A Distributed Digital Library Architecture Incorporating Different Index Styles

Rodger J. McNab
Ian H. Witten
Stefan J. Boddie

Department of Computer Science,
The University of Waikato,
New Zealand
email {rjmcnab, ihw, sjboddie}@cs.waikato.ac.nz

Abstract

The New Zealand Digital Library offers several collections of information over the World Wide Web. Although full-text indexing is the primary access mechanism, musical collections can also be accessed through a novel melody retrieval system. In offering this service over a three-year period, we have had to face many practical challenges in building, maintaining, and administering diverse collections of different kinds of information, involving different search and retrieval systems, with different user interfaces.

This paper describes the design of the software we have built to support the service. Interface server programs provide a uniform interface between search engine and client, irrespective of the nature of the collection. Search engines that embody completely different index styles operate under a single distributed framework—we describe as examples MG, a full-text retrieval system, and MR, a melody retrieval system. A flexible protocol for communicating between an interface server and a search engine is defined. The resulting architecture simplifies library administration and the creation of new collections by providing a unified framework under which vastly different user interfaces and search engines can co-exist in a distributed computing environment.

1 Introduction

The New Zealand Digital Library (NZDL) is a freely-accessible facility that makes available a dozen or so quasi-independent collections of information over the World Wide Web (at http://www.nzdl.org)—many of which involve several Gbytes of on-line information [9]. Operated as a research project in the Computer Science Department at Waikato University in New Zealand since 1995, the system is continually evolving, and both the number and variety of collections offered are increasing rapidly. The collections are intended as independent demonstrations of digital library technology, rather than a unified library in their own right. We believe that this style of organization will become common in digital libraries of the future, because different kinds of material are most useful when indexed and made accessible in ways that take account of their individual nature.

As a digital library grows and the variety of different kinds of material in it expands, problems of administration and maintenance become increasingly severe. Most of the collections we currently maintain comprise text, the primary access mechanism being full-text retrieval [10]. However, some collections contain music, for which the primary access mechanism is a tune retrieval interface that allows a user to sing or hum a melody and have it (and others like it) located in the collection [4, 5]. Initially, special-purpose code was written to embed these search engines within a World Wide Web interface to the digital library as a whole; however, this imposed a significant administration and maintenance burden. This paper describes a uniform architecture that was developed to accommodate these two search and retrieval mechanisms, and the vastly different document types that they involve. This architecture is quite general and will be able to incorporate other retrieval mechanisms in the future—for example, an image collection accessed by content-based queries (for example [8]).

The structure of this paper is as follows. We begin by looking in detail at the two search and retrieval programs currently in use. We then identify the requirements for the architecture, and the dimensions of flexibility that it needs to have. We define what we mean by a *digital library*

collection and base our architecture on supporting this notion. Then we describe the design of the digital library system, which centers around three kinds of program—*builders*, *searchers*, and *interface programs*—and the *protocol* with which interface programs communicate with searchers. Finally, we review some other digital library architectures and discuss differences with the one that we have developed.

2 Example search and retrieval systems

We begin by examining the two search and retrieval systems that are currently used in the New Zealand Digital Library: MG, a full text retrieval system, that is used to index ordinary textual collections; and MR, a music retrieval system, to search collections of melodies. Although currently only these two are in use, our architecture is designed to be general enough to accommodate other search and retrieval systems.

Figure 1 shows a query page for the Computer Science Technical Reports collection, which uses MG, and a query page for the Melody Index collection, which uses MR. Although the interfaces are very different—for example, the music interface accepts audio queries in several MIME types and returns documents as a GIF image of notated music, or in several audio formats—we realized that they nevertheless had a great deal in common. The interface must be able to display appropriate query options and return both query results and target "documents" back to the user in a variety of forms. Here we provide a brief description of these two indexers to convey the radical differences between them.

2.1 Full-text retrieval using MG

The MG system [10] is a freely-available full-text retrieval system that makes efficient use of disk resources by storing the index, and the text that is indexed, in compressed form. Typically text compresses to 25% of its original size, and the compressed index, which is stored at the level of granularity of documents rather than words, occupies around 7% of the size of the original text. This leads to a total storage requirement for the indexed collection of approximately one-third of the size of the original text alone.

