The design of Greenstone 3: An agent based dynamic digital library

Katherine J. Don, David Bainbridge, and Ian H. Witten
Department of Computer Science
University of Waikato
Hamilton, New Zealand
{kjdon,davidb,ihw}@cs.waikato.ac.nz

1. INTRODUCTION

Developed over the last five years, the Greenstone open source digital library software enjoys considerable success and is widely used [11, 13]. Its strengths include international language support and a flexible document importing process that handles different formats—HTML, Word, PDF, PostScript, and E-mail messages, to name but a few. Images, video and audio require accompanying textual metadata. Each digital library collection is designed individually and configured at "build time" when the collection is formed. Runtime flexibility is more limited. Beyond minor adjustments to layout and formatting, modifications—for instance altering or extending functional behavior—require editing (and recompiling) the source code.

Perhaps the most striking feature of the digital library field, at least from an academic point of view, is the irreconcilable tension between two extremes: the very rapid pace of technological change and the very long-term view that libraries must take. The dizzying rate of change in the core technology presents a great practical problem when conducting research in the field—particularly if it is to be useful! Digital libraries must reconcile aspirations to surf the leading edge of technology with the literally static ideal of archiving material "for ever and a day." The existing Greenstone system is a mature, widely-used technology that runs reliably even on obsolete computer systems that prevail in developing countries (e.g. Greenstone CD-ROMs run under Windows 3.1!), can be installed by people without any technical experience other than a minimal level of computer literacy, and can be used to build new collections by librarians without any computer specialization (e.g. UNESCO has run Greenstone courses for librarians in India, and is planning one in Africa). It would not be in this state today had it not been designed several years ago, long before many of today's technologies—particularly XML and the spate of accompanying standards—had been developed to a state where they were usable.

This paper describes a new design aimed at improving the dynamic nature of the Greenstone toolkit, in terms of how it organizes content and how it provides services, whilst simultaneously lowering the overhead incurred by collection developers when accessing this flexibility. The design is based on contemporary standards such as XML (and, in particular, XSL Transforms), contemporary implementation methodologies such as communicating agents, contemporary software practices such as simple protocols (like SOAP), contemporary cross-platform development strategies (Java), contemporary schemes for modularization and dynamic software update. Most important of all, the new design is informed by our experience with the current Greenstone system and the problems and challenges faced by real users, real collection developers, real librarians.

The structure of the paper is as follows. First the requirements for the digital library software are established. Next we review related software architectures, which exhibit a surprising range of implementation strategies and technologies deployed. These stem, in part, from the different aims of the various projects and the developers' different conceptions of what a "digital library" is. Nevertheless, common themes recur which we draw out in the context of our own requirements. We proceed to describe an agent-based software architecture called Greenstone3 that meets the needs that have been identified, highlighting similarities and differences from other solutions. Fundamental to the approach is the use of XML throughout for data representation, combined with XSL Transforms (XSLT) to provide a flexible mechanism for adjusting the functionality of the runtime system without having to modify and recompile the source code. To promote cross-platform independence and facilitate the dynamic loading of agents' objects, the implementation uses Java. Following this we describe an initial configuration of the system that uses three server-side agents to provide backward compatibility with collections built using the previous version of Greenstone, and go on to demonstrate some of the dynamic features of the system.

2. REQUIREMENTS

Over the years the Greenstone digital library software has been employed by many users internationally to develop a wide variety of digital library systems. An early application, and still a major one, is collections of humanitarian information in the form of CD-ROMs that run on any Windows computer and provide a web browser interface to a collection stored locally—and also serve that information over any network to which the computer is connected, such as a local intranet or the Internet. (There is no restriction to Windows—Greenstone is developed on Linux—except that the prepackaged CD-ROMs self-install easily on Windows systems.) There are now over twenty of these collections, and they have been distributed widely, particularly in developing countries [14]. Beyond this, many other styles of collection have been developed under Greenstone, including the following.

- Countless easy-to-form personal collections based around an individual's on-line content in their local file system—E-mail, photographs, Word, PDF, PowerPoint, etc.
- A large-scale bibliographic catalog of the BBC radio and TV archives, structured as several subcollections with seamless cross-collection searching, in daily use by the BBC.
- A kids' digital library that provides individual pupil workspaces and a teacher mode for privileged actions such as adding pupils' own stories to the repository and updating bulletin boards.
- Mixed media collections such as a local public library oral history project (streaming audio, text and images), a national collection of newspapers in the Maori language (page facsimiles, electronic text, and commentaries), a collection of MTV clips (video and text), and several music collections that support direct content-based music queries for particular melodic fragments ("query by humming").

Experimental interfaces include a Venn diagram tool for graphically formulating Boolean queries, and a bibliographic visualization tool that plots matching citations on an x-y grid based on publication year and ranked relevance score to the query terms [1]. These make use of a CORBA-based protocol to support distributed client-server interaction.

Some of these experimental variants, while perfectly functional, are implemented inelegantly, exposing limitations caused by certain aspects of the design. A notable deficiency is the immutable nature of the index files generated during the building process, which makes incremental adjustments to a collection expensive. Another is the low level of functional customization supported by the runtime system—although minor presentation tweaks are easy, more extensive changes involve modifying and recompiling the source code.

This led to the following requirements for an improved design.

Backwards compatibility. Naturally we wish to retain the existing system's strengths. This is accomplished by ensuring that the new design is backward compatible, which has the added benefit of providing existing developers and users with an easy migration path.

