NIST Special Publication 500–229

Computer Systems Technology

U.S. DEPARTMENT OF COMMERCE
Technology Administration
National Institute of Standards and Technology

NIST

Z39.50 Implementation Experiences

Editors:

Paul Over
William E. Moen

Ray Denenberg
Lennie Stovel

The National Institute of Standards and Technology was established in 1988 by Congress to "assist industry in the development of technology . . . needed to improve product quality, to modernize manufacturing processes, to ensure product reliability . . . and to facilitate rapid commercialization . . . of products based on new scientific discoveries."

NIST, originally founded as the National Bureau of Standards in 1901, works to strengthen U.S. industry's competitiveness; advance science and engineering; and improve public health, safety, and the environment. One of the agency's basic functions is to develop, maintain, and retain custody of the national standards of measurement, and provide the means and methods for comparing standards used in science, engineering, manufacturing, commerce, industry, and education with the standards adopted or recognized by the Federal Government.

As an agency of the U.S. Commerce Department's Technology Administration, NIST conducts basic and applied research in the physical sciences and engineering, and develops measurement techniques, test methods, standards, and related services. The Institute does generic and precompetitive work on new and advanced technologies. NIST's research facilities are located at Gaithersburg, MD 20899, and at Boulder, CO 80303. Major technical operating units and their principal activities are listed below. For more information contact the Public Inquiries Desk, 301-975-3058.

Office of the Director
- Advanced Technology Program
- Quality Programs
- International and Academic Affairs

Technology Services
- Manufacturing Extension Partnership
- Standards Services
- Technology Commercialization
- Measurement Services
- Technology Evaluation and Assessment
- Information Services

Materials Science and Engineering Laboratory
- Intelligent Processing of Materials
- Ceramics
- Materials Reliability[1]
- Polymers
- Metallurgy
- Reactor Radiation

Chemical Science and Technology Laboratory
- Biotechnology
- Chemical Kinetics and Thermodynamics
- Analytical Chemical Research
- Process Measurements[2]
- Surface and Microanalysis Science
- Thermophysics[2]

Physics Laboratory
- Electron and Optical Physics
- Atomic Physics
- Molecular Physics
- Radiometric Physics
- Quantum Metrology
- Ionizing Radiation
- Time and Frequency[1]
- Quantum Physics[1]

Manufacturing Engineering Laboratory
- Precision Engineering
- Automated Production Technology
- Intelligent Systems
- Manufacturing Systems Integration
- Fabrication Technology

Electronics and Electrical Engineering Laboratory
- Microelectronics
- Law Enforcement Standards
- Electricity
- Semiconductor Electronics
- Electromagnetic Fields[1]
- Electromagnetic Technology[1]
- Optoelectronics[1]

Building and Fire Research Laboratory
- Structures
- Building Materials
- Building Environment
- Fire Safety
- Fire Science

Computer Systems Laboratory
- Office of Enterprise Integration
- Information Systems Engineering
- Systems and Software Technology
- Computer Security
- Systems and Network Architecture
- Advanced Systems

Computing and Applied Mathematics Laboratory
- Applied and Computational Mathematics[2]
- Statistical Engineering[2]
- Scientific Computing Environments[2]
- Computer Services
- Computer Systems and Communications[2]
- Information Systems

[1]At Boulder, CO 80303.
[2]Some elements at Boulder, CO 80303.

NIST Special Publication 500–229

Z39.50 Implementation Experiences

Editors:

Paul Over
Computer Systems Laboratory
National Institute of Standards and Technology
Gaithersburg, MD 20899-0001

Ray Denenberg
Z39.50 Maintenance Agency
Library of Congress
Washington, DC 20540

William E. Moen
School of Library and Information Sciences
University of North Texas
Denton, TX 76201

Lennie Stovel
The Research Libraries Group, Inc.
1200 Villa Street
Mountain View, CA 94041-1100

Computer Systems Laboratory
National Institute of Standards and Technology
Gaithersburg, MD 20899-0001

September 1995

National Institute of Standards and Technology Special Publication 500-229
Natl. Inst. Stand. Technol. Spec. Publ. 500-229, 125 pages (Sept. 1995)
CODEN: NSPUE2

U.S. GOVERNMENT PRINTING OFFICE
WASHINGTON: 1995

For sale by the Superintendent of Documents, U.S. Government Printing Office, Washington, DC 20402

Preface

ANSI/NISO Z39.50, a communications protocol for information retrieval in a client/server environment, is widely implemented in the United States and increasingly in Europe - with interest growing elsewhere as well. Maintained for ANSI/NISO by the Z39.50 Maintenance Agency at the Library of Congress, the standard is driven by the consensus of its implementors as represented by the active members of the Z39.50 Implementors' Group (ZIG). The ZIG exchanges information about implementation problems and solutions (e.g., through energetic discussion on the Z39.50 list and at meetings), but until now, only a few members have found time to reflect on and write about their experiences with implementing Z39.50.

While substantial explanatory material has been added to the Z39.50 standard in recent revisions, the protocol specification is not intended as a repository for supporting material such as discussions of how to get started, design alternatives, lessons learned, etc. But the need for such ancillary documentation increases as the standard grows and the number of implementations rises. The following collection of papers on Z39.50 implementation experiences is offered as a small contribution toward meeting this need in the hope that it will help new developers to build on the work of more experienced Z39.50 implementors and encourage additional contributions of a practical nature to the literature on Z39.50.

The first three papers draw on the authors' extensive experience with particular implementations to provide generic guidance on designing and implementing Z39.50 clients and servers. In *Z39.50 for Full-Text Search and Retrieval* Margaret St. Pierre (Blue Angel Technologies) describes the construction of a minimally conformant client/server and its modular extension. John Kunze (U.C. Berkeley and the U.S. National Library of Medicine) focuses in *Basic Z39.50 Server Concepts and Creation* on some fundamental design decisions in implementing a basic server. Ralph LeVan's (OCLC) article, *Building a Z39.50 Client*, provides a tutorial on building a Z39.50 client and includes sample source code.

The next two papers also offer generic guidance but with a narrower focus. In *Implementing Explain* Denis Lynch (TRW Business Intelligence Systems) writes on how a Z39.50 server's Explain database can be used to help users and clients discover and take advantage of a Z39.50 server's resources. Makx Dekkers's (Pica) contribution, *Implementing Z39.50 in a multi-national and multi-lingual environment*, reflects on some of the problems encountered as Z39.50, with its Anglo-American origins, is applied to information retrieval in an international environment.

The third group of papers discusses some publicly available resources that have grown out of particular implementations. *Use of Z39.50 for Search and Retrieval of Scientific and Technical Information* by Les Wibberley (Chemical Abstracts Service) recounts the growing pains of Z39.50 in meeting the needs of scientific users and the development of the Scientific and Technical Attribute Set (STAS). Kevin Gamiel's and Nassib Nassar's (CNIDR) article, *Structural Components of the Isite Information System*, outlines technical topics related to a publicly available open information system (Isite) - in particular its document model.

The remaining four papers speak to design goals and implementation decisions in specific information retrieval applications using Z39.50. In *Z39.50 - implications and implementation at the AT&T library network* Robert Waldstein (AT&T Bell Laboratories) looks at the changing expectations of users in one large corporate environment and the development of the Z39.50 standard to meet those expectations. *The Implementation of Z39.50 in the National Library of Canada's AMICUS System* by J. C. Zeeman (Software Kinetics Ltd.) describes the architecture of an integrated bibliographic system including details about the modular search engine and the management of its interface to a relational and a full-text database. Terry Sullivan and Mark Hinnebusch (Florida Center for Library Automation) discuss the implementation of Z39.50 services to maximize platform independence in *Developing a Multi-Platform Z39.50 Service*. Peter Ryall (Lexis-Nexis) writes in his article, *Use of Z39.50 for the Delivery of Current Awareness Products*, about the application of a variety of Z39.50 services to help users deal with information overload.

Our thanks to all the authors for finding time to collect and refine their thoughts and to the other readers - Lennie Stovel (The Research Libraries Group, Inc.) and Bill Moen (University of North Texas) - for their careful review of the articles and many thoughtful suggestions for improvements.

Ray Denenberg (Z39.50 Maintenance Agency) Paul Over (NIST)

Table of Contents

Z39.50 for Full-Text Search and Retrieval
Margaret St. Pierre 1

Basic Z39.50 Server Concepts and Creation
John A. Kunze 11

Building a Z39.50 Client
Ralph LeVan 23

Implementing Explain
Denis Lynch 49

Implementing Z39.50 in a multi-national and multi-lingual environment
Makx Dekkers 57

Use of Z39.50 for Search and Retrieval of Scientific and Technical Information
Les Wibberley 61

Structural Components of the Isite Information System
Kevin Gamiel and Nassib Nassar 71

Z39.50 - implications and implementation at the AT&T library network
Robert K. Waldstein 75

The Implementation of Z39.50 in the National Library of Canada's AMICUS System
J. C. Zeeman 85

Developing a Multi-Platform Z39.50 Service
Terry Sullivan and Mark Hinnebusch 99

Use of Z39.50 for the Delivery of Current Awareness Products
Peter N. Ryall 111

Z39.50 for Full-Text Search and Retrieval

Margaret St. Pierre
Blue Angel Technologies
saint@bluangel.com

Abstract

For search and retrieval of full-text, image, and multimedia information over a heterogeneous network, a non-proprietary standards-based communications protocol is mandatory. In order for the protocol to achieve general acceptance and ubiquity within the rapidly evolving world of distributed information access, it is imperative that the implementation of this protocol be simple and modular, yet rich enough in functionality to meet the growing demands of the information age.

This paper describes implementation experience with the ANSI/NISO Z39.50-1995[1] information retrieval protocol as the communications protocol of choice for distributed information access. The requirement for simplicity is achieved with an implementation of a baseline set of Z39.50 services. The described base-level functionality is sufficient to demonstrate interoperable search and retrieval functionality with a number of other Z39.50 implementations. The requirement for modularity and richness in function is achieved through the incremental implementation of features such as Generic Record Syntax, Element Set Specification, and the Explain Facility.

1 Introduction

The ANSI/NISO Z39.50-1995 Standard is the result of a culmination of the many requirements of a large number of contributing implementors. It was designed to be a comprehensive information search and retrieval protocol specification. The development of a complete ANSI/NISO Z39.50-1995 implementation may at first appear to be a major undertaking. In practice, most implementors begin with a simple baseline implementation, verify interoperability against other implementations, and then incrementally expand in functionality with additional Z39.50 features. This paper describes how such an approach can be used to develop a Z39.50 implementation for use in the search and retrieval of full-text, image, and multi-media information.

Section 2 provides a description of a baseline client and server implementation, and provides recommendations for modular additions of Z39.50 features to this baseline. Section 3 introduces the modular addition of Generic Record Syntax (GRS) for supplying structured database records. Section 4 covers the requesting of structured database records using the Element Specification Format (ESPEC). An extension for providing additional server-specific information using the Explain Facility is described in Section 5. Finally, Section 6 provides guidance for additional modular extensions of Z39.50 functionality.

This paper is one of a series of implementation papers put together by NIST (National Institute of Standards and Technology) and the Z39.50 Maintenance Agency. This paper does not address data communications nor implementation tools. These topics are covered in other papers in this series.

2 Baseline Implementation

A baseline implementation of Z39.50 is a conformant implementation as described in the conformance section of the Standard (refer to Section 4.4.1 "*General Conformance Requirements*"). The only requirements for minimal conformance are the Init, Search, and Present Services, and the Type-1 query. The section below provides guidance for creating a simple baseline implementation, complete with example pseudo-application protocol data units. This baseline implementation should prove to be interoperable with most full-text Z39.50 implementations.

[1]ANSI/NISO Z39.50-1995 -- Information Retrieval (Z39.50): Application Service Definition and Protocol Specification.

The main components of the baseline implementation are the Init, Search, and Present Services. These services provide the ability to negotiate initialization information, perform a search on a database, create a result set of database records that match the query, and retrieve one or more records from the result set. The query is a simple Type-1 query containing a single term, and database records are returned in a result set as ASCII text, called *Simple Unstructured Text Record Syntax* (SUTRS). Although the Standard supports the ability to include more than one database in a Search Request, a baseline server implementation need only support searching a single database at a time. And finally, a conformant implementation provides support for creating and accessing a single result set of database records, called the *default* result set; support for multiple result sets is optional and may be added at a later time.

An example of a simple Init Request sent by the client is as follows:

```
protocolVersion:        Version 1 and 2
options:                Search and Present
preferredMessageSize:   50000
exceptionalRecordSize:  50000
```

These comprise the mandatory components of the Init Request. For simplicity, the optional components have been omitted. It is desirable, however, for interoperability testing and usage statistics gathering, to include an Implementation Id, Name, and Version.[2] The Init Response returned by the server includes the negotiated values of the Init Request parameters, and a Boolean flag indicating whether or not the server accepts the connection.

Of the three services, the Search Service contains the largest number of mandatory parameters. Some client implementations expect to use the Search Request to obtain the first few of records in the result set, commonly called a *Piggybacked-Present*, and request additional records later using the Present Service. Other client implementations request no records during the Search Service, but instead use the Present Service for the retrieval of all records. The former approach may provide improved performance, while the latter approach simplifies the implementation.

[2]Some server implementors consider it anti-social if this information is not included in an Init Request.

A simple Search Request example follows:

```
smallSetUpperBound:       0
largeSetLowerBound:       1
mediumSetPresentNumber:   0
replaceIndicator:         true
resultSetName:            "default"
databaseName:             database name
query:                    Type-1
  attributeSet:           1.2.840.10003.3.1
  rpn:                    Operand
    attrTerm:             AttributesPlusTerm
      attributes:         (empty list)
      term:               octet string
```

For this Search Request, the client indicates that it does not want any piggybacked records in the response by setting the small set upper bound to 0 and large set lower bound to 1. It requests that the server perform a search on a database whose name is *database name*, where the search term is supplied as an *octet string*. The example query is the simplest query to formulate consisting of a Type-1 query, using the Bib-1 attribute set, and containing a single operand, no attributes, and a term. Most implementations use the Bib-1 attribute set for providing a well-known set of search access points, such as a Title or Author search. As described in Section 5 (see also the Lynch article in this series), the Z39.50 Explain Facility provides a means for clients to discover the search access points available on a specific server.

An example of a Search Response that may have resulted from the above Search Request is shown below:

```
resultCount:              15
numberOfRecordsReturned:  0
nextResultSetPosition:    1
searchStatus:             true (i.e. success)
presentStatus             success
```

No records were returned since no records were requested in the Search Request. The number of items in the result set is 15, and the search completed successfully. The next-result-set-position parameter is not particularly useful, but is mandatory.

Finally, the Present Service provides a means for retrieving records from the result set. The Present Request optionally specifies an element set name and preferred record syntax, which if not included, default to whatever the server selects. In a baseline client implementation, it is best to explicitly specify the preferred record syntax, since a server may select a syntax not supported. The following sample Present Request asks for the delivery of the first database record in the result set as ASCII text by requesting a preferred record syntax of SUTRS.

```
resultSetId:                "default"
resultSetStartPoint:        1
numberOfRecordsRequested: 1
recordComposition:          simple
  elementSetNames:          "F"
preferredRecordSyntax:      1.2.840.10003.5.101
```

A full record is requested by specifying an element set name of "F". Alternatively, the element set name may be specified as "B", referring to a request for a brief record. The information provided in a brief database record is defined by the server. Typically a brief record contains enough information for the user to determine if the database record is of interest, and if so, the client then requests the full database record. Some server implementations treat a request for a brief record as identical to a request for a full record, and thus return the entire record. In any case, a conformant server implementation must be able to respond to requests for both the full and brief element sets.

A Present Response to the above request follows:

```
numberOfRecordsRet:     1
nextResultSetPosition: 2
presentStatus:          success
records:                list of NamePlusRecord
  name:                 database name
  record:               external
    direct-reference:   1.2.840.10003.5.101
    encoding:           single-ASN1-type
      ANY:              ASCII text of record
etc.
```

This example details the successful return of a single database record, where the database name must be included only with the first record. If a server does not support the requested element set name or preferred record syntax, a well-behaved server should return a failed present status and a non-surrogate diagnostic.[3]

Once the baseline implementation is completed and interoperability has been verified against one or more other implementations, optional functionality omitted from the baseline may be added as needed. Support for Bib-1 attributes, Boolean and proximity operators, multi-database search, piggybacked present, and named result sets can all be incrementally added. The addition of many of these features to a server implementation often depends on the functionality available in the underlying search engine. The client, on the other hand, should not be designed to rely on the availability at any given server of these additional features.

3 Sending Structured Data

In practice, many databases contain records composed of both structured and unstructured information. Often it is useful to be able to convey both structured and unstructured information in a database record to a client. An intelligent client can then make wise use of this structured information, particularly when there is a need to compare common components of the structured information across databases residing on disparate servers over a wide-area network.

Suppose, for example, each state in the U.S. is responsible for maintaining and serving its own database of criminal records, where each criminal record is made up of structured information such as the criminal's name, birthdate, eye and hair color, date of last offense, and some images such as a fingerprint and a photograph. In addition, the criminal record may also contain less structured information, such as a list of prior criminal offenses, police reports, psychological history, etc. An investigator researching a particular crime can then search across any number of these databases and obtain a uniform view of the structured data even though the data is obtained from one more separately maintained servers.

[3]In general, a diagnostic message may appear in place of a record as a *surrogate* diagnostic, or in place of all records as a *non-surrogate* diagnostic.

As another example, consider a storefront database whose records contain items such as product name, product description, cost, and cost unit (e.g. U.S. Dollar, Japanese Yen). A bargain shopper client can be designed to search any number of storefront databases and to locate the top three suppliers providing the best price.

Generic Record Syntax (GRS) is a Z39.50 record syntax used to transfer database records containing any amount of structured or unstructured information from a server to a client. This section provides a brief overview the various components of a generic record, and provides a detailed example of how to extend the baseline implementation to include GRS records. For completeness, refer to Appendix "*RET: Z39.50 Retrieval*" of the Standard for a more thorough examination of this topic.

Elements and Tags

A Generic Record is made up of one or more hierarchically organized elements, where an *element* is a component of a database record. Each element is *tagged*, where the tag acts as an identifier for the element.

The tag associated with each element may be a *numeric* or a *string* tag. When a numeric tag is used, it reflects a common understanding between the server and client regarding the meaning of the element associated with the numeric tag. The Standard provides two sets of numeric tags: a set of tags used for meta-information about the record, called *tagSet-M*, and a set of tags used for generic information called *tagSet-G*. Examples of tags from the tagSet-M include Score and Date of Last Modification, whose numeric values are 18 and 16, respectively. Examples of tags from the tagSet-G include Title and Author, with numeric values of 1 and 2, respectively. See Appendix "*TAG: TagSet Definitions and Schemas*" of the Standard for a complete definition of the tag sets.

In contrast to a numeric tag whose meaning is intrinsically understood by both the client and the server, a string tag conveys meaningful information to the user (not to the client though) regarding the associated element. In practice, a string tag is used for tagging elements that may be only locally known to a particular database or database record. String tags provide an extensible means for including additional structured elements in a database record where the elements are not commonly recognized or well-known. For example, in an encyclopedia database composed of a number of volumes of information, each database record may contain an element with string tag of "volume".

Database Schema

Each database is associated with a *schema* which defines the collection of tags used in the database records. Numeric tags may be selected from tagSet-M or tagSet-G, or alternatively, they may be defined specifically for a given database or set of databases. When databases are designed to share a common schema, even though the databases reside on different servers over a wide-area network, the common structured elements can be meaningfully compared. A database schema for a criminal record or a storefront could easily be defined using tags from tagSet-M and tagSet-G, where applicable, and defining a new set of tags where necessary.

At the time of this writing, there are two published schemas, WAIS (Wide Area Information Servers) and GILS (Government Information Locator Service), that are well-known in practice and are used in a number of databases today. The WAIS schema makes use of tags from the tagSet-M and tagSet-G: title, name, date, rank, score, local control number[4], and URx[5]. It also makes provisions for database-specific tags by allowing arbitrary string tags to be used to define any additional elements of the database record. A client searching across multiple databases that use the WAIS schema can expect to obtain the tags defined in the WAIS Schema, and thus the client can present database records uniformly to the user regardless of which server delivered the database records. The WAIS schema was designed to be general enough for use in most full-text databases.

For the GILS schema, a large number of government agencies have agreed upon a set of data elements and corresponding tags common to government information locator records. A GILS record contains information about a specific source of government in-

[4]The local control number, or record identifier, is an opaque string defined by the server that identifies the record on that local server.

[5]A client may want to use the Uniform Resource Identifier (URx) to identify and remove duplicate records, particularly when a search is performed over multiple databases residing on different servers. For example, if a client is searching two databases containing records gathered from a WebCrawler, there is a greater chance of duplicated records.

formation. The WAIS and GILS schemas share many of the same tags from tagSet-M and tagSet-G, such as title, local control number, and URx. In addition, the GILS schema includes a GILS tagSet, which contains tags such as the originating government agency and government information distributor name, organization and address.

Variants

Database information is often available in a number of display formats, languages, character sets, etc. Using GRS, element data can be made available in one or more variants, where a *variant* is an alternate representation of the same element data. For example, in the criminal database, the police report element may be available in both plain text, MS-Word, and PDF formats. In a multilingual storefront database, the product description element may be available in English, Spanish, and French. Variants provide a mechanism for capturing additional meta-information about the available representations of the element.

For a given element, each variant provided by a server contains a *variant identifier*. The variant identifier serves to distinguish a specific variant from other variants of an element. The variant identifier can be used by the client within a Present Request to specify which variant of an element is requested. The implementation example described below demonstrates the use of the variant identifier to obtain a specific variant of an element of a database record.

Implementation Example

A natural extension to the baseline implementation is the inclusion of a GRS module for delivery of structured and unstructured information associated with a database record. In practice, the delivery of GRS records usually occurs in two main steps. In the first step, the client requests a number of database records, where each record contains only a small set of *primary* elements, such as the title or author, and a *skeleton* of the remainder of the record, which describes any additional elements that are available for retrieval, but does not include the actual data. The primary elements contain enough information about the database record to allow the user to determine if the other elements (described by the skeleton) of the database record are of interest. If a specific element of a database record is of interest, the second step is the retrieval of a variant of an element of a database record. Variations on this basic two-step process are explored further in the next section.

Suppose, for example, a search resulted in a result set of 100 database records. The first step might be to request all the primary elements, a skeleton of the remaining elements, and any available variant information. This is embodied in a request for the "VARIANT" element set (that is, the element set name "VARIANT" is statically defined to mean "primary elements, skeleton of remaining elements, and variant information"). An example of a Present Request follows:

```
resultSetId:               "default"
resultsetStartPoint:       1
numberOfRecordsRequested:  100
recordComposition:         simple
  elementSetNames:         "VARIANT"
preferredRecordSyntax:     1.2.840.10003.5.105
```

where the preferred record syntax is now GRS. If a server does not support GRS or the "VARIANT" element set, it should return a present status of failure and a non-surrogate diagnostic. If the Bib-1 diagnostic code 227 is returned, meaning no data available in requested record syntax, the client may wish to revert to a Present Request with a preferred record syntax of SUTRS. If the Bib-1 diagnostic code 25 is returned signifying that the specified element set name is not valid for the specified database, the client may instead try the "B" element set name.

Suppose that all 100 records are delivered in the Present Response. An example of a Present Response is shown below:

```
numberOfRecordsRet:     100
nextResultSetPosition:  0
PresentStatus:          success
records:                list of NamePlusRecord
  name:                 database name
  record:               external
    direct-reference:   1.2.840.10003.6.105
    encoding:           single-ASN1-type
      ANY:              generic record
  record:               external
etc.
```

A simple example *generic record* obtained from a criminal database is provided below.

Tag Type	Tag Value	Content
2 *generic*	1 *(title)*	"John Doe"
1 *meta*	16 *(dateOfLastMod)*	"19950507080559"
3 *local*	"Fingerprint"	*noDataRequested (NULL)*
3 *local*	"Police Report"	*noDataRequested (NULL)*
3 local	"Photograph"	*noDataRequested (NULL)*

Other tagged elements could have also been supplied depending on what information was stored in the database. The *title* and *dateOfLastMod* contain content, whereas the skeleton elements, the fingerprint, police report, and photograph do not. Instead, the skeleton elements contain meta-data indicating supported variants. The main reason for not returning the content associated with the skeleton elements is that these elements tend to be large, and are often available in more than one variant. Furthermore, the user usually is initially interested in browsing the brief data associated with the descriptive elements of each record, and not the data associated with the content elements of all records.

Suppose the police report is a 705,051-byte MS-Word document. The meta-data for this element would contain a supported variant given as follows:

```
Variant:      triples
  Triple 1:
    Class:    1 (variantId)
    Type:     2 (variantId)
    Value:    variant identifier
  Triple 2
    Class:    2 (BodyPartType)
    Type:     1 (ianaType/subType)
    Value:    "application/ms-word"
  Triple 3:
    Class:    7 (Meta-data returned)
    Type:     2 (size)
    Value:    705051 bytes
```

where *variant identifier* uniquely identifies the variant for this element. It is useful to supply the client with the size of the variant, particularly if the element is large, as is often the case with multimedia data.

A variant may also contain an optionally specified variant set identifier (not to be confused with a variant identifier), which defines the classes, types, and values that make up the variant. Refer to Appendix "*Var: Variant Sets*" for the definition of the Variant-1 variant set, identified by the object identifier 1.2.840.10003.12.1. In practice, it is assumed that the variant set is Variant-1, and thus the variant set identifier is omitted from the variant.

If the client wishes to retrieve the variant associated with this element, an example of a Present Request is specified as follows:

```
resultSetId:                 "default"
resultsetStartPoint:         record number
numberOfRecordsRequested:    1
recordComposition:           simple
  elementSetNames:           variant identifier
preferredRecordSyntax:       1.2.840.10003.5.105
```

For this example, *record number* is the position of the requested record in the result set, and *variant identifier* is the variant identifier string for the "application/ms-word" variant of the "Police Record" element. A Present Response to this request would contain one GRS record, where the record contains one tagged element. The content for the element would contain the MS-Word version of the "Police Record" element.

Tag Type	Tag Value	Content
3 *local*	"Police Report"	*MS-Word Document (octet string)*

Suppose an element larger than the negotiated exceptional record size is requested. In this case, the server returns as much of the element as will fit into the Present Response without exceeding the negotiated exceptional record size. The server also includes, with the element meta-data, a target token: a string created by the server to refer to the next piece of the element. It is specified in GRS using Variant Class 5 Piece, Type 7: target token.

For the client to retrieve the next piece of the element, the element set name of the above Present Request is modified to use the target token.

```
resultSetId:                "default"
resultsetStartPoint:        record number
numberOfRecordsRequested:   1
recordComposition:          simple
  elementSetNames:          target token
preferredRecordSyntax:      1.2.840.10003.5.105
```

The GRS functionality presented in this section describes a simple set of features useful for extending the baseline implementation to send structured data. In addition to the described functionality, GRS includes a rich set of additional features that could be incrementally added to enhance the quality of the structured data, including hierarchically structured records, usage restrictions, and search term highlighting.