The MG software runs on most Unix systems, and on 32-bit Windows machines. It supports ranked and Boolean queries, all combinations of stemming and casefolding, and allows the user to specify the maximum number of documents to be returned from a query. While very efficient in storage, having indexes at the level of granularity of whole documents is something of a restriction, and although MG does support a subsidiary paragraph-level index as well this is not enough to give the flexibility we require for most collections in the New Zealand Digital Library. Consequently, we build a separate index for each part of the document that a user might wish to search. Thus we have indexes for complete documents, individual pages, paragraphs, articles, abstracts, authors, titles, and so on. This has the advantage of allowing fielded search as well.

Most of the collections in the New Zealand Digital Library are textual and use MG as their underlying search engine.

2.2 Melody retrieval using MR

The MR system [5] is a novel scheme for searching musical melodies: it matches sung (or hummed) input to a database of tunes. Options are provided to restrict attention to subsets of the database, to choose one of two matching algorithms, to match anywhere within a tune or beginnings only, to match using musical intervals or up-down-same contour, to ignore or take account of note durations, to transcribe using fixed tuning or try to adapt to overall drift in the user's intonation, to specify minimum rest and note lengths, and to specify the speed of the music. The reason for many of the options is that users differ in their musical ability and in the accuracy with which they remember tunes, so it is necessary to provide flexibility to cater for this variation.

At present, just one collection is based on this retrieval system, a Melody Index that comprises a database of nearly ten thousand international folk tunes from various countries. There are just over half a million notes in the database, and the average length of a melody is around fifty notes. The database is segmented into the following parts:

North American (and British) folksongs (1700 tunes)
German ballads and folksongs (5500 tunes)
Chinese ethnic and provincial songs (2100 tunes)
Irish folksongs (50 tunes),

and each part can be searched separately.

3 Requirements

We seek a design that is flexible enough to accommodate diverse collections of information. In order to get started, a "model" of a collection is necessary, a model that characterizes collections in an open way that is not restrictive. Our experience with the wide variety of requirements posed within the New Zealand Digital Library led us to the following model.

All collections are made up of *documents*. Documents come in a variety of formats: we presently accommodate

plain ASCII, PostScript, PDF, HTML, SGML, and Microsoft WORD for textual documents; REFER, BIBTEX, and USMARC for bibliographic information; and various internal formats for musical information. Collections must invariably undergo some *building process* to make them suitable for display, search and retrieval. This might involve converting the documents to another format, and identifying subparts that require their own *indexes*. Amongst the current collections are indexes for complete documents, individual pages, paragraphs, articles, abstracts, authors, titles, subjects, publication details, references, and motions. The building process also involves preparing some *statistics* about the collection.

Access points to a collection define ways in which it can be searched by a user, via some user interface. These access points generally correspond to the collection's index or indexes (although it is possible that some collections will be searched directly, without indexes being created in advance). The interface enables a user to submit a query and receive a list of results in return. The search might be based on a textual string, an audio stream, a graphics file, or (conceivably) any other object: mechanisms for specifying the query data (or its whereabouts) and submitting it are needed. The list of results returned to the user specifies matching documents or parts of documents. The user examines this list and retrieves some of the documents it mentions—or information about them. The results of a search can be expressed in different forms. The returned documents may be the original documents that comprise the collection, or a transformation of them—perhaps in another format, or a subpart, or some associated information (size, location, price, etc.). The format might depend on the original document type.

Some means of *browsing* the collection is necessary. Inevitably collection-dependent, this may take the form of a hierarchical browser, or a table of contents, or a list of sites. Some collections may be browsable in several ways. Collections may have *controlled access*: public or private. Public collections are available to all; private ones reveal their existence to a restricted audience. Access to the whole of a private collection, or to parts of it, might be subject to user authentication. *Billing* is also a necessary requirement and could form part of the user authentication processes. Additionally, a collection may or may not be *active*, that is, actually up and running. *Descriptive information* (in the form of text) must also be associated with each collection: a title, brief summary, and description of each index. This information may be provided in different languages.

