Search Middleware and the Simple Digital Library Interoperability Protocol

Andreas Paepcke, Stanford University (paepcke@cs.stanford.edu)
Robert Brandriff, California Digital Library (bob.brandriff@ucop.edu)
Greg Janee, University of California at Santa Barbara (gjanee@alexandria.ucsb.edu)
Ray Larson, University of California at Berkeley (ray@Sherlock.SIMS.Berkeley.EDU)
Bertram Ludaescher, San Diego Supercomputer Center (ludaesch@SDSC.EDU)
Sergey Melnik, Stanford University (melnik@db.stanford.edu)
Sriram Raghavan, Stanford University (rsram@cs.stanford.edu)

1. Introduction

The development of novel information applications is reaching an impasse. HTML forms for searching the Web are fine for traditional, form-based interfaces to information. But what if we wish to develop more intuitive interfaces that reach across multiple information sources, or are more specialized for particular sources?

For example, we might want to enter queries about human body parts by having the user point to the respective spot on the screen image of a body; or we might wish to combine a molecular layout tool with searches over a database of chemical compounds. These are examples where potentially elaborate applications must be written to mediate between the user and the information source, even if advanced HTML features or JavaScript are used in the user interface itself. A similar need for application support arises when small handheld information devices are used to access backend information sources. Standard Web browsers are frequently inadequate for the small displays of such equipment. Again, specialized applications must manage user input, and must interact with the backend search machinery.

The problem in creating such applications is that no generally agreed upon programmatic interface exists for accessing information sources. Rather than focusing on innovative user level facilities, programmers must expend effort on accommodating unnecessarily different information source access methods, or even resort to screen-scraping of Web pages in order to retrieve information.

There is, then, a need for what we call 'search middleware'. The term refers to protocols and associated software packages that enable information application writers to access information sources easily. Search middleware is reponsible for transporting queries and results, and for negotiating the parameters of search interactions.

Perhaps the most widely known search middleware is the Z39.50 standard [1]. It defines a broad range of facilities, such as a standard machine representation of queries, and an extensible collection of document attributes that may be used in queries, and for the retrieval of document fragments. There has been, however, somewhat of a culture clash between the comprehensive, often complex approach of Z39.50, and the generally light-weight approaches typical in the design of Web related protocols.

We have tried to reach a compromise between a full-scale, all encompassing search middleware design such as Z39.50, and the 'anything goes' approach typical for ad hoc search interface designs on the Web. The result is the Simple Digital Library Interoperability Protocol (SDLIP, pronounced S-D-Lip). The protocol was developed jointly with the Universities of California at Berkeley and Santa Barbara, with the San Diego Supercomputer Center (SDSC), and the California Digital Library (CDL). The design also benefited greatly from input by a related emerging IETF standard on Distributed Authoring, Search, and Locating (DASL) [2]. Together, we analyzed previous search middleware designs, and engaged in long discussions. These discussions very often centered around what **not** to include in SDLIP. Reaching a decision on which features to leave out in order to preserve simplicity was usually the most painful portion of the design process. Decisions were greatly helped by our insistence on early implementation and documentation. As the design evolved, we tracked it with a prototype. This self-imposed process taught us early on, and continuously, whether we were in danger of including too much. The resulting protocol is documented on our Web site.

After completion of the specification, UC Berkeley created SDLIP access to the Berkeley Environmental Digital Library document collection. Berkeley also created a gateway from SDLIP to Z39.50, enabling access to the University of California's MELVYL catalog which covers UC library holdings, and to many holdings of the California Digital Library's extensive collections of digital resources. These include electronic journals, databases, reference texts, and archival finding aids. This bridge between SDLIP and Z39.50 further expands the coverage to other Z39.50 compliant servers including, for example the Library of Congress. SDSC is using SDLIP to provide search interfaces to the Metadata Catalog (MCAT) of their Storage Request Broker (SRB) and to the XML-based information mediator (MIX), thereby facilitating access to further sources including, for example, the AMICO image collection.

Access to Web based resources include a people finder, and a film review site. We also implemented SDLIP access to the Dienst protocol, which enables searches over distributed technical reports (NCSTRL).

2. SDLIP -- A Search Middleware Design

Figure 1 identifies where in a digital library SDLIP is used.