4 Requesting Structured Data

There may be times when a client requires greater control over requesting elements of a database record. Suppose for example the client wishes to request the title, author, and date of last modification from a set of database records. This section describes Z39.50 extensions enabling a client to request specific elements of a structured database record.

A constraint imposed by Version 2 is that the record composition in the Present Request must be a simple element set name. One way to allow the client to request multiple elements in a Version 2 implementation would be for the server to define a new simple element set name for the client to use in the Present Request. For example, the server could define a new element set name called "*modzilla*" that returns the title, author, and date of last modification. Unfortunately, this is not a generally extensible mechanism for obtaining any arbitrary set of elements from a database.

Another possibility is to define an element set name that is made up of a list of requested elements separated by spaces. The new element set name would then be called "title author date". This approach is also unacceptable since the element set name now contains implicit structure, which is in violation of the primitive nature required of the element set name.

The ultimate solution is to upgrade the baseline implementation to Version 3, and to use a record composition of complex in concert with Element Set Specification (ESPEC). This enables the client to explicitly request any number of elements from one or more database records.

Upgrading the baseline implementation to Version 3 requires a modification to the Protocol Version parameter of the Init Request and Response. There are a few other minor differences between Version 2 and 3, for example, in the specification of the query and diagnostic record. For the most part, these differences are small and can be easily accommodated.

During the Present Request, the client can explicitly request various components of a database record. An example of a Present Request containing a complex record composition is given below.

```
resultSetId:                "default"
resultsetStartPoint:        record number
numberOfRecordsRequested:   1
recordComposition:          complex
  selectAlternativeSyntax:  true
  generic:
    elementSpec:            externalEspec
      direct-reference:     1.2.840.10003.11.1
      encoding:             single-ASN1-type
        ANY:                element spec
preferredRecordSyntax:      1.2.840.10003.5.105
```

A simple example of an *element spec* used to request the title, author, and date of last modification follows:

```
Espec-1:               elements
  elementRequest:      simpleElement
    simpleElement:     TagPath
      tag:
          tagType:     2 (generic)
          tagValue:    1 (title)
    simpleElement:     TagPath
      tag:
          tagType:     2 (generic)
          tagType:     2 (author)
    simpleElement:     TagPath
      tag:
          tagType:     1 (meta)
          tagValue:    16 (dateOfLastMod)
```

As with the baseline and GRS modules, the implementation of the ESPEC module could later be extended to include support for additional features of ESPEC, such as requests for specific variants of an element, hierarchical elements, wild things, and wild paths.

5 Explaining the Server

When a client encounters a new server for the first time, it is useful to be able to probe the server, for example, to obtain a list of available databases, or a list of search attributes or retrieval elements available for particular database. These capabilities are particularly important for full-text databases where search attributes and record structure may differ from database to database. This section describes how the Z39.50 Explain Facility can be used to obtain information from a server. It describes how an implementation of the Explain Facility can be developed on top of the baseline implementation and incrementally extended as needed.

The implementation of the Explain Facility is a logical extension of the existing search and present services of the baseline implementation. It requires the addition of a new database, called "*IR-Explain-1*", a new set of search attributes (Exp-1), and a new record syntax (Explain). Obtaining server information amounts to formulating a Type-1 query using the Exp-1 attributes, searching the IR-Explain-1 database, and retrieving Explain records.

Explain is made up of 15 categories, each of which provides different information about the server. The *TargetInfo* category, for example, supplies general information about the server, and the *DatabaseInfo* category supplies database-specific information. Because each category can be implemented independently, there is no need to provide support for all categories, and new categories can be added as needed. For interoperability, the *CategoryList* category provides a convenient mechanism for a client to determine what categories are supported by a server.

Below is an example of a non-piggybacked Search Request of the IR-Explain-1 database. The query uses the Exp-1 attribute set, and requests the CategoryList category from the Explain database.

```
smallSetUpperBound:          0
largeSetLowerBound:          1
mediumSetPresentNumber:      0
replaceIndicator:            true
resultSetName:               "default"
databaseName:                IR-Explain-1
query:                       Type-1
  attributeSet:              1.2.840.10003.3.2
  rpn:                       Operand
    attrTerm:                AttributesPlusTerm
      attributes:
        attributeElement:
          attributeType:     1 (Use)
          attributeValue:    1 (ExplainCategory)
      term:                  "CategoryList"
```

The above Search Request should result in at most a single database record. An example of a Present Request for this record follows, where the preferred record syntax is Explain.

```
resultSetId:                 "default"
resultsetStartPoint:         1
numberOfRecordsRequested:    1
recordComposition:           simple
  elementSetNames:           "B"
preferredRecordSyntax:       1.2.840.10003.5.100
```

A Present Response to the above request is:

```
numberOfRecordsRet:      1
nextResultSetPosition:   0
PresentStatus:           success
records:                 list of NamePlusRecord
  name:                  database name
  record:                external
    direct-reference:    1.2.840.10003.5.100
    encoding:            single-ASN1-type
      ANY:               explain record
  record:                external
etc.
```

where *explain record* is composed of a list of the categories supported for this server.

An example of an Explain record containing the CategoryList category is shown below.

```
explain record:       category list
  category list:
    category info:
      category:       "CategoryList"
    category info:
      category:       "TargetInfo"
    category info:
      category:       "AttributeDetails"
    category info:
      category:       "ElementSetDetails"
```

From the information obtained in the CategoryList, a client can determine what other categories are supported by the server. In the example shown, the TargetInfo, AttributeDetails, and ElementSetDetails categories can now be obtained from the server. If the server were to add support for additional categories at a later time, the client would be able to determine this the next time it retrieves the CategoryList category.

6 Additional Extensions

In summary, this paper has described a baseline implementation of Z39.50 and how to incrementally extend this baseline. Other features of Z39.50 can also be implemented as modular extensions. For example, if database security is a concern, the Access Control Facility can be added without the need to modify the original baseline (other than updating the Options in the Init Service). Similarly, if sorting a result set is a requirement, the Sort Facility can be implemented and included as a separate module. Because the capabilities are negotiated during the Init, if a client or server does not support a particular capability, interoperability is still guaranteed.

Basic Z39.50 Server Concepts and Creation

John A. Kunze

8 August 1995

University of California at Berkeley
and
U.S. National Library of Medicine
jak@violet.berkeley.edu

ABSTRACT

The Z39.50 protocol is a standard network language for searching and retrieving records from remote databases. The Z39.50 client/server session model provides multiple abstract views of records, depending on whether searching, retrieval, or element selection is taking place. The underlying network stream that carries user queries, database records, diagnostics, and protocol control information is structured according to the Basic Encoding Rules (BER) as applied to human readable specifications written in the Abstract Syntax Notation, ASN.1.

Several important decisions face creators of Z39.50 servers. Building the BER transport mechanism may be done from scratch or with software tools compiled from ASN.1. Developing a model for managing result sets (records resulting from a query) is required by the stateful nature of Z39.50. A switch may need to be designed to route a query to one of several database engines for resolution, depending on which engine or database management system administers the database(s) being searched. In order to respond to search cancel requests, a server's input system must be at least partially asynchronous. Performance requirements may favor a multi-threaded design over a simpler single-threaded design.

Introduction

This paper leads network programmers through the basic concepts and steps in setting up a networked server that conforms to the Z39.50 standard for searching and retrieving records from remote information systems. Section 1 deals with concepts and section 2 with creating the server.

The reader is presumed to be familiar with the Z39.50-1992 specification [1] of the standard with which we will be primarily concerned. Access to the appendices of the Z39.50-1995 specification [2] will also help round out some concepts that are used but not made explicit in Z39.50-1992. This paper supplements the standard with an eye to helping an implementor build a simple server.

1. Z39.50 Concepts

1.1. The Main Parts of a Z39.50 System

A *user* interacts with a computer program, the *client*, which exchanges network messages with a remote computer program, the *server*. The client acts on behalf of a user, which is often a person, but is sometimes another program, for example, a CGI script [3] that converts requests received by an HTTP [4] server into requests suitable for a Z39.50 server (here the CGI script functions as a so-called gateway). The server acts on behalf of one or more information providers. So far this describes any number of networked client/server systems, but a critical distinguishing feature of a Z39.50 system is the network messaging language, or *protocol*. In this paper we will focus on the server of a Z39.50 protocol system.

A Z39.50 server program is itself a system of three main parts. It contains a *protocol engine* which manages the reading and writing of Z39.50 network messages, known as PDUs (Protocol Data Units). The server protocol engine is called to perform network input or output by the *control module*, which routes requests to and responses from one or more *database engines*. A database engine executes queries, creates search result sets, and stores them for the purpose of returning records on demand, often relying on a DBMS (database management system) underneath it.

For some implementations it may be a goal to keep the protocol engine, control module, and database engine independent. In practice, however, this is difficult because generalizing an access paradigm to connect multiple communication styles up to multiple search systems tends to require costly simplifying assumptions or complex conversion mechanisms. To the extent that independence is achieved, it becomes easy to re-use the protocol engine with different DBMSs and to make a given database engine accessible on the network via multiple protocols.

1.2. Basic Z39.50 Session Activities

A complete Z39.50 session may be anything from a single request/response exchange of PDUs (network messages) to a complex series of exchanges to refine a set of search results before retrieval. The Z39.50 session itself is called an *association*, and it takes place over a network connection whose set-up and tear-down are described outside the standard in [5]. Note that this description ([5]) is not the one to which the standard directs the reader, but it does match the prevailing Internet implementation environment. While Z39.50-1992 was conceived in an OSI framework [6], current practice calls for a TCP/IP transport [7]. This means that most implementors disregard Z39.50-1992 references to the OSI concepts of Association Control Service Element and Presentation Context (most of which have been eliminated in Z39.50-1995). Important concepts from OSI that remain valid for implementors are ASN.1 [8] and BER [9], whose roles are described below.

Once a TCP connection is set up, a Z39.50 session is established with a client Init request followed by a server response indicating that the session connection is accepted. A session may be terminated simply by closing the TCP connection.

A client Search request contains a query that the server executes to create a result set of records satisfying the query. A subsequent client Present request asks the server to return some of the result set records, selecting elements and structural layout according to certain specifications. The server returns records in a Present response. As an optimization, an initial subset of records may be returned with the Search response, a feature informally called "piggybacking."

In formulating a response that includes records (such as for Present), a server performs Element Selection, which is a process of deciding, often under client direction, how to constitute a record before returning it. The full record contains all relevant information, including potentially large elements, such as an image or the full text of a document. Element selection allows a client to glance at many records through relatively small summary element sets before requesting the full element set for those records only that the client user wants to look at more thoroughly. The idea is to minimize network transmission by keeping the summary records small enough for bulk transfer while anticipating that the user, basing decisions on perusal of summary information, will request only a few fully constituted records.

Init, Search, and Present are the only top level protocol features needed for the basic server. No advanced features such as Access Control and Resource Control are required here, nor are any features specific to Z39.50-1995, such as Explain and Segmentation. All features described here conform to Z39.50-1992 and should work in Z39.50-1995 implementations as well.

1.3. The Abstract Record

The term *record* in Z39.50 always refers to an abstraction of a record that a server makes visible to the world via Z39.50. By never referring to the internal database record structure, Z39.50 avoids confining applicability to any one particular DBMS, but this then requires that the server implementor develop a map between it and the abstract record. An *element* is a component of a record. There are few restrictions on elements. For example, a record may contain any number of elements, including zero, and its elements may overlap or repeat.

Similarly, there are few restrictions on records. There is, however, an important unstated assumption that any two records from the same database have a similar configuration of elements. While wildly different sets of elements between two records would leave a Z39.50 server operational, it would undermine predictability for the client, particularly in the area of element selection.

For the purpose of this discussion, a server database record has three *personas* − for Search, Element Selection, and Present − and each persona comes in several choices. This complexity is needed to accommodate the different ways that a record can or might be broken down, indexed, re-arranged, and displayed. Again, the data personalities that a server chooses to show are purely abstractions for external consumption via Z39.50; they imply nothing about layout or composition of the internal database records behind them.

1.3.1. The Search Persona

A specific choice of Search persona defines Query semantics. This is a collection of traits, accessible elements, and expected behaviors for a database when running a Query. For our purposes, Query semantics are given by a text describing (a) some numbers to use in referring to each data element that might be searched and (b) a description of those elements. The text may be seen as a table with one row per element; each row lists a concept and the number(s) that Z39.50 needs to transmit a reference to that concept.

This kind of table is called an *attribute set*. Different attribute sets show different public search points for the same data. An attribute set called Bib-1 was originally designed for searching bibliographic data. Another attribute set called STAS-1 [10] is used for technical and scientific data (it imports Bib-1). For the basic server implementor, designing a search point table for each database using the Bib-1 attributes is completely adequate. The Bib-1 attributes are listed in appendices of both the Z39.50-1992 and Z39.50-1995 specifications.

1.3.2. The Element Selection Persona

The Element Selection persona assumes a particular division of a record into tagged elements, from the point of view of retrieval. It is a collection of traits, accessible elements, and expected behaviors for a database when building up a retrieval record from elements of the abstract record. In its fullest form as specified in Z39.50-1995, it is given by both a text that dictates general structural rules (such as how element hierarchies are formed) and, most importantly, a table listing each element tag next to a description of it. Such a table is called a *tag set*.

Element selection takes place prior to laying out a record for return. This is when a server, often under client direction, decides which of the available elements to include. A client may request element selection using an element specification that contains either a named set (such as ''F'' for Full), a sequence of element tags, or both. Whether to include an element is ultimately up to the server and depends on several factors, notably on delivery constraints dictated by the requested Present persona (which is essentially the record syntax, described below).

This discussion assumes the simplified element selection mechanism described in Z39.50-1992, which allows only named element sets but not individual element tags. In particular, the basic server need only define for each database its own element sets corresponding to the names

F	(Full) all available record elements, and
B	(Brief) restricted set summarizing record.

1.3.3. The Present Persona

A specific choice of Present persona defines Information semantics. This is a collection of traits, accessible elements, and expected behaviors for a database when requesting delivery of records. For our purposes, Information semantics are given by (a) a table describing the kind of information within each data element that may be included in a returned record and (b) a text describing the actual bit-level, serialized layout (that is, in a network data stream, not in memory) of those data elements. They are called, respectively, (a) an Abstract Syntax and (b) a Transfer Syntax.

A particular combination of these two is called a *record syntax*. Different combinations provide different retrieved views of the same data. Note that data elements that are searched may bear little relationship to elements that are returned.

The potential complexity in all this generality is mitigated by the small number of abstract syntaxes and the common practice of employing only one transfer syntax per abstract syntax. The simplest is SUTRS (Simple Unstructured Text Record Syntax), which consists of one string designed to hold multiple lines of text formatted by the server, thereby greatly easing the display burden for the client software. Another common syntax is USMARC (United States MAchine Readable Catalog) [11], used in many bibliographic systems. Basic servers that deliver either or both of unstructured text and bibliographic records need only consider supporting these two syntaxes.

1.3.4. Merging Personas

One consequence of the abstract record having multiple personas is a powerful separation of Search, Present, and Element Selection functions that enables the kind of task-specific semantic mapping among elements that large scale systems require. Another consequence, however, is that the split personas define no sense of element equivalence between personas unless the attribute sets and tag sets are defined to draw an explicit relationship.

For example, in some systems a query on an Author (Search persona) might return records with an Author element (Present persona) containing data that appears unrelated to the query term. This may be because the match succeeded on other elements that the server deemed related to Author, or because the user's term was mapped to a synonym that matched (e.g., Mark Twain may return Samuel Clemens). In some information domains exact element equivalence across personas would be useful.

One goal of STAS-1, which names both an attribute set and a tag set, is to maintain element identity between personas. While it does not require every search, selection, and retrieval element to carry the same tag for each database, it does allow the database provider to preserve the correspondence whenever that may be meaningful.

1.4. The Roles of ASN.1 and BER

The text describing the Abstract Syntax (section 1.3.3a above) may include an abstract record structure specified in ASN.1 (Abstract Syntax Notation 1) [8]. ASN.1 is not much more than a scheme for writing down the kind of data structuring and typing information afforded by most programming languages, but it is abstract in that it is independent of programming language and machine architecture.

The text describing how the Transfer Syntax (section 1.3.3b above) is used to represent the abstract syntax may not be needed because as long as there is an ASN.1 specification, the bit-level serialized layout can be derived from it. The rules for deriving the layout in Z39.50 are called BER (Basic Encoding Rules) [9]. The abstract syntax for SUTRS, simple as it is, consists of exactly one ASN.1 InternationalString, which is a kind of character string that holds any number of text lines. One particular abstract syntax that does not include an ASN.1 specification, but instead relies on a separate text to describe the transfer syntax, is USMARC [11].

From the point of view of the programmer, ASN.1 is not directly used by a running system, but instead primarily affects the system under construction. It influences decisions as to what real data structures will hold the elements coming from and going to the serialized network data stream. Of particular importance is the core Z39.50 ASN.1 protocol specification as a set of PDUs, since they contain all other structures, including Queries and Records. Programmers study these abstract PDUs closely when writing the protocol engine that interprets and builds the corresponding real PDUs.

Encoding and decoding subroutines have to be written that convert data between real structures and the serialized stream format dictated by BER. In some implementations, an ASN.1 compiler generates program code for both the data structures and the conversion routines. More discussion of this subject appears in section 2.2.

A strength of ASN.1 is that it provides a clear, machine independent way to express structuring and ordering of protocol elements. The BER algorithms ensure that arbitrary hierarchical data structures in text or binary will be transmitted over a serial byte stream without loss of information. On the other hand, because the encoded stream itself is binary, it cannot easily be entered from a keyboard or output directly onto a display device. This makes it harder to tinker with Z39.50 servers than with some other servers. For example, much of an HTTP [4] server can be tested by simply establishing a terminal session with the server and typing in HTTP protocol client requests (which are text-based) from the keyboard.

1.5. The Z39.50 Query

The protocol allows for several different query types, but for a basic server it is adequate to support only the Type-1 query, which we refer to as the Query. Accompanying the Query in a Search request is a list of database names. Inside the Query is an attribute set name plus a boolean expression tree, each leaf of which is either a result set name or an actual search term list. Often the tree received consists of just one leaf that contains a single term list corresponding to a straightforward search, one that might, for example, be expressed by a traditional-looking command sequence such as `find author="Mark Twain"`. Non-leaf tree nodes (branches) indicate one of the boolean operations AND, OR, or AND-NOT, where the three children are, respectively, the left operand, right operand, and operator. For historical reasons the Type-1 query is also known as the Reverse Polish Notation Query, or RPNQuery.

The heart of the Query is the search term list, which is a Term (one or more words) and an indefinite-length list of attributes that the client bundles with the Term. Each attribute consists of two numbers: a type and a value. They identify, respectively, an attribute category and subcategory. In the Bib-1 attribute set, for example, the attribute 1,30 associates the Term with a Use(1) category of Date(30). In another attribute set the attribute 1,30 might mean something other than Date. Here is a sample search term list.

```
Mark Twain
1,1003  4,1  5,100  3,1  6,1  2,3
```

For completeness, this particular list includes one attribute from each of the six categories, though often a client omits several categories. Taken left to right, the attributes (integer pairs) identify a search access point (search index) for which a string of words, ''Mark Twain'', submitted in a Query will be treated, respectively, as an

```
author,
with words structured as a phrase,
no truncation,
occurring at beginning of a field,
not needing an entire subfield,
and compared for equality.
```

1.6. Statefulness, Complexity, and Z39.50

Z39.50 is one of several protocols that allows a client program to transmit user queries to a remote server program and to receive server responses, the ultimate aim being to display results to the user. Unlike several well-known stateless protocols, such as HTTP and Gopher [12], Z39.50 is *stateful*, in the following sense. A server using a stateless protocol (such as HTTP) treats each request as if from a client with which it has never communicated before; in other words, it maintains no memory, or *state*, regarding the client. In contrast, a stateful protocol (such as Z39.50) is conducted over a session for which the server keeps track of things like user identification and search results as they accumulate over the course of the session.

One obstacle facing every implementor is the perceived complexity of the standard. Z39.50 probably owes this perception to three factors: (a) it is a formal national and international standard [13], (b) it grew out of the library automation community, whose highly methodical approach to storing and indexing information might not be immediately appreciated by the non-library-oriented implementor, and (c) it contains references to the stunningly comprehensive ISO OSI [6] layered network model.

Experience with the standard usually reveals the essential simplicity beneath its densely detailed specification. Z39.50 is stateful and it is complex. These two facts may be viewed as weaknesses or strengths. Without promoting either view it may be said that Z39.50 was designed to solve a complex problem and that stateless protocols were designed to solve simpler problems.

2. Creating a Z39.50 Server

2.1. Before Starting

You need to ask yourself whether you want to build a server from scratch or on top of an available software base. At the time of writing, server packages were freely available from the Clearinghouse for Networked Information Discovery and Retrieval (CNIDR) the National Library of Canada, and the University of California at Berkeley. For information on how to obtain them, you may access the World-Wide Web "Z39.50 Pointer Page" [14]:

```
http://ds.internic.net/z3950/z3950.html
```

Whatever you decide in creating your own server, it is recommended that you track protocol development and establish contact with other implementors. One way to start becoming involved is to subscribe to the Z39.50 Implementors Group (ZIG) mailing list, `z3950iw@nervm.nerdc.ufl.edu`. To do so send an e-mail message to `listserv@nervm.nerdc.ufl.edu` with the body of the message containing

`sub z3950iw` *your_first_name your_last_name*

An official register of Z39.50 implementors [15] is available and you may wish to have your organization's name listed in it.

2.2. Whether to Use an ASN.1 Compiler

An important decision is whether to build your own BER encoding and decoding routines or to have an ASN.1 compiler build them for you. It would do so by translating the ASN.1 specification for PDUs, record syntaxes, queries, etc. into program source code. You may wish to consider the following issues in reaching this decision.

The problem to be solved is translating Z39.50 PDUs, which encapsulate all other data, between their network format and the form in which the server programmer can make use of them via program variables. This amounts to making the coded byte stream conveniently available to the internal memory of a running server program, a process called decoding. Most of what follows about decoding applies in a straightforward way to the inverse process, called encoding.

Decoding is generally done in three steps (as is encoding). First, a PDU originally encoded by the client is read into a server buffer as a contiguous sequence of bytes that includes a header from which the decoder can deduce when the last byte of the PDU has been received. Second, this flat form of the PDU drives construction of a generalized tree that reveals the hierarchical structure inside the PDU buffer. Finally, the leaves of the tree are explored to discover which actual PDU elements have been received.

ASN.1 compilers generate data structure definitions and source code for high level programming languages (such as C or PL/1). Generally each abstract structure definition produces both a real data structure definition (e.g., an ASN.1 SEQUENCE becomes a C `struct`) to contain the corresponding PDU element and a decoding routine (plus another for encoding).

To perform the last step above, the programmer calls a compiler-generated, top level general PDU decoding routine, which in turn calls the appropriate specific PDU decoding routine, which calls other routines, and so forth depending on what is found at each branch and leaf of the tree. At the end of this process, what is left is (a) a PDU buffer, (b) a tree whose leaves point to individual PDU elements, and (c) another tree of structures corresponding closely to the ASN.1 specification and containing fully decoded leaf elements. (With a clever compiler the leaves of both trees will point back into the buffer since it is expensive to make and keep copies.)

An advantage of using an ASN.1 compiler is that each PDU is rigorously checked for syntactic correctness and the PDU is fully decoded in one fell swoop. Another possible advantage is that the decoding routines can be re-created automatically when the ASN.1 specification changes. Since the programmer must still alter by hand the server code that references the changed structures, this advantage would be certain if routines built by hand used a second tree of structures just as the compiler-generated routines do (c, above). In practice, however, hand-generated routines only use the one generalized tree (b, above), so the amount of code that needs changing in either case is roughly equivalent.

A disadvantage of using an ASN.1 compiler is that it can be very inflexible with regard to experimental elements or element sequences not given by one unified ASN.1 specification (such as when a server supports Z39.50-1992 and Z39.50-1995 simultaneously). It may be hard to make your compiler ignore unknown PDU elements, and when it rejects a PDU sometimes recovery is impossible. Another situation in which recovery can be difficult using compiler-generated code is when an ASN.1 structure (such as a record in the Generic Record Syntax) spans more than one record, which will likely happen one day if you support full Z39.50-1995 Segmentation.

The advantage of rigorous syntax checking becomes less significant in mature interoperation environments where the majority of errors will be semantic. Besides the time and space used to build and maintain a second tree (c, above), there is also a potential inefficiency in decoding everything at once because received elements often go unused.

Even if you use an ASN.1 compiler, becoming familiar with the rudiments of BER can help you understand how to use your internal data structures best and how to read PDU log files when debugging. An excellent package of low-level BER routines is freely available from OCLC [16] for implementors writing their own encoding and decoding routines. Two ASN.1 compilers that are freely available are SNACC [17] and ISODE's *pepsy* [18].

2.3. Getting Started Online

Before you can bring up a server on the network, you will need to locate a set of TCP tools. On many platforms they are already provided with your operating system (e.g., the UNIX socket library).

It is also imperative to locate a Z39.50 client with which to test your server as you build it. Unless you build your own client as well, you may wish to read [14] for information about freely available clients. Existing servers that are known to be functioning correctly can be valuable for gaining comparison experience and simple reality checks.

Once you have any sort of server program ready to test (e.g., just to test Init), you will need to make it ready to accept incoming client connections. One way is to have the server code itself listen on a particular TCP port, and then have your client try to make the network connection to that port from either the same computer or a different computer. Another way is to use an existing ''super-server'' that listens on a number of ports and starts up your server upon sensing an incoming connection to a port that you will have specified in advance.

This second way is particularly useful in the UNIX environment because it allows the server code to be written without having to know whether its input and output are to a socket, a file, or a terminal; this can be useful for debugging, when you may want to start your server by hand without reconfiguring the super-server. Under UNIX the super-server is called the *inetd* daemon and the way to make your server known to it is to add an entry such as

```
z39.50 stream tcp nowait nobody
     /usr/local/irserver irserver
```

(all on one long line) into the system file */etc/inetd.conf*. The standard TCP port for Z39.50 is 210, so under UNIX, for example, you can make this fact known to the system by adding the line

```
z39.50   210/tcp   ir
```

to the file */etc/services*. Setting up the underlying TCP connection involves straightforward coding consistent with widely available HTTP servers.

2.4. Design the Control Module

Two main issues to resolve early are (a) to what degree the server's input will need to be asynchronous and (b) whether the server will be single-threaded or multi-threaded. A server is asynchronous if it can detect and act upon the arrival of a new request before it finishes processing the current request. A server is single-threaded if at most one client connection is active at a time.

For a basic server without Access Control or Resource Control, the easiest design is purely synchronous and single-threaded. Highlights of the simple synchronous server control scheme are:

1. Block program until request PDU arrives.
2. Execute request and formulate response.
3. Send response PDU and go back to step 1.