We need to accommodate *distributed processing* within the architecture. Search engines for different collections should be capable of residing on different computers, and it should be possible for multiple copies of a search engine to work in parallel on several processors. Under this distributed architecture, individual organizations can maintain their own collections of information; but there is a consistent interface for accessing them. Moreover, a user can query several collections simultaneously.

It must be possible to make *different user interfaces* to the same collection, so that interfaces can be provided for different user populations (such as the visually impaired). System maintainers must be able to develop and test new user interfaces while existing ones are still running. Also, other applications may use the same index/search engine sub-system—for example, alerting services (for example [11]) or data mining schemes (for example [1]).

Finally, a mechanism is necessary for learning about other collections, so that user interfaces can draw the user's attention to the existence of new collections as they come up, without requiring any special effort by the system maintainers.

4 Design

An architecture that satisfies these requirements has been designed and implemented as a foundation for the New Zealand Digital Library. The design, shown in Figure 2, consists of three programs—a *builder*, a *searcher* and an *interface*—and a *protocol* for communicating between the *interface* and each *searcher*. Every collection of information must have a *searcher*, however, it may not need a *builder*. There is no limit on the number of *searcher* programs that a given *interface* program can connect to. The crux of the new architecture is the separation of interface programs from the actual search engines, and the definition of a protocol for communicating between the two. We return to this below in the section on interface programs.

To incorporate different index styles, often with vastly differing needs, the builders and searchers use a flexible, object-oriented program structure. Object classes with all the basic features needed for building and searching are defined, but the functionality they provide is selectively overridden by defining subclasses for particular search and retrieval systems, and these are finally overridden by collection-specific subclasses that accomplish any tasks peculiar to a collection. This object hierarchy is shown in Figure 3 for three example collections: the Computer Science Technical Reports collection (CSTR), the Frequently Asked Questions collection (FAQ), and the Melody Index collection (MELDEX). Further search and retrieval systems can easily

be incorporated into this framework by deriving other search and retrieval subclasses.

Each object structure is encased in a simple wrapper program that provides a consistent command-line interface, no matter what indexing or searching subsystem lies beneath. This mechanism greatly simplifies maintaining the collections, because there are fewer programs for administrators to come to terms with. The class hierarchy is written in object-oriented PERL, although any language that allows communication over the Internet would serve. Communication between the objects and the search engines themselves is handled by piping information back and forth, allowing full language flexibility. In fact, the actual search engines are written in C (for MG) and C++ (for MR).

4.1 Builders

Builders are responsible for accomplishing all the tasks that are necessary to make a collection ready for serving to users. This may include converting documents to a different format, building indexes, and preparing statistics about the document collection. All this work is specific to both the collection and the search and retrieval software that is being used for it. There are four methods in the *builder* class: *init*, *build_indexes*, *make_auxiliary_files*, and *cleanup*. *Init* parses command-line options, initializes variables used in the building process, removes old indexes, converts files to a useable format, caches archives onto local storage, and records building dates. *Build_indexes* creates the files that are needed to search a collection and obtain search results. *Make_auxiliary_files* prepares statistics for the collection and performs other functions such as gathering tables of contents, site lists, and so on. *Cleanup* removes temporary files and performs other general housekeeping operations.

4.2 Searchers

Searchers are responsible for performing all the tasks involved in actually serving collections to users. This includes providing access points to a collection, retrieving documents from it, and obtaining any general information that relates to the collection. It also includes provision for obtaining information about other collections that are available. When invoked, a searcher calls an *initialize* method in the *search* class, kills any other searcher for the same collection, and initializes its communication port. It then waits for requests and responds by calling the appropriate method in the *search* class. When the server is shut down, a *cleanup* method is called.

