SDLIP + STARTS = SDARTS
A Protocol and Toolkit for Metasearching

Noah Green
ngreen@cs.columbia.edu

Panagiotis G. Ipeirotis
pirot@cs.columbia.edu

Luis Gravano
gravano@cs.columbia.edu

Computer Science Dept.
Columbia University

ABSTRACT

In this paper we describe how we combined SDLIP and STARTS, two complementary protocols for searching over distributed document collections. The resulting protocol, which we call SDARTS, is simple yet expressible enough to enable building sophisticated metasearch engines. SDARTS can be viewed as an instantiation of SDLIP with metasearch-specific elements from STARTS. We also report on our experience building three SDARTS-compliant wrappers: for locally available plain-text document collections, for locally available XML document collections, and for external web-accessible collections. These wrappers were developed to be easily customizable for new collections. Our work was developed as part of Columbia University's Digital Libraries Initiative–Phase 2 (DLI2) project, which involves the departments of Computer Science, Medical Informatics, and Electrical Engineering, the Columbia University libraries, and a large number of industrial partners. The main goal of the project is to provide personalized access to a distributed patient-care digital library.

Categories and Subject Descriptors

H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*Information Filtering, Query Formulation, Search Process, Selection Process*; H.3.5 [**Information Storage and Retrieval**]: Online Information Services—*Data Sharing, Web-based Services*; H.3.7 [**Information Storage and Retrieval**]: Digital Libraries; H.2.4 [**Database Management**]: Systems—*Textual Databases, Distributed Databases*; H.2.5 [**Database Management**]: Systems—*Heterogeneous Databases*

1. INTRODUCTION

The information available in electronic form continues to grow at an exponential rate and this trend is expected to continue. Although traditional search engines like AltaVista can solve common information needs, they ignore the often-valuable information that is "hidden" behind search interfaces, the so-called *"hidden web."*

One way to access the information available in the hidden web is through *metasearchers* [12, 18, 24], which provide users with a unified searchable interface to query multiple databases simultaneously. A metasearcher performs three main tasks. After receiving a query, it determines the best databases to evaluate the query (*database selection*), it translates the query in a suitable form for each database (*query translation*), and finally it retrieves and merges the results from the different sources (*results merging*) and returns them to the user using a uniform interface.

Metasearching is a central component of the Digital Libraries Initiative–Phase 2 (DLI2) project at Columbia University, which involves the departments of Computer Science, Medical Informatics, and Electrical Engineering, the Columbia University libraries, and a large number of industrial partners (e.g., IBM, GE, Lucent). The project is named PERSIVAL (PErsonalized Retrieval and Summarization of Image, Video, And Language resources) and its main goal is to provide personalized access to a distributed patient-care digital library. The information needs vary widely among the users of the system. We have to provide access to all kinds of collections, ranging from Internet sites with consumer health information to the Columbia Presbyterian Hospital information system, which stores patient-record information and other relevant resources. Metasearching is further complicated by the different access methods used by each source, which range from public CGI-based interfaces to proprietary access methods used inside the Columbia Presbyterian Hospital. Key features of the project are the abilities to access all these distributed resources regardless of whether they are available locally or over the Internet, to fuse repetitive and conflicting information from multiple relevant sources, and to present concisely the retrieved information. Exploiting such a variety of information sources is a challenging task, which could benefit from information sources supporting a common interface for searching and metadata extraction.

Rather than defining yet another new protocol and interface that distributed sources should support, we decided to exploit existing efforts for our project. More specifically, we built on two complementary protocols for distributed search, namely SDLIP [20] and STARTS [11], and combined them to define SDARTS (pronounced "ess-darts"), which is the focus of this paper.

SDLIP (Simple Digital Library Interoperability Protocol) is a protocol jointly developed by Stanford University, the University of California at Berkeley and at Santa Barbara, the San Diego Supercomputer Center, the California Digital Library, and others. The SDLIP protocol defines a layered, uniform interface to query and retrieve the results from each searchable collection through a common interface. SDLIP also supports an interface to access source metadata.

SDLIP is optimized for clients that know which source they wish to access. In contrast, the focus of STARTS (STAnford protocol proposal for Internet ReTrieval and Search) is on metasearching. A crucial component of STARTS is the definition of the specific metadata that a source should export to describe its contents. This metadata includes the vocabulary (i.e., list of words) in the source, plus vocabulary statistics (e.g., the number of documents containing each word). These summaries of the source contents are useful for the metasearchers' database selection task.

SDLIP and STARTS complement each other naturally. SDLIP has a flexible design that allows it to host different query languages and metadata specifications. The major parameter and return types of its methods are passed as XML, and the DTDs for this XML allow for extensions and instantiations of the protocol. Thus, SDLIP can easily host the main components of the STARTS protocol. The result of this combination is SDARTS, which can be regarded as an instantiation of the SDLIP protocol with a specific query language, and, more importantly, with the richer metadata interface from STARTS, which is useful for metasearching.