In a basic server program the scheme will probably be fleshed out with a simple timeout and checks for termination and errors. The following somewhat over-simplified C program fragment illustrates this. It uses the UNIX `select(2)` system call [19] to wait no more than a specified timeout period for input to arrive.

```
. . .
for (;;) {
   /* for select, clear bit mask for read */
   /* file descriptors; set our input bit */
   FD_ZERO(&rfds); FD_SET(input, &rfds);
   maxd = input + 1;  /* last bit to check */
   if (!select(maxd, &rfds, 0, 0, timeout)) {
      printlog("read timed out");
      exit(1);
   }
   /* beware: timeout only guaranteed we */
   /*      would get one byte of the PDU */
   switch (getPDU(input, &request)) {

   case END_OF_INPUT:
      printlog("end of session");
      exit(0); /* normal termination */
   default:
   case NOT_A_PDU:
   case UNSUPPORTED_PDU:
      printlog("garbage or unsupported PDU");
      exit(1); /* session abort */
   case INIT_REQUEST:
      printlog("init");
      status = init(request, &response);
      break;
   case SEARCH_REQUEST:
      printlog("search");
      status = search(request, &response);
      break;
   case PRESENT_REQUEST:
      printlog("present");
      status = present(request, &response);
      break;
   }
   if (status != OK) {
      printlog("internal error");
      exit(1);
   }
   putPDU(output, response);
}
. . .
```

If you are strictly interested in the basic server you may skip to the next section. If you intend to enhance your server beyond the simple synchronous scheme it makes sense to plan early. Some asynchronous ability will be needed if you intend to allow canceling a search in progress, as this requires acting on a TriggerResourceReport request from the client while the database engine is toiling away. Detecting the arrival of a byte of input is easy, but detecting that what arrived was a triggering PDU is harder. It requires that the server be able to set aside a PDU that turns out to be a non-triggering PDU, in other words, to put it in a queue. This could be the case if two or more request PDUs arrive back-to-back (a feature allowed in Z39.50-1995).

Also, a complex subsystem such as a database engine cannot simply be interrupted and made to return from an arbitrary program instruction. It can, however, return from various states in the subsystem where the programmer is willing to insert a check (such as testing a global variable set by the control module upon arrival of a relevant PDU) for a cancel so that the cleanup and retreat, if necessary, may be orderly.

Keeping a queue for PDUs is indispensable in the multi-threaded case. The main reason for implementing a multi-threaded server on computers where a single-threaded design is also an option is to improve performance. It is relatively easy to design and run a single-threaded server: the process starts up when a client attempts a connection, is used exclusively by that client, and is then terminated when that client releases the connection. This carries with it the cost of creating and destroying each server process, which becomes more significant as the frequency of connections rises. At such times having only one server process that is multi-threaded to handle many connections becomes appealing.

The drawback is that careful error monitoring, strict memory usage accounting, and general programming quality become critical, because a program abort now affects many users, not just the user whose request caused the abort.

2.5. Design the Protocol Engine

The protocol engine's job is to read and write PDUs under direction from the control module. It needs to keep track of the evolving protocol state as PDUs are sent and received. For example, it might provide a check on the control module to prevent a Present response from being attempted when a Search response is called for. It might also look for sundry protocol violations, such as inconsistent or illegal parameters (such as conflicting values for smallSetUpperBound, largeSetLowerBound, and mediumSetPresentNumber).

Session tear-down can seem anti-climactic to the implementor since there is nothing for the protocol engine to do but return. Because the basic Z39.50-1992 system has no access to the Z39.50-1995 Close request, normal and abnormal termination may be indistinguishable. If the server drops the TCP connection before the client drops it, that is a server abort. If the client drops the connection when it is waiting for a server response, that is a client abort. But if the client drops it when the server is waiting for a request PDU, it is impossible to know if that was a client close or client abort.

2.6. Design the Database Engine

A database engine, possibly one of several, is called by the control module to create the result set for a Query and to retrieve records from result sets. It is the ultimate arbiter of all questions regarding what is and is not supported by the server for a given database, mostly dependent on the DBMS underneath it. This means that while the protocol engine can screen out client protocol errors on a syntactic and superficial semantic level, all other interpretive actions, including most error situations, are rightly the domain of the database engine. Because each database engine will have different capabilities, too much protocol engine error checking could pre-empt DBMS functionality.

The database engine is responsible for implementing all aspects of the record personas mentioned earlier. It defines which attribute sets, tag sets, element sets, abstract syntaxes, and transfer syntaxes will be supported for each database. It must therefore keep tables that map actual database elements to

(a) search access points – to build indexes and recognize incoming attribute combinations,

(b) element selection tags – to choose the elements for the element sets Full and Brief, and

(c) element return tags – to identify elements returned in records (such as field tags for USMARC).

One tricky question is whether the server will handle more than one database per search, and as before the answer must be left up to the database engine in question. Which engine to ask can itself be a problem when two databases in the search are managed by different engines, but in all probability the combination will not be supported. If only one database engine is involved, all databases in the requested combination will need to support the particular persona (as specified via PDU parameters) that the client is approaching. This may be easy when searching a set of related archives, for example, but difficult when searching a personnel directory in combination with a chemical database.

2.7. Process Init

When an Init request PDU arrives, the server must follow up by sending an Init response indicating either acceptance or rejection. For many servers, this is a formality because not much useful feature or buffer information is gained in the process. Many servers and clients currently interoperate well only with Z39.50-1992 Search and Present, regardless of what other features are negotiated. Those that are able to allocate I/O buffers dynamically have little need for negotiated buffer sizes.

The idAuthentication Init parameter, however, is of particular interest to servers that need authentication without using Access Control. If it is received, tagged as an ASN.1 VisibleString and containing two text substrings separated by a slash character ('/'), the first substring is taken to be a userid and the second to be a password.

2.8. Process Search

When a Search request PDU arrives, the control module finds the database engine that administers the databases in the database list and hands the Query over to it. The database engine then decides if it supports the requested database combination.

A model for result sets must be developed. This includes the concept of intermediate result set, which is the set of records matching a subexpression within the Query. Intermediate result sets are needed during Query evaluation, which culminates in the creation of the top level result set named in the search request. As a service, the server may wish to make them available after Query completion (a feature supported in Z39.50-1995), which means that they must occupy the same name space as the client-named sets. It is also helpful to keep track of which top level set caused the creation of which intermediate sets. For the basic server there is probably no need to support intermediate result sets.

You will have to decide how result set existence will be communicated among the various server modules that require it (e.g., a global linked list might be used). Some servers may support result sets that persist between sessions (the Extended Services of Z39.50-1995 support this) in order to allow connections from stateless gateways (such as from HTTP) to retrieve records resulting from a search in a prior session.

Finally, the server must be prepared to handle a Search request that returns some result set records piggybacked onto the Search response. It makes sense to structure this record-retrieving function so that it may be re-used when processing a Present request.

2.9. Process Present

A Present request arrives designating a range of records to retrieve. Obtaining records from a result set is usually done through the intermediary of the DBMS that created it. For even though the search that created it is over, the DBMS cannot relinquish control since the set might be referenced in a subsequent boolean search expression. Besides, it is usually too expensive to externalize DBMS records except on client demand. If other program modules (such as other DBMSs) need to know whether a result set exists, the DBMS that created it will have to externalize that fact somehow.

Most of the problems with Present will already have been solved if you implemented piggybacked records for the Search response. This includes handling element sets and record syntaxes.

To support retrieval of SUTRS, the server need only render a set of record elements as text with a maximum line length of approximately 72 characters. The maximum is approximate in order to give clients an idea of what length to plan for, but not to preclude the possibility of the occasional long line for which the server has decided that preserving data integrity outweighs aesthetics (e.g., not breaking a long row of a formatted table). To support the USMARC record syntax, you will need to refer to the various standards comprising USMARC [11].

2.10. Did You Get It Right?

Your server is essentially ''right'' if it interoperates with three independently developed clients. If you do not have access to as many clients as you would like to test, you may wish to invite connections from implementors by sending a message to the ZIG mailing list (section 2.1).

To prepare for test or public access to your server, you will want to write a server description document defining server parameters such as Internet hostname, port number, and listing available databases. For each database, describe the attributes, element sets, and record syntaxes supported. Once in the hands of client users, that description opens up your Z39.50 server to the Internet.

3. Conclusion

This paper has covered a number of key ideas behind the Z39.50 international standard protocol for searching and retrieving records from networked information systems. These include the way that Z39.50 allows database records to adopt many different personas depending on whether the current operation involves search, retrieval, or element selection. The abstract form of each Z39.50 PDU is given by an ASN.1 description that the implementor uses to write software converting PDUs between internal memory and the serialized byte stream encoding dictated by BER.

Implementors of Z39.50 servers need to consider several issues carefully. One of these is whether to implement from scratch or on top of existing software packages for ASN.1 and for BER, if not for Z39.50 itself. The server control module will have to be conceived with models in mind for result set management and database engine switching. A server may be synchronous or asynchronous, or single- or multi-threaded. These are among the many decisions that will have an impact on server functionality and performance.

4. References

[1] ANSI/NISO Z39.50-1992, *Information Retrieval Service and Protocol: American National Standard, Information Retrieval Application Service Definition and Protocol Specification for Open Systems Interconnection*, 1992. ftp://ftp.cni.org/pub/NISO/docs/Z39.50-

1992/www/Z39.50.toc.html (also available in hard copy from NISO Press Fulfillment, P.O. Box 338, Oxon Hill, Maryland 20750-0338; phone 800-282-6476 or 301-567-9522; fax 301-567-9553).

[2] ANSI/NISO Z39.50-1995, *ANSI Z39.50: Information Retrieval Service and Protocol*, 1995. http://lcweb.loc.gov/z3950/agency

[3] NCSA CGI, *The Common Gateway Interface*, National Center for Supercomputing Applications. http://hoohoo.ncsa.uiuc.edu/cgi/overview.html

[4] Berners-Lee, T., *Hypertext Transfer Protocol (HTTP)*, CERN, November 1993. http://ds.internic.net/internet-drafts/draft-ietf-http-v10-spec-00.txt

[5] RFC 1729, *Using the Z39.50 Information Retrieval Protocol in the Internet Environment*, December 1994. http://ds.internic.net/rfc/rfc1729.txt

[6] ISO 7498, *Open Systems Interconnection – Basic Reference Model*, International Standards Organization.

[7] RFC 793, *Transmission Control Protocol*, September 1981. ftp://ds.internic.net/rfc/rfc793.txt

[8] ISO 8824, *Information Processing Systems - Open Systems Interconnection - Specifications for Abstract Syntax Notation One (ASN.1)*, Omnicom, Inc., Vienna, VA, 1987.

[9] ISO 8825, *Information Processing Systems - Open Systems Interconnection - Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1)*, Omnicom, Inc., Vienna, VA, 1987.

[10] STAS-1, *Scientific and Technical Attribute Set*, Maintenance Agency, CNIDR. http://stas.cnidr.org/STAS.html

[11] Network Development and MARC Standards Office, *USMARC Concise Formats for Bibliographic, Authority, and Holdings Data*, Cataloging Distribution Service, Library of Congress, Washington, DC, June 1988

[12] RFC 1436, *The Internet Gopher Protocol*, University of Minnesota, March 1993. ftp://ds.internic.net/rfc/rfc1436.txt;type=a

[13] ISO 10162/10163 International Organization for Standardization (ISO). Documentation – Search and Retrieve Service/Protocol Definition, 1992.

[14] http://ds.internic.net/z3950/z3950.html, *Z39.50 Resources – a Pointer Page*, Robert Waldstein, wald@library.att.com.

[15] Register of Z39.50 Implementors, available from the Z39.50 Maintenance Agency, Ray Denenberg, ray@rden.loc.gov. http://lcweb.loc.gov/z3950/agency

[16] *OCLC BER Utilities*, Ralph LeVan, rrl@oclc.org. ftp://ftp.rsch.oclc.org/pub/BER_utilities

[17] SNACC, *Sample Neufeld Asn.1 to C Compiler*, University of British Columbia. ftp://ftp.cs.ubc.ca/pub/local/src/snacc

[18] *ISODE pepsy ASN.1 compiler*, ISO Development Environment Consortium. ftp://ftp.psi.com/isode

[19] UPM select, *Select – synchronous I/O multiplexing*, UNIX Programmers' Manual.

The name UNIX is a trademark of AT&T.

Building A Z39.50 Client

Ralph LeVan

OCLC Online Computer Library Center Inc.

6565 Frantz Rd.

Dublin, OH 43017

email: rrl@oclc.org

Abstract

The core functionality for a Z39.50 Client Application is described. This core functionality consists of Connection, Initialization, Search, Present and Disconnection. A Z39.50 Client API is described which provides the core functionality. Also included are brief descriptions of TCP/IP, the abstract syntax ASN.1, BER records and USMARC records. Code for implementing the Client API, TCP/IP access, encoding/decoding BER records and decoding USMARC records is freely available.

1. Introduction

Z39.50, the ANSI/NISO Information Retrieval Protocol, is perceived by potential implementors as being difficult to implement. I will demonstrate that this is not so by developing a Z39.50 client during the course of this article. The code produced, while copyrighted, is freely available for anyone to use.

In this article, I will stick to the "core" functionality of Z39.50; features that are widely implemented and have the greatest chance of interoperability. You will learn how to initialize a Z39.50 session, how to do searches using simple Boolean operators (type-1 queries) and how to retrieve USMARC and simple text (SUTRS) records. To do this, I will show you how to build a Z39.50 Client Application Program Interface (API) which will allow you to embed Z39.50 client functionality in your applications. I will show you how to build Z39.50 messages and how to send and receive them using standard TCP/IP socket protocols. I will also give you a simple tool for displaying USMARC records. Finally, I will wrap all these tools up in a simple Z39.50 client (*zdemo*).

This article is intended primarily for implementors. It is sprinkled liberally with C code fragments. The complete source code is available at OCLC's anonymous FTP site. (See the section on Source Code Availability at the end of the article.)

2. The Z39.50 Standard

2.1 Who Developed It?

The Z39.50 standard was initially developed in the library community. It was built to satisfy a requirement to search and retrieve USMARC-formatted bibliographic records. Those roots still show today: the core attribute set for Z39.50 (which includes the list of types of things that can be searched for) is named bib-1 and the most widely interoperable record syntax is still USMARC. However, the standard has grown considerably beyond the original modest requirements. Today there are organizations using Z39.50 to deliver full-text documents based on natural language queries. Other organizations support complex chemical structure searching and display.

2.2 Who Maintains It?

The Z39.50 standard started life as the product of a standards committee. The committee considered its work complete with the successful balloting of the original 1988 version of the standard. At that point a Maintenance Agency was appointed by the National Information Standards Organization (NISO) and the original committee was disbanded. Members of the Z39.50 committee met occasionally to discuss possible implementation of the standard and in 1990 the Z39.50 Implementors Group (ZIG) was founded. Today, changes to the standard are developed jointly by the ZIG and the Maintenance Agency. Because the standard is being enhanced by real implementors, the standard now reflects their real-world requirements.

2.3 Where Can I Get It?

The Maintenance Agency for the Z39.50 standard is the Library of Congress. It maintains an anonymous FTP server at ftp.loc.gov where many documents related to Z39.50 are available. Among those documents is the latest version of the standard. Paper copies of the standard can be purchased directly from NISO. Contact them by phone at (800) 282-NISO.

3. Z39.50 Overview

Unlike other Internet protocols such as HTTP or WAIS, Z39.50 is a session oriented protocol. That means that a connection to a Z39.50 server is made and a persistent session is started. The connection with the server is not closed until the session is completed. Session oriented applications are often called "stateful" applications and transaction oriented applications are often called "stateless".

A session oriented protocol is considerably more efficient than a transaction oriented protocol that requires that the connection with the server be reestablished with every message. Session orientation also allows clients iterative refinement of search result sets and multiple record retrieval requests against the same result set. It also allows the client and server to negotiate behavior, such as the kinds of services it needs, and to have that negotiation persist for the duration of the session. In HTTP, much of the message traffic from the client contains descriptions of preferred server behavior that needs to be repeated with every transaction.

In its simplest form, Z39.50 is a synchronous protocol. That is, the client sends a message to the server and waits for the server to respond. The client that is developed in this article (*zdemo*) will use this form. It is possible to negotiate much more complex behavior. The client can have multiple outstanding requests to the Z39.50 server and the Z39.50 server can interrupt those client requests with requests of its own that must be responded to before the original client request can be completed. The *Client API* will not negotiate for that functionality, but it can be readily extended to provide it.

4. Z39.50 Messages

There are two logical parts to the definition of Z39.50 messages (called Protocol Data Units or PDU's in the standard). First is the definition of the content of the messages and second is the encoding rules for converting the logical content into a physical message that can be transmitted. In Z39.50, the messages are defined in the Abstract Syntax Notation 1 (ASN.1) grammar and the encoding rules are defined by the Basic Encoding Rules (BER).

4.1 Defining The Message: Abstract Syntax Notation 1

ASN.1 is an ISO standard (ISO 8824) for defining the content of messages. It is used to define all the ISO protocol messages and is used in the Internet world to define Simple Network Management Protocol (SNMP) messages. ASN.1 is a very rich language. What follows is a simple description of ASN.1; seek a higher authority for a more definitive description.

ASN.1 defines records as being composed of combinations of atomic and constructed data types. The atomic data types are things like INTEGER and BITSTRING. You will recognize them in ASN.1, because they are usually in capital letters. Constructed data types are things like Queries and Options. They always begin with an initial capital letter.

All data types have a number (usually called a *tag*) assigned to them. The tags for atomic data types are assigned by the BER encoding rules. The tags for constructed data types are assigned in the ASN.1 where they are defined and are specified inside square brackets.

Because tags are simply numbers, there is the possibility the two applications will choose the same tags to mean the different things. One possible way to avoid this would be to reserve ranges of tags for ASN.1 data types. Instead, ASN.1 defines four types of tags: *UNIVERSAL, APPLICATION, CONTEXT* and *PRIVATE*. *UNIVERSAL* tags are expected to be recognized wherever they are used in a record. (i.e., a tag of *[UNIVERSAL 8]* is always an *INTEGER*.) *CONTEXT* tags can have different meanings in different contexts. A tag of *[CONTEXT 1]* might be a query in one part of a record and a count in another. The meaning of the tag is defined by its context.

For example, the ASN.1 definition *ReferenceId ::= [2] IMPLICIT OCTETSTRING* defines a constructed data type named *ReferenceId*, whose tag is 2. The type of tag was not specified and defaults to *CONTEXT*. The *ReferenceId* is composed of the atomic data type *OCTETSTRING*. The *IMPLICIT* in that statement says that the tag for the *OCTETSTRING* must not be included inside the *ReferenceId.*

If *IMPLICIT* had been omitted from the above definition (i.e., *ReferenceId ::= [2] OCTETSTRING*) then both the context tag (*[2]*) and the *UNIVERSAL* tag (*[UNIVERSAL 4]*) would have been encoded in the message. Thus, the use of the *IMPLICIT* keyword in the definition allows for smaller encodings.

ASN.1 includes constructs for grouping data types together. These constructs include CHOICE (pick one of the things that follows), SEQUENCE (the things that follow must be provided in the order specified) and SET (the things that follow can be provided in any order.)

4.1.1 EXTERNAL's, OBJECT ID's and ISO Registration

ASN.1 allows the developer to specify that a constructed datatype being referenced is not defined in the current body of the ASN.1. The keyword for specifying this is *EXTERNAL*. *EXTERNAL*s are used throughout the Z39.50 standard. They are the mechanism used to provide extensibility and flexibility in the standard. Saying that a field is defined externally to the standard allows a company to use private data in that field that only their clients and servers will understand. (This is an interoperability problem for other clients and servers, but there are often good reasons for wanting to do this.) It also allows the ZIG to agree on extensions to the standard simply by agreeing on the contents of fields defined *EXTERNAL* to the standard.

*EXTERNAL*s provide flexibility by allowing *Object Identifiers* to be used to make selection from a broad range of possible choices. For example, *RecordSyntax* is defined as *EXTERNAL* in Z39.50, which means that any of a number of possible choices (e.g., USMARC, SUTRS, GRS) can be specified.

EXTERNAL objects, when they arrive in a message, have an *OBJECT IDENTIFIER*. The *OBJECT IDENTIFIER* provides an identification number that allows the message decoder to understand the contents of the object. *OBJECT IDENTIFIERS* are represented symbolically as strings of numbers, separated by periods ('.'). 1.2.840.10003 is the *OBJECT IDENTIFIER* for the Z39.50 standard itself.

Object Identifiers are controlled by the International Standards Organization (ISO). Object Identifiers would have no value as identifiers if they were not unique. Normally, ISO issues Object Identifiers, but once ISO issued an Object Identifier for Z39.50, the Z39.50 Maintenance Agency was authorized to issue subordinate Object Identifiers for Z39.50 objects. Thus, all Z39.50 Object Identifiers begin with the Object Identifier for the standard itself.

4.2 Encoding the Message: The Basic Encoding Rules

Z39.50 messages are encoded according to the Basic Encoding Rules (BER), ISO 8825. BER defines records as being composed of a triple of values: a tag, a length and a value (TLV). The tag portion of the triple includes bits that specify the type of tag (*UNIVERSAL* or *CONTEXT*) and whether the value portion of the tag is primitive data or is composed of more TLV triples. This recursive definition of a record allows for the construction of arbitrarily complex hierarchical records.

I know of two ways to construct BER records. The first way is with an ASN.1 compiler. The compiler reads the ASN.1 definition and produces source code in a programming language such as C or C++. The programmer can then fill in a structure in that language with the values that are to be encoded and the code produced by the ASN.1 compiler reads that structure and builds the BER record. The strong advantage of this method is that you're reasonably confident that the resulting BER record does in fact encode the ASN.1 properly.

OCLC chose not to use an ASN.1 compiler, but instead produced utilities to construct the BER records directly. OCLC has made those utilities publicly available, as well as the Z39.50 Client API. The reasons for choosing not to use an ASN.1 compiler stem mostly from the maturity of the compilers when OCLC first started implementing Z39.50 in 1988. Those reasons are given in greater detail in the documentation accompanying the BER utilities. Directions for getting the BER utilities can be found at the end of this article.

4.2.1 The BER Utilities

The BER utilities allow the programmer to build a tree structure that describes the contents of the record, instead of filling in a record-specific structure and having a record-specific routine construct the BER record. Each node in the tree contains the tag for the data it describes and either a pointer to data or a pointer to another node in the tree. A node in the tree is a **C** structure of type **DATA_DIR**. Routines are provided to construct the tree and to encode the primitive data types such as BITSTRING and INTEGER. Once the tree is built, a utility routine (*bld_rec()*) is called to construct the BER record.

When a BER record is received and decoded by an application, one of these tree structures is produced. To examine the contents of the BER record, simply traverse the tree. This puts the interpretation of the record much more in the hands of the programmer.

5. *ZDEMO* and the Client API

Zdemo is going to be a simple client. It will establish a connection to the Z39.50 server, send an **InitRequest** and wait for an **InitResponse**. It will then sit in a loop waiting for the user to enter searches, record display requests or a Quit command. Commands will consist of a single letter (**S** for Search, **D** for record Display and **Q** for Quit.) Arguments to the commands can follow the command and the default command is Search, when the command is omitted (i.e., **S DOG** and **DOG** are equivalent commands).

The *Client API* is nearly as simple. It consists of the routines *InitRequest()* and *InitResponse(), SearchRequest()* and *SearchResponse()* and *PresentRequest()* and *PresentResponse()*. The request routines take parameters that correspond to the fields in the Z39.50 requests. The response routines take a BER record as their only parameter and return a pointer to a response-specific structure with fields in it that correspond to the fields in the Z39.50 response. The encoding and decoding of the requests and responses will depend on the BER utilities.

6. Establishing the Z39.50 Connection

The vast majority of Z39.50 servers are accessible via TCP/IP, so our client will need to know how to connect to a server via TCP/IP. The usual way to perform TCP/IP functions is with "sockets". Sockets provide the tools and structures for establishing TCP/IP connections and for sending and receiving messages. Sockets have some of the characteristics of files, in that they are opened, read from and written to. In the UNIX world, the relationship between files and sockets is very close; it is less so in the MS Windows world.

For our purposes, only the simplest features of sockets will be used. We will need to know how to convert a host name into an IP address, open a socket, send a message, wait for a return message, determine how many bytes of message are waiting, read a message and close the socket. The complete code for opening and closing a connection to a Z39.50 server is contained in *irpconn.c* at OCLC's anonymous FTP site. (See the section on Source Code Availability at the end of this article.) The code for writing a Z39.50 request, waiting for the response and then reading the response is contained in *doirp.c*.

Windows Sockets are similar enough to standard UNIX sockets that I have provided support for them as well. Sprinkled throughout *irpconn.c* and *doirp.c* you will see fragments surrounded with "#ifdef WINDOWS" and "#endif". These sections contain the support for Windows Sockets.

The routine to make the connection is named *connect()*. It gets passed the name of the host machine for the Z39.50 server and the port where the server is listening. The standard port for Z39.50 is 210, but few of the servers actually listen at that port, so *zdemo* (our client program) will need to accept the port number as an argument. In turn, *zdemo* will get the host name and port as arguments that are passed to it, though, with modification, *zdemo* could read this information from a configuration file.

For MS Windows applications, the first step is to initialize *winsock.dll*, the dynamic link library that contains the sockets routines. This is done by calling *WSAStartup()*, passing it the lowest acceptable version number of the Windows Sockets standard. In our code, *zdemo* will ask for version 1.1. If either there is no *winsock.dll* available or it does not support version 1.1 of the Windows Sockets standard, then *connect()* will write a diagnostic message and return a failure indication.

The next step in establishing the connection will be to convert the host name into an IP address. This is done by calling *gethostbyname()*, passing it the host name. If successful, it will return a structure which contains data that will be used in creating the socket. If *gethostbyname()* fails, then *connect()* will write a diagnostic message and return a failure indication.

Next, the socket is created. This is done by calling *socket()*, telling it that the client will be using it to communicate via TCP/IP. If *socket()* fails, then *connect()* will write a diagnostic message and return a failure indication.

Next, the connection to the server is established by calling *connect()*, passing it the socket and a structure containing the IP address and port number. If *connect()* fails, then *connect()* will write a diagnostic message and return a failure indication. If it succeeds, then *connect()* returns a pointer to the socket and is done. A TCP/IP connection has been made to the Z39.50 server.

6.1 ZDEMO

So far, our source code for *zdemo* looks like this:

```
void *socket;

int main(int argc, char *argv[])
{
    char password[20], server_name[100], userid[20]",
            *usage="usage: zdemo -h[hostname] [-pport#] "
            "[-uuserid/password]";
    int     i, port=210;

    get_args(argc, argv, server_name, &port, userid, password);
    printf("Talking to Z39.50 server on port %u of host '%s'\n", port,
        server_name);
    /* initialization code */
    if( (socket=irp_connect(server_name, port))==0 )
    {
        printf("unable to connect to server %s\n",
        server_name?server_name:"");
        exit(1);
    }
}
```

7. Initialization

The first Z39.50 service is Initialization. The client and server use this service to negotiate the other Z39.50 services and options that are to be provided. They also get to negotiate the preferred message size and exceptional record size. In addition, the client can provide a userid and password.