There are four types of documents that a searcher deals with: *boilerplate*, *collection information*, *summaries*, and *target documents*. *Boilerplate*[1] is used for any collection-specific text needed by an interface program—this text may be available in more than one language. In a well-designed collection, all text specific to that collection should be stored as boilerplate documents, rather than (for example) as string constants in the interface program. *Collection information* refers to documents relating to the collection that are not indexed. Examples include tables of contents, site lists, collection icons, short descriptions of the collection, collection-specific help, and other information generated on the fly—like a musical staff showing a melody that was sung as an input query. *Summaries* are short descriptions of target documents in the collection, often available at different levels of detail. They are used both when browsing a collection and when displaying search results to give a small amount of information about hits. Finally, *target documents* are those that make up the collection, either the original documents or versions in other formats.

4.3 Interface programs

Interface programs mediate between the user and the searchers. They allow information to be obtained about a collection, searches to be performed, and documents to be retrieved. Interface programs are designed so that they can run on a machine different from the one that hosts the searcher. Interface programs communicate with each searcher through a well-defined communication protocol that can provide all the information needed to present a complete view of the collection to a user. They divorce the user interface from the searching, allowing individual user interfaces to be created without the restriction of a pre-determined information format (such as HTML). Using the information provided by the searcher, an interface program can present an attractive, informative interface to a collection of information on a remote computer with little or no special configuration effort.

Interfaces have been written that allow command-line access (mainly for debugging and performing information retrieval experiments) and end-user access over the World Wide Web. The latter involves several pieces of software: the user's Web browser, an HTTP server, a CGI script, and an interface server program. The last three could be combined if efficiency became an issue. If the occasion arose—as it might for interfaces to people with special

[1] "Boilerplate" is a journalistic term for repeated items of canned text (like a newspaper masthead), used so often that they are stored as pre-prepared printing plates.

needs—a full window-based user interface could also be created.

4.4 Protocol for communication

The communication protocol defines how interface programs communicate with searchers; this is similar in spirit to the well known Z39.50 protocol [6] but incorporates an added degree of generality that is needed to implement the architecture. Unlike Z39.50, the protocol we have designed is stateless. This generally simplifies implementation. However, it does make certain tasks, like authentication, more difficult. Messages are sent from the interface program to the search program using a remote procedure call mechanism. Arguments are serialized, passed across a socket interface, and de-serialized. The procedure—a method within the *search* class—is called and the result is serialized, returned through the socket and de-serialized by the interface mechanism.

There are three types of message that all search classes must be able to respond to: requests for *general information*, requests for a *search*, and requests for *documents*. The message type is prepended to the message name, giving a uniform naming convention. The messages, split into these three categories, are summarized in Table 1: they are explained in more detail below.

Because all collections on a single computer are included in the same file structure, once the location of one collection is known it is possible to ascertain what other collections are present on the same machine. This gives a convenient mechanism for discovering the existence of collections. When an interface program starts up (and at periodic intervals after that) it sends messages to all the collections which it has information on, requesting information on other collections on the same machine. The information that is returned about each collection includes its *identifier*, the *host machine* it runs on, and the *port* it uses for communication. When a collection is added, its existence will (eventually) be noticed by every interface program, providing it is aware of another collection on the same machine. The automatic nature of the notification process greatly simplifies the administration involved in collection creation.

Once the interface program has constructed a list of collections, it can acquire further information about each. This allows it to present the user with an appropriate menu of collections. *ThisCollection* and *Boilerplate* information messages are requested when the interface program is exploring all the collections. The *SearchOptions* message is used so that the interface program can present appropriate choices for a user's query, and obtain the information necessary to specify the query. Other messages like *CollectionInfo* and *SummaryOptions* are needed when query and results pages are displayed. Finally, the *DocumentTypes* message is used to determine an appropriate format when the user requests a document.

This protocol, like any other, is critically dependent on semantics—a common understanding of what the various terms mean. Using it, well-engineered interfaces can be created for a large range of diverse collections: however, it does require the interface program's author to understand all the various options. For this reason defaults are given for all search, summary, and document-type options. This allows even a simple interface program to communicate with a powerful searcher.

When the user has selected all necessary options and entered the information required to perform a search, which in most cases constitutes a list of query terms, the interface program makes the corresponding request to the searcher. The response includes any information that relates directly to the query. This often involves optional information: for example, if the searcher expands terms before matching them, the expanded query may be returned; or if an audio track has been transcribed before matching it, the transcription may be returned. Typically, the list of returned documents is saved by the interface program and delivered a page at a time using the document summaries.