To simplify making document collections compliant with SDARTS, we have developed a software toolkit that is easily configurable. This toolkit includes software to index locally available collections of both plain-text and XML documents. Also, to be able to wrap external collections over which we do not have any control and which do not support SDARTS natively, the toolkit includes a reference wrapper implementation that can be augmented for new external collections with relatively small changes.

SDARTS and its associated software toolkit, which are the focus of this paper, provide the necessary infrastructure for metasearching and for incorporating collections into our digital libraries project with minimal effort. In Section 2 we describe in detail the metasearching tasks and the challenges involved. In Sections 3 and 4 we describe STARTS and SDLIP respectively, the two protocols on which we have based SDARTS. The resulting protocol is described in Section 5 and the toolkit and reference implementations are presented in Section 6. Finally, Section 7 provides further discussion of our overall experience.

2. METASEARCHING

As we briefly mentioned in the Introduction, metasearching mainly consists of three tasks: *database selection*, *query translation*, and *result merging*.

- **Database Selection:** A metasearcher might have hundreds or thousands of sources available for querying. An alternative to broadcasting queries to every source every time is to only send queries to "promising" sources. This alternative results not only in better efficiency but in better effectiveness as well. Selecting the best sources for a given query requires that the metasearcher have some information about the contents of the sources. Some database selection techniques rely on human-generated descriptions of the sources. More robust approaches rely on simple metadata about the contents of the sources like their vocabulary (i.e., list of words) together with simple associated statistics. Research in this area includes [10, 13, 21, 12, 18, 24].
- **Query Translation:** Metasearchers send queries to multiple different sources, which requires translating the queries to the local query language and capabilities supported at each source. Query translation is facilitated if sources export metadata on their query capabilities, on whether they support word stemming, on the attributes or fields available for searching (e.g., author and title), etc. Research in this area includes [6, 7, 8].
- **Result Merging:** Sources typically use undisclosed document ranking algorithms to answer user queries, which makes the combination of query results coming from multiple sources challenging. Furthermore, even two collections that use the same ranking algorithm might rate a common document quite differently depending on the other documents present in each collection. Result merging is facilitated if sources export metadata on their contents and query results. Research in this area includes [22, 5].

A metasearcher needs information about the underlying collections to perform the tasks above successfully. Consequently, there is a need for a layer on top of the information sources that will mask source heterogeneity and export the right metadata to metasearchers. One protocol that was designed to solve exactly this problem is STARTS, which we briefly review next.

3. STARTS: A PROTOCOL PROPOSAL TO FACILITATE METASEARCHING

STARTS [11] is a protocol proposal that defines the information that a source should export to facilitate metasearching. By standardizing on a common way of interacting with clients and by defining what information a document source should export, metasearching becomes a much easier task. Of course, the exported information alone is not a panacea and does not solve the metasearching problems, but at least it can make these problems more tractable. Specifically, STARTS defines the information that should be included in the queries to the sources, the format of the query results, and the metadata a source should export about its contents and capabilities [11].

For historical reasons, the original STARTS specification used Harvest's SOIF format [3] to encode queries, results, and metadata. Also, STARTS did not define explicitly how the information is transported. For our project, we defined an encoding of all the STARTS information in XML. All STARTS queries, result sets, and metadata objects are then represented as XML documents. This means that all STARTS elements can be easily manipulated by available XML technologies, such as XSL, SAX, etc. Additionally, the transformation from one XML dialect to another can be easily achieved using off-the-self tools, thus it is easy to support other query languages by just adding a thin layer