7.1 Negotiation

Z39.50 supports a simple negotiation mechanism. The client proposes values in the **InitRequest** and the server responds with the actual values. If the client is unhappy with the returned values, its only option is to close the session.

7.1.1 Version

There are now three versions of Z39.50. Version 1 was defined in 1988. It was implemented at only a few sites and was completely superseded by Version 2, which introduced ASN.1 and BER encoding to the standard. Version 2 was defined in 1992. The 1995 version of the standard defines both Version 2 and Version 3. The reason for this is that the ZIG wanted Version 3 to be backward compatible with Version 2 and wanted a single document that defined both. The ZIG did not want developers to have to have two documents to develop a server capable of interoperating with either Version 2 or Version 3 clients. So, both versions are defined in Z39.50-1995 and all the compatibility rules for the two versions are defined there as well.

The version of the standard that the client wants to use is one of the things that is negotiated. The client sends a bitstring with a bit turned on for each version of the standard that the client understands. The server responds with a similar bitstring. The highest version of the standard that the client and server have in common is the version in effect for the session. If the client and server have no supported version in common, then the server will return an empty bitstring and fail the **InitRequest**. The client can deduce the reason for the failure from the empty **Version** bitstring in the **InitResponse**.

7.1.2 Options

The client and server negotiate the services and options that they want through the **Options** bitstring. These are specified by turning on the appropriate bits in the bitstring. All of the Z39.50 services can be negotiated; that is, the client can request that they be made available by the server. The server can deny these services by turning off the appropriate bit in the bitstring when it is returned in the **InitResponse**. Options that can be negotiated include such things as support for named result sets or concurrent operations.

7.1.3 Message Sizes

The client also specifies a **Preferred-message-size** and an **Exceptional-record-size**. The **Preferred-message-size** will be exceeded by the server only when the client requests a single record and its size exceeds the **Preferred-message-size**, but not the **Exceptional-record-size**. The purpose of this is to allow the client to control the maximum size of a normal message from the server, but to allow it to occasionally accept large records.

The server may respond to the proposed values with alternative values in the **InitResponse**.

7.2 Other Initialization Parameters

The client can provide a userid and password in the **InitRequest** and can also provide information identifying the client software itself. Lastly, the **InitRequest** contains a placeholder for information defined externally to the standard.

All Z39.50 request definitions include an optional **referenceId**. This is an arbitrary string of bytes that the client can send that the server is required to return with the response. Its intent is to help the client identify the returning response in an asynchronous message environment. While **referenceId** can hold any number of bytes, the *Z39.50 Client API* allows only a C language **long** value to be used.

7.3 The InitRequest

The **InitRequest** is created by a call to the *InitRequest()* routine. It takes a **referenceId**, a **preferredMessageSize**, an **exceptionalRecordSize**, an **id** and a **password** as parameters. It does not accept

options as a parameter, since the Client API always negotiates for the most functionality that it can handle.

InitRequest() returns a pointer to an allocated area in memory that contains the BER encoded **InitRequest.**

The prototype for *InitRequest()* looks like this:

```
unsigned char *InitRequest(
        long referenceId,
        long preferredMessageSize,
        long exceptionalRecordSize,
        char *id,
        char *password);
```

7.3.1 Encoding the Request

The easiest way to understand the *InitRequest()* routine is to walk through it line by line, showing the ASN.1 that is being encoded and providing commentary. The **C** code is indented and in bold. The ASN.1 is in italics and the commentary is in normal text.

Normally when I code using the BER utilities, I use preprocessor variables to hold the tag values. The preprocessor variable **InitRequest** would be defined as **20**. I do this for readability. But in the code below, the commentary explains what is going on in the code, and I want you to be able to see the correlation between the code and the ASN.1, so I am omitting the preprocessor variables. If you get the code from our FTP server, you will see proper preprocessor variables instead of constants.

```
CHAR *Init_Request(long referenceId, long preferredMessageSize,
            long exceptionalRecordSize, char *id, char *password, long *len)
/*
        referenceId has no particular meaning to the Client API.  You can put whatever
        value you want into it, and it will be returned in the response.  id and password
        can be either NULL or "".  len will contain the length of the encoded request
        when InitRequest() returns.
*/
{
        static char *protocol_version="yy";  /* versions 1 and 2 */
/*
        When you want Version 2, you have to ask for Version 1 too.  (This is to allow
        interoperability with ISO 10163).
*/
        static char *options_supported="yy"; /* search and present only */
        /*****************************************************/
        /*              build an IRP Init request       */
        /*****************************************************/
        dir=dmake(20, ASN1_CONTEXT, 30);
initRequest  [20] IMPLICIT InitializeRequest,
/*
        Make a DATA_DIR tree for assembling the parts of our message.  The first two
        arguments specify the tag and tag type for the root of our tree.  They correspond
        to the first tag in the ASN.1 definition of an InitRequest.  The 30 tells dmake()
        that we expect to see 30 nodes in our tree.  If that number is exceeded, then the
        BER utilities will automatically increment the size of the tree by that amount.
        dir, the value returned by dmake(), is a pointer to the root of the tree.
*/
        if(referenceId)
            daddchar(dir, 2, ASN1_CONTEXT, (CHAR*)&referenceId, sizeof(referenceId));
```

```
referenceId  ReferenceId OPTIONAL,
/*
        ReferenceId is defined later in the standard as:
                ReferenceId ::= [2] IMPLICIT OCTETSTRING
        If a non-zero referenceId has been provided, then add it to the request.  The first
        argument to daddchar() is a pointer to the parent of the field being added.  The
        next 2     arguments are the tag and tag type of the referenceId.  The last two
        arguments are a pointer to the referenceId and its length.  The referenceId is
        being passed to the server as a string of bytes (an OCTETSTRING in ASN.1.)
*/
        daddbits(dir, 3, ASN1_CONTEXT, protocol_version);
protocolVersion  ProtocolVersion,
/*
        protocolVersion is defined later in the standard as:
                protocolVersion ::= [3] IMPLICIT BITSTRING
        daddbits() encodes ASN.1 BITSTRINGs.  Here, we're encoding the ProtocolVersion.
*/
        daddbits(dir, 4, ASN1_CONTEXT, options_supported);
options  Options,
/*
        Options is defined later in the standard as:
                Options ::= [4] IMPLICIT BITSTRING
*/
        daddnum(dir, 5, ASN1_CONTEXT, (CHAR*)&preferredMessageSize,
                sizeof(preferredMessageSize));
preferredMessageSize  [5] IMPLICIT INTEGER,
/*
        daddnum() encodes ASN.1 INTEGERs.  Here, we're encoding the preferredMessageSize.
*/
        daddnum(dir, 6, ASN1_CONTEXT, (CHAR*)&exceptionalRecordSize,
                sizeof(exceptionalRecordSize));
exceptionalRecordSize  [6] IMPLICIT INTEGER,
        if(id && *id)
        {
                char *t;
                DATA_DIR *subdir;
/*
                We'll use subdir to keep track of subtrees in our DATA_DIR tree.
*/
                int len=strlen(id)+1;
/*
                We need to figure out how long the id and password are and then add 1 for the
                '/' separator character.
*/
                if(password && *password)
                        len+=strlen(password)+1;
                else
                        password="";
                t=(char*)dmalloc(dir, len+1);
/*
          dmalloc() malloc's space that is freed automatically when the DATA_DIR tree
          is freed. In this case, the "+1" is for the NULL that sprintf() will put at the end
          of the string.
*/
          strcpy(t, id);
          if(password && *password)
               sprintf(t+strlen(t), "/%s", password);
          subdir=daddtag(dir, 7, ASN1_CONTEXT);
idAuthentication  [7] ANY OPTIONAL,
/*
          daddtag() adds a tag without any data. It returns a pointer to the node that
          was added to the tree to hold the tag.
*/
          daddchar(subdir, ASN1_VISIBLESTRING, ASN1_UNIVERSAL, (CHAR*)t, len-1);
/*
          The ANY is recommended later in the standard to be encoded as a CHOICE,
          one option of which is:
               open VisibleString,
          Add the id and password with an IMPLICIT ASN.1 data type of
          VISIBLESTRING.
*/
     }
     daddchar(dir, 110, ASN1_CONTEXT, (CHAR*)"1995", 4);
implementationId  [110] IMPLICIT InternationalString OPTIONAL,
     daddchar(dir, 111, ASN1_CONTEXT, (CHAR*)"OCLC IRP API", 12);
implementationName  [111] IMPLICIT InternationalString OPTIONAL,
     daddchar(dir, 112, ASN1_CONTEXT, (CHAR*)"1.0", 3);
implementationVersion  [112] IMPLICIT InternationalString OPTIONAL,
/*
     Tell the server what kind of client is talking to it.
*/
     return bld_rec(dir, len);
/*
     bld_rec() malloc's the amount of space needed to hold the BER record, assembles
     the BER record in that area and returns a pointer to that area, which is finally
     returned by InitRequest().
*/
}
```

7.3.2 Transmitting the Request

Zdemo transmits the BER requests by calling *doirp()*, passing it the pointer to the BER request and the pointer to the socket returned by *connect()*. *Doirp()* sends the request to the Z39.50 server, waits for the response to the request from the server and returns a pointer to that response.

Doirp() starts by determining the length of the request. It does this by calling the BER utility *asn1len()*. It uses that length to drive a **while** loop where the length represents the number of bytes of the request waiting to be sent.

Doirp() sends data to the server by calling the socket routine *send()* and passing it the socket, a pointer to the request and the number of bytes to send. *Send()* returns the number of bytes actually sent. The pointer to the request is incremented by that amount and the length is decremented by that amount. If the length goes to zero, then the complete request has been sent and *zdemo* falls out of the **while** loop. If *send()* indicates an error, then *doirp()* prints an error message and quits, returning an error indication.

Next, *doirp()* needs to wait for the response from the server. The socket utilities are prepared to handle much more complicated tasks than *zdemo* is requiring of them, so some of the tools that it uses seem overly complicated for this purpose. The mechanism for waiting for a message is one of those tools. The socket utilities allow an application to have many active sockets open and allow you to wait until any of them have a message. To do this, the application has to construct a list of sockets to be waited on. Two preprocessor macros are used to construct the list: *FD_ZERO()* and *FD_SET()*. *FD_ZERO()* initializes an empty list, and *FD_SET()* adds sockets to the list. After the list is built, the routine *select()* is called, passing it the list of sockets to be waited on. The *select()* call sits inside a **while** loop; sometimes *select()* returns with an indication that it has not received anything yet.

After *doirp()* has gotten the indication that a message is available, it calls *ioctl()* to determine the amount of data that has been received. It then calls *recv()* to read the data. It passes *recv()* the socket, a pointer to a buffer to hold the incoming message, and the number of bytes it wants to read (which it got from *ioctl()*.) *Recv()* returns a count of the number of bytes that it actually read. If that count is zero, then there was probably some failure in the connection and *recv()* will print an error message and return with an error indication.

Often, TCP/IP has to break large messages into smaller messages to transmit them. That means that when *doirp()* gets a message, it might be the first of many messages that comprise a complete Z39.50 response. The BER utilities provide a routine, *IsCompleteBER()*, which gets passed a pointer to a buffer with a BER encoded message and a count of the number of bytes in the buffer. *IsCompleteBER()* returns an indication of whether a complete message is in the buffer. If the message is complete, then *IsCompleteBER()* also returns the actual size of the message, which might be less than the amount of data in the buffer, since it is possible for more than one message to have been received at one time.

If the message was not complete, then *IsCompleteBER()* also returns the number of bytes remaining to be read to complete the message. Sometimes *IsCompleteBER()* reports that the message is not complete and there are zero bytes waiting to be read. This means that *IsCompleteBER()* cannot determine the remaining length and *doirp()* should just wait for more data to arrive. Either way, *doirp()* sits in a loop, reading more data, until *IsCompleteBER()* reports that a complete message has arrived. When that happens, *doirp()* returns a pointer to the buffer containing the message.

At this point, *zdemo* has sent our **InitRequest** and received an **InitResponse**.

7.4 The InitResponse

The most important field in an **InitResponse** is the **result** field. It tells the client whether its **InitRequest** has been accepted by the Z39.50 server. If it has a non-zero value, then a Z39.50 session has been successfully established. If it is zero, then the Z39.50 server has rejected our session. Unfortunately, there is no explicit mechanism for the server to tell why it is rejecting our **InitRequest**. We'll have to deduce the reason from the other values returned in the **InitResponse**.

7.4.1 Decoding the Response

The Z39.50 Client API provides the routine *InitResponse()* to decode the **InitResponse** from the Z39.50 server. It is passed a pointer to the **InitResponse** and returns a pointer to a structure containing information from the **InitResponse.**

The first step in decoding any Z39.50 response is to decode the BER encoded message. The BER utility *bld_dir()* does this. Its job is to build a DATA_DIR tree that reflects the structure of the message. Typically, to decode the message, we'll just traverse the tree. I use a **for** loop to do this. I set the loop variable to the first child in the tree and loop through all its siblings. Inside the loop I use a **switch** statement to test for the possible tags that might have been in the message.

Again, as with the *InitRequest()*, the easiest way to understand the *InitResponse()* routine is to walk through it line by line, showing the ASN.1 that is being encoded and providing commentary. The **C** code is indented and in bold. The ASN.1 is in italics and the commentary is in normal text. I have also repeated the practice of replacing preprocessor variables with constants to emphasize the correspondence between the **C** code and the ASN.1.

```
INIT_RESPONSE *InitResponse(CHAR *response)
{
    DATA_DIR far *subdir;
    INIT_RESPONSE *init_response;
    if(!response || !bld_dir(response, dir))
        return NULL;
/*
    If a response was not provided or we were unable to decode the response, then
    return a failure indication. The dir that is being passed to bld_dir() is the same
    one that was created in InitRequest() to hold the message being built there. Dir is
    a global variable and will be used by all the request and response routines.
*/
    if(dir->fldid!=21)
        return NULL;
initResponse  [21] IMPLICIT InitializeResponse,
/*
    If the response wasn't an InitResponse, then return a failure indication. The tag
    in the root node of the tree is the message tag.
*/
    if( (init_response=(INIT_RESPONSE*) calloc(1, sizeof(INIT_RESPONSE)))==NULL)
        return NULL;
/*
    If we can't allocate space to hold the structure describing the InitResponse, then
    return a failure indication.
*/
    for(subdir=dir->ptr.child; subdir; subdir=subdir->next)
/*
    This is our driving loop. The loop variable is initialized to point at the first child
    off the root. As long as there is such a child, process it and then point at its
    sibling.
*/
        switch(subdir->fldid)
/*
    Test for the value of the tag in this node.
*/
        {
            case 2:
referenceId  ReferenceId OPTIONAL,
/*
                ReferenceId is defined later in the standard as:
                    ReferenceId ::= [2] IMPLICIT OCTETSTRING
*/
                memcpy((char*)&init_response->referenceId, (char*)subdir->ptr.data,
                    (int)subdir->count);
/*
```

```
     Just save the referenceId in the INIT_RESPONSE structure.  Only the calling      application will be inter-
ested in it.
*/
                         break;
                    case 4:
options  Options,
/*
                         Options is defined later in the standard as:
                              Options ::= [4] IMPLICIT BITSTRING
*/
                         init_response->options=dgetbits(subdir);
/*
                         dgetbits() decodes encoded BITSTRINGs.  It returns a character string
                         with a 'y' for every bit that was turned on, and a 'n' for every bit that
                         was turned off.
*/
                         break;
                    case 5:
preferredMessageSize  [5] IMPLICIT INTEGER,
                         init_response->preferredMessageSize=dgetnum(subdir);
/*
                         dgetnum() decodes encoded INTEGERs.  It returns a long, which we will
                         save in the INIT_RESPONSE structure.
*/
                         break;
                    case 6:
exceptionalRecordSize  [6] IMPLICIT INTEGER,
                         init_response->maximumRecordSize=dgetnum(subdir);
                         break;
                    case 12:
result [12] IMPLICIT BOOLEAN,
                         init_response->result = (int)dgetnum (subdir);
/*
                         BOOLEANs are encoded as INTEGERs, so dgetnum() is used to decode
                         them.  A non-zero value means TRUE and a zero value means FALSE.
*/
                         break;
               }
          }
     return init_response;
}
```

7.5 ZDEMO

The following code gets added to *zdemo*:

```
INIT_RESPONSE *init_response;
     long               len;
     unsigned char      *request, *response;

/*
     Build the InitRequest.
*/
     request=InitRequest(0, 16384, 500000L, userid, password, &len);
/*
     Send the request and get the response.
*/
     response = do_irp(request, socket);
     if(!response)  /* If we did not get a response, then quit. */
     {
          printf("unable to send init request\n");
          exit(2);
     }
/*
     Decode the response.
*/
     init_response=InitResponse(response);
     if(!init_response || !init_response->result)
     {  /* If the response was not decodable, or if the InitRequest failed, then quit. */
          printf("init failed\n");
          exit(3);
     }
```

8. Searching

Z39.50 allows highly specific searching of databases. The specificity of Z39.50 queries is one of the standard's great strengths. Other protocols, such as WAIS or Gopher, support "magical" searching. The user enters some kind of free text query and "magic" happens. The same query on another server might produce completely different results, because different "magic" happened. The user is at a loss to determine why the records were retrieved. The user is also unable to control the search. The user is unable to specify that she wants to find records where the word **SMITH** appeared in the title, but not as an author. These weaknesses have all been overcome with Z39.50.

Another strength of Z39.50 queries is the persistence of their results for the duration of the Z39.50 session. With other protocols, the results of the query must be sent immediately to the client. That's fine, if the database is small and the result sets are always small. When the databases are large, that is not practical. The user needs the ability to fetch and examine some of the records and still be able to ask for other records later. Better yet, if the result set is large, the user would like to be able to apply restrictors to the result set and produce a smaller, hopefully more pertinent, result set.

8.1 Result Sets

In order to reference a result set after it has been produced, the result set must have a name. In Z39.50, the client provides the name of the result set with the query: the client names the result set. Every query can have a different result set name, allowing the client to reference any number of previous result sets. But few, if any, servers allow an unlimited number of result sets. When a client has exceeded the number of supported result sets, the server might delete old result sets arbitrarily.

In fact, some servers allow a client to have only one result set. In that case, they do not really support named result sets. To get around the apparent contradiction of the client being able to name result sets and the server being unable to support named result sets, the ZIG agreed on the result set name "**default**". This is the result set name that must accepted by servers that do not otherwise support named results sets. If all queries sent to such a server are named "**default**", then the client has only one result set that it can refer to.

Unfortunately, in Version 2 of the standard, the client can not tell whether the server will allow result set names other than "**default**". The only way to tell is to use a different result set name. If the server cannot support named result sets, it will fail the search and return an error code indicating the problem. The client will then know that "**default**" will be the only acceptable result set name. In Version 3, support for named result sets is one of the options that can be negotiated at initialization time.

If the client uses the same result set name twice, the server should replace the previous result set of the same name with the new result set. To keep that from happening accidentally, the client is required to set a flag in the **SearchRequest** indicating that the result set is to be replaced.

8.2 Attributes

In "magic" searching systems, query terms are unqualified. That is, the user types in a term, but provides no extra information about the term to indicate its semantic meaning. Systems that provide more specific searching usually provide the concept of an "index". So the user can say that the term provided should be considered to be an author or a word from a title. But this is only a single piece of qualifying information that can be provided with the term.

The Z39.50 developers wanted a richer mechanism than simply indexes. They wanted to provide many dimensions of qualification to the term. The word they chose to describe these additional qualifications on a term is "attribute". A term can have many attributes. One of those attributes could be Use, which roughly corresponds with indexes. The Use attribute allows the client to specify how the term would have been used in the records to be retrieved. For example, the term was Used as an AUTHOR or TITLE. Another attribute is Structure; the term is supplied according to a particular structure. The structure might be that the term is a WORD or a PHRASE.

8.2.1 Attribute Sets

Since the developers understood that they could not predict all the attributes that implementors would want, they created the idea of an attribute set. An attribute set defines a collection of attributes. Implementors are free to invent their own attribute sets, but the developers provided a starter set of attributes and packaged them in an attribute set named **bib-1**.

Attribute sets are identified by an Attribute Set ID, which is just an Object Identifier. All Attribute Set ID's begin with 1.2.840.10003.3; the Attribute Set ID for the bib-1 attribute set is 1.2.840.10003.3.1.

The **bib-1** attribute set contains 6 types of attributes: Use, Relation, Position, Structure, Truncation and Completeness. These attributes are explained in great detail in the **bib-1** attributes documents, available at the Library of Congress' FTP site. The only attributes discussed in this article will be Use and Structure.

Attribute types in an attribute set are identified by a number. In the **bib-1** attribute set, Use is attribute type 1 and Structure is attribute type 4. The values that an attribute can have are also identified by a number. This means that it takes two numbers to specify an attribute for a term: the attribute type and the attribute value. For example, every Use attribute, such as AUTHOR or TITLE, has a number. (AUTHOR is 1003 and TITLE is 4.) These numbers are specified in the Attribute Sets appendix of the standard. At last count, there were 98 different Use attributes specified, and that list can be extended at any time.

8.3 Query Terms and Attributes

Terms can have one or more attributes associated with them. In the ASN.1 for the standard, this association is called **AttributesPlusTerm** and consists of an **AttributeList** and a **Term**. An **AttributeList** is defined as a *SEQUENCE* of **AttributeElement** which are in turn defined as a pair of *INTEGER*s consisting of **attributeType** and **attributeValue**. These pairs of numbers are exactly the numbers described above.

In Version 2, all the attributes in the query have to come from the same attribute set. During the development of Version 3, it soon became clear that this was a problem. How could the user formulate a query asking about AUTHORs (a **bib-1 Use** attribute) and BOILINGPOINTs (a **Use** attribute from an chemical attribute set)? In Version 3, the attribute set ID can be specified for every **AttributeElement**. That means that you can mix attributes from a number of attribute sets.

8.4 Query Grammars

Z39.50 defines several query grammars, each one identified by a number. Type-0 queries are for private query grammars. Sometimes clients and servers from the same organization prefer to use that organization's own query grammar. At OCLC, a number of our clients know how to use the query grammar of our database engine and pass those queries to the Z39.50 server as type-0 queries.

Type-1 queries are the only widely accepted queries. Support for them is mandatory in Z39.50. Type-1 queries are described in more detail later.

Type-2 queries use the query grammar from the ISO Common Command Language (ISO 8777). This grammar has severe extensibility limitations and probably should not be used. ISO CCL queries can always be sent as type-0 queries.

Type-100 queries use the query grammar from the ANSI/NISO Common Command Language (Z39.58). This grammar is closely related to, and has the same problems as, the ISO Common Command Language.

Type-101 queries are an extension of type-1 queries to support proximity searching. With Version 3 of the standard, type-1 queries are identical with type-101; but they remain distinct in Version 2.

Type-102 queries are still being defined. They are intended to support some of the features of query grammars that support ranking.

8.5 Reverse Polish Notation Queries (type-1)

Type-1 queries are called Reverse Polish Notation (RPN) queries. Reverse Polish Notation is a way of representing Boolean queries by specifying first the operands and then the operator. Normal query grammars let you specify an operand, then an operator and another operand. This is called an infix notation. The problem with infix notations is that you end up having to use parentheses to specify the order of evaluation of the operators and operands. Reverse Polish Notation does not have that problem.

The search (**DOG OR CAT**) **AND HOUSE** would be expressed as **DOG CAT OR HOUSE AND** in Reverse Polish Notation and the search **DOG OR** (**CAT AND HOUSE**) would be expressed as **DOG CAT HOUSE AND OR** in RPN. The query is evaluated left to right. Every time you encounter an operator you process the two operands to the left and replace the operator and operands with the result of evaluating them. In the first example, the **OR** is associated with **DOG** and **CAT**. After **DOG OR CAT** is evaluated, the result is put back into the query. The **AND** then has that result and **HOUSE** as its operands.

Reverse Polish Notation queries can be easily represented as trees, with the operators as roots and branches and the operands as leaves. That is the sense in which type-1 queries are Reverse Polish Notation. They are not text strings as in the examples above. They are trees defined recursively in ASN.1. A type-1 query can either be an operand or an operator with two operands. An operand can either be a term or a type-1 query. This recursive definition allows for arbitrarily complex queries.

We need some way to pass a query into our Z39.50 Client API. To do this, we'll use real Reverse Polish Notation. Terms will be optionally followed by a slash '/' and then a Use attribute value. They can also be followed by an optional slash and a Structure attribute value. Terms can be surrounded by double-quotes. The following are all examples of legal query terms: **DOG** (no Use or Structure attribute specified), **DOG/21** (dog as a subject heading), **DOG/21/2** (dog as a subject heading and a structure of WORD) and **"DOG HOUSE"/21/1** (dog house as a subject heading and a structure of PHRASE).

8.6 *Database Names*

The client must specify what database or databases the server is to search. The Z39.50 standard allows multiple databases to be specified in a search request. Unfortunately, this is another feature that cannot be determined at initialization time. One way the client can find out if the server supports multiple database names is to try it and see if a diagnostic is returned, but the lack of a diagnostic does not necessarily mean that all the databases were searched. Some of the servers just ignore the extra database names. This feature is not available in the Client API.

8.7 *Piggy-backed Presents*

It is possible to request that records be returned automatically with the **SearchResponse**. This is called a piggy-backed Present. Piggy-backed Presents are supported in the Client API but are not supported by *zdemo* and are beyond the scope of this article. *Zdemo* will provide hard-coded values for those parameters in its call to *SearchRequest()*.

8.8 *The SearchRequest*

The **SearchRequest** is created by a call to the *SearchRequest()* routine. It takes a **referenceId**, a **replaceIndicator**, a **resultSetName**, a **databaseName**, a **query,** and a **query_type**.

The **referenceId** is a C language **long** value and has the same meaning as in *InitRequest()*. The **replaceIndicator** is an integer and has either a zero or non-zero value for **FALSE** and **TRUE** respectively. The **resultSetName** can be any character string. The **databaseName** is a character string whose value is determined by the server.

The conversion of the **query** parameter into a Z39.50 **query** is probably the trickiest code in the *Client API*. The **query** is passed as a character string, but its evaluation is dependent on the **query-type**. If the **query-type** is 0, then the **query** is assumed to be in a private query grammar and is passed through to the Z39.50 server exactly as received by *SearchRequest()*.

If the query-type is 1, then *SearchRequest()* is expecting a string with a Reverse Polish Notation query in it. The terms can be surrounded with double-quotes. This is important if the term consists of multiple words, as in a phrase search. The term can also be followed by an optional slash ('/') and a Use attribute value. The Use attribute value can also be followed by another optional slash and a Structure attribute value. There is no default Use attribute value and the default Structure attribute value is WORD.

For example: to search for books about slavery by Mark Twain, you could enter the search:

slavery/21 "twain, mark"/1003/1 and

which asks for records with "slavery" as a subject heading and "twain, mark" as an author phrase.