Levels of customization. To match the different categories of people involved in constructing digital libraries—e.g. content developer, collection editor, workflow designer, software developer—different levels of customization are required. For instance, a content developer may wish to include source documents in a new format; a collection editor may seek influence over diverse issues of presentation; a workflow designer may want the system to function in a novel way. Ideally, the work required to accomplish such adjustments should be compatible with the skill base associated with each job category.

Software modularity. To facilitate development and long-term management of the software, code modularization—a mantra of any software engineering approach—is essential. This is promoted by adopting off-the-shelf technology such as a database system, indexing tools, and page rendering software; and by the use of standards.

Service based. Basing a digital library around a set of services is another way to accomplish modularity—in this case, modularity of function.

Distributed architecture. A rich digital library infrastructure can only be supported by a distributed architecture, and the addition of an open protocol helps to foster interoperability. Running a digital library on a single host machine becomes a trivial degenerate case.

Future compatibility. Libraries are long-term institutions with a mandate for preservation, and it is essential that old collections can be presented by future versions of the system. This is a more ambitious requirement than mere extensibility which, although an admirable quality in any design, does not necessarily ensure that future versions can safely interact with current ones.

Dynamic. Many aspects of the library should be dynamic. This is an umbrella term covering both *dynamic content*, whereby documents and metadata can be added, revised and removed while a repository remains on-line, and *dynamic configuration*, permitting presentational issues to be adjusted and services to be added at runtime.

Integrated documentation. Large-scale software systems such as digital libraries benefit immensely from the use of an integrated documentation system.

Self-describing modules. This goes a stage further than the previous item: modules must describe themselves in a machine-readable format so that other modules can interact with them without the need for explicit control.

Computer environment integration. A digital library but should mesh well into a user's existing computer environment. Full integration makes a digital library become a seamless component of each user's work environment.

Generalized abstraction. Viewed in the abstract, documents and metadata in a digital library are resources maintained by a content management system. By generalizing the design to this level, the utility of the tool extends beyond the traditional boundaries of a digital library and further enhances the role it plays in a user's on-line work environment.

3. RELATED WORK

Recent years have seen a steep rise in the number of digital library solutions available, making it impractical to review them all here. Instead we concentrate on reviewing pertinent strategies that have been successfully deployed. We divide projects into two groups: open source systems, and academic research efforts. A third category, proprietary systems, is not discussed because insufficient design information is available.

All approaches below use off-the-shelf technology to implement significant parts of the system, in particular a database (typically relational) to store and retrieve metadata, and an indexing tool if full-text searching of document content is to be provided. A web server capable of running CGI scripts is another widely-used component, because the most prevalent form of digital library user interface acts through a web browser. These components are bound together by custom-written software that implements the workflow constituting the digital library system.

Table 1 lists four open source digital library systems, the last being our own, and summarizes the implementation platforms, principal implementation technologies, and component standards.

EPrints is designed for archiving on-line reports, particularly academic research papers. Indexing and a subject hierarchy are based around a relational database of bibliographic metadata that a site administrator can adjust to suit the sort of reports being stored. Source documents are linked to their corresponding bibliographic record so that they can accessed at runtime as a side-product of record retrieval. Through a network of web pages, end-users (who are often authors of papers in the archive) and the administrator participate in a workflow cycle of submission, optional editing control, and deposit—thereby accomplishing the developers' stated aim of a "self-archiving" methodology.

Using exactly the same implementation technology, Koha[1] focuses on a different community—public librarians and their users. This results in a system whose services are strikingly different to EPrints but whose underlying techniques are similar. In addition to bibliographic information (expressed in MARC format) in the relational database, recorded items are extended to include membership, reading list, acquisition, and budget information, amongst other things. However, the same basic arrangement of web forms is used to support a workflow of submission and editing (password protected if required). From the user's perspective, Koha is like the graphical, web-based OPAC systems one has come to associate with public libraries.

DSpace, a combined venture by MIT and Hewlett Packard, is a software tool aimed at the digital content needs of institutions. Strongly influenced by the the Open Archival Information Systems (OAIS) reference model [6], it supports submission, searching, browsing and retrieval. Searching includes full-text retrieval, which, through the use of Apache's Lucene indexing tool, allows for incremental updates. Considerable design effort has been placed on the submission workflow.

We now move on to describe related digital library research projects.

The University of Michigan's Digital Library project UMDL has developed an agent based architecture, strongly founded on artificial intelligence techniques, to accomplish a wide ranging generality [10]. Written in C++, agents reason—using a frame-inheritance structure—about services and content provided by other agents. Messages between agents are passed over CORBA using the Knowledge Query and Manipulation Language (KQML). A "market economy" analogy is used to help coordinate the interaction of agents through buying and selling services.

The Dienst software is produced by a long-running digital library research initiative at Cornell University [4]. It defines a rich set of services through a protocol that supports distributed source repositories. Services include document submission, indexing, querying, information gathering, and user registration. Written in Perl, it incorporates a sophisticated document model that permits logical structuring of documents and multiple forms. Its influence can be seen in the Open Archives Initiative [5].

The Simple Digital Library Interoperability Protocol (SDLIP) from Stanford University has interfaces that provide four different services: searching, accessing results, metadata and delivery [8]. Several digital library systems use this protocol, and, in addition, gateways to Dienst and Z39.50 have been developed. The design has a simple API that devolves much responsibility concerning parameter details to an XML encoding. This enhances the flexibility of the parameter passing mechanism. For instance, parameters can be optional, and new ones can be added at a future date without having to alter the API.