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE starts:scontent-summary SYSTEM "http://www.cs.columbia.edu/~dli2test/dtd/starts.dtd">
<starts:scontent-summary
         xmlns:starts="http://www.cs.columbia.edu/~dli2test/STARTS/"
         version="Starts 1.0"
         stemming="false"
         stopwords="false"
         case-sensitive="true"
         fields="false"
         numdocs="19997"
>
...
        <starts:field-freq-info>
                <starts:field type-set="basic1" name="body-of-text"/>
                <starts:term>
                        <starts:value>algorithm</starts:value>
                </starts:term>
                <starts:term-freq>75</starts:term-freq>
                <starts:doc-freq>34</starts:doc-freq>
...
```

Figure 1: A small fraction of a STARTS metadata object for a document collection.

that translates the query format of another query language to STARTS. Another strong reason to transform STARTS to XML format is that SDLIP, the protocol that we review next, uses XML to encode information, so this transformation makes combining the two protocols easier. We refer to the "XMLized" STARTS version as *STARTS XML*.

Figure 1 shows part of the metadata object that summarizes the contents of the "20 Newsgroups" document collection [2], which is frequently used in the machine learning community. This collection consists of 19,997 articles posted to 20 newsgroups. 34 of these articles contain the term "algorithm" in their body, as the metadata record in the figure indicates. Also, the record shows that this term appears 75 times over these 34 articles. [1] A metasearcher can use these content summaries to select what sources to send a given query (i.e., for database selection). More specifically, a metasearcher can estimate the number of matches that a given query will produce at each source. As a simple example, given a query on "algorithm" a metasearcher might conclude that it does need to contact a source with very few or no documents containing that word, as reported in the corresponding STARTS content summary for the source. (See [12, 18, 24] for research on how to exploit this type of information for database selection.)

4. SDLIP: SIMPLE DIGITAL LIBRARY INTEROPERABILITY PROTOCOL

SDLIP [20] is a layered protocol that defines simple interfaces for interoperability between information sources. Its designers define it as "search middleware," lighter and easier to use for web-related applications than standard middleware protocols like Z39.50 [1].

The main purpose of SDLIP is to provide uniform interfaces to information sources for querying and retrieving results, and taking care of the transport of the data across the network. SDLIP defines the interfaces that a source or a wrapper of a source should implement so that it can be accessed by an SDLIP-enabled client. An implementation of the SDLIP interfaces is called a *Library Service Proxy (LSP)*, and functions as a lower-level wrapper around one or more underlying collections. Figure 2, borrowed from [20], depicts the role of SDLIP in a digital library architecture. LSPs mask access differences among the information sources in the digital library, and present these sources to clients through a uniform interface. Additionally, SDLIP places one more layer above the LSP, namely network transport, via HTTP/DASL or CORBA, to define exactly how the servers and the clients should communicate with each other. The SDLIP protocol supports both a stateful and a stateless mode of operation. Just as STARTS is a stateless protocol, we decided to use only the stateless version of SDLIP in our project. Often web search over distributed collections does not justify the added complexity of supporting stateful protocols.

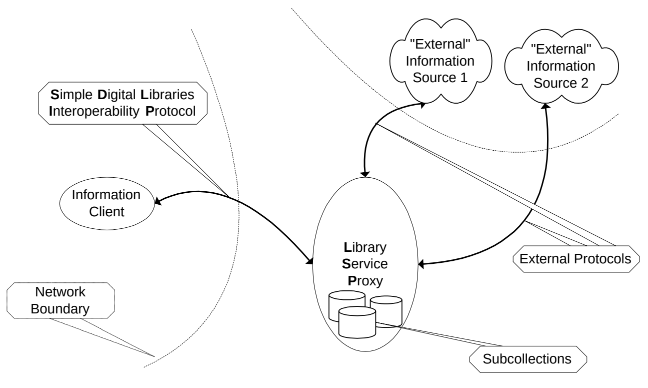
"External" Information Source 1
"External" Information Source 2
Simple Digital Libraries Interoperability Protocol
Information Client
Library Service Proxy
External Protocols
Network Boundary
Subcollections

Figure 2: The role of SDLIP in a digital library architecture with autonomous sources and wrappers (taken from [20]).

In a typical deployment of SDLIP, the LSP is a wrapper that knows how to interact with the underlying collections and exports a uniform SDLIP interface. The LSP interface is divided into three parts: the *search* interface, the *results*

[1]Although XML is quite verbose to describe these statistics, we should note that these XML objects can be compressed effectively [16], and this compression can take place before a potential transmission to a client.

interface, and the *metadata* interface. The search interface defines the operations for submitting a search request to an LSP. The result interface allows clients to access the result of a search. Finally, the metadata interface allows clients to question a library service proxy about its capabilities and contents. For each interface, SDLIP supports, by design, a limited set of operations. This design decision is based on the observation that the SDLIP interfaces can be enhanced as needed through inheritance.

While inheritance is a useful tool for extending the SDLIP interfaces, there is another mechanism for extending the protocol's capabilities. The methods of the three interfaces described above are designed to pass XML documents as their major parameter and return types. The conditions for an SDLIP search, for example, would be specified not by primitive or object-based parameters, but rather as an XML "property sheet" of search parameters. SDLIP's grammar for this XML is extensible, making it easy to host additional metasearching-related features within SDLIP simply as some dialect of XML.