Messages in the third section of Table 1 are used to obtain the various documents that are accessible through the search program. Two of them, *Boilerplate* and *Summaries*, return lists because these documents are frequently required in groups and are generally brief. The other two, *CollectionInfo* and *TargetDocument*, return single documents because they are usually only required individually and are generally longer.

5 Other architectures: Review and comparison

In this section we review other architectures and comment on how they differ from that presented here.

5.1 NCSTRL: Networked Computer Science Technical Reports Library

The Networked Computer Science Technical Reports Library, or NCSTRL (pronounced "ancestral"), is an amalgam of two earlier projects for gathering together technical reports in computer science (WATERS and DIENST), which in turn were inspired by a scheme (UCSTRI) for searching the README files that often appear in FTP archives of technical reports. A distributed architecture, it provides users with four kinds of service [3]: a *repository service*, an *index service*, a *meta service*, and a *user interface service*. All services communicate

using a protocol that was developed in the DIENST project.

Review. The *repository service* stores documents. Documents are uniquely specified by a string called a *handle* (or "unique document identifier"). Document handles have two parts: a naming authority and a string provided by that authority. Naming authorities are hierarchically organized, with period symbols used to separate each name. Handles are similar in intention to universal resource names (URNs) on the World Wide Web. The document model allows a single document to reside in several different formats, and, where appropriate, provides access to individual pages of documents. Formats are intended to express the *purpose* of a document rather than its representation, although the possible formats—which include PostScript, plain text, OCR'd text, scanned images, in-line images, document structure files, and HTML—seem to indicate a rather low-level interpretation of "purpose." The actual representation of the document is represented as a MIME type corresponding to its format.

The *index service* provides a facility for searching a set of document descriptions and returning a list of handles of those that match, or for returning bibliographic information about a particular document. Boolean searches are permitted, and *title*, *author*, and *abstract* fields are supported. Stemming and casefolding appear to be always in force. The *meta service* provides a directory of other service locations. Through it one can list the resources involved in the library, such as, index servers, repositories, and organizations that publish reports. The *user interface service* provides a form that allows a fielded search to be specified, and web pages to browse the collection by date or author. An important feature is that it can perform parallel searching of multiple document indexes.

NCSTRL is built on the DIENST protocol, an open protocol that uses HTTP for information transmission. Although designed specifically for technical reports, this protocol should be applicable to other types of collection. It embodies the document model described above, and it seems that other index and user interface services could be provided, based on this protocol. A serious problem with the distributed nature of the architecture is that parts of the index become unavailable when servers go down; however, it is planned to provide some fault tolerance by replicating index records on a backup server, thus making it possible to search the information on a site when it is down (but not retrieve the reports stored there).

Comparison. The principal difference between NCSTRL and the NZDL architecture is that NCSTRL regards a collection as made up of many different distributed servers, whereas we view a library as comprising many different distributed collections. NCSTRL is oriented towards a particular type of collection—technical reports—that are inherently distributed amongst the organizations that publish them.

In contrast to NCSTRL, each of the collections in the NZDL is centralized as a particular information repository, although it may have been gathered from several sources on the Internet. Moreover, for any particular collection a certain piece of indexing software is responsible for maintaining all access points to it, which simplifies the design considerably. Access to individual collections is "learned" from the collections themselves, which reduces the need to publish each new collection's availability. Other differences include the fact that in the NZDL architecture formats do not relate to intent; the unrestricted range of formats enables multimedia searching; and no structure (e.g. pagination) is imposed on documents except by the indexing software that processes them. As befits its bibliographic orientation, NCSTRL permits field searching whereas we rely more heavily on ranked full-text search.

5.2 Harvest

HARVEST is a distributed digital library system that intends to make efficient use of Internet resources [2]. First, *gatherer* programs are run to collect documents from the repositories where they reside. These programs filter the raw information, reducing the amount that needs to be transmitted. This information is then conveyed to a *broker*, which indexes it and provides an interface to the collection. Indexes can be replicated efficiently on different sites on the Internet using a *replicator* program. Brokers can also build indexes by filtering information held by other brokers, which allows, for instance, the creation of a subcollection containing Artificial Intelligence technical reports from a full Computer Science technical report index.