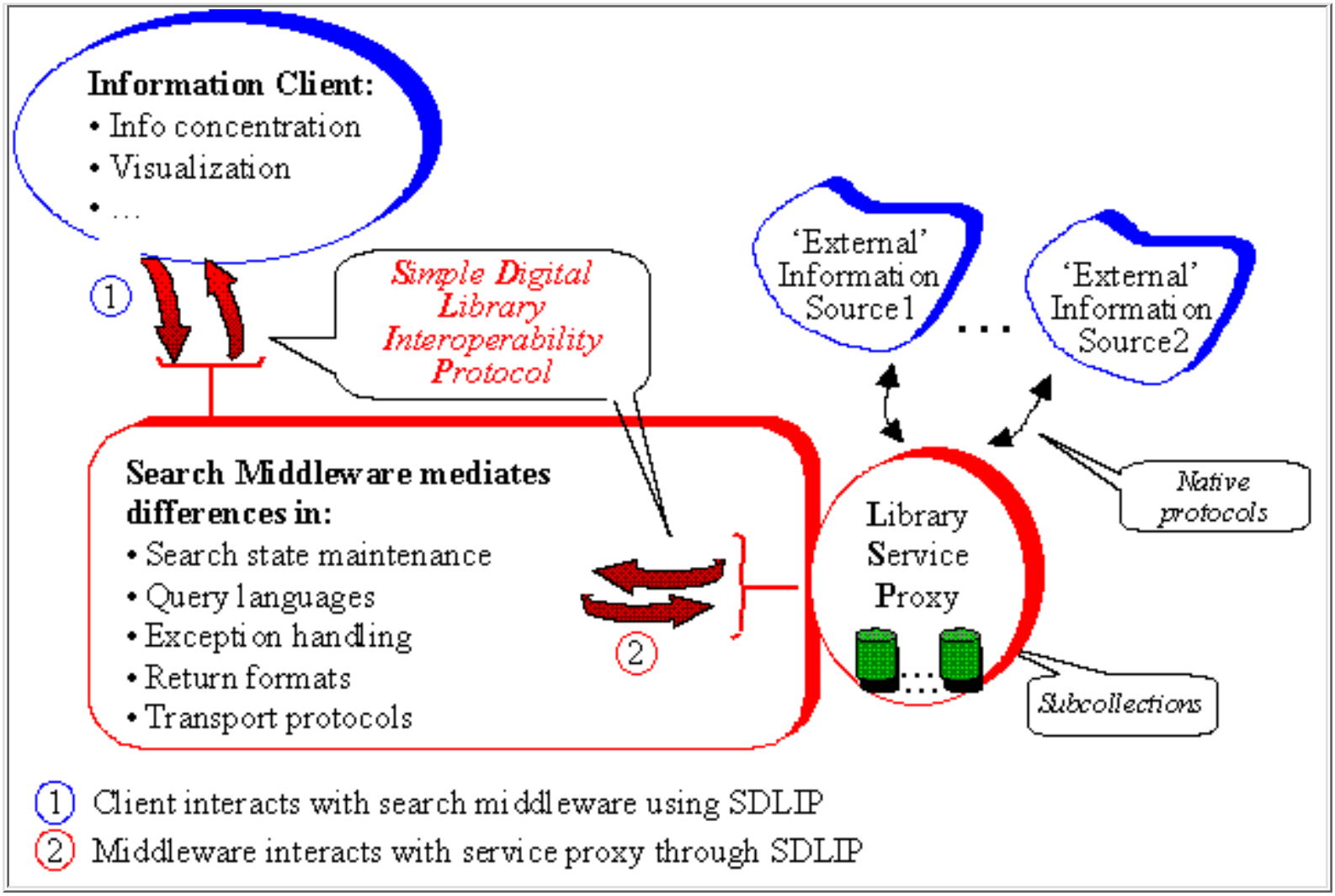

Figure 1: The Role of SDLIP in a Digital Library Architecture With Autonomous Sources and Wrappers

Note in Figure 1 that the information to be served is stored in repositories that do not (necessarily) implement SDLIP. This is a typical scenario, because information sources are often autonomously maintained, and do not present uniform interfaces to programs trying to extract information from them. Examples for external, non-conforming information sources are Web search engines, library catalogs, and commercial information providers, such as Nexus-Lexus, or the Dialog Corporation.

The 'Library Service Proxy' (LSP) in Figure 1 wraps two external sources. Through its back end, the LSP interacts with the external services via the transport and higher-level protocols required for these services. One LSP may thus serve out multiple 'subcollections'. At the front end, the proxy supports SDLIP. Of course, an information source may itself provide SDLIP access. In that case, the client can interact directly with the source.