As mentioned above, SDLIP is optimized for clients that know which source they wish to access. In contrast, STARTS specifies, among other things, the metadata that sources should export for metasearching. This makes merging these two complementary protocols desirable. More specifically, we describe next how we instantiate the SDLIP interfaces with STARTS XML to obtain an expressive protocol for effective metasearching.

5. THE SDARTS PROTOCOL

In this section, we describe how we combined SDLIP and STARTS XML into the SDARTS protocol. Section 5.1 outlines our rationale, while Section 5.2 gives an overview of SDARTS.

5.1 SDARTS Design Rationale

SDARTS combines SDLIP and STARTS XML into an expressive protocol for distributed search. We now discuss our choice of these two existing protocols as the basis for SDARTS.

Our decision to adopt SDLIP was influenced by the fact that SDLIP is already in use by other DLI2 projects around the United States, and the ability to interoperate with resources made available by other digital libraries efforts is of course attractive. Additionally, the fact that there are already implementations that allow SDLIP clients to access the contents of sources supporting alternative protocols like Z39.50 [1] (but not vice versa) was another important factor in our choice of SDLIP as our "middleware" architecture. (Other emerging related efforts, notably the "Open Archives Initiative" [19], were still under development and not in stable form when we developed SDARTS.) Finally, we believe that the agreement on common interoperability protocols is a major factor in the success of efforts like the DLI2 projects.

At the same time, STARTS specifically describes the information that should be exchanged between sources and a metasearcher. STARTS includes support to plug in other attribute sets for searching like the Dublin Core [23] and Z39.50's Bib-1 [1] attribute sets. In fact, STARTS built on these efforts and already supports some of their most useful features. Additionally, STARTS specifies the metadata that sources should export for effective metasearching. Other protocols that define related metadata are the GILS Z39.50 profile [9] and Z39.50's Exp-1 [1] attribute sets. Again, STARTS built on these protocols. In particular, STARTS defines that the sources should export vocabulary frequency information (Section 3). SDLIP does not specify this kind of metadata, which is useful for metasearching. However, SDLIP is designed in such a way that it can easily incorporate additional capabilities, which can be exploited by clients that are aware of them. Thus, it is natural to enrich SDLIP with the STARTS components.

We standardized on STARTS XML as the XML "dialect" for SDLIP to exchange the extra information not included in the original version of SDLIP. SDARTS follows all of SDLIP's original DTDs, which include placeholders that can be exploited to include the necessary STARTS XML objects (e.g., the `getPropertyInfo()` method in SDLIP's metadata interface returns the `<propList>` element that can be used for this purpose). Since the vocabulary statistics for a source might be large, and given that a client other than a metasearcher might not need them, SDARTS returns only the URL where the content summary resides as part of the metadata for a source. Then, a metasearcher can download the summary outside of SDARTS with the given URL.

By standardizing on one XML format for SDLIP, we have created an architecture where an LSP is divided into two pieces: a standardized front-end that never needs to change, because it only has to deal with one dialect of XML, and an abstract back-end, which is implemented for specific underlying collection types. Thus, a programmer would need to write only this back-end implementation. Furthermore, this standardization and predictability of the back-end LSP makes it easier for us to create configurable reference implementations of the wrappers for frequently occurring collection types, at the expense, of course, of reduced flexibility.

Another factor that influenced our decisions was our intention to design a software design kit (SDK) for developers to create SDARTS-compliant wrappers. The main design goal was ease of implementation. To be sure, we wanted any programmer to be able to implement our standard wrapper interface from scratch, and create custom wrappers fine-tuned to their underlying collections. But in addition, we wanted to create reference implementations for some common collection types. Moreover, we wanted these implementations to be adjustable *without any additional writing of code* whenever possible. Thus, SDARTS includes a toolkit with reference implementations of wrappers for frequent types of collections. These wrappers are all configurable with text files. These files tell our wrappers things like how to index the documents in a locally available collection, how to transform queries into forms that external collections can understand, and how to translate results passed from collections. Once again, we chose XML as the format for these files. Later, in Section 6 we focus on the various reference implementations, and the pros and cons of our configuration-driven approach.

5.2 Overall SDARTS Architecture

Figure 3 shows the final SDARTS architecture, as it would be used over the Internet to make one wrapped collection available. The client layer is on the left side of the diagram, and the wrapped collection is on the right side.

