaction model. And we demonstrated how this technology can be used to help with the specific heterogeneity problem of multiple online payment schemes.

All five thrusts of the Stanford Digital Library project's work leave room for a wide variety of future work, some of which is currently in preliminary stages. At the user interface level we are working on the problem of interactively configuring the use of library services to accomplish a task, and of re-using and sharing the results of such efforts. In the information finding thrust, current work focuses on the problem of users needing to query multiple services for the same information, without having to contend with disparate query languages and result schemata. In the area of support for economic activity, problems of security and privacy are being considered. In the infrastructure thrust we continue to develop protocols that allow highly flexible distribution of information among machines, while providing satisfactory response time. Agent work is being pursued in the area of profile-based information filters.

References

[1] *The Common Object Request Broker: Architecture and Specification*. Object Management Group, ftp://omg.org/pub/CORBA, December, 1993.

[2] Martin Röscheisen, Christian Mogensen, and Terry Winograd. *Platform for Third-Party Value-Added Information Providers: Architecture, Protocols, and Usage Examples*. Number http://www-diglib.stanford.edu/diglib/pub/reports/commentor.html. Stanford University, November, 1994. Updated April, 1995.

[3] Martin Röscheisen, Christian Mogensen, and Terry Winograd. Interaction Design for Shared World-Wide Web Annotations. In *Proceedings of the Conference on Human Factors in Computing Systems*, 1995.

[4] Luis Gravano, Hector Garcia-Molina, and Anthony Tomasic. The effectiveness of GlOSS for the text-database discovery problem. In *Proceedings of the 1994 ACM SIGMOD Conference*, 1994. http://gloss.stanford.edu.

[5] Tak W. Yan and Hector Garcia-Molina. SIFT — A Tool for Wide-Area Information Dissemination. In *USENIX Technical Conference*, 1995. http://sift.stanford.edu.

[6] N. Shivakumar and H. Garcia-Molina. SCAM: A Copy Detection Mechanism for Digital Documents. In *Proceedings of the Second Annual*

Conference on the Theory and Practice of Digital Libraries, 1995.

[7] Marko Balabanovic, Yoav Shoham, and Yeogirl Yun. An Adaptive Agent for Automated Web Browsing. *Journal of Visual Communication and Image Representation*, 6(4), December, 1995.

[8] *Information Retrieval: Application Service Definition and Protocol Specification*. ANSI/NISO, April, 1995. Preliminary Final Text.

[9] Ramana Rao, Bill Janssen, and Anand Rajaraman. *GAIA Technical Overview*. Xerox PARC, December, 1994.

[10] Robert Kahn and Robert Wilensky. A Framework for Distributed Digital Object Services. Number http://www.cnri.reston.va.us/home/cstr/arch/k-w.html, 1995.

[11] B.C. Neuman and G. Medvinsky. Requirements for network Payment: The NetCheque perspective. In *Proceedings of IEEE COMPCON'95*, March, 1995.

[12] S. Cousins, S. Ketchpel, A. Paepcke, H. Garcia-Molina, S. Hassan, and M. Röscheisen. InterPay: Managing Multiple Payment Mechanisms in Digital Libraries. In *Proceedings of the Second Annual Conference on the Theory and Practice of Digital Libraries*, 1995.