As in *InitRequest()*, *SearchRequest()* returns a pointer to an allocated area in memory that contains the BER encoded **SearchRequest.**

The prototype for *SearchRequest()* is:

```
unsigned char *SearchRequest(
      long referenceId,
      int replaceIndicator,
      char *resultSetName,
      char *databaseName,
      char *query);
```

I will not walk through the code this time. You have already seen BER encoded messages produced; the searches are not any more exciting. The code is provided if you want to examine it.

8.9 *The SearchResponse*

The **SearchResponse** is processed by *SearchResponse()* and it, like *InitResponse()*, takes the BER record returned by the Z39.50 server as its only parameter and returns a pointer to an allocated structure which contains the fields of the **SearchResponse**. The prototype for *SearchResponse()* is:

```
SEARCH_RESPONSE *SearchResponse(
      CHAR *response);
```

and the SEARCH_RESPONSE structure looks like this:

```
typedef struct
{
      long referenceId;
```

```
    int searchStatus;
    long resultCount;
    long resultSetStatus;
    long error_code;
    char *error_msg;
} SEARCH_RESPONSE;
```

The **referenceId** is the same one provided to *SearchRequest()*.

searchStatus contains either a zero to indicate that the search failed or a non-zero value to indicate success.

If **searchStatus** indicates that the search succeeded then **resultCount** will contain the count of the number of records that satisfy the search and the value of **resultSetStatus** will be undefined. A value of zero in **resultCount** is not an indication that the search failed, only that there are no records in the database that meet the search criteria.

If **searchStatus** indicates that the search failed, then the value of **resultCount** is undefined and **resultSetStatus** will indicate if there are any records available for retrieval. Typically **resultSetStatus** will contain the value 3 which indicates that there is no result set available, but other values are potentially available and defined in the standard. **error_code** and **error_msg** should contain values; otherwise they will contain 0 and NULL respectively. The values for **error_code** and **error_msg** are described in the Error Diagnostics appendix of the standard.

8.10 ZDEMO

Before *zdemo* can generate a search, it needs a simple command processor. Remember that commands to *zdemo* are going to be single letters, so parsing the commands will be easy. *Zdemo* will need a loop for getting commands from the user. A command of ‘q’ or an end-of-file indication from the input stream will end the loop. Inside that loop, *zdemo* will test for a single letter command and if there is none, then it will assume that a search is being requested. It will then switch on the value of the command and call a routine to handle the command.

Our driving loop looks like this:

```
char cmd, input[1000];
while(gets(input))
{
    strlwr(input);
    if(input[0])  /* did we get any input? */
        if(input[1]==‘ ‘)  /* was the second character a blank? */
            cmd=input[0];
        else
            cmd=‘S’;  /* assume that they want to search */
    else
        cmd=‘ ‘;  /* no command */

    if(cmd==‘q’)
        break;  /* exit the loop */

    switch(cmd)
    {
        case ‘s’:  /* explicit search command */
            zsearch(input+2);  /* +2 to skip command and blank */
            break;
        case ‘S’:  /* implicit search command */
            zsearch(input);
    }
}
```

In addition, the routines that *zdemo* calls will need some clues about the behavior of the Z39.50 server. For instance, some servers will not accept any **resultSetNames** except “default”. *Zdemo* will be told this through arguments that are passed to it at startup time. In the case of the “default” resultSetName, *zdemo* will look for an argument of “-d” to indicate that it must use the “default” **resultSetName**.

```
char resultSetName[20];

void zsearch(char *query)
{
    long                    len;
    SEARCH_RESPONSE *search_response;
    unsigned char           *request, *response;
    static int              search_num=1;

    if (MustUseDefault)  /* global variable */
        strcpy(resultSetName, “default”);
    else
        sprintf(resultSetName, “Search%d”, search_num++)

    request=SearchRequest(0, TRUE, resultSetName, database_name, query, &len);

    response = do_irp(request, socket);
    search_response=SearchResponse(response);
    printf(“%ld records found.\n”, search_response->resultCount);
    if(search_response->searchStatus)
        printf(“Search Successful! :-)\n”);
    else
    {
        puts(“Search Failed! :-(“);
        printf(“Error_code=%ld, message=’%s’\n”, search_response->error_code,
            search_response->error_msg ? search_response->error_msg :
            ”None provided”);

        if(search_response->error_code==22)
        {
            puts(“Must use ResultSetName of \”default\””);
            puts(“Resetting internal flags; please try again”);
            MustUseDefault=TRUE;
        }
        if(search_response->error_msg)
            free(search_response->error_msg);
     }
    free(search_response);
    free(response);
}
```

9. Retrieval

The Z39.50 implementors clearly saw retrieval as a weakness in Version 2 of the standard. Many of the enhancements in Version 3 center around retrieval. Included in these enhancements are the ability to ask for specific parts of a record, to ask about the contents of a record and to specify a prioritized list of desired record syntaxes. But, even without these enhancements, Z39.50 supplies perfectly acceptable mechanisms for retrieving records. Since this article is concentrating on core functionality, the *Client API* will only use those retrieval features available in Version 2.

Version 2 allows clients to ask for a specific range of records from a result set in full or brief forms and to specify a single record syntax. The most common record syntaxes are USMARC and SUTRS. USMARC is the record syntax used in the U.S. library community to exchange cataloging information and SUTRS is a Simple Unstructured Text Record Syntax, invented by the ZIG. Both of these record syntaxes will be discussed in greater detail later.

9.1 Result Sets Revisited

In Z39.50, result sets are modeled as containing ordered lists of pointers to records. This does not mean that a server is actually supposed to create lists like that; it means that the client can act as if that were true. The ordering of the result set is important, although the type of ordering is not. Whether the records are in rank order or chronological order or sorted by title is unimportant. What is important is that the client can ask for the n'th record in a result set and always get the same record from the same result set.

To retrieve records from a result set, the client specifies the name of the result set and the relative record number of the record in the result set. The first record in a result set is record number 1. In the C programming languages the first record would naturally be record number 0, so it is important to remember that that is not true here.

To ask for several records, the client can specify a single relative record number for the first desired record and a count of the number of records to be returned. This only allows for a single list of adjacent records to be returned. With Version 3 comes the ability to specify multiple ranges of records in a single request. This will allow the user to request the first, third and ten thousandth records from a result set and the client will be able to satisfy the request in a single transaction with the server.

9.2 Element Sets and Element Set Names

The fields in a record are called *elements* in Z39.50. A collection of elements would be an *element set* and if that collection of elements had a name, it would be an *element set name*. In Version 2, element set names are the only mechanism available to specify the elements desired from a record. Version 3 includes rich mechanisms for identifying and specifying the elements in a record, but element set names are sufficient for many purposes.

The standard only specifies two element set names: "F" for Full records (all elements included) and "B" for Brief records. Brief records are a problem. The standard is rightly silent on the elements that constitute a brief record. But, that leaves the client developer at the whims of the server developers as to the fields that can be displayed in a brief record. Unless I am sure that a particular server returns all the fields that I want to display in a brief record, I usually ask for full USMARC records and throw away the fields that I do not need. That technique will not work if SUTRS records have been requested, since they consist of a single field.

9.3 Record Syntaxes

A *record syntax* is simply the way that records are encoded. There are a number of record syntaxes recognized in Z39.50. *Object identifiers* are used to specify record syntaxes, so record syntaxes must be either registered with the maintenance agency or be registered as nodes of an implementor's private object identifier tree. As mentioned above, there are two widely recognized record syntaxes; USMARC and SUTRS. I'll describe them in detail below, but it is worth mentioning the other record syntaxes listed in the standard. Understanding what these other syntaxes are and where they are intended to be used is useful in understanding where the implementors of the standard are taking it.

9.3.1 Non-core Record Syntaxes

9.3.1.1 Other MARC Syntaxes

There are a number of variants on the MARC record syntax. In the United States, the Z39.50 developers tend to forget that fact and refer to USMARC as simply MARC. But, there are 14 other MARC record syntaxes recognized by the standard and they will be supported by many of the commercial servers as Z39.50 services are implemented in Europe. For the most part, these are national MARC syntaxes (e.g., UKMARC, CANMARC and FINMARC) which encode support for local cataloging standards, but there are also some internationally recognized MARC syntaxes (e.g., UNIMARC and INTERMARC.)

9.3.1.2 Explain

Successful interoperation of Z39.50 clients and servers in Version 2 is based on a priori agreements between the two parties. The client had no mechanism for determining what Use attributes were going to be supported by the server for searching nor what record syntaxes were going to be supported for retrieval. The client had to be told this information through some process outside of the standard. Currently, most of the server hosts provide human readable documentation that can be used to statically configure a client. The Explain service provides the mechanism that allows those things to be determined dynamically.

The Explain service is implemented as a database that can be queried by the client. Access to the records in this database is primarily gained through search keys defined by the standard. The contents of these records, which contain things like Use attributes and record syntaxes supported are defined by the Explain record syntax.

9.3.1.3 OPAC

OPAC (Online Public Access Catalog) records were an attempt to allow holdings information to be transmitted along with bibliographic records (usually sent in USMARC format.) They were not widely implemented and a number of non-standard mechanisms for transmitting holdings information were developed instead.

9.3.1.4 Summary

Summary records were developed as part of an effort to bring the WAIS retrieval software into compliance with Z39.50. WAIS was based on the 1988 version of Z39.50, with a number of private extensions. Among these extensions was the ability to provide brief record information in a more standardized way than the simple Brief Element Set Name provided by the standard.

9.3.1.5 GRS

The Generic Record Syntax is at the heart of most of the growth areas of Z39.50 implementation. The other record syntaxes described so far have limited structural flexibility (you cannot have really complex fields) and rigid semantics (everyone knows what to expect in every field.) What was needed was a record syntax with great flexibility and the ability to transmit both elements with semantic understanding and elements with no semantic understanding.

GRS was invented for this purpose. It supports arbitrarily complex hierarchical records and elements that can carry numeric tags from any number of well-known name spaces as well as string tags intended to carry field "names" that might be of use to a human viewing them, if not of use to the software receiving them.

GRS is being heavily used by the Chemical Abstract Service to provide their complex chemical records which include things like chemical structure information. In addition, the GILS (Government Information Locator Service) profile uses GRS records as the most flexible way to transmit Information Locator records and the CIMI (Coalition for the Interchange of Museum Information) group is looking to use GRS records to transmit their information.

9.3.2 USMARC

USMARC can be quite daunting, at first. Fields are tagged numerically and there is little pattern to the tagging. If you do not know what the tags mean, you are out of luck. To complicate things more, some of the fields can repeat and others cannot: but some of the non-repeatable fields have other, repeatable, fields that the extra data can go into. (e.g., The first author of a book might be placed in a 100 field, a nonrepeating field, but subsequent authors would be put into 700 fields.)

There are actually three different sets of rules combined to form USMARC records. The first is the encoding standard; ANSI Z39.2. It describes the physical encoding of all MARC records (at least that is the theory.) The second is the tagging rules: what data goes in what fields. Finally come the formatting rules for the data (e.g. names should be entered last name first with a comma separator.) Fortunately, as client developers, it is not necessary to worry about the formatting rules.

The encoding rules are straightforward. The records are theoretically encoded as 7-bit ASCII, but I've seen many private characterset extensions that use 8-bit ASCII. The record begins with a fixed format *leader* that describes the length and type of the MARC record and well as describing some of the encoding options that will be used in the record. The leader is followed by a *directory* that describes what fields are contained in the records, the offset from the beginning of the data that the field can be found at and the length of the field. Fields can have tags in the range 1 through 999.

Finally comes the data itself. Fields with tags 1 through 10 have a fixed format. Fields with tags 11 through 999 have subfields. The subfields do not have additional subfields. Subfields have single character tags and the tags are primarily alphabetic, but digits and even punctuation characters are sometimes used. The fields and subfields are separated by separator characters.

I have provided a routine to help with the decoding of the USMARC records; marc2dir(). It takes a USMARC record and decodes it as if it were a BER record. Even if you decide that you do not want to use the BER Utilities, this routine will give you a leg up on the decoding of USMARC records. In addition, I have provided a table at the end of this article that lists a large number of USMARC fields and their subfields and the labels that are commonly put on them when displaying them to non-librarians.

9.3.3 SUTRS

The Simple Unstructured Text Record Syntax exists to provide a minimal level of data communication. SUTRS records are essentially preformatted records. The intent is to allow the client to ask the server to format its data in a manner suitable for display to a human. The assumption is that the server probably has a better idea of how its data should be formatted than the client does, especially if they have no other record syntaxes in common.

SUTRS records are simply a single field of ASCII characters with a newline character at least every 72 characters. As the name states, there is no structure within that single field. The client should not try to parse the field looking for subfields.

9.4 The PresentRequest

The **PresentRequest** is created by a call to the *PresentRequest()* routine. It takes a **referenceId**, a **resultSetName**, a **resultSetStartPoint** and **numberOfRecordsRequested**, an **ElementSetName** and a **preferredRecordSyntax**.

The **referenceId** is a **long** and has the same meaning as in *InitRequest()*. The **resultSetName** will be one of the **resultSetNames** used in a previous successful call to *SearchRequest()*. The **resultSetStartPoint** is the relative record number from the resultSet of the first desired record. **numberOfRecordsRequested** is the count of the number of sequential records requested. The sum of **resultSetStartPoint** and **numberOfRecordsRequested** minus 1 should be less than or equal to the resultCount for the resultSet. **ElementSetNames** will be set to "F" or "B", depending on whether Full or Brief records are desired. **preferredRecordSyntax** is set to the Object ID of either USMARC or SUTRS. Preprocessor variables of **MARC_SYNTAX** and **SIMPLETEXT_SYNTAX** are provided for this purpose.

As in *SearchRequest()*, *PresentRequest()* returns a pointer to an allocated area in memory that contains the BER encoded **PresentRequest.**

The prototype for *PresentRequest()* is:

```
unsigned char *PresentRequest(
        long referenceId,
        char *resultSetName,
        long resultSetStartPoint,
        long numberOfRecordsRequested,
        char *ElementSetNames,
        char *preferredRecordSyntax);
```

9.5 The PresentResponse

The **PresentResponse** is processed by *PresentResponse()* and it, like *SearchResponse()*, takes the BER record returned by the Z39.50 server as its only parameter and returns a pointer to an allocated structure which contains the fields of the **PresentResponse**. The prototype for *PresentResponse()* is:

```
PRESENT_RESPONSE *PresentResponse(
        CHAR *response);
```

and the PRESENT_RESPONSE structure looks like this:

```
typedef struct
{
        long referenceId;
        long presentStatus;
        long numberOfRecordsReturned;
        long nextResultSetPosition;
        char recordSyntax[50];
        struct record
        {
                long len;
                char *record;
        } *records
        long error_code;
        char *error_msg;
} SEARCH_RESPONSE;
```

The **referenceId** is the same one provided to *SearchRequest()*.

presentStatus contains either a zero to indicate that there was no error during the PresentRequest or it contains a status code describing the type of problem encountered during the PresentRequest.

A value of 5 in **presentStatus** means that no records were returned and the **PresentRequest** completely failed. If this happens, there should be an **error_code** and possibly an **error_msg** explaining why the **PresentRequest** failed. The other possible values indicate why fewer records than requested where returned. Those values are described in detail in the standard. The values for error_code and error_msg are described in the Error Diagnostics appendix of the standard.

The **numberOfRecordsReturned** contains the count of records returned by the server. It should be equal to the **numberOfRecordsRequested** from the *PresentRequest()*. If it is not, then **presentStatus** should have had a value other than 0.

The **nextResultSetPosition** is set to the value that should be used as the **resultSetStartPoint** in the next *PresentRequest()* to retrieve the next sequential record.

recordSyntax will be set to the Object ID of the record syntax used by the server for the records returned. It should be the same as the **preferredRecordSyntax** used in the *PresentRequest()*.

records will contain an array of pointers to and the lengths of the records returned. The number of pointers in the array will be equal to **numberOfRecordsReturned**, even if the server accidentally returns fewer records than it claims. If this happens then the pointer will be set to NULL.

9.6 ZDEMO

Zdemo needs four things to allow it to do **PresentRequests**. It needs a way for the user to specify the **resultSetStartPoint** and **numberOfRecordsRequested,** a way to specify the **preferredRecordSyntax**, a way to specify the **ElementSetName** and a way to display the records returned.

The **preferredRecordSyntax** is specified with a new command (**r**) that takes as its single argument either the word **USMARC** or the word **SUTRS**. A global variable is set based on the argument. The default value for **preferredRecordSyntax** is **USMARC**.

The **ElementSetName** is specified with a new command (**e**) that takes as its single argument either the word **FULL** or the word **BRIEF**. A global variable is set based on the argument. The default value for **ElementSetName** is **FULL**.

The **PresentRequest** is initiated and the **numberOfRecordsRequested** and **resultSetStartPoint** are specified with a new command (**d**) that takes two optional numbers representing the **resultSetStartPoint** and **numberOfRecordsRequested** respectively. The default value for both numbers is 1.

The code in *zdemo* for parsing the two new commands is trivial and looks much like the code added to handle the search (s) command, so it will not be shown here.

Zdemo will call a new routine, *zread()* to handle the **PresentRequest**. The code for *zread()* looks like this:

```
void zread(char *parms)
{
    long                    i, numrecs=1, whichrec=1;
    PRESENT_RESPONSE        *present_response;
    unsigned char           *request, *response;

    if(*parms)  /* were any arguments provided */
    {
        char *t;
        whichrec=atoi(parms);
        if( (t=strchr(parms, ' ')) != NULL)
            numrecs=atoi(t);
    }

    request=PresentRequest(0, resultSetName, whichrec, numrecs,
        ElementSetName, preferredRecordSyntax);

    response = do_irp(request, socket);

    present_response=PresentResponse(response);
    if(!present_response)
    {
        printf("Did not get a PresentResponse!\n");
        return;
    }

    numrecs= present_response->numberOfRecordsReturned;
    printf("%ld records returned\n", nRecs);
    switch(present_response->presentStatus)
    {
        case IRP_success:
            printf("Present successful\n");
            break;
        case IRP_partial_1:
        case IRP_partial_2:
        case IRP_partial_3:
        case IRP_partial_4:
            printf("Partial results returned\n");
            break;
        case IRP_failure:
            printf("Present failed\n");
            break;
    }
    for(i=0; i<numrecs; i++)
        if(present_response->records[i].record)  /* did a record really get returned? */
        {
            char *end, *ptr;
            if(strcmp(present_response->recordSyntax, SIMPLETEXT_SYNTAX)==0)
            {  /* SUTRS records have a BER wrapper around them */
                DATA_DIR *temp=dalloc(3);
                bld_dir(present_response->records[i].record, temp);
                ptr=(char*)temp->ptr.data;
                end=ptr+(int)temp->count;
                dfree(temp);
            }

            if(strcmp(present_response->recordSyntax, MARC_SYNTAX)==0)
          {  /* convert the MARC record to a SUTRS-like record */
                ptr=formatmarc(present_response->records[i].record);
                end=ptr+strlen(ptr);
            }

            while(ptr<end)
            {  /* print each line in the record */
                char *t=strchr(ptr, '\n');
                if(t)
                    *t='\0';
                puts(ptr);
                if(t)
                    ptr=t+1;
                else
                    ptr=end;
            }

            free(present_response->records[i].record);
      }

    if(present_response->error_code)
    {
        printf("Error_code=%ld, message='%s'\n", present_response->error_code,
            present_response->error_msg ?
            present_response->error_msg:"None provided");
        if(present_response->error_msg)
            free(present_response->error_msg);
    }

    free(response);
    free(present_response);
}
```

9.6.1 Displaying USMARC Records

Decoding USMARC records is beyond the scope of this article, but the code to accomplish it is provided as part of *zdemo* at OCLC anonymous FTP site. (See the section of Source Code Availability at the end of this article.)

10. Terminating the Z39.50 session

In Version 2 of Z39.50, both the client and the server are allowed to terminate the session at any time, simply by dropping the TCP/IP connection between them. The routine *disconnect()* has been provided to do this. It accomplishes this by closing the socket with a call to the *fclose()* routine (one of the standard C i/o routines.)

11. Summary

This article has described the elements of Z39.50 necessary to create a simple client. Many of the more complex elements have been mentioned in enough detail that you should have some idea if you need them. Hopefully the code provided and its discussion have shown you that while it is not trivial to build Z39.50 applications, neither is it terribly complex.

12. Source Code Availability

The source code for the Z39.50 Client API and *zdemo* is available via anonymous FTP at ftp.rsch.oclc.org in the pub/SiteSearch/z39.50_client_api directory. A copy of this article, all the source code and user documentation for the Client API can also be found in that directory.

The BER utilities used by the Client API can be found on the same host in the pub/BER_utilities directory.

OCLC maintains their copyright to all these materials, but they have been made freely available to all developers.

12.1 License

Implementing Explain

Denis Lynch
TRW Business Intelligence Systems
dml@bis.trw.com

Abstract

The Explain facility is the primary mechanism for Z39.50 clients to discover servers' capabilities. Explain-based clients can dynamically configure their user interface (or other search capabilities) to exactly match individual servers. This allows generic clients to access a wide range of Z39.50 server, and allows any client to adjust to changes in server configuration.

The Explain facility is defined by an abstract record structure and attribute set, with no additional protocol mechanisms. Still, effectively using Explain is a relatively large undertaking. This paper describes the most important issues to be considered, and suggests the most important features to implement first.

Introduction

The Z39.50 protocol allows clients and servers to work together to provide users with tailored access to information. This flexibility presents a challenge to client developers: how will a client know how to deal with a specific server? The traditional choices have been:

- Make the client very general, but very simple, providing only "least common denominator" access.
- Build the client with specific knowledge of the server(s) it will access, providing tailored access to a limited set of information sources.

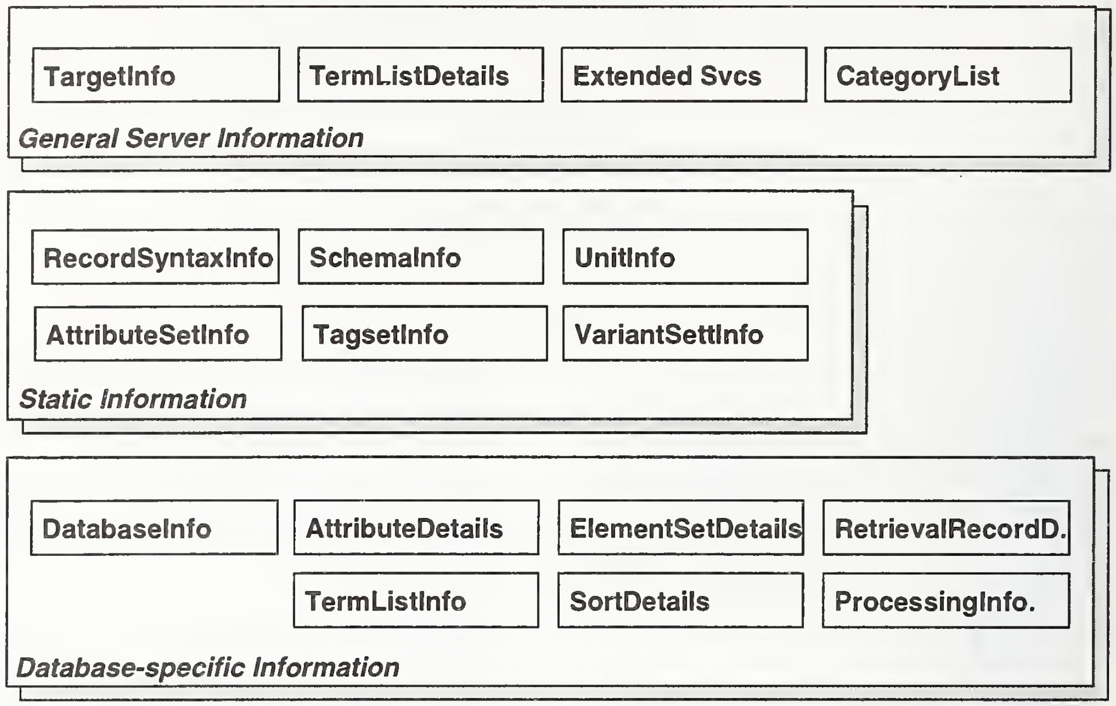
Early in the development of Z39.50 it became apparent that the least common denominator was not useful for most applications. On the other hand, building server knowledge into clients is cumbersome and dangerous: even the most stable environments change; a client with obsolete server knowledge will give its users nasty surprises. The Z39.50 environment reveals this problem more than some others precisely because of the specificity that it supports. For example a user might have a request like "Return Canadian MARC records for all the items in this database with MESH subject starting with 'symptom'". But while such a specific request will get exactly what the user wants when the server supports the specific search access point and record syntax, few servers do so.

Collectively the Z39.50 Implementors' Group (ZIG) determined that dynamic discovery of server capabilities was the most promising way to get good client behavior. The best way to communicate the dynamic data from servers to clients was seen to be the very search and retrieval facilities that are the core of the Z39.50 protocol. Server characteristics were divided into categories, and database record structures were defined to contain the information in each category.

Explain records organize human-readable information as well as information to be used internally by client software. For example, the server description record (TargetInfo) includes separate human-readable items for an overall description, usage restrictions, and operating hours as well as items client software can use directly such as network addresses and the maximum number of result sets supported.

Capturing server information in a searchable database makes supporting Explain relatively easy, since both clients and servers will necessarily have most of the required capabilities. It also meant that the ZIG's energies could be devoted to defining the dynamic information requirements without concern for new protocol capabilities.

Implementation experience has proven that sophisticated clients can be built that use only Explain information to configure their user interface and key operating characteristics.



Overview of the Explain information structure

The remainder of this paper describes the Explain database structure and how clients and servers can use Explain information effectively. The paper does not exhaustively cover the Explain database structure. Readers are referred to the 1995 Z39.50 standard, in particular section 3.2.10, Appendix 3 section ATR.2, and Appendix 5 section REC.1.

The Explain Database

The Explain database operates within the Z39.50 protocol exactly like any other database. Each server's primary Explain database is named IR-EXPLAIN-1; a server may have additional Explain databases as well (see the Surrogate Explain section below). Because predictable behavior and content are important to the internal workings of clients, Explain databases are searched using a dedicated attribute set, and records are retrieved in a dedicated ASN.1 syntax. (Other alternatives, such as the Generic Record Syntax, were considered, but the concreteness and specificity of ASN.1 and the Explain-1 attribute set were compelling.)

An Explain database contains three basic kinds of records:

- Static information that should not vary among servers, e.g. attribute set definitions
- Database-specific information, e.g. available record syntaxes
- Information that applies generally to the server, e.g. extended services available.

General server information

Four types of records apply across databases:

- The TargetInfo record describes the server as a whole
- The CategoryList lists the kinds of Explain records contained in the database
- ExtendedServicesInfo records describe each Extended Service supported by the server
- TermListDetails describe each term list supported in the target (term list names appear in TermListInfo records for specific databases; multiple databases can share the more detailed TermListDetails descriptions).