Carnegie Mellon University's Informedia project is one of the leading digital library systems for video content [9]. Indexing is accomplished through a mixture of speech recognition processing, image similarity matching, and video optical character recognition. A recent development is to read a document's metadata out of the database in XML form, and then apply style sheet transforms (XSLT) to finalize presentation within a web browser interface [2]. This allows the interface's appearance to be tailored without modifying the runtime system's source code.

3.1 Discussion

It is encouraging to see a considerable intersection between our requirements and the abilities and features of the various projects reviewed above, which implies a shared notion of what factors are important. Yet it is also interesting to observe how differently these projects manifest themselves in their implementation. We do not offer any reasoned ex-

[1] The name is a Maori word meaning gift or donation, chosen to reflect the open source nature of this project.

Table 1: Implementation summary of sample open source digital library systems.

System	Platform	Technology	Supported standards
www.EPrints.org	Unix	mySQL, PERL	Open Archives
www.Koha.org	Unix	mySQL, PERL	Z39.50, MARC
www.DSpace.org	Java; runs on Unix	postgreSQL, Lucene (indexing)	Open Archives
www.greenstone.org	Unix, Windows, MacOS X	GDBM, MG (indexing), PERL (building), C++ (runtime)	Z39.50, Open Archives

planation for this, except to suggest that it is perhaps driven in part by social factors—both within a development team and in their perception of the intended users.

Common to all projects is a segmented service-based approach to providing the required functionality. At the simplest level this involves supplying different CGI scripts for each core ability, encapsulating a primitive notion of segmented services. Given the range of applications and domains the projects cover, segmentation is clearly a successful strategy.

Generality of the system is perhaps the most important factor in our design. Most designs present a fixed set of services and are therefore constrained in the sort of functionality they can provide. In Dienst, a representative example that has matured over several years of development, the services, though fixed, permits a wide range of applications. For instance, it has been used in the *Making of America* project, where documents are predominantly image-based and browsing is a strong feature of the interface, and in NCSTRL, which in contrast features full-text indexing of a federated network of technical report repositories.

At the other extreme is UMDL, whose architecture is so general that it would not be out of place as an implementation technology for the semantic web. There are, however, factors to be wary of that stem from this generality. For instance, the authors themselves report on how difficult it is to design the ontologies necessary to support agent reasoning [10, 3].

Clearly, the two extremes have advantages and drawbacks. Dienst's more contained approach, with a fixed set of services, makes it easier for content providers to establish a site and make minor adjustments to their taste. However, to go much beyond that would be a major undertaking requiring considerable programming commitment.

Our requirement of generality precludes an approach that enforces a fixed set of services. Nor do we believe that the UMDL approach is conducive to a wide-ranging community of developers who participate in the extension and augmentation of service functionality. Moreover, while we acknowledge that some of these approaches allow customization, none cater for different levels of customization. What we require is an approach that lies somewhere between the two design extremes and also satisfies our requirement for different customization levels. This is is accomplished by building upon some of the ideas that appear in the Informedia and SDLIP projects, as detailed in the next section.

4. ARCHITECTURE

We now describe the new software architecture, called Greenstone3. It has evolved from the requirements above, our own experience of developing digital libraries systems, and from studying other open source software and research projects. We decided that the best way to meet the challenges posed by the list of requirements was to reimplement Greenstone using an agent topology. For portability reasons Java was chosen as the implementation language, and XML-encoded messages were used to express all communication between agents.

At the highest level, this description is strongly reminiscent of UMDL. However, as we shall see, our requirements have led us to a significantly different infrastructure, and as soon as we delve into detail all similarity disappears. In particular, the role that we have developed for the agent component is less ambitious, and favors modules with pre-programmed functionality.

A typical basic Greenstone3 digital library system is made up of a "back end," which we call a digital library *site*, coupled to a "front end" that provides the user interface, known as the *receptionist*. A simple stand-alone example is shown in Figure 1. The Receptionist's point of contact with the server is the MessageRouter (MR) agent—all communication with the site occurs through this module.

The digital library back end in Figure 1 contains two collections, *demo* and *fao*, and a cluster of collection-formation services. All functions that the digital library provides are called "services." For example, *AddDocument* is a service that adds a document to a collection; *ImportCollection* imports into the system all documents associated with a collection, converting them where necessary from their original form; *BuildCollection* builds all indexes and browsing structures that are associated with a collection; *ActivateCollection* makes a newly-built collection active, so that it can be seen by digital library users. These particular services are related: they are all concerned with creating a digital library collection. Related services may be grouped together into a "service cluster" also managed as an agent, thereby establishing a hierarchy to services if desired. In Figure 1 the services just listed are accessed through the *Collection-Formation* ServiceCluster module.