Although the HARVEST architecture is distributed, brokers do not support distributed information retrieval: the indexing software and user interface must run on the same machine. There is, however, a special central broker, called the *HARVEST server registry*, that contains information about each *gatherer*, *broker*, *cache*, and *replicator* on the Internet. This central index can help users find a suitable broker that is likely to contain the information they are seeking. It was also planned to develop tools to permit recursive query evaluation—allowing users to search several servers identified by the registry simultaneously.

Review. A broker consists of five software modules: *collector*, *registry*, *storage manager*, *index/search engine*, and *query manager*. The collector is responsible for obtaining new information, the registry stores information about each object, the storage manager archives the object on disk, the index/search engine indexes and retrieves the objects, and the query manager provides a World Wide Web interface to the index/search engine.

HARVEST defines a generic index/search engine interface that can accommodate a variety of "backend" search engines. Two search programs have been developed specially for HARVEST—*Glimpse*, which supports small indexes and flexible queries, and *Nebula*, which sacrifices index size for speed. The only requirements placed on the index/search engine are that they must support Boolean queries and allow incremental updates. The query manager implements a consistent interface that is oriented towards the World Wide Web; it also allows a user direct access to each index/search engine to take advantage of its special features. Unfortunately the query interface does not retain state across queries, so query refinement is not possible.

The HARVEST system is highly customizable—with moderate programming effort. This flexibility is shown in the wide range of demonstration collections implemented with the software—a Networked Information Discovery and Retrieval collection, a Computer Science Technical Report collection, a PC Software collection, and a collection of World Wide Web home pages.

Comparison. HARVEST focuses on making good use of Internet resources. However, our experience in the New Zealand Digital Library is that only around half of the collections we are currently developing contain information obtained over the Internet, and even fewer have a requirement to be updated over the Internet. We feel that this situation will reflect digital libraries of the future, with most of the content coming from local material or from private or in-house collections. Although HARVEST's broker system does accommodate different index styles, it is unclear whether the design is able to incorporate the diverse index and search engines needed for multimedia collections. Finally, the fact that the index/search engine is combined with a user interface means that cross-collection searches cannot be made, multiple interfaces are hard to make, other services cannot be built upon the search subsystem, and it could not operate at a remote site.

5.3 Stanford's Infobus

Inspired by the concept of a computer bus, the InfoBus being developed by the Stanford Digital Libraries Group defines a standard mechanism for interacting with many different distributed information servers [7]. The basic approach is to model the way in which interface programs interact with different library services, and develop protocols and infrastructure general enough to implement the model. A general *service client* connects to a *library service proxy* which acts as a gateway, translating messages into formats that can be understood by the *library service*.

Review. A *library service proxy* is created for each different library service protocol (for example, Knight-Ridder's Dialog information service, World Wide Web information sources, Z39.50, and Oracle's ConText summarization tool). The service client connects to the appropriate library service proxy using Stanford's digital library interoperability protocol. This protocol is implemented using CORBA objects.[2] Each library service proxy contains an object that implements a standard set of methods, for example *open session*, *open database*, *search* and *quit*. The proxy utilizes the polymorphism in object-oriented languages to implement methods that take appropriate action for the particular protocol that the library service is using. For example, a call to *open session* on a TELNET library service proxy would open a TELNET connection to the requested service, whereas, a call to *open session* on a Z39.50 library service proxy would open a Z39.50 connection to the requested service. Using this abstraction, a service client can connect to any library service as long as there is a library service proxy that can translate to the appropriate protocol.

This interoperability is intended to shield users from the underlying details of communication with diverse networked resources, allowing them to navigate "information space" in a uniform way.