Static Information

One of the sources of Z39.50's power and flexibility is the degree to which important concepts are modularized into entities that may be defined externally to the standard. Two such entities—Attribute Sets and Record Syntaxes—are central to using Z39.50. Four additional entities play important roles in Version 3: unit systems, database schemas, record tag sets and variant sets. Instances of all six of these entities are identified by an object identifier or a unique name.

Since the definition of each of these entities (for example a specific attribute set) is intended to be universal, clients and servers could have complete *a priori* knowledge of the definitions. In practice, though, clients can allow users to use servers without having built-in knowledge of the semantics of these entities. For example, a client can use information from a UnitInfo record to allow a user to enter a numeric term and specify that it is a length in furlongs, where the unit type and unit (i.e. "length" and "furlongs") are transparent to the client.

The static information categories contain enough details about the entities to support this kind of transparent client behavior.

Like all Explain records these static descriptions can contain human-readable text that clients simply display to users, allowing users to understand aspects of the server's operation that the client software does not necessarily understand.

Database-specific Information

Most of the dynamic content of an Explain database relates to a specific database on a server. Each database is described by a DatabaseInfo record; additional records describe specific aspects of the database:

- AttributeDetails lists the attributes that can be used in a Type-1 or Type-101 query, and how those attributes can be used in combination.
- TermListInfo lists the term lists ("indexes") that apply.
- ElementSetDetails and RetrievalRecordDetails describe record retrieval options.
- SortDetails lists the available sort options.
- ProcessingInformation allows a server to provide clients with specific instructions such as search forms and record formatting.

Surrogate Explain Databases

A single Explain database describes a single information service. There are many reasons that a Z39.50 server might wish to provide Explain data about a service other than itself (for example if the other service does not have an Explain database). We refer to such a server as a "surrogate Explain server," and the corresponding database as a "surrogate Explain database." The mechanism for this is straightforward:

- The surrogate Explain server provides its own Explain data in the standard place: database IR-EXPLAIN-1.
- The surrogate Explain server provides Explain data for additional servers as separate databases. By convention the names of surrogate Explain databases should be the URLs for the server they explain. (That is not a requirement, and there are times when other names *must* be used.)
- Each surrogate Explain database should naturally be described in the IR-EXPLAIN-1 database. Only the DatabaseInfo record is required, and two of its elements are especially important to clients:
  - explainDatabase (a Boolean flag) informs the client that the database is an Explain database
  - lastUpdate gives clients an indication that cached data may be obsolete.

When a server follows these conventions a client can determine what surrogate Explain data is available by searching IR-EXPLAIN-1 for

> ExplainCategory = "DatabaseInfo"
>
> AND
>
> ExplainDatabase AlwaysMatches (*any term*).

The client is likely to need the list of databases in any case, so it can simply search for ExplainCategory = "DatabaseInfo"; then Brief records will identify all the server's "normal" databases and surrogate Explain databases.

Using Explain in a Client

Even the most basic Z39.50 client can use dynamic information from Explain to great advantage. The most obvious application is search attributes, but the opportunities are much greater.

- Network addresses. If the Explain data is coming from a surrogate Explain server, the surrogate Explain server might recommend a network address different than the destination server name. It is possible that a non-surrogate server would provide alternative addresses—but these would only be discovered after a connection to the original address had succeeded!

- Target operation parameters. The TargetInfo record contains more detail about the server than Init negotiation provides. A client can use the namedResultSets and multipleDBsearch flags for named result sets and multi-database searching to avoid sending searches that the server will fail. (Clients can learn that named result sets are not supported during initialization if Version 3 is negotiated, but not if Version 2 is negotiated.)
- Searching.
  - Whenever a TermListInfo record is available for a database a client should use the listed termLists as the primary search access points. For each access point (or index) to a database, the TermListInfo record includes a title for each term list, intended for display to the user (possibly in multiple languages), an indication of the cost of using that access point, and whether the term list is scannable. The attributes in a term lists's TermListDetails record specify which attribute combinations address that term list.

    A server administrator identifies term lists as specific ways to search a database, so distinct term lists can be presumed to be truly different. On the other hand, the same administrator will list as many attribute values as possible in AttributeDetails, even though some are treated identically.
  - AttributeDetails lists all legal attributes for a database; its attributeCombinations lists how those attributes can be combined in a single operand. Attribute names and descriptions can be found in AttributeSetInfo records; database-specific descriptions may be included in the AttributeDetails.
  - The associatedDbs element of a database's DatabaseInfo record lists the databases that can be searched in combination with that database. (Similar information may be available in the TargetInfo record's dbCombinations element.)
- Retrieval. The ElementSetDetails and RetrievalRecordDetails records associated with a database identify the record syntaxes and element requests that are sensible for that database. Knowing the full list of alternatives allows a client to choose appropriate element set names, even if the standard "F" and "B" are not appropriate. (This could happen, for example, if "F" and "B" are supported only for a record syntax the client can't accept.)
- Scan. Scan requests use attribute combinations to address term lists. A client may choose to attempt to scan with any supported attribute combination, but using only term lists with the scanable flag set will avoid unnecessary errors.
- Processing instructions. These are currently usable only by private agreement between the client and server. General purpose formats will be defined in the future.
- Extended Services. Each Extended Service supported by a server is described by an ExtendedServicesInfo record. Clients can use this record to learn whether ES packages will be retained in the ES database, and whether the request can be issued with the wait flag.
- Access requirements. Both the TargetInfo record and DatabaseInfo records may contain AccessInfo elements. Clients can use this access information to determine if they can usefully access the server and specific databases. The most important element in this regard is restrictedAccess. If restrictedAccess indicates that access to a server or database is restricted, the client will be able to access the server or database successfully only if it supports one of the listed accessChallenges. In some cases a client might learn that a database doesn't support any of the query types the client supports. (This is unlikely, since essentially all clients and servers support Type-1 queries.)
- Human-readable descriptions. The various description fields—particularly in TargetInfo and DatabaseInfo records—should be available from a client's Help system. The information in these fields may explain to a user why operations behave as they do, as well as describing the contents of the databases.

Getting Explain records

Explain databases are accessed using the search and retrieval facilities of the Z39.50 service. The exp-1 attribute set includes USE attributes that are tailored for searching Explain databases. Each exp-1 attribute corresponds to a specific element in an Explain record. Some of the attributes correspond to elements that appear in more than one type of record (e.g. ExplainCategory and DatabaseName), while others correspond to elements that currently appear in only one type of record (e.g. ExtendedServiceOID).

Several attributes require specific search terms, as described in the definition of exp-1.

Exp-1 only defines USE attributes; attributes of other classes are the same as Bib-1 attributes.

Explain databases can be accessed effectively using the Version 2 present facility, in particular simple element set names. The Explain record syntax definition identifies the elements that must appear in brief records (element set name "B"), these include all of the identifying elements of the record. Full records (element set name "F") must contain all elements present in the database. The element set names "description", "specificExplain" and "asn" are also defined in certain cases as detailed in the syntax definition.

Search

Finding desired records in an Explain database is generally very simple. The canonical search is:

> ExplainCategory = *category*
> AND
> *identifier* = *value*

For example

> ExplainCategory = "DatabaseInfo"
> AND
> DatabaseName = "CATALOG"

A few records have multi-field keys (e.g. ElementSetDetails and ProcessingInformation) which means that known-item searches require additional terms.

As with most things, it is best to keep the searches as simple as possible. TRW's Explain software, for example, never sends attributes other than USE, and always uses the general choice for encoding search terms.

Retrieval

The safest policy is to assume that a server has 100% recall, but unknown precision. Clients should therefore be prepared to accept records that weren't asked for. In particular, the desired record may not be the first one in a result set. (Perhaps even more annoying: the search may return many records, but *none* is the desired record!) It is wise for the client to remember the result set positions of these "unwanted" records in case they become wanted later (e.g. if a target returns many DatabaseInfo records when only one was wanted, the client can avoid later searches by retaining the positions of the "unwanted" records).

To minimize round-trip delays, all searches should request piggyback presents. In most cases the piggyback should only request brief records, but known-item searches (searches that expect to find no more than one record) could set a small-set size of one, and request full records for the small set.

Explain servers may not handle present requests exactly as issued. The two most obvious things clients should be prepared to handle are:

- Piggyback present is not required, and some common servers do not honor it. Clients should be prepared to issue present requests if a search response contains no response records.
- Brief records are not required, so clients should examine the records they have received before issuing a redundant present request. If any non-brief elements are present the record isn't brief. It would be most irregular for a server that doesn't provide brief records to make other element set distinctions, so in practice it is safe to assume that non-brief records received for "B" are full records.

Although it isn't strictly necessary, specifying the preferred record syntax as Explain is a reasonable safety measure.

Handling errors

The most common error received from Explain searches is "Database does not exist." This should be noted carefully—there's no reason to try again! A server may react in other less helpful ways, for example "Unsupported attribute set" or "No records syntaxes available." It is simplest to assume that almost any error means that the server doesn't support Explain at all. (Bib-1 error 27—Result set no longer exists—is one exception.)

Once connected, servers frequently time out idle connections. Since Explain is best handled "behind the scenes," the connection should simply be re-established the next time it is needed.

Default configuration

The conventions described in this section are used by TRW's software. They have been proposed to the ZIG as possible implementors' agreements.

For servers that provide many similar databases it may be simpler to describe them only once. The easiest way to do this is to describe a database named "Default". A client that finds no record of a particular type (e.g. AttributeDetails) for a specific database should try to find the same record for the database "Default". In many cases that record will have already been retrieved as fallback for a different database.

A surrogate Explain server can provide default client configuration information for specific destination servers, as well as providing fallback default behavior. To obtain information about a destination server from a surrogate Explain server, a client searches a sequence of Explain databases. (An Explain database is searched only if the surrogate Explain server's IR-Explain-1 database has a DatabaseInfo record for that Explain database.)

- The URL for the destination server is used as a database name (e.g. the database name "z39.50s://rlg.stanford.edu" is used when looking for information about RLG's server)
- A similar string is used to find a database of default information to be used for the specific access (e.g. "z39.50s://default" for Z39.50 access)
- A fallback database is used for generic default information ("default://default")

The default Explain databases are structured exactly like other surrogate databases.

Minimizing impact on servers

Explain searches require roughly the same amount of server attention as real information searches. This makes it important that clients behave reasonably! Here are a few techniques:

- Cache explain records, and don't ask the server for information that has already been retrieved. Generally it's not a good idea to "download" a server's whole Explain database—just cache the records that are retrieved in the normal course of operation.

  The biggest problem with cached records is knowing when to throw them away. The strategy TRW adopted is to delete all cached information about a database whenever the CommonInfo in a DatabaseInfo record has an update (or creation) date more recent than the corresponding DatabaseInfo record in the cache. This works well because the DatabaseInfo records are retrieved from the server on a regular basis (see the next item), and the CommonInfo is in brief records.
- Keep lists, and check the appropriate list before asking the server a question. This technique applies especially to DatabaseInfo records. A client can avoid searches by noticing that there is no DatabaseInfo record for the relevant database (e.g. when looking for AttributeDetails for a specific database, or for surrogate Explain information about a specific server). One scheme for managing these lists is as follows:
  - Initialize the list before any records are retrieved from the cache. (Each cache will have several lists.)
  - As records are retrieved from the cache, add them to the appropriate list.
  - When a specific record is needed, retrieve records from the cache until either the record is found or the whole cache has been retrieved.
  - When all the records have been retrieved from the cache and the needed record hasn't been found, ask the server. But instead of asking the server for a specific record, ask for the list (e.g. "all DatabaseInfo records"). As the records from this list are retrieved the local cache can be checked for out-of-date information.
  - After the server's list has been completely retrieved a record that isn't in the list doesn't exist—there is no need for additional searches.
- If a surrogate server is supplying the Explain data, check for the destination server's surrogate Explain database in the list of Explain databases before accessing the surrogate database.
- Remember result set positions of records to avoid redundant searches. As Explain records are encountered in Present responses, the client should retain the result set name, result set position, and identifying information (ExplainCategory plus, for example, DatabaseName). When the client needs a record that was incidentally received, this retained information lets the record be Presented from the existing result set without another Search.

Building an Explain Database

"Using Explain in a Client", above, points out most of the issues with populating an Explain database. This section points out a few additional considerations for server administrators.

- Make use of default configurations wherever possible. In a surrogate Explain system it makes sense to provide default TargetInfo records, for example to make clients presume that multiple database searching is not supported.
- Consider carefully what attributes to include in Explain. There is no reason to list attributes that are treated as synonyms just for compatibility. Clients that rely on dynamic configuration will be misled by these attribute aliases; clients that don't use the dynamic configuration won't notice that the aliases are missing.
- Provide term list information if at all possible.
- Include AttributeCombinations to let clients filter out illegal search requests. The AttributeCombinations structure is designed to avoid combinatorial explosion by specifying patterns that describe a set of similar combinations, for example:

- With USE attributes 4, 20, 21, 62 and 1000 the RELATION attribute may be omitted or value 3 may be supplied, and the STRUCTURE attribute may be omitted or values 1, 2 or 6 may be supplied
- For USE attributes 30, 31, 32, 1011 and 1012 the RELATION attribute may be omitted or values 1, 2, 3, 4 or 5 may be supplied, and the STRUCTURE attribute may be omitted or values 5 or 100 may be supplied.

- There is no need to include individual attributes in an AttributeSetInfo record if no databases on the server use those attributes.
- If a CategoryList record is provided, it should list only categories for which at least one record is available. The primary purpose of the CategoryList is to allow clients to learn about Explain extensions supported by the server, but it can also allow clients to skip searches for records that the server doesn't have.

Serving Explain Records

Any fairly capable Z39.50 server should be able to process Explain requests. Unlike user-initiated requests, Explain requests are generated from low-level client code. They will generally be fairly simple, and will not take advantage of Explain information themselves. Servers should therefore make every attempt to process Explain requests without returning the kind of diagnostics that might help a human user refine a request.

There are a few specific requirements:

- The Explain records syntax and attribute set must be supported.
- Since Explain searches will be generated automatically by the client software, support for named result sets is nearly mandatory. The cost of retaining the result sets will surely be less than the cost of re-executing searches.
- If Version 3 is negotiated, the search engine must be prepared to process search terms sent in the OBJECT IDENTIFIER choice.
- In Version 2 or Version 3, the search engine must be prepared to deal with object identifier terms sent as character strings as specified is Appendix ATR.2, note (4).
- Support of element sets other than "F" is not required. But a Brief record should *never* contain any non-brief elements: clients need to be able to look at a received record and determine whether it is Full. As described above, clients will reasonably presume that records with non-brief elements are non-brief records.
- Support for piggyback present is not required.
- Unlike processing user-requested searches, a server should simply ignore Explain search operands with unsupported attributes. If the server fails the search instead, the client will (at best) reformulate a broader query without the offending attribute. Ignoring the unsupported operand avoids this inevitable round trip.
- A useful Explain service can be provided with only a few attributes. The minimal set of attributes is very short:
  - ExplainCategory
  - DatabaseName
  - TermListName (if term lists are provided)

  The next attributes to add are the identifiers from other records supported by the server:
  - AttributeSetOID
  - RecordSyntaxOID
  - TagSetOID
  - SchemaOID
  - ExtendedServiceOID
  - VariantSetOID
  - UnitSystem

  The next tier of attributes is:
  - HumanStringLanguage (if there is more than one)
  - DateChanged
  - DateAdded
  - DateExpires

Only a very simple search engine and no specialized database are required. The Explain records can simply be stored as BER-encoded files. The TRW Explain Editor and server add some small twists to this:

- Each Explain database is stored as a separate directory.
- Within an Explain Database directory, subdirectories are created to store records pertaining to specific databases.
- Records that are not specific to a single database are stored in the top-level Explain Database directory.
- Two special files are created in every directory:
  - title.trw contains the title of the database. Our software makes no use of directory names because of character set and length restrictions.
  - contents.trw contains a list of all the files in the directory along with the values of the searchable fields (including creation/update dates). This isn't much data, but it allows the directory to be indexed or searched without accessing the actual Explain records.

Conclusion

The Explain database structure appears daunting to new readers in size and complexity: its ASN.1 definition is slightly larger than the definition of the body of the Z39.50 protocol, and the records are closely interrelated (like the protocol itself, the Explain database was designed for economy and modularity).

This paper has described how the most general parts of the Explain database are used in practice, and how software can be designed to provide and access Explain databases efficiently. The paper is based on experience gained in building a reasonably complete end-to-end Explain system: a graphical editor for creating and maintaining the data, server extensions to provide the data, and a client that relies exclusively on Explain records for configuration.

The author can be reached at dml@bis.trw.com.

Implementing Z39.50 in a multi-national and multi-lingual environment

Makx Dekkers
Pica - *Centrum voor bibliotheekautomatisering*
Leiden, The Netherlands
(dekkers@pica.nl)

Abstract

Z39.50 provides a very useful tool for intersystem communication, but it also demonstrates that differences in language and culture have an impact on the scope and usefulness of international services.

A number of problems arise when implementing Z39.50 in a multi-national and multi-lingual environment. In the U.S. implementers group, these problems are not always obvious.

Problems that are being identified by current implementers in Europe include differences in character sets between countries, different sorting order of characters in different languages, different rules for conversion from 8-bit to 7-bit ASCII for indexing dependent on country and language, difficulties in translating system messages, variety of MARC formats and differences in cataloguing rules between countries.

Internationalisation of the standard can solve some of the problems. Hopefully, through the implementation and use of EXPLAIN some of the others can be explained to users. The aim should be to make it possible, through Z39.50, to provide services to a wide international audience, respecting the multitude of cultures and languages in the world.

Introduction

As a result of the possibilities offered by the introduction of network technology in the past decades, exchange of information through communication between computer systems has become an everyday phenomenon in today's world.

With the success of Z39.50 in North America, organizations in non-English speaking areas are becoming interested and now are identifying problems with the essential Anglo-American scope of some aspects of Z39.50. Obvious examples are: the use of basic 7-bit ASCII, and the relation with the Anglo-American Cataloguing Rules AACR2. It should be stressed that the issue is not a American-European dichotomy but rather one of language, i.e. English versus non-English communities.

While it is clear that Z39.50 provides a very useful tool for intersystem communication, experience from international projects indicates that solutions will have to be found for national differences such as language, character sets, cataloguing rules and data formats, to make the standard globally acceptable.

Problem areas

It should be stated that many of the problems that are identified by non-English implementers are not inherent to Z39.50, but are merely made visible because of the new possibilities of intersystem communication.

A major category of problems is that of external incompatibilities. These problems are related to cultural differences: national bodies are responsible for definition and maintenance of national rules for bibliographic descriptions; more fundamentally, different countries use different languages with different character sets and sorting rules.

Almost all countries have bodies that define the national rules for cataloguing of publications. Large bibliographic utilities sometimes define their own (additional) rules. Although some coordination is taking place, such as with the AACR2 rules, the independence of these national bodies introduces some fundamental incompatibilities when organisations in different countries want to interchange bibliographic descriptions through Z39.50.

An illustrative example of this is the treatment of multi-volume publications; under some rules these are catalogued as one single record with repeated elements, under other rules they are described as separate entities with relations between them. Another example is the use of standard phrases in national language within the cataloguing rules, such as for title changes for journals (in Dutch cataloguing the description would contain the phrase "Voortgezet als:"). International exchange of such bibliographic descriptions would ideally involve automatic translation; however, this is not being done in practice.

For searching, standardised keyword lists are usually defined in national language and subject code systems are agreed in a national context. These different language and country related rules and practices cause incompatibilities that are difficult or sometimes impossible to overcome.

A number of interworking problems are associated with differences in character sets. There are many scripts being used in the world and most of the existing library systems are unable to handle them all. Transcription rules or character set conversions will sometimes lose information since they are not always one-to-one reversible. Transcription rules are generally dependent on the combination of source and target language, e.g., Russian Cyrillic will be transcribed differently in the Netherlands, Germany and France. Furthermore, transcription rules sometimes change over time. Especially in searching, it is difficult for a user to determine what transcription should be used: will it be Nabokov, Nabokow, Nabokoff? How does one search for the person referred to in the Netherlands as Aleksandr Isajevitsj Solzjenitsyn? In practice, in current multi-national communication, the textual information is transferred in 7-bit ASCII (ISO 646). This limits the scope of communication to languages with a Latin character set and even then the results are fundamentally insufficient.

In sorting, the situation is even more complicated. The same character set may be used in two different languages, yet the sorting order might be different. In some languages "o-umlaut" is sorted as "oe", in others it might appear at the end of the alphabet. Even when the same language is used in two countries, there might be differences in sorting order of names: in Belgium a personal name of "Van Dam" will appear under "V", in the Netherlands under "D".

All these practices make it difficult to predict what the results of search commands will be; sometimes items will not be found that are in the database, sometimes they will just appear in unexpected places in a sorted list. Users do not always realise these difficulties and the occasional user might turn away in disappointment.

Users would like to look at title descriptions that were found as a result of the search action. This is not a trivial task. The target database records might be stored in a format that the user is unfamiliar with. The systems will then have to provide some form of format conversion. Even between MARC formats, which are at least structurally compatible, format conversions almost invariably lose some of the information. If for some reason format conversions are not possible, the user will either see an unfamiliar format or a description can only be displayed as unstructured text, which might be fine for an end-user doing reference work, but would be close to useless to a librarian.

Finally, there are also some elements of the standard itself that pose problems to users in multi-lingual environments. For several services, the target system may convey diagnostic information to the origin. The standard does not prescribe the origin behaviour when such messages are received, so the origin might not know anything better to do than show the diagnostic information to the user. This information could be in a language the user is not familiar with. Obviously, the standard is not supposed to prescribe any external behaviour but to users this is not really helpful.

Solution scenarios

In practical implementations of the standard, solutions to the above problems take a very pragmatic approach. In one way or another, they all take the lowest common denominator and make the best of it. This either leads to solutions that cut away all complexity and settle for very limited functionality, or to highly parameterised implementations of the form: "If talking to A do this, if talking to B do the other". The language and character set problems are commonly 'solved' by standardizing on English and basic ASCII.

For real solutions, a first essential step is for developers and implementers of the standard to become aware of the problems that are introduced by its use in multi-national and multi-lingual environments. This awareness can not be of a simple theoretical nature; it can only lead to practical steps if there is a business case to justify extra investments.

One general step towards solving the incompatibilities outlined above is the introduction of negotiations, dynamic conversions, and powerful explanation techniques. Negotiation aims at establishing a mutually agreed environment. If this cannot be achieved, the data that is exchanged can be converted from one format to another. If that is impossible, the user should receive some information to explain why the result is not as expected and to suggest alternative actions to maximise the user's efforts.

In the final text of the 1995 version of the standard, a mechanism has been incorporated for negotiations between origin and target. This mechanism may be used to negotiate the character set used for textual information and the language of messages. This is an essential improvement over the 1992 standard, allowing multi-lingual systems to take advantage of their capabilities across a Z39.50 communication. The character sets that can be used are ISO 10646 and ISO 2022, or mutually agreed private character sets. If negotiation cannot be completed successfully, the situation is basically that of the 1992 standard: the target determines the language and the only safe assumption for the character set is that it will be 7-bit ASCII.

For the data formats, it is clear that national or local rules will remain important to determine the way information is stored in databases. Possibly some of the MARC formats will become widely accepted as an exchange format; at the moment, USMARC and UNIMARC seem to be the dominant formats in several projects. It is also clear that this will not be a solution for systems not governed by purely bibliographic rules that do not usually store or export their data in MARC. A positive development is that some European projects are building table-driven, public domain toolkits for format conversions. Although 100% accuracy in conversion cannot be achieved, this might help in broadening the scope of Z39.50 interoperability.

In areas where negotiation or conversions cannot solve the problems, the use of the Explain facilities defined in the standard would have to provide the solution. This facility is probably the most powerful feature of Z39.50. It can be seen from the complexity of the Explain facility that the problems are manifold. Through Explain, the client is given information that can be used to help the user to better understand what goes on behind the scenes and to allow him or her to make sense of the results of certain actions. The problem, of course, is that what makes perfect sense for one user might be completely illogical to another. At best, Explain will provide a general information level to an average user. Still, this is better than nothing and fortunately, all messages in Explain have been designed for multi-lingual environments. As a drawback, the maintenance of an Explain database can be a considerable task.

Even with the above solutions, there will remain elements that cannot be negotiated, converted, or explained during a session. There will always be out-of-band bilateral agreements between Z39.50 partners and there will always be situations where two Z39.50 systems cannot communicate to provide a useful service to users. As an example of the latter, imagine that the Z39.50 target provides access to a Chinese database and the origin has no way of displaying Chinese characters to the user. It should be clear that a standard like Z39.50 is just a vehicle for communication and will never be capable of solving all the problems that exist; neither should it try to do this.

Conclusion

Z39.50 provides a very useful tool for intersystem communication but it is clear that differences in language and culture have an impact on the scope and usefulness of international services. Internationalisation of the standard can solve some of the problems. Hopefully through the implementation and use of Explain some of the others can be explained to users. The aim should be to make it possible, through Z39.50, to provide services to a wide international audience, respecting the multitude of cultures and languages in the world.

Use of Z39.50 for Search and Retrieval of Scientific and Technical Information

Les Wibberley

Chemical Abstracts Service
les.wibb@cas.org

Abstract

A common perception of Z39.50 is that it defines a simple information retrieval standard for bibliographic data. But in its latest version, Z39.50-1995, the standard has evolved into a rich set of interoperable services that can be used for client/server-based search and retrieval within literally any information discipline.

This paper addresses the use of Z39.50 for search and retrieval of scientific and technical information. This topic is explored within the context of an implementation of Z39.50 by Chemical Abstracts Service (CAS). The paper explores some of the practical implementation issues encountered, the solutions applied, and lessons learned.

Introduction

Z39.50 is an American National Standard that specifies an interoperable protocol and services for information search and retrieval. The Z39.50 protocol specifies formats and procedures governing the exchange of messages between a client and server, enabling the client to request that the server search databases for information that meets specified criteria, and to retrieve some or all of the identified information.

This paper addresses the use of Z39.50 for search and retrieval of scientific and technical information. It explores some of the practical implementation issues encountered, the solutions applied, and lessons learned. Specific topics include the use of Attribute Sets, Record Syntaxes, Element specification, Segmentation, OtherInformation, and Extended Services. These topics are explored within the context of an implementation of Z39.50 by Chemical Abstracts Service (CAS).

A brief history of Z39.50

The Z39.50 protocol was originally proposed in 1984 for search and retrieval of bibliographic information. The first version of Z39.50 was prepared by a committee of the National Information Standards Organization (NISO), and was approved as an ANSI standard in 1988 [1]. Early implementations of Z39.50-1988 included WAIS (Wide Area Information Servers) and OCLC systems. Within this paper, Z39.50-1988 will be referred to as "V1."

As interest in Z39.50 broadened, the Z39.50 Implementors Group (ZIG) was established in 1990. The formation of the ZIG was a positive step for Z39.50, since it allowed new versions of the standard to be guided, driven, and defined by the needs and experience of implementors. This lent a practical balance to the academic and theoretical viewpoints that have traditionally influenced standards.