As far as the digital library user is concerned, a "collection" is a focused group of documents with a uniform means of ac-

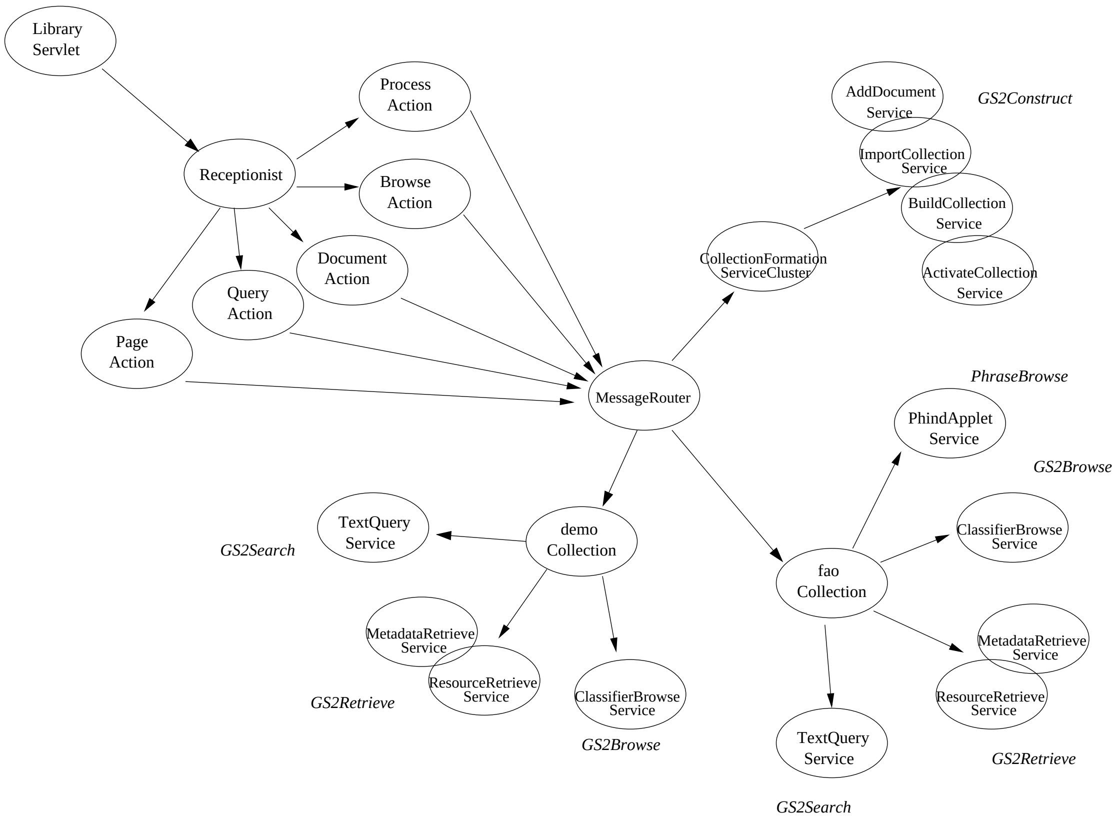

Figure 1: A simple stand-alone site.

cess. For the system, it is a service cluster that groups a set of services that are related by the set of data they work on. For example, the *demo* collection towards the lower left of Figure 1 contains three agents providing four services to the user. The agents are called called "Search," which provides a *TextQuery* service, "Retrieve," which provides a document retrieval service *ResourceRetrieve* and a metadata retrieval service *MetadataRetrieve*, and "Browse," which provides a metadata browsing service *ClassifierBrowse*. There may well be different agents that provide the same services in different ways; these particular ones provide searching and browsing for Greenstone2 ("GS2") collections.

The Web-based front end at the upper left of Figure 1 centers around the Receptionist, which is the point of contact for the interface generator. A servlet (labeled "Library Servlet") takes HTTP commands in the form of URLs and arguments and translates them into XML for the Receptionist. The Receptionist is capable of executing various different actions, each of which involve one or (usually) many calls to the digital library's MessageRouter (center of Figure 1). Figure 3 shows a screenshot of a sample action, the "about this collection" action.

The configuration in Figure 1 is a very simple example of a digital library structure. In practice, there may be many digital library sites, possibly involving distributed computers. Each site has a structure similar to the back end in Figure 1 (that is, all but the upper left-hand quadrant). Different sites may know about each other and can gain access to each other's collections by forwarding requests. There can also be different user interfaces to the library. Figure 1 shows a simple web-based interface, but there may be others, ranging from applets that display documents in different ways to alert services that recognize when new information becomes available in one of the collections and formulate appropriate E-mail to users. Although in the simplest case the front and back ends (receptionist and site server) are compiled together into a single executable process, MessageRouters have the capability to communicate across a distributed network with other MessageRouters and with Receptionists, using a defined protocol. The following subsections elaborate on this structure.

4.1 Modular structure

The new architecture utilizes independent modules called "agents" that communicate via a single method call:

$$\mathrm{XML}_{out} = \mathrm{process}(\mathrm{XML}_{in}).$$

Both input and output are expressed in XML. This decision shifts attention from the design of an Applications Programming Interface (API) to the design of XML forms that encode the equivalent information. The advantage is modularization: the XML specifications can be modified locally and communication will proceed effectively according to the new scheme provided only that all affected modules are altered appropriately. Moreover, a degree of robustness to future changes is accomplished by adopting a "defensive" programming approach to the way in which parameters are sought. In contrast, in a system where future changes require altering the API, all modules—some of which may be on remote machines—usually have to be recompiled to reflect the update.

Modules are conceptualized as agents that have, or have access to, certain functionality. An agent may respond to a message by processing it itself, forwarding it to another module, or a combination of the two.

If agents are on different computers, communication normally takes place using SOAP (Simple Object Access Protocol) although other protocols are possible. Figure 2 shows a Greenstone system whose local site has no collections or services of its own. Instead, the MessageRouter (MessageRouter 1 in the diagram) talks to two other sites using SOAP. The local MessageRouter (#1) has two Communicator modules that enable it to make SOAP requests; the remote sites (#2 and #3) each have a SOAP server that listens for such requests and fulfills them.