In the diagram, we see two possible clients: a standard SDLIP client, since SDARTS uses the SDLIP interfaces and transport layer, and the `SDARTSBean`, a client component that we developed that simplifies access to SDARTS and en-

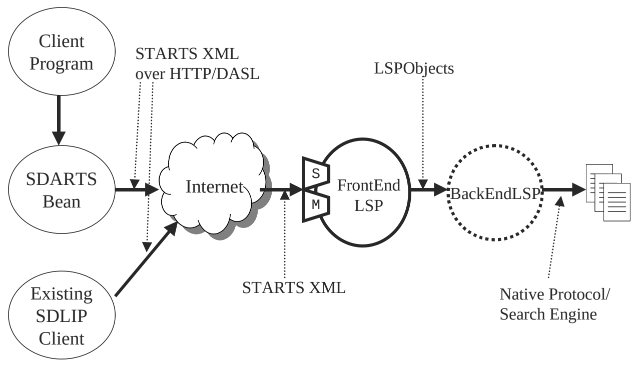

Figure 3: The use of the architectural components of SDARTS to query an SDLIP-compatible database over the Internet. (The "S" and "M" stand for SDLIP's "Search" and "Source Metadata" interfaces, respectively.)

hances SDLIP with the metadata information that can be used by metasearchers. For simplicity, we chose to use the HTTP/DASL protocol as transport layer and not to implement at this point the support for CORBA. In the diagram we show both prospective clients using the HTTP/DASL protocol for the communication. All messages passing between the clients and the top level of SDARTS are formatted in STARTS XML. As we discussed earlier, the benefits of this are:

- XML is an easily parsed and transformed language standard, and is ideal for client use; and
- Lower layers of SDARTS need only be implemented once, as the incoming XML protocol is already known.

The top layer of the server side is a Java component called the `FrontEndLSP`. This component implements the major SDLIP interfaces, and is for all intents and purposes an SDLIP LSP. The `FrontEndLSP` parses the incoming STARTS XML requests using the Simple API for XML (SAX), and generates various search objects that implement the common `LSPObject` interface.

These `LSPObjects` are just data containers; basically, they are objectified STARTS XML documents, and are passed to the lower layer. They represent the requests and responses that the system fulfills and produces. By standardizing on this object representation of the various SDARTS operations, we simplify the task for the prospective wrapper implementors. They will not need to parse any incoming XML or generate any outgoing XML, but instead they can just examine and create `LSPObjects`, whose simple interfaces make it easy to understand queries and generate results. Thus, the wrapper programmer needs only to implement the abstract `BackEndLSP` interface, which accepts and returns `LSPObjects`. This implementation reads `LSPObjects` that come from the `FrontEndLSP`, and generates other `LSPObjects` that it returns. Each `LSPObject` already knows how to represent itself as STARTS XML, so it is easy for the front-end to transform them into responses to a SDARTS client. In short, back-end developers do not really need to know anything about STARTS, SDLIP, or even XML in general. All they need to understand is what the back-end interface supports, what the `LSPObjects` are and how to read them, and, of course, how to handle the collection they are wrapping.

An alternative design choice for SDARTS could have used the Document Object Model (DOM), which provides an existing framework and implementation for converting XML documents into Java objects. While DOM does simplify the task of objectifying XML documents, it does so at the expense of specificity and performance. DOM objects have very generic interfaces. For example, if the `BackEndLSP` accepted DOM objects instead of `LSPObjects`, then wrapper programmers would have to remember what to expect when calling each of the generic getter methods on the DOM objects. They might have to make redundant calls in order to ascertain what data was actually available, and would certainly have to memorize what data types to expect. There would be much casting overhead during each interaction. In short, they would have to perform significant extra work to extract the necessary query data. As mentioned above, our overriding design goal was that it be relatively simple to implement the wrappers. As such, we opted to specify our own object representation of the XML documents, `LSPObjects`, with well-known method names and well-understood types.

The decisions made during the development of SDARTS resulted in a protocol that made the implementation of wrappers a relative easy task, even for a moderately experienced programmer. However, our goal was to facilitate even further the development of wrappers for existing collections. For this reason we have created reference implementations of wrappers for common collection types that can be easily configured, as described next.

6. CONFIGURABLE REFERENCE IMPLEMENTATIONS

In this section, we describe the reference wrapper implementations that we developed for common collection types. To wrap a collection, we can use the closest of these implementations and adapt it by defining simple XML-based files. Currently, we have three reference wrapper implementations:

- A wrapper for unindexed text documents residing in a local file system (Section 6.1);
- A wrapper for unindexed XML documents residing in a local file system (Section 6.2); and
- A wrapper for an external indexed collection fronted by a form-based WWW/CGI interface (Section 6.3).

6.1 `TextBackEndLSP`: Unindexed Text Document Collections

The first wrapper we created is for locally available collections of documents with no index over them. For this, we leveraged an open-source Java search engine known as Lucene [17] to index and search such collections. In its internal architecture, our wrapper hides the Lucene engine behind a more abstract interface, so in practice other search engines could be used with this wrapper.

Figure 4 shows the basic structure of the wrapper, known as `TextBackEndLSP`. The back-end administrator needs only write one file: `doc_config.xml`. This tells the wrapper where

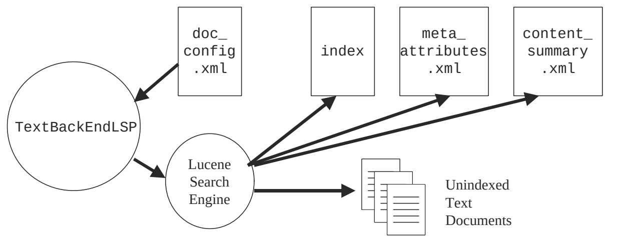

Figure 4: Structure of the `TextBackEndLSP` wrapper.