Enhancements proposed by Z39.50 implementors were coupled with changes necessary to align Z39.50 with its international counterpart standard, ISO 10162/10163 [2]. This work led to the second version of Z39.50, approved as ANSI standard Z39.50-1992 [3]. The improved interoperability and functionality of this "V2" standard triggered a large number of successful implementations.

Development of the third version of Z39.50 began in late 1991. Several major enhancements and extensions were proposed by implementors for this third version, to support a wider scope of information retrieval activities. From December 1991 through September 1994, a progression of "V3" drafts was developed by the Z39.50 Maintenance Agency, based on ZIG proposals. Each draft underwent careful scrutiny by implementors, and was discussed at length over the ZIG electronic mail list and at the ZIG meetings. Z39.50-1995 [4] was balloted in the fall of 1994. At the time of this paper's writing, June 1995, all ballot objections and comments had been resolved, and final approval of Z39.50-1995 was underway. Since one of the goals of Z39.50-1995 is to support interoperability with Z39.50-1992, it includes specification of both "V2" and "V3" of the protocol.

CAS interest in Z39.50

Chemical Abstracts Service (CAS) is a world leader in scientific and technical information, with heavy concentration on chemistry-related sciences [5]. In additional to its traditional publishing and CD-ROM products, CAS builds and licenses scientific and technical databases, and provides online access to these and other licensed databases through STN International, the Scientific and Technical Information Network. CAS also develops and licenses search and retrieval software for accessing scientific and technical databases.

As a member of NISO, CAS has tracked and voted on the Z39.50 standard since its inception. CAS interest in

Z39.50 was due to its potential for providing a single robust, standard, interoperable protocol for search and retrieval. Since Z39.50 is based on the client/server model, the user interface, protocol, server, search engine, and database management components can each be treated as independent modules, providing greater architectural flexibility. Like other commercial online services, CAS has traditionally used a proprietary user command language and protocol for accessing its databases and services. Z39.50 appeared to be a potential candidate as the protocol for the next generation of client/server systems, offering both expanded functionality and interoperability.

Z39.50-1988 was too limited to be used for interoperability between sophisticated information retrieval systems, especially in a commercial context. However, by 1991, increasing interest in Z39.50 indicated that it might develop into a widely supported standard within the industry, especially if it could be enhanced to support a broader context of information retrieval. At that point, CAS joined the ZIG and became an active participant in the development of both V2 and V3 of Z39.50.

Z39.50 Implementation Issues

CAS started an experimental implementation of Z39.50 in 1991, in order to evaluate its potential for providing interoperable access to scientific and technical (scitech) information. In the process of this evaluation, several issues were identified.

1. Protocol scope and extensibility

Initially, Z39.50 would be used to provide access to existing databases using an existing search and retrieval system. It would therefore be necessary to build a gateway to translate between the Z39.50 protocol and the existing search system's protocol.

The first issue encountered was the fact that the functionality defined by Z39.50 V2 was a small subset of the functionality of the search and retrieval system to which it provided access. And the protocol was not extensible to support additional functionality, without sacrificing interoperability.

There was a lack of extensibility at two levels. First, there was no way to add supplementary data to the Protocol Data Units (PDUs) defined for a given service within the protocol. And secondly, there was no interoperable way to add supplementary services to the protocol.

2. Attribute Set limitations

Using the Z39.50 protocol, a client sends its Query in a Search Request Protocol Data Unit (PDU) to the server, which executes the search and returns a Search Response PDU back to the client, indicating the results of the search operation.

Z39.50 supports several query types within its Search Request PDU, but the one required query type is the Type 1, or Reverse Polish Notation query. The Type 1 query allows any number of search terms to be combined with boolean logic. Each search term can be qualified by one or more Attributes, which identify characteristics of the term, such as: how the term is to be used (Use Attribute); whether it is truncated (Truncation Attribute); how it is structured (Structure Attribute); etc. Z39.50-1992 defined a "bib-1" Attribute Set, which included Use Attributes for most of the common bibliographic fields, but did not include Use Attributes for other information disciplines, such as scitech information.

Therefore, the second major issue encountered was how to identify and characterize scitech query terms within a Z39.50 Search Request.

3. Search service limitations

In addition to the search term Attribute limitations, there were also several other implementation issues related to searching.

a. The V2 Search Request only allowed one Attribute Set to be specified. This precluded the possibility of mixing Attributes from two or more Attribute Sets in the same Search Request.
b. There was no mechanism for specifying the datatype of a given term within a Type 1 query. Since bibliographic searches generally required support for textual terms only, this had not been a major problem. However, for scitech searching, terms might be expressed as integers, real numbers, externally defined structures, binary data, or a variety of other datatypes.
c. There was no way to specify the "units" in which a search term was expressed. For example, a search term specifying a boiling point may need to qualify whether the term value is expressed in degree Celsius, Fahrenheit, or Kelvin.
d. There was no way to send or return additional search information. V2 PDUs contained no extensibility features for carrying information not explicitly defined by the ASN.1 structure of the PDU. Thus, information about the "Type" or "Scope" of a search could not be expressed in the Search Request. And additional information about the search results (such as how many "hits" were found for each term in the query) could not be returned in the Search Response.

4. Element Specification

Within the Z39.50 model, once a search is completed, the matching records from the database are represented by a Result Set, logically containing one "record" per "answer." A Z39.50 client may retrieve one or more records from the Result Set, using the Present service. Within V2, the only way for the client to specify the particular data elements to be retrieved from the Result Set records is to specify an ElementSetName in the Present Request PDU. Only two ElementSetNames were pre-defined by the standard: Brief and Full, both essentially defined by the server. Other ElementSetNames also were to be "primitive" in nature, meaning that a name would designate a pre-defined set of retrieval elements.

Database records within scitech databases tend to contain a large number of data elements, and the particular selection of elements to retrieve may vary considerably, depending on a given client's or user's needs. Pre-defining unique ElementSetNames for each combination and permutation of the retrievable elements for each database supported by a given server is a tedious and error-prone task. Furthermore, there was no defined mechanism for the client and server to share a common understanding of these ElementSetNames in a way which would ensure general interoperability.

5. Record Syntax limitations

When a Z39.50 client retrieves Result Set records from the server, it may request a particular packaging of the elements by specifying a preferred Record Syntax in the Present Request PDU. The Record Syntax is intended to ensure preservation of the information content and semantics, as information is transferred from the server to the client.

In the early implementation stages, there was only one type of Record Syntax defined by the Z39.50 standard: USMARC and the various other national MARC formats. MARC is an old but revered format that was designed to carry bibliographic data in a rather limited tagged field format. The MARC Record Syntax has served the library and bibliographic community quite well, but is completely inadequate for carrying scitech information. However, at the time, the scitech community had not defined any appropriate syntaxes or formats that could be used as a general interoperable Record Syntax within Z39.50.

Retrievable scitech information may be available in a wide variety of forms, including character strings, binary strings, integers, real numbers, tables, images, complex data structures, and others. A given scitech element may be available in various forms and may be expressed in different unit systems. The form in which a client retrieves a given scitech element may depend on what it plans to do with it. For example, a client retrieving scitech information may manipulate it, display it, save it, print it, or feed it into local software to process it. The form in which an element is retrieved for display may be very different than the form useful for local processing.

There may be complex hierarchical relationships between individual elements within a retrieved record. A given logical Result Set "record" may actually consist of a complex hierarchy of records, each containing a complex hierarchy of elements. The amount of data retrieved for a single scitech Result Set record (or even a given element within a record) could be very large, and may need to be retrieved in pieces. It is useful to be able to retrieve metadata (data about data) in addition to the data elements themselves. Finally, numeric information such as integers need to be encoded in a standard manner when transferred between computers with different hardware architectures, to ensure data portability.

Few of these needs were addressed by MARC. Therefore, a new, flexible Record Syntax was needed to support retrieval of scitech information. In addition, a simple Record Syntax was needed to retrieve pre-formatted information which a simple client could simply display for the user, without any particular understanding of its content or semantics.

6. Other Retrieval limitations

In addition to the Element Specification and Record Syntax limitations, there were other limitations to the Present service for retrieving scitech information using Z39.50 V2.

a. There was no way to request retrieval metadata.
b. There was no way to request different elements and Record Syntaxes for different databases, when the Result Set was created by a search against multiple databases.
c. There was no way to request a particular form of a given element.
d. There was no way to request particular sub-trees of a hierarchical data structure.
e. There was no flexible way to return large Result Set records. Since there was no concept of record segmentation, a record larger than the message buffer provided by the client simply could not be retrieved.

Approach to the issues

The CAS approaches to these Z39.50 implementation issues fell into three categories:

- o enhancement of the standard through active participation in the ZIG;
- o development of interoperable formats and conventions to support exchange of scitech information;
- o definition of external data structures for use within the standard.

The primary approach was to actively participate in the evolution of the Z39.50 standard, by working with other implementors within the ZIG to propose and evaluate new functionality and extensibility enhancements that addressed the basic limitations of the protocol. Examples of this work include the addition of the OtherInformation structure, the Scan service, and the Extended Services in V3 of Z39.50.

The second approach was to work with other organizations to develop open, interoperable formats and conventions needed to support the exchange of scitech information.

These formats and conventions were designed to be plugged into the Z39.50 protocol in a standard and interoperable manner. Examples include STAS, the Scientific and Technical Attribute and Element Set [6], and CXF, the Chemical eXchange Format [7].

The third approach was to work with other implementors and partners to develop and propose externally-defined data structures to be used within the protocol to carry additional information needed for commercial scitech information systems. Some of these structures were proposed for public adoption, while others were intended for more limited contexts of interoperability. Examples include the GRS-1 (the Generic Record Syntax), AdditionalSearchInformation, and the SetUserParm Extended Service.

CAS began its implementation of Z39.50 in 1991, and continues to evolve and expand that implementation over time. CAS initially based its implementation on V1, then V2, and finally V3 of the standard. Each of these versions presented its own set of issues, which had to be addressed within its own context. In several cases, the next version of the standard included opportunities for better solutions to a given problem than possible in the previous version. Since most current implementations of Z39.50 are based on V2, interoperability goals dictate that, wherever possible, solutions to issues be applicable to both V2 and V3. Therefore, some approaches to the issues represent compromises between the functionality available in V3 and the need to interoperate with V2 implementations.

Solutions to the issues

1. Protocol scope and extensibility

The first extensibility challenge was that Z39.50 V2 provided no way to carry supplementary data in the Protocol Data Units (PDUs) defined for a given service within the protocol. V3 addresses this problem by adding an optional OtherInformation structure to each Z39.50 PDU. This allows a given Z39.50 service to carry along externally-defined information that augments the core information fields explicitly defined within the PDU. The OtherInformation structure allows the externally-defined information structure to be uniquely identified by an Object IDentifier (OID) to improve interoperability and avoid any ambiguity.

A simple example of the use of this feature within the CAS Z39.50 implementation is to allow a language code to be carried within the OtherInformation field of the Scan Request PDU. This allows the client to specify the preferred national language for the output of a Scan operation against a multi-lingual thesaurus.

The second extensibility challenge was that Z39.50-1992 lacked several necessary services, and provided no interoperable way to add supplementary services to the protocol. Z39.50-1995 addresses this in two ways: (1) addition of new services such as Scan and Sort which are closely related to search and retrieval; and (2) addition of the Extended Services facility.

CAS has leveraged both of these new protocol features to provide better access to its existing databases and services. As an example of the first case, CAS has implemented the Scan facility to allow term expansion within both database indices and thesauri.

In the second case, CAS has leveraged the Extended Services facility extensively to support both "standard" Extended Services (those specified in Z39.50-1995), and "local" Extended Services. An example of a "local" CAS Extended Service is the Analyze Extended Service. Analyze allows the client to perform an analysis of the content of one or more records within a Result Set, based on specified data elements and their values. This information can then be used to help the user select the information of greatest value.

2. Attribute Set limitations

The issue here was how to identify and characterize scitech query terms within a Z39.50 Search Request. The solution to this issue was to define a new Scientific and Technical Attribute and Element Set (STAS).

STAS defines both an Attribute Set and an Element Set. The STAS Attribute Set supports the use of scientific, technical, and related search terms within a standard Type 1 or Type 101 Query carried within a Z39.50 Search Request. The STAS Element Set supports identification and selection of data elements retrievable from scientific, technical, and related databases using a Z39.50 Present Request. The STAS Attributes and Elements are also useful within other Z39.50 services such as Scan and Sort.

CAS originally developed STAS as part of its research project on the use of Z39.50. In September 1994, co-sponsors CNIDR [8], Dialog [9], FIZ Karlsruhe [10], and CAS announced the public availability of STAS as an open, public definition. As such, any interested party may freely use and contribute to STAS. STAS maintenance and registration functions are provided by CNIDR. The Z39.50 Maintenance Agency has assigned the STAS Attribute Set a standard public Object Id, which is listed in Appendix ATR of the Z39.50-1995 standard along with bib-1 and others.

The Search service supported by Z39.50 V2 has a limitation that influenced the approach taken in defining STAS. A V2 Search Request allows Attributes from only a single Attribute Set to be used in a given RPN query. And yet it is a practical requirement to support searches containing both bibliographic and scitech search terms within the same query. This limitation required definition of a single Attribute Set that contained both bibliographic and scitech Attributes.

Therefore, the STAS Attribute Set is defined as a superset of the bib-1 Attribute Set, and implicitly imports all Attributes specified by the bib-1 Attribute Set. Additional

STAS Attribute types and values are assigned identifiers that are outside of the range assigned to bib-1 Attribute types and values. As new Attribute types and values are added to the bib-1 Attribute Set, they automatically become part of the STAS Attribute Set.

The STAS Attribute and Element Set definitions are evolving from the ongoing effort of defining Attribute and Element mappings for existing scitech databases. Wherever a valid mapping can be defined between existing bib-1 Attributes and a database's search fields, bib-1 Attributes will be used. For each database search field that has no equivalent bib-1 Attribute, a new STAS Attribute will be defined.

3. Search service limitations

a. The V2 Search Request allowed specification of only one Attribute Set within an RPN query.

This issue was addressed in two ways: (1) the definition of STAS as a superset of the bib-1 Attribute Set addressed the problem within the V2 context; and (2) expanding the V3 RPN query to allow specification of multiple Attribute Sets provided a long-term robust solution.

Z39.50 V2 supports a single Attribute Set ID field in the RPN query within a Z39.50 Search Request. Search Requests using STAS will specify the STAS Attribute Set Object IDentifier (OID) in this field. This allows use of both bib-1 and other STAS Attributes within the query.

Z39.50 V3 allows optional specification of the Attribute Set Id for each search term, and even for each Attribute. This feature of V3 allows STAS to be used in combination with the bib-1 and/or any other Attribute Set(s). Bib-1 Attributes may be explicitly identified as such, and other STAS Attributes may be identified by the STAS Attribute Set Id.

b. There was no mechanism for specifying the datatype of a given term within a Type 1 query.

The short term (V2) approach to this issue was to define STAS Attributes explicitly enough to provide strong hints about the datatype of the term. In retrospect, the disadvantage of this approach was the proliferation of similar Use Attributes for a term with the same semantics but different datatypes, forms, or formats.

The long term solution was to expand the RPN search term definition within V3 to support explicit data typing of the term contents. With this capability, the current STAS philosophy is to move away from datatype-specific Use Attributes.

c. V2 provided no way to specify the "units" in which a search term was expressed.

The short-term (V2) approach to this issue was to carry the units indicators along with the term value within the RPN query term. Although this works adequately within a limited context, the lack of publicly defined conventions for expressing units in this manner limits the interoperability of this approach.

The long-term solution to this problem is the explicit support for units within V3. CAS defined and proposed an IntUnit structure for specifying values with units. IntUnit allows specification of an Integer value, qualified by a scale factor, Units System, Unit Type, and Unit. After some discussion and modifications, this IntUnit structure has been incorporated into several parts of the V3 standard. In particular, the IntUnit is one of the supported datatypes for RPN search terms, thus allowing a search term to be expressed in explicit units.

d. There was no way to send or return additional search information.

Since V2 Search PDUs lacked the capability for carrying additional search information, indicators about the "Type" or "Scope" of a search could not be expressed in the Search Request.

The V2 approach to this issue was to define new STAS Attribute Types to express Search Type and Search Scope. For example, currently defined STAS Search Types include Substructure, Closed Substructure, Family, and Exact Searches. And currently defined Search Scope values include Full File, Sample File, Range, and Subset Searches.

Although these indicators are generally expressed globally for an entire query, this approach allowed the flexibility of specifying Search Type and Scope at the subtree or even the term level of a given query.

The V3 approach to this issue was the addition of a new optional AdditionalSearchInfo field in the Search Request PDU. Although this V3 feature has not been leveraged yet, it will eventually provide a more robust solution.

A second issue was that additional information about the search results, such as how many "hits" were found for each term in the query, could not be returned in the Search Response.

The V3 solution to this problem is a new optional AdditionalSearchInfo field in the Search Response PDU. CAS leveraged this V3 feature by defining an external structure for carrying various types of information about the search results, including how the server interpreted the query and how many "hits" were found for each term in the query. This structure was proposed to the ZIG, and following discussion and modification, was added to the V3 standard.

4. Element Specification

The CAS solution to the Element Specification issue within the V2 context was to define a simple syntax for expressing any combination of elements to be retrieved. This syntax is called STETSEN (the Scientific and Technical Element Set Names) [11]. STETSEN draws on STAS, by using the STAS Element Numbers to identify individual elements to be retrieved. Just as a unique STAS Use Attribute Value can be defined for each database search field, a unique STAS Element Number can be assigned to each database retrieval field (retrievable element).

A given combination of retrieval elements is expressed using the STETSEN syntax by a character string containing a list of the corresponding STAS Element Numbers, separated by commas or spaces. These STAS Element Numbers may be combined with other ElementSetNames such as Full (F), Brief (B), or target-defined names. Since a STETSEN ElementSetName is simply a character string, it can be legally carried in the ElementSetName field within either the V2 or V3 Present Request PDU.

The general Element Specification issue was addressed in a more robust manner in V3. CAS worked with other implementors within the ZIG to define several new mechanisms within the V3 Present service to support more flexible and powerful mechanisms for element retrieval. These include the CompSpec, eSpec-1, Variant, and Schema features. Although CAS has not yet fully leveraged these new features, they provide very powerful retrieval capabilities, and will be implemented in the future.

5. Record Syntax limitations

Since externally-defined Record Syntaxes can be flexibly "plugged into" both V2 and V3 of Z39.50, the basic issue was addressed by defining new Record Syntaxes that met the requirements for scitech information. Related issues were addressed within the context of the V3 Present service via mechanisms such as the Schema concept and the CompSpec structure.

One fundamental issue that was clarified during the development of these V3 mechanisms was the fact that a Record Syntax consists of both an Abstract Syntax and a Transfer Syntax. The Abstract Syntax (often expressed in a formal notation, such as ASN.1) specifies the content, semantics, and structure of the record [12]. The Transfer Syntax (usually defined by a set of encoding rules such as BER) ensures that the information in the record is successfully conveyed over a network in a portable and unambiguous manner [13]. When USMARC and the other MARC formats were the only supported Z39.50 Record Syntaxes, these distinctions were less critical, and not well articulated. But development of new Record Syntaxes to support non-textual information forced developers to better articulate this concept within the Z39.50 standard.

CAS initially worked with a small group of Z39.50 implementors, led by John Kunze (University of California Berkeley) to develop a new "info-1" Record Syntax. The goal of info-1 was to flexibly carry tagged elements of multiple datatypes and formats as well as metadata about those elements. CAS implemented at least three generations of this concept, starting with info-1, then GRS-0, and finally GRS-1. After several years of discussions, implementations, and refinement, this work has evolved into the Generic Record Syntax-1 (GRS-1), as specified in V3. GRS-1 is a very powerful Record Syntax that supports flexible delivery of literally any type of information of essentially arbitrary complexity. GRS-1 supports tagged elements, metadata, hierarchical data structures, unit specification, and information about the particular form (variant) of individual elements. Use of the BER standard to encode GRS-1 records ensures strong data portability across networks and computing platforms.

CAS has upgraded its Z39.50 implementations to use GRS-1 extensively in delivering scitech information via the Present service. When STAS is used in combination with GRS-1, the Tag Numbers used to identify elements carried in a GRS-1 record are the STAS Element Numbers. The STAS Element Numbers therefore constitute a standard Z39.50 TagSet that has been registered in the V3 standard.

Use of STAS Element Numbers within GRS-1 leverages a STAS convention, wherein the same number space is used to assign STAS Use Attribute Values, STAS Element Numbers, and STAS Tag Numbers. Within many databases, there are often retrieval fields (elements) that correspond to search fields (Attributes). It is often useful for a client to be able to relate a retrieval field with a corresponding search field. A database field that can be both searched and retrieved is assigned the same value for its STAS Use Attribute, its STAS Element Number, and its STAS GRS-1 Tag number.

In addition to GRS-1, there was also a need for a simple Record Syntax for delivering pre-formatted textual information for display. CAS worked with a small group of other implementors to propose and refine SUTRS (the Simple Unstructured Text Record Syntax), which is now a registered Record Syntax defined within V3. SUTRS is especially useful as a "lowest common denominator" Record Syntax between clients and servers that have minimal knowledge of each others' data or conventions.

Finally, CAS needed a standard interoperable format for exchanging detailed chemical information, and worked with other organizations to develop the Chemical eXchange Format (CXF). CXF is defined in ASN.1, encoded using BER, and may be used either as a Record Syntax or as an Element Syntax for a tagged element within GRS-1. In the interest of maximum scitech interoperability, CAS has submitted CXF to the industry as an open definition, available for use by any interested organization. CAS uses CXF extensively in the search and retrieval of chemical information via Z39.50.

6. Other Retrieval limitations

There were other limitations to the Present service for retrieving scitech information using Z39.50 V2.

a. There was no way to request retrieval metadata.

In conjunction with the development of the GRS-1 Record Syntax, CAS also worked with other implementors to develop a complementary element specification mechanism for use within V3. This work resulted in the V3 eSpec-

1 structure, which may be used within the CompSpec structure of the V3 Present Request.

eSpec-1 allows the client to request retrieval of element metadata, with or without the corresponding data. GRS-1 provides the complementary ability to deliver the metadata, with or without the corresponding data. And the V3 TagSet-M defines a set of Tags which can be used to identify metadata carried within GRS-1 in an interoperable manner. The combination of these new mechanisms allows a client to dynamically discover characteristics such as the size, cost, and copyright restrictions of information, prior to retrieving it.

b. There was no way to request different elements and Record Syntaxes for different databases, when the Result Set was created by a search against multiple databases.

This need has been addressed by development of the new CompSpec structure of the V3 Present Request. This new structure allows the client to specify a particular combination of Record Syntax and element specification for each database from which the Result Set was created.

c. There was no way to request a particular form of a given element.

The V3 eSpec-1 structure allows the client to request a particular form of a given element for retrieval via a concept called "Variants." V3 defines Variant-1, a standard Variant Set which identifies a number of classes and types of variants such as national language, body type, size, etc. GRS-1 provides the complementary ability to deliver the requested variant of the element as well as the corresponding "applied variant" identifiers. And the V3 metadata capabilities already mentioned allow dynamic discovery of the available variants of a given element before retrieving it. The combination of these new V3 mechanisms allows a client to dynamically negotiate and retrieve the best form of information for its needs.

d. There was no way to request particular sub-trees of a hierarchical data structure.

The V3 eSpec-1 structure allows the client to request retrieval of specific elements or subtrees within a hierarchical data structure, using the concept of TagPaths. A TagPath specifies the path through a data structure, where each node in the path is identified by a tag. ESpec-1 allows the client to specify a given element or subtree for retrieval by specifying its TagPath. GRS-1 provides the complementary ability to deliver the requested element or subtree as well as the corresponding Tags representing the Path. And the V3 Schema concept provides the client and server with a common understanding of the hierarchical database structure by documenting it using TagPaths.

e. There was no flexible way to return large Result Set records.

The new Segmentation features of the V3 Present Facility address this problem. Records larger than the message buffer provided by the client can now be retrieved, by breaking them up into pieces, which are delivered in Segment Request PDUs. CAS is one of the first implementors of Segmentation, including segmentation of GRS-1 records, using a recently defined Fragmentation Syntax.

A second V3 capability was added to support the retrieval of "pieces" of individual elements, using the eSpec-1, GRS-1, and Variant mechanisms. A client may retrieve a single large element, such as a large image, in pieces by specifying its retrieval using eSpec-1. The particular piece and its size can be specified using Variants. And GRS-1 identifies the specific element, its piece and its size upon delivery.

Lessons learned

In the course of implementing Z39.50, several lessons were learned.

1. Standards can be enhanced and expanded, but it is not easy.

A given standard rarely meets all the needs of a given implementor. In some cases, the best way to address the shortcomings of a key standard is to actively participate in its development. However, influencing the scope and functionality of an evolving standard such as Z39.50 is neither easy nor inexpensive. It takes time, effort, resources, patience, persistence, and commitment. For some implementors with advanced requirements, it may actually be simpler to design and implement a proprietary protocol which does exactly what is needed. However, we have previously learned that a world of proprietary protocols does not promote the flow of information. The interoperability provided by Z39.50 promotes the unencumbered flow of information, and opens up many technical and business opportunities which would not be possible with the use of proprietary protocols. In the case of Z39.50, our investment in active standards participation was successful and will be leveraged.

2. Implementation experience leads to better standards.

Having participated in several other standards development efforts, it is the author's opinion that the process for developing the Z39.50 standard was an unusually successful one. A lot of the credit for this goes to the Z39.50 Maintenance Agency and the ZIG. The administrative and political hurdles were kept at a manageable level, allowing technical needs to be addressed in a timely manner. An active role by Z39.50 implementors in defining and expanding the standard within the ZIG added a practical influence to the process. In several cases, implementors such as CAS designed and implemented new features and services before proposing them to the ZIG for inclusion in the Z39.50 standard. In other cases, early implementation of features proposed within a working draft of the standard helped refine and improve their definition, leading to a better specification. This active participation by implementors coupled with early implementation experience resulted in a better Z39.50 standard.

3. The Attribute, Element, and metadata problems are difficult, and cannot be solved by a protocol alone.

Some of the major remaining interoperability challenges for implementors of Z39.50 revolve around the need for unique and unambiguous identification of information, both in search queries and retrieved answers. The ambiguities and inconsistencies in the use of the bib-1 Attribute Set and the MARC record syntax are the most visible aspects of the problem. However, an underlying source of the problem lies in the original indexing policies used to identify information. Different organizations index and identify information in different ways. Mappings of the bib-1 Attributes into search elements and mappings of retrieval elements into MARC records are not consistent across databases or organizations. This results in reduced interoperability of information. Definition of a protocol such as Z39.50 cannot alone solve this problem. Definition of metadata standards, improved indexing standards, and unambiguous Attribute and Element Sets are also needed.

CAS has attempted to avoid many of the MARC and bib-1 Attribute Set interoperability problems by defining and using the STAS Attribute, Element, and Tag Sets in conjunction with the SUTRS and GRS-1 Record Syntaxes. However, for protocol consistency, STAS also inherits some of the characteristics of the bib-1 Attribute Set. It is therefore expected that STAS will continue to evolve, as additional experience is gained with its use. Other Attribute, Element, and Tag Sets will probably be defined to address other information disciplines. A new bibliographic Attribute Set may eventually evolve to replace bib-1. In summary, this particular area of information interoperability is a challenging one, and will require additional work in the future.