A potential downside of expressing the programming interface structure in XML is execution efficiency. The input (XML_{in}) and output (XML_{out}) of each agent call can be either a serialized String representation, which is the primary representation method, or a Document Object Model (DOM), which is a tree that represents the parsed XML string. The serialized String requires parsing every time it is transmitted to an agent, and is therefore more expensive than the DOM approach, which effectively caches the parsing information in the tree structure; the latter, however, cannot be natively transmitted between agents over a distributed network.

4.2 Dynamic configurability

Digital libraries need to be dynamic. Administrators will routinely want to add new collections, or new user interfaces, or completely new kinds of service, to a running digital library without having to bring it down and restart it.

The digital library back end is built around a central MessageRouter module that provides a way of gaining access to any collection or service. When new collections come up, they can be registered with the MessageRouter in order to make themselves visible throughout the system. Alternatively, the MessageRouter can periodically poll the site to discover new collections. All requests that users make are passed to the MessageRouter, which forwards them to the appropriate agent for processing. Requests are synchronous; the requesting process is blocked until the result is received.

The most basic request, which any agent must respond to, is "describe-yourself" (in fact, the ability to respond to "describe-yourself" is really what defines an "agent" in our system.) The MessageRouter, for example, responds to this request with an XML document which specifies the collections that are available locally, and the other Greenstone sites that it knows about (whose collections may also be listed). Its response also describes any service clusters or single services that are provided by the MessageRouter itself—for example, cross-collection searching or collection formation capabilities. As a second example, a collection agent, on receipt of a "describe-yourself" message, returns collection-specific metadata and a list of services that the collection provides.

A plain "describe-yourself" request will return a complete description. Through the XML request syntax, a qualifier can be added to the request which asks for a particular facet of the complete description, thereby achieving communication economy.

Using these facilities, a user interface agent can ask a MessageRouter for a list of local collections, remote sites and their collections, and then ask each collection for a list of the services it provides. The XML documents containing this information could be amalgamated and presented to the user as an XML form that actually implements the services that are represented.

4.3 Interacting with the user

The MessageRouter, together with the services to which it gives access, forms the core of the digital library system. Clients can be written that call in a variety of ways upon the services that are provided.

An important form of client is one that implements user interaction through a Web browser, which—as remarked above—is the standard way of communicating with the digital library. The user makes a request by clicking a URL or submitting a Web form. This request is intercepted by a servlet that invokes the Receptionist module. The Receptionist represents the user's normal point of contact with the system: based on the input, it creates XML messages which it passes into the digital library system through the MessageRouter. The responses are gathered together and translated into the form of a Web page for presentation to the user.

The Receptionist receives from the servlet an XML representation of the arguments in the URL. One of these arguments is the Action, which, along with the Subaction argument determines what information must be requested from the MessageRouter to fulfill the request. Table 2 shows the list of actions needed to provide a Greenstone3 configuration that is backwards compatible with the previous version of Greenstone, namely Greenstone2.

The Receptionist includes a Java class for each action. These classes know nothing about the collections, services, or other sites that are available in the digital library system. Instead, they decode the arguments in the URL to determine what information must be requested, and send it through the MessageRouter. A single action often generates several different requests. For example, to generate the standard Greenstone home page, the PageAction must query the MessageRouter for a list of its collections. Then, for each one, collection metadata such as its icon and title must be retrieved. The XML results returned by these requests are composed together into one large XML tree, to which is appended system configuration and translation information. The resulting structure is converted, using XSLT files appropriate to that particular action, to an HTML page for presentation to the user.