The basic interaction is for the client to request a search across the network. Part of the request specifies how many documents are to be returned initially, once the search will be complete. The request also specifies which portion of each document is to be returned. For example, the client might ask for authors and titles of the first 10 documents to be returned right away. The client may later request more documents of the result, or it may request additional portions of the documents already delivered.

The protocol details can be obtained from the SDLIP documentation. In the following, we examine four of the features that must be considered when designing search middleware. These features are the

maintenance of search state, management of protocol complexity and extensions, query language formats, and the transportation of search requests and results. A longer version of this article also discusses load balancing and exception handling.

We describe how other search middleware, like Z39.50, handles these design issues. Also included in some of the comparisons are protocols such as CORBA [3], DCOM [4], and HTTP [5], which are not in themselves search middleware, but are often used as building blocks for highly customized, one-of-a-kind solutions.

3. State Maintenance

The state maintenance feature determines whether searches are 'one-shot deals', or whether clients may submit a query, retrieve a portion of the result, and then refer to the result set later on for follow-up exploration. The decision of whether information servers maintain result sets for clients has far reaching consequences.

Stateful servers are very efficient for clients who engage in highly interactive result set exploration. This is especially true for servers that provide fine grained access to document fragments. For example, a text document server might allow clients to ask separately for a document's author, title, abstract, or publication date. The initial query might request just the title of each result document. Based on the title list, the client might request more document attributes for some of the results. Such requests are easily filled when the server maintains result sets. However, implementations pay for this convenience in two ways. Servers are more difficult to scale up as the number of simultaneous users rises. If result sets must be cached, excessive space requirements may have to be managed, especially in high-volume situations.

The second difficulty that arises is that clients might never issue follow-up requests, and thus tie up server resources indefinitely. Typically, clients are required to close search sessions explicitly to signal the server that resources can be freed. If such a release is never received, the server must recover on its own.

World-Wide Web solutions generally use stateless servers. Any state to be maintained is moved to the client and stored there. Special identifiers, called 'cookies' may be used to help the server restore portions of a search context in successive interactions with a client. Cookies are passed from the server to the client, and are stored there. They contain all the information the server needs to 'remember' about a previous interaction with the client. When the client returns to the information source, the server may request the cookie, and use it to restore session state.

This approach to state maintenance is sufficient for simple applications, like standard Web search engine requests where results are merely references to documents located elsewhere. For richer interactions, server side statelessness can be very expensive, because the restoration of search context for each followup request implies repeated replication of search effort at the server side.

SDLIP addresses the issue of state maintenance with a 'parking meter' approach. With each search request, clients include the amount of time they wish the server to maintain the corresponding result set. In its response, the server returns the amount of time it is willing to grant for that request. A stateless server might, for example, reply with a time of zero, to indicate that the server does not maintain state at all. Or it might offer a somewhat smaller time span than what the client requested. The degree of state maintenance committment is thus determined by the server, rather than the client. This is appropriate, since it is the server which must marshall the corresponding resources. Once the degree of state management is thus

established, an imaginary parking meter clock begins to tick. Once the clock reaches zero, the server is free to discard state. If the client wishes to extend the amount of time the result set is available, an `extend timeout` operation is available for requesting additional result set maintenance time. The server may again, grant the request, reply with a smaller timeout, or simply reply with a zero, indicating that it is unable to retain the result set any longer.

4. Complexity Management and Extensibility

It is crucial for any software to be easily understood and maintained. But this is particularly important for any kind of middleware, which is typically used by many applications. A variety of approaches to complexity control have been tried for protocol design. The most obvious is the exclusion of features. The decision to drop features is often painful. But the pain may well be predominantly the designer's. Frequently, only a fraction of the features that make specifications bulge are used by a large majority of clients. A disadvantage of the exclusion approach is that it makes protocol design very difficult, because one wrong decision can render the result useless. If a key feature is discarded, the protocol may not fill crucial needs for too many applications.

Another approach to complexity control is to define and implement a rich core functionality, and then to specialize and limit this functionality for particular purposes. Z39.50 *profiles* fall into this category [6]. For example, the ZDSR profile [7] customizes Z39.50 for searching over document metadata and retrieval characteristics of search engines, such as the engine's ranking algorithms. Similarly, the GILS profile customizes Z39.50 for searches over geographic information. The advantage of profiling over the feature exclusion approach is that a protocol can be made all encompassing, yet able to be pared down for use in particular domains. A potential pitfall is that the rich core set can still convey the impression of complexity. At a minimum, implementations must be provided that hide the core and provide applications with a simple interface. In sofar as a profile restricts the features an implementation provides, another danger is the potential for interoperability breakdowns. In the worst case, profiling can deteriorate into creating a set of disjoint protocols. This is, of course, a danger with any extensible approach.