```
<doc-config re-index="true">
          <path>/home/dli2test/collections/doc1/20groups </path>
          <linkage-prefix>http:// localhost/20groups</linkage-prefix>
          <stop-words>
                    <word>the</word>
                    <word>a</word>
          </stop-words>
          <field-descriptor name="author">
                    <start>
                              <regexp>^From: </regexp>
                    </start>
                    <end>
                              <regexp>$</regexp>
                    </end>
          </field-descriptor> . . . . . . . .
</doc-config>
```

Figure 5: Portion of the contents of `doc_config.xml`.

the documents are, what fields should be indexed, and where to find the fields in the documents. `TextBackEndLSP` uses the file to index the documents offline, so that when the SDARTS server is available, the collection is fully searchable. During indexing, the wrapper also generates the two metadata files, `meta_attributes.xml` and `content_summary.xml`; these contain the standard STARTS metadata, and can be returned in response to requests from the front-end. The file `doc_config.xml` is itself in a very simple XML format (see Figure 5). An important point is the use of regular expressions to tell the wrapper how to extract the values associated with each searchable field like author or title. In our sample file, the author field of a document is found on a line that starts with the string "From:".

This format is simple enough for a non-programmer to edit by hand, and a GUI administration tool can certainly easily generate it. It has some limitations; it assumes that all documents in the collection are in the same format. In addition, field data must be contiguous within a document. This is especially a problem with XML documents, so we created a separate wrapper, described below, to deal with them.

6.2 `XMLBackEndLSP`: Unindexed XML Document Collections

This wrapper for documents formatted in XML is quite similar to its plain-text counterpart; the key difference is that to index collections of XML documents a slightly more complicated configuration file is needed. Here, the configuration file `doc_config.xml` only tells the wrapper where to find the documents. The administrator must then write a

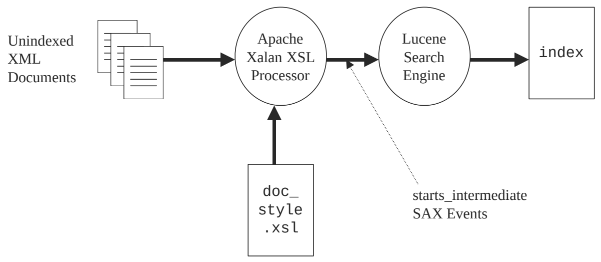

Figure 6: Indexing XML documents for the Lucene search engine, using an XSLT stylesheet to locate the fields in each document.

second file, `doc_style.xsl`, which is an XSL Transformations (XSLT) stylesheet. It is this second file that tells the wrapper how to find the fields in the documents during the indexing process.

Figure 6 illustrates the indexing process in this wrapper. The `doc_style.xsl` file should be written to transform an XML document from the collection into a new XML format we devised called `starts_intermediate`. This is a simple subset of STARTS XML that describes STARTS documents. The transformed document is never materialized; rather, the transformation emits SAX events that ultimately tell the search engine indexer how to index the document. Like the text document wrapper, this process assumes that all documents in the collection are structured similarly.

When we set out to devise this and the final wrapper, we believed that writing XSLT stylesheets was easier than designing and implementing Java objects; while this is true, XSLT is still non-trivial. In Section 7, we assess XSLT's suitability for making wrappers easy to implement and configure.

6.3 `WWWBackEndLSP`: WWW/CGI Collections

`WWWBackEndLSP` is the most complex of the three wrappers. It is intended for autonomous, remote collections fronted by HTML forms and CGI scripts. Such collections include search engines such as Google, AltaVista, and thousands of other web-based searchable collections.

There are two major issues in creating a generic, configurable wrapper for such collections:

- How to convert a query into the proper CGI-BIN invocation onto a search engine; and
- How to interpret the HTML results returned by such an engine.

For metasearching, there is also the question of how to extract metadata from such engines, since most search engines do not provide any such metadata. Our current implementation relies on the wrapper administrator to write a metadata file (the `meta_attributes.xml` file) with the information specified by STARTS XML. In the future, we could automatically generate at least an approximation of the content summaries by using the results of research on metadata extraction from "uncooperative" sources [4, 15].

We decided that the best way to make this wrapper configurable without additional Java coding was through the use of XSLT stylesheets and the `starts_intermediate` format. We extended it to be able to describe CGI invocations