It is not necessary to use HTML to interact with the Recep-

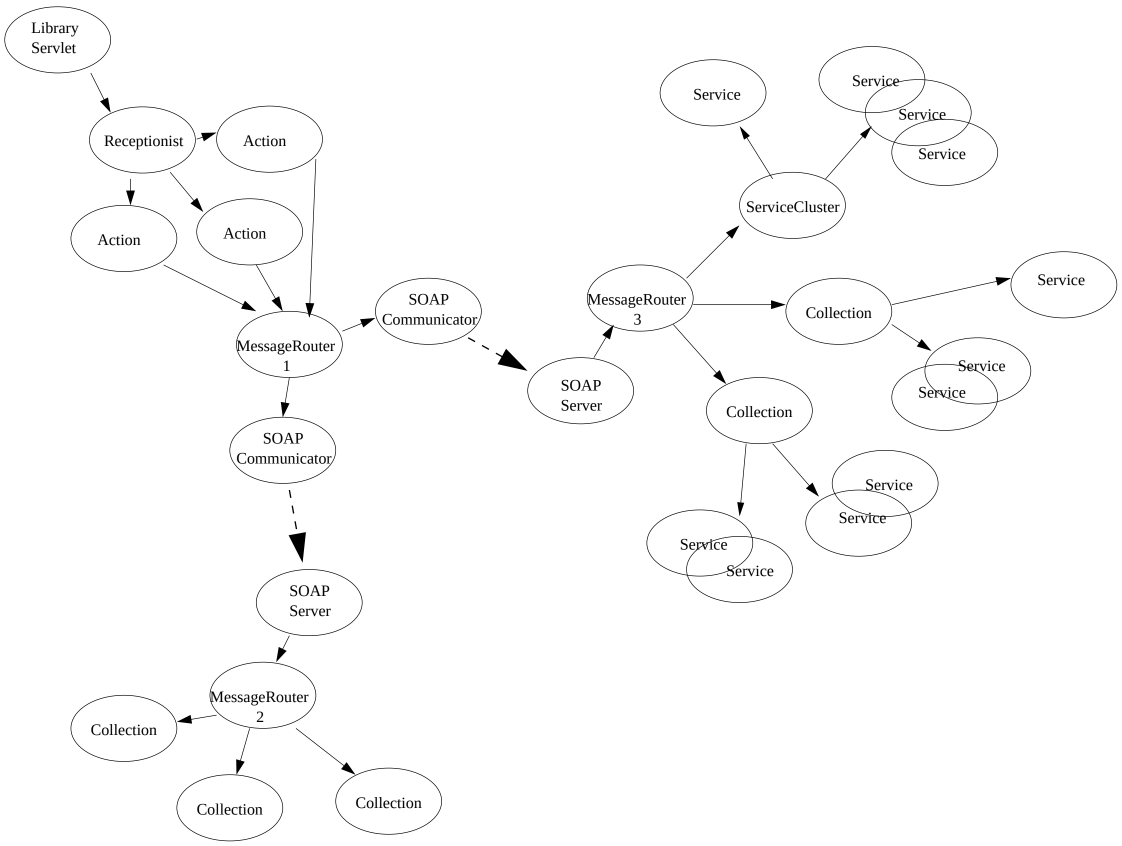

Figure 2: A more complex digital library configuration distributed over several servers.

tionist, and other clients use different means of interaction. An output type specifier is included in each Receptionist request and, using XSLT modes, different output formats may be generated—such as XML or WML.

4.4 Data in the system

Data in the digital library system is divided into *documents* and *resources*. The former represent primary data expressed in XML upon which indexing and browsing is based, while the latter represent supporting data such as an audio recording from which a textual transcript has been obtained.

For example, a book that has been added to a collection is represented by an XML document. The document contains metadata associated with the book, for instance title, author and publisher. XLinks are used to associate resources or other documents. Any images embedded in the book are expressed as resources belonging to that document. The original representation of the book, such as a PDF file, would also be an associated resource. As well as resources, other documents may be associated with the book—such as a translation of the same book into a different language. This translation is a document in its own right, and the original document links to it.

Documents are indexed, but resources are not. This means that it is documents that are discovered through searching and browsing. Resources, on the other hand, can only be found via the containing document. Both can be retrieved through the XML communication mechanism.

The metadata and/or content of the document need not be stored with the document identifier—it may, for example, be represented implicitly in a large file that contains a compressed representation of all documents. However, it is necessary that knowledge of the document's identifier is sufficient to reconstruct the complete document. In Greenstone2, the equivalent information is shared between the compressed text files and a database that contains metadata (expressed using GDBM, that is, the Gnu Database Manager[2]).

Documents are not restricted to textual articles, books, and other text files. A collection could contain images, in which case each image would have a corresponding document whose content pointed to the image file. A document could be a sequence of other documents, such as a PowerPoint show of individual slides.

5. SAMPLE AGENT NETWORK

As an initial test of this software architecture we have developed a set of services that provide backward compatibility to collections built using the old system, Greenstone2. These are provided by the three agents shown for the *demo* collection in Figure 1, called "GS2Search," "GS2Retrieve," and "GS2Browse" (which together provide querying, retrieval,

[2]www.gnu.org

Table 2: Actions in the new architecture that implement backwards compatibility.

Action	Subaction	Description
p (page)	home	Display the home page for a site
	about	Display the "About" page for a collection or service cluster
q (query)	Text	Simple text query
	Field	Form query
d (document)		Display a document
a (applet)	d	Display an applet
	r	Process an applet's request to its service (pass the response back to the applet, without formatting)
b (browse)	Classifier	Greenstone2-style metadata browsing
pr (process)	d	Display the form for a command type (asynchronous) service, e.g. import a collection
	r	Send a request to start the service
	s	Send a status request to see how the service is progressing

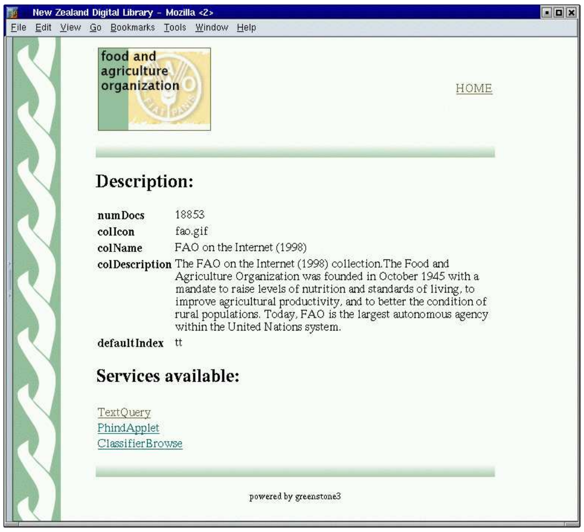

Figure 3: Top level page to the *fao* collection.