A third approach to making protocols less complex and still very adaptable, is the notion of metaobject protocols (MOPs) [8]. MOP based protocol implementations make each component of a protocol into an object with its own (metalevel) interface. The protocol implementation can then be modified, making it simpler, or more complex. For example, a search engine implementation might be modularized to include a query parser, a request dispatcher, a database access module, a ranking unit, and a result set manager. In a MOP-based approch, the behavior of each module would be controllable through its metalevel interface. The query parser, for example, might be programmable to allow some query operators, and to reject others. Or the request dispatcher interface might contain a switch that allows a metalevel programmer to control whether search clients are allowed to include requests for quality of service with their searches.

By programming at the metalevel, maintenance staff of a MOP-enabled search engine would therefore be able to change the interface through which client applications interact with the engine. MOP based protocol implementations are thus highly adaptable to special purpose uses. One downside is that very advanced programming techniques are required to obtain high enough performance. Without these techniques, all of the client/server interactions in effect run through an interpreter.

SDLIP accomplishes a degree of complexity control through the partitioning of operations into coherent interfaces. Figure 2 shows how the SDLIP operations are divided into three such interfaces.

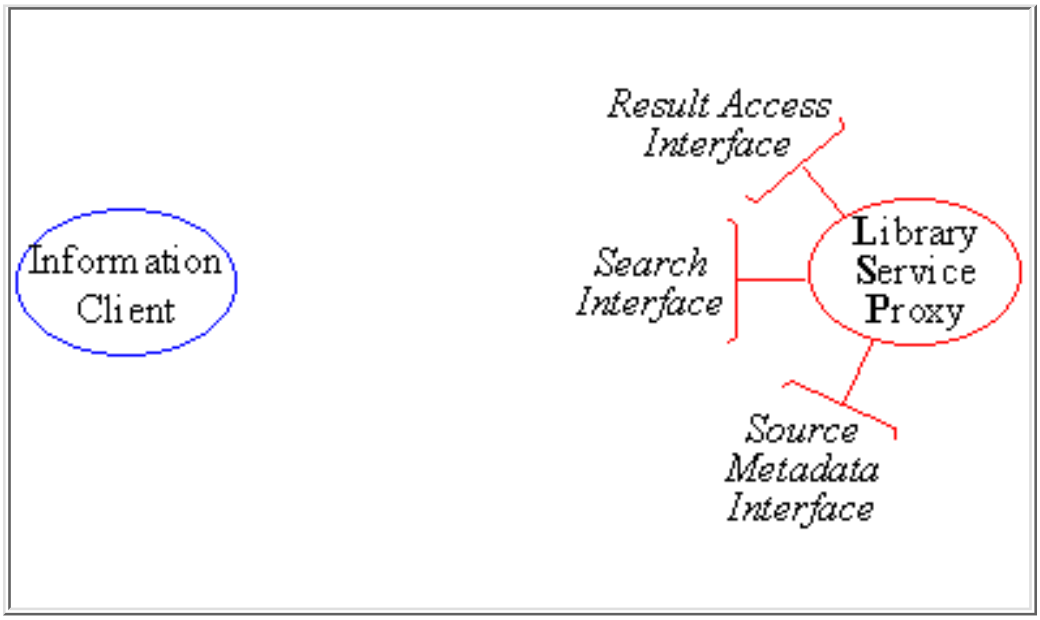

Figure 2: SDLIP Consists of Operations Grouped into Three Interfaces

Each interface contains at most four or five operations. Parameters and return values use XML syntax.

The search interface on the service contains the operations needed for submitting a search request to the service. The result access interface allows client applications to request the set of result documents. The source metadata interface, finally, allows clients or services such as metasearch engines to question a library service proxy about its capabilities. This might include a list of the subcollections served by the LSP, or the attributes that may be searched.

The partitioning into interfaces has three advantages. First, the interfaces make it clear which role each operation plays, and for which participants and phases of the search transaction the operation needs to be implemented.