Comparison. The biggest disadvantage of this scheme is that a library service protocol must be built for each service that a user wants to connect to—this becomes a problem with the quickly expanding World Wide Web where services multiply daily, each one using different conventions. Although the Stanford architecture seems at first to be quite different to the NZDL, they do exhibit some similarities. Both use objects to accommodate diversity—the InfoBus deals with heterogeneous communications protocols, the NZDL deals with heterogeneous indexes and media types. In fact it would be possible to use the *search program* like a *library service proxy* and derive classes which connect to a stateless service like a commercial Web search engine, for

[2] CORBA is a distributed object standard developed by the Object Management Group.

instance—although this would be inefficient for state-based protocols like Z39.50. Again, there seems to be no indication of how well the architecture can handle document types other than text.

6 Conclusion

We have described a new architecture for digital libraries, one that is flexible enough to deal with many different collections served through a wide variety of searching or indexing engines. We take the collection as our basic unit and assume that any given collection is served through a particular search discipline. The search engine is encapsulated within a searcher program that provides important additional facilities over and above search itself: facilities both for the interface program with which it communicates (such as what query options are supported, their possible values and default values), and for the user (such as target documents, summaries, and information about the collection). The searcher communicates with the interface program that provides the user interface through a well-defined communication protocol. Information about the existence of other collections is also disseminated through this mechanism. The design is flexible enough to accommodate search engines for completely different media, and this has been demonstrated by implementing services for text retrieval and for music melody retrieval.

This work has many points of contact with other digital library projects, both in aims and in mechanisms. Like Stanford's InfoBus it uses objects to achieve flexibility of functionality and to maximize the use of common code segments. Like NCSTRL it is distributed, although in our case it is whole collections, and the software that accesses them, that is distributed, rather than parts of a collection. Like HARVEST it is open-ended and flexible, but care is taken to separate user interface from searcher so that different interfaces can communicate with the same search mechanism. Studying these earlier systems has enabled us to design and implement a new and powerful architecture, one that is already simplifying collection building and maintenance, and is expected to provide an effective testbed for user interface design and for the creation of novel library services within the New Zealand Digital Library.

References

[1] Apté, C., Damerau, R. and Weiss, S. (1994) "Automated learning of decision rules for text categorization." *ACM Trans Office Information Systems* **12**(3), 233–251.

[2] Bowman, C. M., Danzig, P. B., Hardy, D. R., Manber, U. and Schwartz, M. F. (1995) "The Harvest information discovery and access system." *Computer Networks and ISDN Systems* **28**, 119–125.

[3] Davis, J. R. (1995) "Creating a networked computer science technical report library." *D-Lib Magazine*, September 1995, http://www.dlib.org/dlib/september95/09davis.html.

[4] McNab, R.J., Smith, L.A. and Witten, I.H. (1996a) "Signal processing for melody transcription." *Australasian Computer Science Conference*, 301–307, Melbourne, Australia; January.

[5] McNab, R.J., Smith, L.A., Witten, I.H., Henderson, C.L. and Cunningham, S.J. (1996b) "Toward the digital music library: tune retrieval from acoustic input." *Proc Digital Libraries '96*, 11–18.

[6] National Information Standards Organization (1995) *Information retrieval (Z39.50): Application service definition and protocol specification*. ANSI/NISO, Bethesda, Md.

[7] Paepcke, A., Cousins, S. B., Garcia-Molina, H., Hassan, S. W., Ketchpel, S. P., Röscheisen, M. and Winograd, T. (1996) "Using distributed objects for digital library interoperability." *IEEE Computer* **29**(5), 61–68.

[8] Smith, J. R. and Chang, S.-F. (1996) "VisualSEEK: a fully automated content-based image query system." *ACM Multimedia Conference*, Boston, MA; November.

[9] Witten, I.H., Cunningham, S.J. and Apperley, M.D. (1996) "The New Zealand Digital Library project." *New Zealand Libraries* **48**(8), 146–152.

[10] Witten, I.H., Moffat, A. and Bell, T.C. (1994) *Managing gigabytes*. Van Nostrand Reinhold.

[11] Yan, T. W. and Garcia-Molina, H. (1995) "SIFT — A tool for wide-area information dissemination." *Proceedings of the 1995 USENIX Technical Conference*, 177–186.