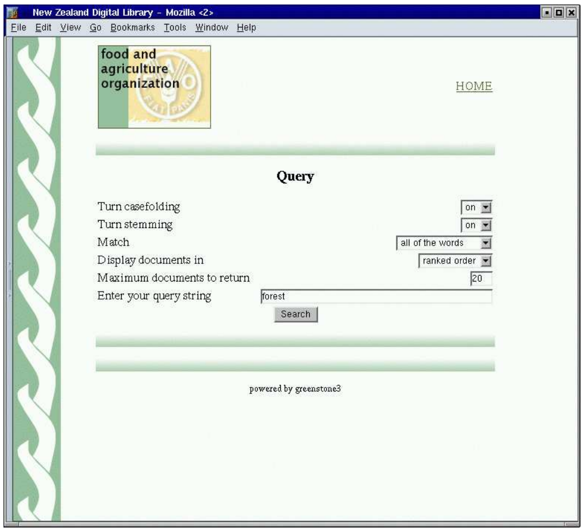

Figure 4: Querying the *fao* collection for texts on "forest" using the standard query page.

and browsing services), along with a fourth (shown at the upper right) called "GS2Construct," which provides a collection-building service for creating Greenstone2-compatible collections. Of course, the point of the architecture is that different agents can be constructed that implement the very same services in quite different ways.

Figure 3 shows the "about this collection" page for the FAO demonstration collection *fao*. This page is normally the first point of contact for someone accessing the collection. At this stage in the development, the page is rather stark: it displays some raw collection-level metadata and a hyperlinked list of the services that are available (calculated at runtime). A more polished version of the software would reformat the raw collection-level information (using XSLT) to present a more informative and graphically pleasing page, including the collection-specific policy statement governing how the content is selected [7].

In Figure 4 the user has clicked on the *TextQuery* link to access the main search page and entered the term "forest" as the query. Figure 5 shows the result of performing the query. The top half of the page reproduces the query and its settings so the user can modify it if desired, and the bottom half shows the ranked order of matching documents. Again, these figures are intended to show the basic functionality: it is anticipated that actual collections will present the raw information more attractively.

Figure 6 shows the result of accessing the *ClassifierBrowse* service from the "about this collection" page of a second demonstration collection, *demo*, and then browsing through the subject hierarchy to reach the point shown. The main task performed by this service is to access items stored in the GDBM database that Greenstone2 uses for collection metadata.

The indexing tool used to support searching in the demonstration collections is MG++, an extended C++ version of

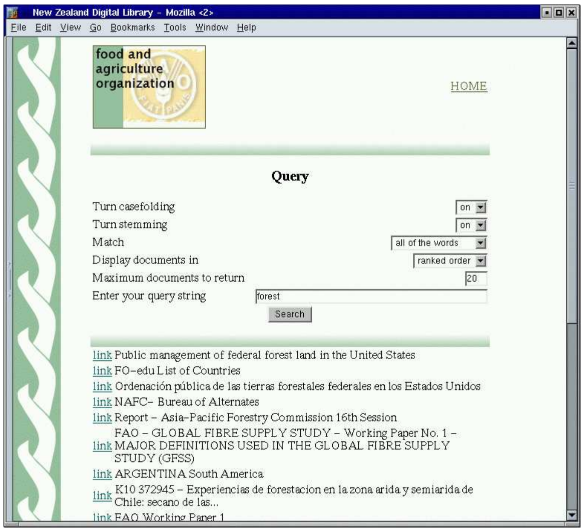

Figure 5: Documents in the *fao* collection that match the query for "forest".

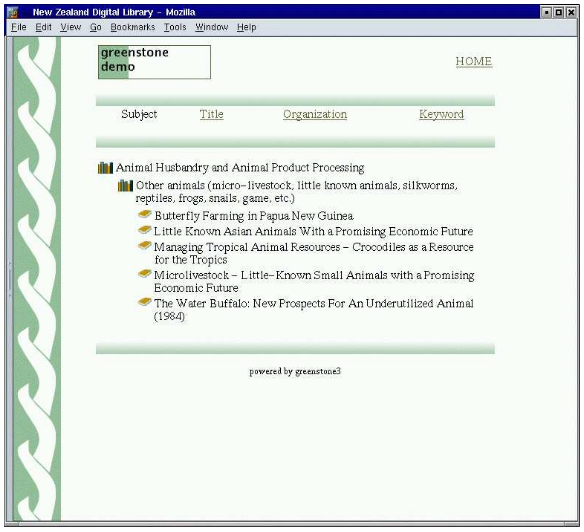

Figure 6: Browsing the Subject hierarchy in the *demo* collection.

the basic MG tool [12] that comes supplied with Greenstone. This has the capability to perform fielded searching—a favorite with librarians—however to be backwards compatible with MG, the basic query service does not include any options to access this functionality. To illustrate the dynamic configurability of Greenstone3, a new service, *AdvancedFieldQuery*, which does access this ability was developed; this was accomplished by writing a new Java agent for the purpose. For a collection to take advantage of this new service, its configuration file needs to be updated to specify this service. This was done for our demonstration collection, and through polling file timestamps, the MessageRouter detects this change, and reloads the collection, causing the new service to be dynamically loaded. If Figure 3 were to be reloaded *AdvancedFieldQuery* would now appear in the list of services. Clicking on the service link takes the user to the snapshot shown in Figure 7. The QueryAction and its associated XSLT are sufficiently general that they can handle the new query service. Local and remote clients can therefore present the new service without requiring modification or recompilation.

5.1 Levels of configurability

We now describe how the use of XSLTs allow for different levels of configurability within the system.

Control of presentational issues is primarily through the XSLTs associated with action agents, which effectively add style sheet information to a structured record. Through this mechanism properties such as header, footer, background, and typeface can be quickly customized to a collection editor's taste. Through more sophisticated XSL transforms, an editor can accomplish more fundamental changes that modify the structure of the generated page, for example adding XSLT that sorts the query results alphabetically by title rather than ranked score, or alternatively formats the results according to OAI syntax in preparation for exporting. Previously such changes required the editing of source code and the restarting of the digital library software after recompilation.