Second, the interface notion enables clean expansion of the protocol in the future. One can subclass the existing interfaces to accommodate more elaborate facilities, or one can add additional interfaces. For example, one could use interface inheritance to add operations to the source metadata interface, if in the future some LSPs wish to export additional metadata, or wish to export that data in some new format. Or one might want to add a whole new interface for financial transactions. Neither of these expansions would impact the existing core protocol. This is very different from the profiling approach: whereas profiling begins with a rich core and then limits it for customization, the inheritance approach begins with a small core, and expands it to accommodate special needs. The hierarchical nature of inheritance then allows protocol compliance statements to be made about any given implementation. One can point to a 'cut-off tier' in the hierarchy and state that everything above it is supported, and everything below it is not.

A third advantage of organizing SDLIP's operations into functionally coherent interfaces is that for some scenarios, or 'configurations', some of the interfaces are not needed at all. Rather than having to list various operations to be dropped for these cases, we can then simply say that interface X is not needed. For example, if a server is stateless, it does not need to implement a result access interface, because all results are returned in response to the query. Leaving out interfaces is like profiling in that it limits, rather than expands, but it does so in 'chunks' of functionality, rather than on an operation by operation basis. The difficulty with this approach is that the partitioning of operations into interfaces must be very well thought

through, so that the operations in one interface do indeed form a coherent collection that makes sense to include or exclude from an implementation.

The minimum a stateless SDLIP server needs to implement is the search interface. Clients can rely on it being present. If a server maintains result sets which clients can access, then the server also needs to implement the result access interface. Clients know what is available, because of the parking meter negotiation: if the server returns a non-zero state maintenance time, then the presence of a result access interface is implied. Though not required, all servers should implement the source metadata interface.

Different Ways of Using SDLIP

Figure 3 illustrates the flexibility of SDLIP's partitioned interface design. The figure shows how SDLIP can be used in three configurations. The simplest is the configuration of Figure 3a. It features one library service proxy serving the information, and a single client application object. The client submits the search request synchronously via the service's search interface. The results are returned as part of that call. The dotted lines in Figures 3a/b indicate a network boundary: entities on the same side of the line are assumed to be in the same address space.

Figure 3b shows a somewhat more sophisticated usage in which the server maintains the result set of the search, at least for a while. Later, the client might, again synchronously, ask for more documents of the same result set (2).