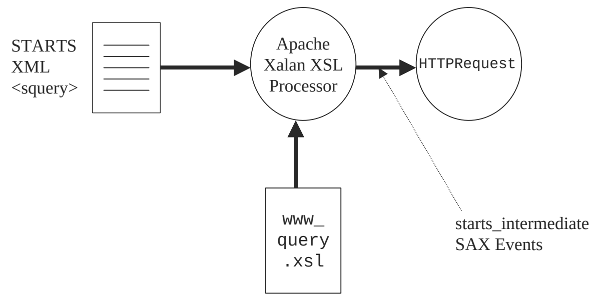

Figure 7: Converting a STARTS query to a CGI request using XSLT.

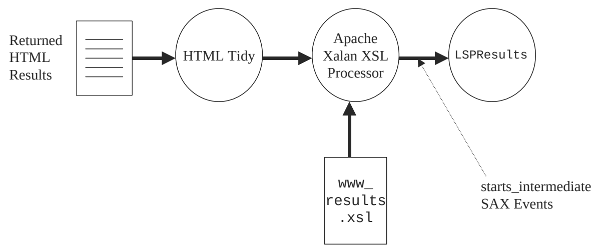

Figure 8: Converting HTML search engine results to a SDARTS object with XSLT.

using an element called `<script>`, which consists of a URL, a method (GET or POST), and a set of name/value pairs that are the script's parameters.

Figure 7 shows how the query is assembled. The administrator first writes an XSLT stylesheet called `www_query.xsl`. This stylesheet is used to map the transformation between a a STARTS `<squery>` and `starts_intermediate <script>`. The result of this transformation is output as SAX events, which ultimately build up a Java embodiment of the CGI invocation called `HTTPRequest`.

Figure 8 shows how the HTML results page returned by the wrapped search engine is parsed. While HTML is a close relative of XML, it is not as well-structured as XML. Hence, it cannot be truly classified as XML, and thus cannot be processed by XSLT. Thus we must first pass it through the HTML Tidy utility [14], which converts the page into well-formed XML (similar but not identical to XHTML). This XML is then transformed using an XSLT stylesheet called `www_results.xsl`, once again written by the administrator. The stylesheet maps the transformation between the XML-formatted results and a `starts_intermediate` result set. Once again, the transformation is output as SAX events, which ultimately build up an `LSPResults` object, which is the standard `LSPObject` subclass returned by a `BackEndLSP` in response to a search query.

Our wrapper is designed to work with search engines that return HTML pages of result records that may have a "more" or "next" button located on them, designed to retrieve further results. This is typical of most, if not all, web-based search engines. The stylesheet can also be written to detect such a button on the search results page, and convert it into a `starts_intermediate <script>` element. The wrapper will then understand that there are more result pages, and will invoke the `<script>` as an additional CGI call. This allows the wrapper to automatically page through a complete set of results from a search engine query.

7. FURTHER DISCUSSION AND CONCLUSIONS

In this paper we have described SDARTS, a protocol that is an instantiation of the SDLIP protocol with metasearch-specific elements from the STARTS protocol. We also reported on a toolkit with wrappers for a variety of heterogeneous collections. All the details (including the source code and the complete documentation of the various SDARTS wrappers that we have implemented) are publicly available at `http://www.cs.columbia.edu/~dli2test/`.

The reference SDARTS wrappers that we have implemented so far are meant to be customized for new collections. To build a wrapper for a new local text collection residing in a local file system, it is straightforward to write the required `doc_config.xml` file (Figure 5). In contrast, we have found that building wrappers for web-based collections and for local XML document collections by writing XSLT stylesheets can sometimes be more involved. XSLT turns out to be quite difficult to master. It is declarative, template-driven, and rule-based, which makes it quite different from the procedural and object-oriented languages most programmers are used to. In addition, writing these stylesheets requires a wide sampling of possible input documents. Many test searches on a web-based collection, for example, should be performed and saved before a wrapper administrator can write a `www_results.xsl` file for it.

The challenge of writing any configuration-driven system is to make it configurable but not so overly complex that writing the necessary configuration files is equivalent to writing a new piece of software. We think that using XSLT in our wrappers meets this challenge, although this is still not a perfect solution. In the end, in our opinion it is still easier than writing and re-writing Java code for parsing HTML or XML, extracting results, and formatting them. Furthermore, it would be easier to create a tool that could generate XSLT stylesheets than it would be to create one that could generate the Java code embodying the same transformation logic. XSLT is an area of active research and development, with a new generation of automatic stylesheet generators, many of them open source, already on the horizon. Therefore, we hope that future versions of our system might include GUI-based wrapper development tools that could simplify the generation of the configuration files.

8. ACKNOWLEDGEMENTS