To take another example, a collection developer may desire a simpler variety of fielded query interface in addition to that shown in Figure 7. This can be accomplished with two XSLTs: the first controls the appearance of the Web page; the second gathers the input data from submitted web form and constructs the necessary message to be transmitted to the *AdvancedFieldQuery*. From there the system continues as usual with the returned result set handled in the normal way. We are also experimenting with XSLTs being transmitted with the message. This introduces, in a configurable manner, the ability for agents to influence other agents in the system.

6. CONCLUSIONS

This paper has presented a new digital library architecture which is radically different from the present Greenstone, but strongly rooted in past success. To meet a broad range of requirements an agent based topology is utilized to provide a flexible infrastructure. Written in Java to promote portability, dynamic loading of objects and internationalization, agents communicate by streaming XML message between each other. Using SOAP this communication can be across a distributed network. Common functionality to all agents is an ability to describe themselves in a machine readable form, and to apply an XSLT to transform messages. The latter is instrumental in providing different levels of configurability, an important ability given the different types of people involved in the life-cycle of a digital library.

The example implementation presented here provides backwards compatibility to collections built with the current version of the software. This satisfies another of the listed requirements, and helps minimize the migration path of existing developers and users. The list of requirements is ambitious, however, and not all of them have been proven yet. For example future compatibility needs more time to establish if the design successfully meets such a criteria. For

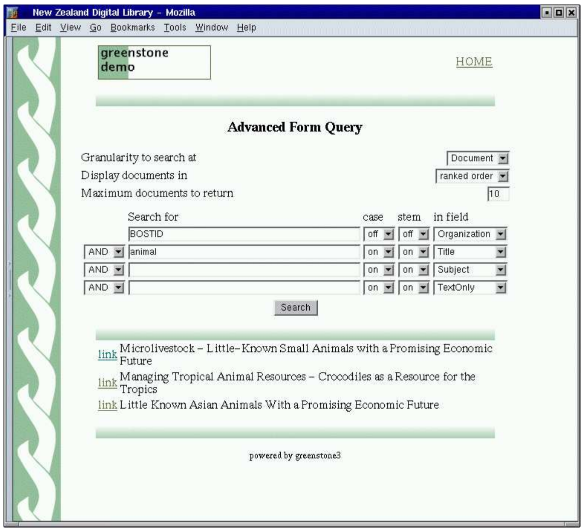

Figure 7: Advanced form query page for the *demo* collection.

computer work environment integration, while nothing in the sample implementation demonstrates this property, the design is capable of supporting the idea through specialized receptionists. Finally, the generality of the architecture to be applied to other content management tasks needs to be verified but this is a topic for another paper.

Acknowledgments

This work is supported by the New Zealand New Economy Research Fund.

7. REFERENCES

[1] D. Bainbridge, G. Buchanan, J. McPherson, S. Jones, M. Mahoui, and I. H. Witten. Greenstone: a platform for distributed digital library applications. In *Proceedings of the European Conference on Digital Libraries*, pages 137–148, Darmstadt, Germany, September 2001.

[2] M. Christel, B. Maher, and A. Begun. Xslt for tailored access to a digital video library. In *Proc. of the first ACM and IEEE joint conference on Digital Libraries*, pages 290–298, Roanoke, Virginia, USA, June 2001.

[3] E. Durfee, D. Kiskis, and W. Birmingham. The agent architecture of the university of michigan digital library. *IEE Software Engineering*, 144(1):61–7, Feb. 1997.

[4] C. Lagoze and D. Fielding. Defining collections in distributed digital libraries. *D-Lib Magazine*, 4(11), Nov. 1998.

[5] C. Lagoze and H. Van de Sompel. The open archives initiative: Building a low-barrier interoperability framework. In *Proc. of the first ACM and IEEE joint conference on Digital Libraries*, pages 54–62, Roanoke, Virginia, USA, June 2001.

[6] B. Lavoie. Meeting the challenges of digital preservation: The OAIS reference model. *OCLC Newsletter*, 243:26–30, Jan/Feb. 2000.

[7] M. Lesk. *Practical digital libraries: books, bytes and bucks.* Morgan Kaufmann, San Francisco, 1997.

[8] 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), Mar. 2000.

[9] H. Wectlar, M. Christel, Y. Gong, and A. Hauptmann. Lessons learned from building a terabyte digital video library. *IEEE Computer*, pages 66–72, Feb. 1999.

[10] P. Weinstein, W. Birmingham, and E. Durfee. Agent-based digital libraries: decentralization and coordination. *IEEE Communications Magazine*, pages 110–115, Jan. 1999.

[11] I. H. Witten, D. Bainbridge, and S. J. Boddie. Greenstone: Open-source DL software. *Communications of the ACM*, page 47, May 2001.

[12] I. H. Witten, T. C. Bell, and A. Moffat. *Managing Gigabytes.* Morgan Kaufmann, San Francisco, 2000.

[13] I. H. Witten, S. J. Boddie, D. Bainbridge, and R. J. McNab. Greenstone: a comprehensive open-source digital library software system. In *Proceedings of the fifth ACM conference on Digital libraries*, pages 113–121, Roanoke, Virginia, June 2000.

[14] I. H. Witten, M. Loots, M. Fernandez-Trujillo, and D. Bainbridge. The promise of digital libraries in developing countries. *The Electronic Library*, 20(1):7–13, 2002.