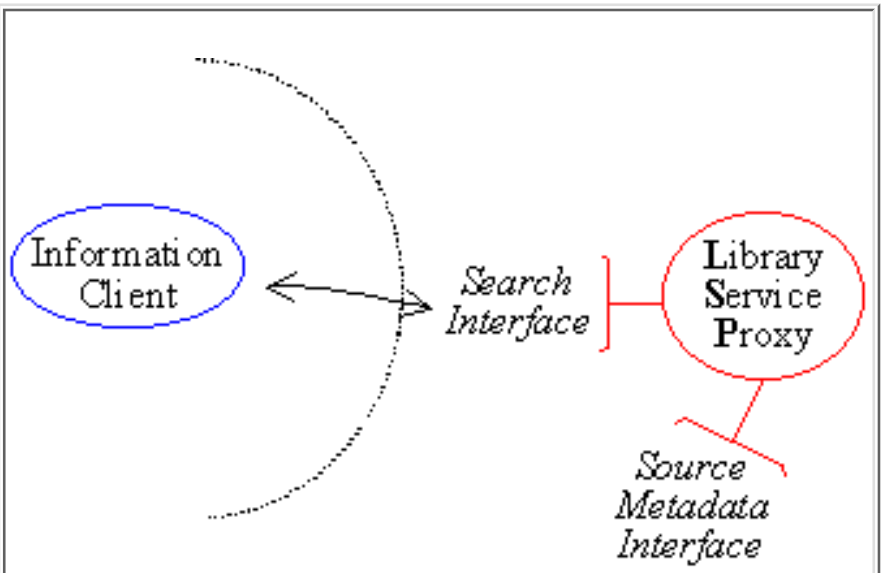

Figure 3a: Synchronous requests for search. Server does not save any state

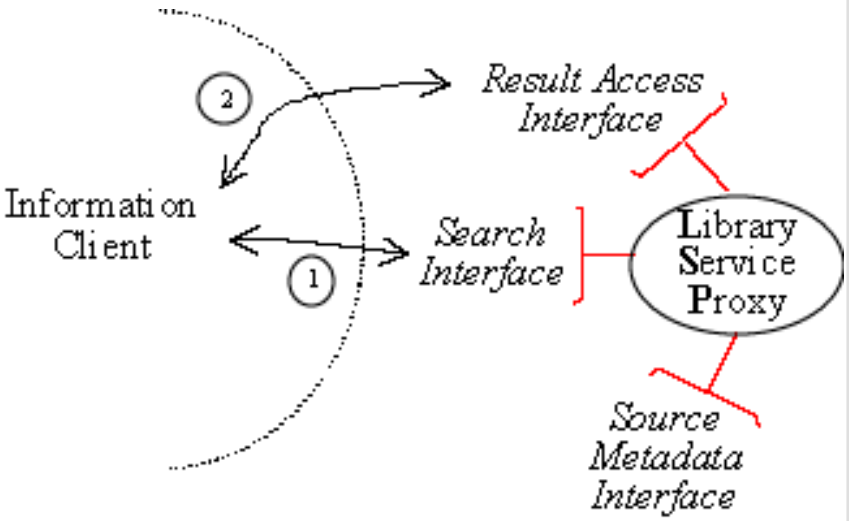

Figure 3b: Synchronous requests for search, and follow-up information

Figure 3: Different Configurations for Using SDLIP

More configurations are possible if the asynchronous SDLIP extensions are also supported.

5. Query Language and Format Neutrality

A major design decision for search middleware is whether search requests should be required to use a particular query language, and which data format will be used for results. Every combination has been tried. Large commercial providers often try to limit clients to a single language; the format of return results is prescribed. When data models are well enough described and agreed upon, such single language/format approach works very well. For example, SQL has dominated protocols for interacting with relational databases. If, however, such standardization and standard adherence is not present, then the single language approach is very limiting.

Another approach is to provide one client-side query language, but to translate queries to other languages that are native to the target search engines. This approach lets search middleware provide clients with easy access to diverse search facilities, while also allowing the use of native languages [9, 10]. One drawback of this approach is that the client-side language must be able to express all the features of all the search facilities. This in turn can lead to excessive complexity in the language. Another difficulty is that the search middleware must 'dumb down' queries that contain sophisticated features which are not supported by the target search facility. For example, a query might include the proximity operator which calls for search keywords to occur next to each other in the result documents. If this operator is not supported at the target, the middleware may need to replace the operator with a Boolean `and`. This, in turn, will result in inappropriate documents to be returned, which then need to be filtered out before they reach the client. Thus, query translation, while very convenient to the client, can be complicated.

A third approach is to translate the client query into an intermediate abstract query representation, which is

then interpreted by each server. Z39.50 uses this method to pass query information from the client to the server. This method also suffers from a very complex query representation, which still may not cover all of the features of a given server.

The easy way out is a compromise which many protocols, including SDLIP, have taken. One can define a simple search language that is guaranteed to be supported by all search services. However, rather than limiting clients and services to this language, the protocol can allow the use of other languages, as long as the search request includes information as to which language is being employed.

Here is an example query expressed in SDLIP's standard, minimal language, called *basicsearch*, which is taken from the DASL internet draft. XML is used for encoding. In the first line, the expression states that this is a basicsearch query. The first line also introduces the XML namespace `Dialog`, associating it with the Dialog Corporation's Web site for reference. The remainder of the expression requests documents whose `author` property equals 'Miller', and whose publication date is '1994'.

```
<basicsearch xmlns="DAV:" xmlns:Dialog="http://dialog.com/">
  <where>
    <and>
      <eq>
        <prop><Dialog:au/></prop>
        <literal>Miller</literal>
      </eq>
      <eq>
        <prop><Dialog:py/></prop>
        <literal>1994</literal>
      </eq>
    </and>
  </where>
</basicsearch>
```

The same query could be expressed in Dialog Corporation's native language like this:

```
<Dialog:StandardQuery>
  au=Miller and py=1994
<Dialog:StandardQuery>
```

This format, of course, is more economical, and may be preferred for clients dedicated to this single information source. But this choice is made at the expense of interoperability. Basicsearch provides a minimal common 'default' query language. SDLIP can transport other query languages just as well. The language used is specified in each search request. Thus, evolving query languages, like W3C-QL98 for querying XML, can be accommodated.