We thank Sergey Melnik and Andreas Paepcke, from the Stanford Digital Libraries project, for their invaluable help and comments. We also thank Sergey Sigelman and Mike Medric for their contributions to the development and debugging of SDARTS code. Panagiotis Ipeirotis is partially supported by Empeirikeio Foundation and he thanks the Trustees of Empeirikeio Foundation for their support.

This material is based upon work supported by the National Science Foundation under Grant No. IIS-98-17434. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

9. REFERENCES

[1] International Standard Maintenance Agency. *Z39.50 Maintenance Agency Page*. Accessible at `http://www.loc.gov/z3950/agency/`. ISMA, 2000.

[2] C. Blake and C. Merz. University of California at Irvine repository of machine learning databases. Accessible at `http://kdd.ics.uci.edu/`.

[3] C. M. Bowman, P. B. Danzig, D. R. Hardy, U. Manber, and M. F. Schwartz. Harvest: A scalable, customizable discovery and access system. Technical Report CU-CS-732-94, Department of Computer Science, University of Colorado-Boulder, Aug. 1994.

[4] J. P. Callan, M. Connell, and A. Du. Automatic discovery of language models for text databases. In *SIGMOD 1999, Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 479–490. ACM Press, 1999.

[5] J. P. Callan, Z. Lu, and W. B. Croft. Searching distributed collections with inference networks. In *SIGIR'95, Proceedings of the 18th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 21–28. ACM Press, 1995.

[6] J. Calmet, S. Jekutsch, and J. Schü. A generic query-translation framework for a mediator architecture. In *Proceedings of the Thirteenth International Conference on Data Engineering*, pages 434–443. IEEE Computer Society, 1997.

[7] C.-C. K. Chang and H. Garcia-Molina. Mind your vocabulary: Query mapping across heterogeneous information sources. In *SIGMOD 1999, Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 335–346. ACM Press, 1999.

[8] C.-C. K. Chang and H. Garcia-Molina. Approximate query translation across heterogeneous information sources. In *VLDB 2000, Proceedings of the 26th International Conference on Very Large Data Bases*, pages 566–577. Morgan Kaufmann, 2000.

[9] E. Christian. Application profile for the government information locator service GILS, Version 2, Aug. 1997. Accessible at `http://www.usgs.gov/gils/prof_v2.html`.

[10] N. Craswell, P. Bailey, and D. Hawking. Server selection on the World Wide Web. In *Proceedings of the Fifth ACM Conference on Digital Libraries*, pages 37–46. ACM, 2000.

[11] L. Gravano, C.-C. K. Chang, H. García-Molina, and A. Paepcke. *STARTS*: Stanford Proposal for Internet Meta-Searching. In *SIGMOD 1997, Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 207–218. ACM Press, 1997.

[12] L. Gravano, H. García-Molina, and A. Tomasic. *GlOSS*: Text-source discovery over the Internet. *ACM Transactions on Database Systems*, 24(2):229–264, June 1999.

[13] D. Hawking and P. B. Thistlewaite. Methods for information server selection. *ACM Transactions on Information Systems*, 17(1):40–76, Jan. 1999.

[14] HTML Tidy. Accessible at `http://www.w3.org/People/Raggett/tidy/`, 2000.

[15] P. G. Ipeirotis, L. Gravano, and M. Sahami. Probe, count, and classify: Categorizing hidden-web databases. In *SIGMOD 2001, Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2001.

[16] H. Liefke and D. Suciu. XMILL: An efficient compressor for XML data. In *SIGMOD 2000, Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 153–164. ACM, 2000.

[17] The Lucene Search Engine. Accessible at `http://www.lucene.com/`, 2000.

[18] W. Meng, K.-L. Liu, C. T. Yu, X. Wang, Y. Chang, and N. Rishe. Determining text databases to search in the Internet. In *VLDB'98, Proceedings of the 24th International Conference on Very Large Data Bases*, pages 14–25. Morgan Kaufmann, 1998.

[19] Open Archives Initiative. Accessible at `http://www.openarchives.org/`, 2000.

[20] A. Paepcke, R. Brandriff, G. Janee, R. Larson, B. Ludaescher, S. Melnik, and S. Raghavan. Search middleware and the Simple Digital Library Interoperability Protocol. *D-Lib Magazine*, 6(3), 2000.

[21] A. Sugiura and O. Etzioni. Query routing for web search engines: Architecture and experiments. In *Proceedings of the Ninth International World-Wide Web Conference*. Foretec Seminars, Inc., 2000.

[22] E. M. Voorhees, N. K. Gupta, and B. Johnson-Laird. The collection fusion problem. In *Overview of the Third Text REtrieval Conference (TREC-3)*, pages 95–104. Department of Commerce, National Institute of Standards and Technology, Mar. 1995.

[23] S. Weibel, J. Godby, E. Miller, and R. Daniel Jr. OCLC/NCSA metadata workshop report, 1995. Accessible at `http://www.oclc.org:5047/oclc/-research/publications/weibel/metadata/-dublin_core_report.html`.

[24] J. Xu and J. P. Callan. Effective retrieval with distributed collections. In *SIGIR'98: Proceedings of the 21st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 112–120. ACM Press, 1998.