In contrast to queries, the format for SDLIP search results is strictly prescribed. Here is an example. The DID is a document ID which is generated by the server, and which can be used to request additional properties of the respective document.

```
<SearchResult>
  <doc>
    <DID>1</DID>
    <propList>
```

```
      <author>Bill Smith</author>
      <author>Frank Miller</author>
      <title>This is My Life</title>
      <abstract>It's been great so far.</abstract>
    </propList>
  </doc>
  <doc>
  ...
  </doc>
</SearchResult>
```

6. Transport Neutrality

Once queries and formats are taken care of, search middleware needs to ensure that information requests and results can be transported between clients and servers over the Internet. There are at least four methods for accomplishing this transport: HTTP, CORBA, DCOM, and specialized, proprietary techniques. HTTP has the advantage of great simplicity, in part because all HTTP commands are human-readable. Disadvantages arise when complex interfaces are involved, with many different operations, and parameters that comprise complex data structures. In those cases, large amounts of error-prone software is needed to marshall the necessary information into and out of formats appropriate for transport over the wire.

For example, consider a medical application which builds up data structures containing information about a patient being transported to a hospital. These data structures might include the patient's address data, measured vital signs, and health history. If this application then constructed a search to retrieve, say, related brain scans, it would be convenient and reliable if the data structures themselves could simply be passed to the search engine as parameters. When this is not possible, the application must extract all the data from the data structures, and must embed it in some other format, possibly transforming numeric data into ASCII characters. Approaches like CORBA and DCOM can be more appropriate for those cases, because they include machinery to manage all of this complexity. The price is added installation complexity and a steeper initial learning curve for application programmers.

In order to ensure widest possible usability, SDLIP is 'transport neutral'. The information required by servers and clients may be transported by any of the three major transport systems. Figure 5 shows the architecture that enables this transport neutrality through straight-forward layered abstraction.

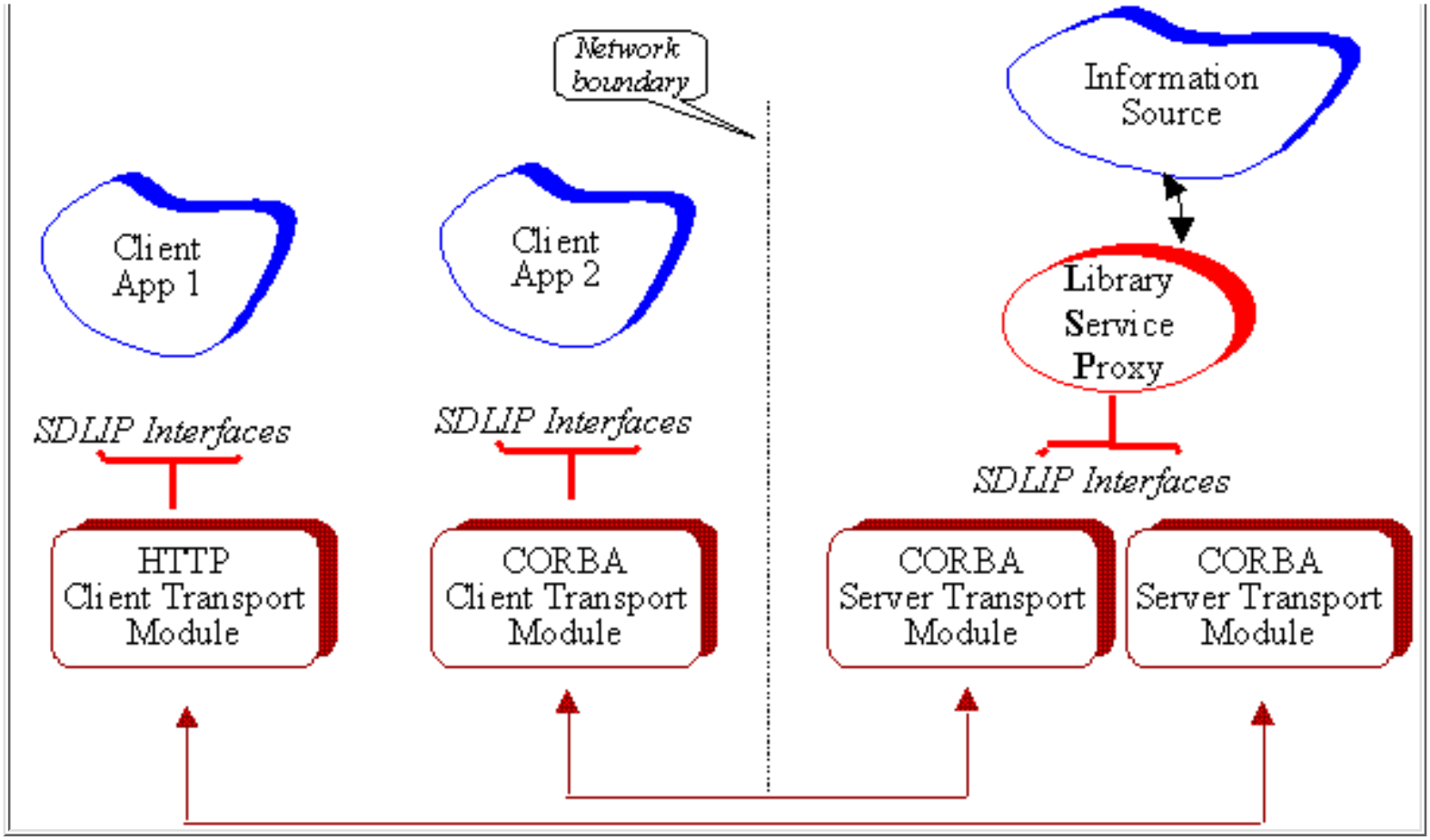

Figure 5: SDLIP Implementation Architecture

The architecture follows the basic CORBA approach, while simplifying many of the details: Client applications communicate with local pieces of software, called *client transport modules*. These modules all present identical SDLIP interfaces, as if they themselves were the servers that the clients wish to access. In reality, the modules simply act as go-betweens to their mirror modules at the server side. Transport modules are written just once by protocol implementers. Application writers do not need to concern themselves with these modules. Communication between the modules may use CORBA, HTTP, or some other protocol. This detail is transparent to the client applications. Transport details are also transparent to the library service proxy. Note that a single LSP may serve multiple server transport modules. This multiplexing arrangement is a great advantage, because it allows a single piece of information source wrapper software to be accessible through HTTP, CORBA, or other transports without effort. The LSP simply presents the SDLIP interface to the transport modules. The modules 'look' to the LSP like local clients. In reality, they are simple relays to the clients across the network. Java based CORBA and HTTP client and server transport modules are available, so that new client/server application builders can focus on the information access, rather than having to worry about transport details.

7. Conclusion

Search middleware enables new information intensive applications to be developed easily. This capability is crucial, if information access, exploration, and sensemaking are to progress beyond their current state. Search middleware design, however, is a delicate balancing act that requires continuous weighing of simplicity and demands for features. We have introduced some of the related design considerations, and have exemplified them with CORBA, Z39.50, HTTP, and SDLIP, a new search middleware that is being

adapted by several participants of the latest Digital Library Initiative (DLI2). Documentation for SDLIP is available at http://www-diglib.stanford.edu/~testbed/doc2/SDLIP/. SDLIP implementations exist for sources such as California Digital Library Collections, UC Berkeley's Melvyl, a metadata server at the San Diego Supercomputer Center, the Networked Computer Science Technical Reference Library (NCSTRL), a movie database, and Z39.50 services, such as the Library of Congress.

References

[1] *Information Retrieval: Application Service Definition and Protocol Specification*. ANSI/NISO, April, 1995. Available at http://lcweb.loc.gov/z3950/agency/document.html.

[2] Saveen Reddy, Dale Lowry, Surenda Reddy, Rick Henderson, Jim Davis, and Alan Babich. *DAV Searching & Locating, Internet Draft*. IETF, June, 1999. Available at http://www.webdav.org/dasl/protocol/draft-dasl-protocol-00.html.

[3] Object Management Group. The Common Object Request Broker: Architecture and Specification. Dec, 1993. Accessible at ftp://omg.org/pub/CORBA.

[4] Microsoft COM Technologies. http://www.microsoft.com/com/tech/DCOM.asp.

[5] HTTP - Hypertext Transfer Protocol. 2000. http://www.w3.org/Protocols/.

[6] *About Profiles*. Library of Congress, January, 1998. Accessible at http://lcweb.loc.gov/z3950/agency/profiles/about.html.

[7] *Z39.50 Profile for Simple Distributed Search and Ranked Retrieval*. Library of Congress, March, 1997. Accessible at http://lcweb.loc.gov/z3950/agency/profiles/zdsr.html.

[8] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[9] Chen-Chuan K. Chang, Héctor García-Molina, and Andreas Paepcke. Boolean Query Mapping Across Heterogeneous Information Sources. *IEEE Transactions on Knowledge and Data Engineering*, 8(4):515-521, Aug, 1996.

[10] Dataware Search and Retrieval. 2000. http://www.dataware.com/technology/.