Futures

The CAS implementation of Z39.50 is an ongoing project. The specific issues and solutions identified in this paper reflect the functionality implemented to date. However, Z39.50-1995 defines a very rich set of services and features, several of which CAS has not yet implemented. In the future, CAS will continue to implement additional Z39.50 features and services, as required by its users and projects. Some representative examples are noted here.

The STETSEN syntax for ElementSetNames has provided an adequate means for specifying element selection within both the V2 and V3 contexts. However, the future direction will be to implement support for the eSpec-1 definition to enhance interoperability and support more granular retrieval specifications. For example, eSpec-1 will allow the client to:

- retrieve and leverage metadata
- discover variants of retrieval elements
- retrieve the optimal form of information
- request the server to package information in an optimal manner

The Explain service supports the discovery of information useful to both the users and clients of Z39.50 services. Explain holds great potential for expanding the intelligence and reach of Z39.50 clients, without building specific database knowledge into the client software. Use of Explain will be especially useful when the client and server have been independently developed or are operated by different organizations. However, the Explain specification has just recently been finalized, and is probably one of the least mature portions of Z39.50-1995. As Explain matures and gains wider acceptance, CAS will add support for Explain in the future.

There are several standard Extended Services defined in Z39.50-1995, which CAS will implement in the future to provide access to advanced features of the CAS search and retrieval systems. For example, the ItemOrder Extended Service defines a mechanism for initiating document orders from a Z39.50 client. CAS will initially use Item Order to convey user requests to the CAS Document Delivery System.

Due to project requirements and timeframes, CAS implemented some V3 Extended Services and data structures prior to their finalization in Z39.50-1995. Since Z39.50 explicitly supports the identification and use of such "local" data structures and Extended Services, they are currently being used between CAS clients and servers, with no compromise in protocol compliance. However, over time, these CAS-defined conventions and Extended Services will be migrated to adhere to the "standard" structures and Extended Services defined by Z39.50-1995. This will improve interoperability with Z39.50 clients and servers implemented and operated by other organizations.

Conclusion

Z39.50 is a very positive example of the value of interoperable standards. It supports a rich set of interoperable services between separately developed clients and servers. Z39.50 has rapidly evolved to address the needs of implementors operating within the context of various information disciplines and commercial search systems. The model of the ZIG working in concert with NISO, ANSI and ISO has proven to be very successful in meeting the needs of the information retrieval community. In recognition of the demonstrated value of Z39.50, it is being used as one of the key access protocols for current and future CAS software projects and products.

References

[1] ANSI/NISO Z39.50-1988, “ANSI Z39.50: Information Retrieval Service and Protocol”, 1988.

[2] ISO 10163—Information and Documentation - Search and Retrieve Application Protocol Specification for Open Systems Interconnection, 1991.

[3] ANSI/NISO Z39.50-1992, “ANSI Z39.50: Information Retrieval Service and Protocol”, 1992. <URL:ftp://ftp.cni.org/pub/NISO/docs/Z39.50-1992/www/Z39.50.toc.html>

[4] ANSI/NISO Z39.50-1995, “ANSI Z39.50: Information Retrieval Service and Protocol”, 1995. <URL:http://lcweb.loc.gov/z3950/agency/1995doc.html>

[5] Chemical Abstracts Service (CAS), a division of the American Chemical Society, is the world’s leading provider of chemical information. <URL:http://info.cas.org/welcome.html>

[6] STAS, the Scientific and Technical Attribute and Element Set. <URL:http://stas.cnidr.org/STAS.html>

[7] CXF, the Chemical eXchange Format, Version 1.0, September 1994, CAS. <URL:ftp://info.cas.org/pub/cxf/specification/cxf.1.0/>

[8] CNIDR is The Clearinghouse for Networked Information Discovery and Retrieval, located at MCNC in Research Triangle Park, N.C. <URL:http://cnidr.org/welcome.html>

[9] Knight-Ridder Information Inc., formerly Dialog Information Services, Inc., a Knight-Ridder company. <URL:http://www.dialog.com/>

[10] FIZ-Karlsruhe is the FachInformationsZentrum Karlsruhe, operator of STN International in Europe. <URL:http://www.fiz-karlsruhe.de/>

[11] STETSEN—The Scientific and Technical Element Set Names, an internal CAS technical paper.

[12] ISO 8824—Information Processing Systems - Open Systems Interconnection - Specification of Abstract Syntax Notation One (ASN.1), 1990.

[13] ISO 8825—Information Processing Systems - Open Systems Interconnection - Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1), 1990.

Structural Components of the Isite Information System

Kevin Gamiel (Kevin.Gamiel@cnidr.org)
Nassib Nassar (nrn@cnidr.org)
Clearinghouse for Networked Information Discovery and Retrieval (CNIDR)

Abstract

This paper discusses various technical topics related to the structural model of Isite, an open information system supporting multiple external protocol-based access mecha nisms, a database-independent search and retrieval API, and an extensible field-based text search engine. Isite was developed by the Clearinghouse for Networked Information Discovery and Retrieval (CNIDR), which is funded by the National Science Foundation.

Two Models in Data Publishing

The broad range of existing distributed data access models can be classified into two main groups: those oriented toward browsing, and those supporting searching capabilities.

Examples of browsing systems are Gopher (University of Minnesota) and World-Wide Web (CERN); publishers with the proper tools can make their data accessible to users through these systems. Browsing systems are based on the concept of navigating through a virtual information space. Such a system is well suited to data retrieval in cases where there are clear relationships among the data. In such systems, it is very important to have some method of organization, since browsing depends on the user's ability to make intelligent navigational decisions. Browsing systems have proved very useful for locating databases on the vastly distributed global Internet, because the "hyper-text" model utilized by Gopher and the World-Wide Web has encouraged a rudimentary kind of organization among related documents.

An information system based on the searching model is desirable when the data being published are difficult to organize. In such cases an automated search on relevant words of text may be far more economical than a browsing model, or at least may assist the browsing system at certain stages.

In practice, a combination of these access models is frequently used. For example, many search-based systems are arranged at the top level with a user interface provided by a browsing system. It is also desirable that the various searching systems conform to Internet protocol standards so that they may easily be integrated with other searching and browsing systems.

Isite

Isite uses the combined model, but also provides multiple access mechanisms via standard protocols. Additionally, using the Search API included in Isite, multiple databases of different formats may be served. Newly installed data bases have the immediate benefit of being accessible through all data retrieval communications protocols supported by Isite.

At present Isite supports four access methods based on standard Internet protocols: Z39.50, World-Wide Web (HTTP), Electronic Mail, and Gopher. The Z39.50 protocol, implemented in Isite in conformance to the ANSI/-NISO Z39.50 version 2 standard, is the primary access point of the system. Z39.50 supplies a rich set of "stateful" services for deep search and retrieval on distributed database servers. It is the internationally accepted open standard for search and retrieval over networks. The other protocols are supported via protocol "gateway" soft ware. A gateway is software that translates between two different protocols. Thus Isite contains an HTTP to Z39.-50 gateway, which converts search queries received by an HTTP server to queries that can be understood by the Z39.50 server.

The Search API (SAPI) provides a layer of abstraction between the Z39.50 server in Isite and the database system that contains the data to be served. The API is a specification for a certain set of functions that must be supported per database system. Thus the Z39.50 server can rely on a consistent communications layer for accessing data stored in various databases. A supported database may be a complex text search or relational system, or it may be a simple utility such as "grep." The essential principle is that the SAPI must normalize the behavior of the database software in order to be accessible by the Z39.50 server. With this layer operational, all access methods available to Isite are extended to databases supported by SAPI.

Implementation

Isite consists of several distinct software packages, including (1) libcnidr, a source code library of commonly used functions, (2) ZDist, an ANSI/NISO Z39.50 version 2 programmer's library, UNIX server, and UNIX client and gateway, (3) SAPI, the Search API, and (4) Isearch, a field-based text search system including a C++ class interface and Iindex/Isearch utilities.

ZDist

The ZDist package contains much of the core of communications access for Isite. Access to SAPI-supported data bases is entirely directed through Z39.50. The current version of ZDist, included in Isite, supports the Initialize, Search, Present, and Close facilities of Z39.50.

The Z39.50 library is written in C and is based on the freely available BER utilities developed by the OCLC On line Computer Library Center, Inc. The server and client/gateway use this library to encode and decode Z39.50 Protocol Data Units (PDU). The application-level network communications source code is distinct from the Z39.50 PDU encoding/decoding layer, and is physically located in the libcnidr library.

The Z39.50 server application is easily configurable by modifying a text file, and the configuration options include executing the server either as a forking daemon or from inetd, specifying a maximum number of simultaneous connections, specifying which databases to serve, and various Z39.50 settings down to the PDU level. The server's overall operation is straightforward: it (1) listens for client connections on a well-known port (usually 210), (2) accepts a connection, initializes with the client, (3) accepts search queries, and (4) interacts with the SAPI to process the query and returns results to the client, after which (5) the client may request that the server "present" the contents of any of the results, and (6) the server contacts the SAPI to retrieve those contents and return them to the client.

The Z39.50 client is designed especially to be the foundation for gateway applications. Thus it is not interactive, nor does it provide a user-friendly interface. In order to build a gateway from a "stateless" protocol such as HTTP to a "stateful" protocol such as Z39.50, state requirements must be hidden from the calling (in this case, HTTP) side. The Z39.50 client is implemented as a single-pass client that initializes, searches, and presents in uninterrupted sequence. The client also conforms to the Common Gateway Interface (CGI), a well-known standard for gateway communication with HTTP servers. The CGI process provides a layer between HTTP and the Z39.50 gateway. Like the Z39.50 server, the client is configurable, allowing the user to specify all Z39.50 PDU-level information, for maximum flexibility.

Electronic Mail Gateway

Implementation of an Electronic Mail gateway is simplified by taking advantage of a feature of the Z39.50 client. Since all PDU-level information may be specified in a text configuration file, a rudimentary gateway can use the exact text of an Electronic Mail message as the client configuration file. The results of the search may then be mailed in reply to the requesting user.

Isearch and the Document Type Model

Isearch is designed as a set of modular components, at the center of which are two groups of C++ classes: the Isearch engine, which encapsulates the functionality of field-based indexing, searching, and presenting, and the Document Type class hierarchy, which defines the behavior of the Isearch engine for certain types of documents. It is the latter that we wish to discuss here.

Document Type classes bind specific functionality in the Isearch engine to the documents being processed. C++ classes defining the field structure and presentation characteristics of certain types of documents are therefore grouped by the common features of those documents. The DOCTYPE base class defines default behavior, and it is invoked during the processing of documents for which no Document Type class has been specified. All Document Types must be derived from DOCTYPE; therefore it is important to understand the implementation of the base class and its interactions with the Isearch engine, which, although straightforward, require some explanation.

There are four essential methods of DOCTYPE: AddFieldDefs(), ParseFields(), ParseRecords(), and Present().

The first three of these are invoked during indexing, and the fourth during searching. AddFieldDefs() provides information to the Isearch engine about the fields it expects to discover during the parsing phase. ParseFields() is called during indexing of each document record, at which time it parses the document and inserts field structure information into the RECORD object. ParseRecords() defines the record structure for files that contain multiple document records; this may be beneficial in cases where record structure is dependent upon other aspects of document structure that must be determined at run-time. Finally, the Present() method defines exactly how documents are displayed in response to requests for various element sets, which allows presentation to be abstracted from the retrieval of field contents.

The Isearch engine calls each of these methods at appropriate times during the indexing and searching processes. In DOCTYPE they are defined for minimal functionality, but they can be overridden within descendent classes. For example, DOCTYPE::AddFieldDefs() and DOCTYPE::ParseFields() contain no source code, and consequently the default behavior of the indexing routines is to treat documents as lacking field structure. No knowledge of field structure is indigenous to the Isearch engine. However, source code to handle field-based searching is present in the engine and has only to be enabled by defining field structure within the Document Type.

The method of building field definitions and structure tables is slightly involved because of the various levels of nesting and data hiding.

An instance of the DFDT (Data Field Definitions Table) class is stored within the Isearch engine classes, and DOC TYPE methods can add new definitions to the table. The table is a list of DFD (Data Field Definition) objects. Each DFD consists of a field name and an attribute list (ATTRLIST), which in turn contains a list of attribute (ATTR) objects that can be used to preserve additional information about each field definition. Field definition information can be added to the engine's internal DFDT object at any point within the Document Type methods, except that field structure information should not be generated for a record if that field structure refers to fields that have not yet been defined. There are two good reasons to generate DFD objects within DOCTYPE::AddFieldDefs(); first, it is a convenient place to collect all field definition information, and secondly, it allows for optimization within the Isearch engine. An example in which it is necessary to generate field definitions in DOCTYPE::ParseFields() is when field information is not known ahead of time, such as in the case where the field name must be derived from contents of the document being processed.

An example that illustrates this last point is the SGML-TAG Document Type, which scans the document text for SGML-like tags and treats the tag name as the field name and the delimited text as the field contents. In order to support maximum variety of field structure and yet avoid the unnecessary overhead of definitions for fields not present in any of the documents, generating field definitions in AddFieldDefs() would require knowledge of field information *a posteriori*. The problem is solved by adding each field definition as it is discovered during parsing of the document records.

Building field structure information is similar to building a DFDT. Field structure is encapsulated in a DFT (Data Field Table) object, which is a member of the RECORD class; thus there is one DFT per document record being indexed. The DFT is a list of DF (Data Field) objects, each of which consists of a field name and an FCT (Field Coordinate Table). The FCT is a list of coordinate pairs (FC) that delimit instances of the field within the document record. Inserting multiple FC objects into the FCT enables support in the Isearch engine for repeating fields, which may be defined as a sequence of multiple occurrences of the same field within a single document record.

At search time these field definitions and structures may be retrieved and used for presentation of document text. The Isearch engine provides a method for locating the contents of a certain field within a certain document record. However, the Document Type architecture creates one additional level of abstraction. A method in the Isearch engine called Present() is accessible to the main application for general purpose high-level presentation of document text, and it yields control to the Present() method of the Document Type associated with the document record that is being accessed. The default behavior, defined in DOCTYPE::Present(), is more or less to treat the element set as a field name, and to retrieve the contents of that field from the Isearch engine. In addition, it interprets the element sets "B" and "F" as requesting "brief" and "full" records, respectively. The purpose of this architecture is to allow DOCTYPE::Present() to be redefined in descendent classes, in order that various forms of presentation may be implemented. For example, element sets may be synthesized from more than one field, retrieved text data may be reformatted for suitable output, etc. DOCTYPE::Present() essentially "intercepts" normal processing of field-based presentation, thus allowing it to be extended in relation to the document being accessed.

It is hoped that the advantages of using the Document Type architecture outweigh the small amount of additional development required to integrate it with existing systems. Since the model is a map of document type behavior rather than a physical document representation, the scope of the representation is unspecified. A Document Type may be so general as to handle all SGML documents, or so special as to be tailored to a particular proprietary document format. The model may even be wrapped around other field parsers, since a Document Type may be defined simply to read field coordinates out of an ancillary file.

The Document Type model benefits from some of the features of object-oriented programming, including extensibility, modularity, code maintainability, data hiding, and the ability to build upon previous work via class inheritance. Documents defined by a Document Type know how to "present" themselves, which minimizes risky and tedious internal modifications to the Isearch engine. In addition, DOCTYPE can be expanded to provide increased access to features of the engine, while descendant classes may add document type-specific functionality.

Since the Isearch engine supports document records of different Document Types within the same database, it is possible to abstract multitype text data by normalizing functionality within the Document Type class methods. For example, a database of international patents may consist of a variety of data formats. Rather than requiring massive data conversion, the Document Type model allows run-time format normalization by supporting customized field parsing and a formatted presentation layer. Thus multitype data can be stored in their native format within the same database, and Search and Present operations can be abstracted from structural differences.

Conclusion

Isite implements a variety of modular architectural structures based, where possible, upon open standards. Used together, they provide a powerful, extensible information system that is capable of remaining compatible with continuously changing paradigms.

Z39.50 – implications and implementation at the AT&T library network.

Robert K. Waldstein

AT&T Bell Laboratories
Murray Hill, NJ 07974

Abstract

The AT&T library organization has developed an interest in Z39.50 for a number of diverse reasons. It is hoped that eventually Z39.50 will help with or solve several classes of problems, ranging from behind the scenes issues resulting from distributed computing architectures to diversity of user interfaces. In addition to helping with known problems, we hope that Z39.50 will give us a flexibility required for a constantly evolving library organization in an international corporate environment.

For Z39.50 to meet our needs the main requirement is that the protocol itself incorporate all the functionality of our existing information retrieval environment. The 1992 (version 2) version of the standard was a major start, but the newer version (1995 version 3) comes much closer to incorporating existing functionality.

The next major requirement is proven interoperability and transparency of database provider to our users. Issues of indexing style, default operations and ways to override defaults, database coverage and loading characteristics become even more apparent in Z39.50 than in the traditional online world.

Our end users, like users everywhere, are expecting interfaces integrated into their regular computing environment. A solution to this problem is a well accepted search and retrieval protocol. Z39.50 is well positioned to become this protocol, and in this belief we have focused our attention on developing a high-quality server for our internal resources.

1. Introduction

The AT&T library organization has changed considerably, and continuously, along with the rest of AT&T in the years since divestiture in 1984. We provide world-class library services to the employees of AT&T worldwide including: technical information needed for research and development, business and marketing needs, as well as manufacturing information and internal newsletters. Some significant user requirements that we must meet as a corporate information service provider are:

- Cost reductions. This is both in terms of our budget and what people are willing to pay for information services and resources.
- Information provided in the user's environment. e.g. integrated workstation, fax.
- Information on demand. Users often want the desired information at the time of request. In addition, the user wants to control the depth and format of the desired information.
- User access to information. User demand for direct access to information is growing, with or without an intermediary's involvement.

The computing environment in which we provide our services has been evolving rapidly. Distributed computing has become a requirement for a flexible environment, both in terms of costs and functionality. In addition, workstations – powerful computer resources at the user's desk – have nearly replaced terminals as access tools. Workstations have not only opened new opportunities, they have changed users' expectations about the "look and feel" of the information presented to them.

The information market has also been rapidly evolving during this time period. Interest in new forms of access licenses, especially site licenses, has been growing among database providers. Databases have begun to expand beyond flat text, into "multimedia" – particularly scanned images. Finally, special interest database providers are appearing on the Internet, covering an entire range of corpuses. These include the human genome, ftp-able files, congressional bills, acronyms, and Library of Congress exhibits. Although providing access (as well as meaning and organization) to all this information may be pushing what some may consider the library's role, our users do come to us expecting our service to include these resources.

Finally, of course, the means by which information is accessed and used is changing. Our users have multiple or no offices. Telecommuting is a growing practice. Information requests and requirements arise while waiting in airports. The growing international user population and information environment make time zones and export laws increasingly important. The importance of security, of computers, information, and customers interests has been growing, especially as the Internet becomes both a carrier and a source of information.

This paper presents technical concerns related to these issues. It will focus on the author's perceptions, and in particular on where and why Z39.50 presents a flexible means of approaching a diverse set of these issues.

2. Overview of the AT&T Library Network and Retrieval Environment

The AT&T library network is from many perspectives based on information retrieval. Our end users search the databases we provide, our information professionals search both internal and external database resources, our publications are based primarily on searching database resources, and our library automation systems are based on database searches and modifications. Our internal database setup since the early 1980s has been Unix–based, with an internally developed database engine called SLIMMER[1]. SLIMMER has been designed to work as a filter – for searching, retrieving, and formatting records from databases; as well as for updating databases. Our library automation systems, including circulation systems, table of contents (TOC) alerting, billing, and photocopy tracking, consist primarily of scripts tying together database retrievals and database updates. These scripts are written primarily in high level languages such as perl[2], AWK[3], and Unix shell(s).

The above conveys a view of our library systems as modules which have, at their lowest level, database retrievals and updates. Until recently, a software module that used a database had to reside on the same computer as the database. From the 1960s through 1990, our library automation systems were mainframe-based. When applications resided on different computers they had the extreme limitation that they could not exchange information with other applications in real time. In the early 1980s this was solved by bringing most of our library applications together on a single computer. This was also when we switched to using the Unix operating system. But by the late 1980s, as our requirements grew, a single mainframe computer proved an inflexible and costly solution.

Our organization took the first step away from this architecture in 1991 when we began using Network File Systems (NFS). In 1991/1992 we moved to a cluster of minicomputers with a common file system using NFS. Unfortunately, this added complexities that we are still dealing with after 3 years:

- Shared database aspects such as shared memory and interprocess control require considerable care.
- Database and record locking has proved a continual problem in a multi-CPU environment.
- The load on the network and the file server is quite high in a large database environment. For example, an application needs to know the number of articles in *Byte* magazine that contain the word "computer". This can be expressed as a search on "Byte AND computer". The inverted file entries for "byte" and "computer" are brought to the computer where the application is running, intersected (ANDed), and the number of records in the intersection saved. To obtain one number, several million bytes of data flow over the network.
- Finally, as our network continues to grow and diversify other network problems are arising. Issues of security with exporting our file systems to computers geographically far away is a concern. In addition, as the computers are more dispersed the speed of the linking network (and the reliability) decreases.

Thus while NFS is good for many of our shared file applications, it has significant limitations for large database applications. A viable alternative approach is the use of database servers.

A database server has a database residing at one place with all the applications accessing the single copy via some robust, flexible technique. When the access technique is a network protocol, this makes the the database server and the application using the database nearly independent. Z39.50 is well-developed protocol that can help meet our internal needs as a database server. In addition, it allows us to use external database servers in a transparent manner.

Increasingly over the last 10 years we have also been mounting databases, both from internal as well as external sources. Most recently we have begun receiving newswire feeds, such as DowVision and AP wires. As our users' data requirements grow, and with them the demand on our organization resources, both computer (e.g. disk space, CPU, backups, security) and human (e.g. database administrators, help lines, tape handlers), our organization is looking constantly at buying database access. But our requirements are high: neither our end users nor staff can be expected to learn multiple interfaces or database setups, plus we require that the location of a database be nearly transparent.

The client-server model is becoming the 1990s' solution to the problems described above. In addition, and perhaps most importantly, it helps minimize the need for people (users at all levels) to learn new interfaces depending on the information resource being accessed. It also opens up opportunities for distributed library applications. It means applications that require database access can be built independent of where the database resource resides.

3. Z39.50 Version 2 Protocol Limitations

This section addresses issues that were of concern in Z39.50-1992 Version 2 – issues that version 3 has resolved. These are protocol limitations – that is, features the standard could not support in the 1992 version. Presenting a simplified user search is probably the easiest way of conveying the first set of issues that arose upon considering Z39.50. Note these issues are mostly resolved in the 1995 version of the protocol.

The following is a simplified user search interaction of a SLIMMER database.

1. The user is presented with a introductory screen presenting the database.
2. The user enters a search, for example "computer retrieval". SLIMMER searches all indexed fields, basically ANDs together the two terms, and tells the user:

   Term "computer" retrieved 21959 items -
   Term "retrieval" retrieved 919 items -
   now 315 in set
3. The user is now presented with ways of reducing the retrieved set; one common method is to restrict the search by field, for example "title".
4. Records are retrieved and displayed.

In the scenario above, the interface knows a fair amount about the database. It needs to know the database name and other relevant information to present the user on the introductory screen. This is probably the first embarrassment to a Z39.50 client implementor; all the interface really knows about the database is the network address; this does not make for a friendly welcome screen. But problems also exist lower in the interaction. Step 3 requires knowing both how to present indexed fields for a given database to a user and how to use them in a Z39.50 search. Step 4 requires knowing the content of a database record and how to present it to the user. A related requirement is knowing how to obtain a given field from a record. For example, if the user says "give me more records by this author" somehow the client software must be able to find the "author" field and know how to use it in a Z39.50 search.

All these issues are solved by the Explain facility in Z39.50-1995. Without Explain, implementations are constrained to conveying database information outside the retrieval session (e.g. by phone or documents). Since our system allows considerable database setup flexibility and change we needed Explain to get started. So we implemented the first stable Explain structure, as proposed in summer 1992. It has proved quite satisfactory for our main needs.

Step 2 above, the search step, also involves a number of protocol features beyond Z39.50-1992. By default SLIMMER searches all indexes. This capacity was not in the 1992 standard but was added shortly thereafter as the Use attribute "any". In addition, note the line:
Term "retrieval" retrieved 919 items - now 315 in set
The intermediate step information about "retrieved 919 items" cannot be conveyed in Z39.50-1992. This required the User Information Format features introduced in the 1995 version.

Finally, step 3 was probably the most controversal issue of a protocol deficiency in Z39.50-1992. SLIMMER carries along information about the fields of a record in which the retrieval terms were used. This allows, for example, a user to search on "einstein" and then based on the number of retrieved records either look at all the records or first reduce the set to those records where "einstein" was used in the "title or subject". This feature exists in many major database providers, as well as in the Common Command language (Z39.58); it can now be done interoperably done in version 3.

Step 4 includes fairly major requirements. It requires the ability to package a record in a Z39.50 message without loss of information. Since SLIMMER has elements (record pieces like author, title) with arbitrary string and numeric tags that can have diverse content, MARC is not an acceptable record package. John Kunze (University of California at Berkeley) proposed a flexible record structure called INFO-1 that supported the functionality required. We implemented and used this structure from 1992 through 1994. This record format evolved into the generic record syntax (GRS), which is part of the 1995 protocol specification. This new format even better suited our needs; in particular it has a clean way to carry a local record key and record dates, part of our basic SLIMMER record.

4. A Security Concern in Client/Server

There is a security requirement that arises once records are delivered into the control of client software. Currently, when our databases contain sensitive data (e.g. social security numbers, passwords) we simply do not display this content. This works quite well as long as we control the user display and interaction with the record. However, once the record has gone out to a user's program we lose control of how the record content is displayed and manipulated. This problem was solved by adding to a database setup a list of fields that are considered sensitive, and these are simply never sent to the client program.

This security approach has solved a number of problems as well as creating new ones. It solved a growing problem: as our interface gave greater control to the user it was becoming more difficult to protect against clever users gaining access to sensitive data. Simply not making the data available is one solution, but is too extreme for some cases. For example, if a personnel record has a special library access password we displayed on the screen

user has library access password.

That is, the password is sensitive, but the fact that it exists is of value and is not sensitive. Solutions to this problem are difficult and have uncertain return, so for now this functionality is an accepted loss in our Z39.50 implementation.

5. Z39.50 Extended Services

When the AT&T library network originally investigated Z39.50 a number of functions were ignored since interoperability was not necessarily a requirement. Many of these have since been proposed and incorporated into Z39.50-1995. Two noteworthy services, item order and database update, are discussed below.

5.1 Item Order

Item order is essential to our users – they complain when databases present records describing materials they cannot easily order. But this doesn't necessarily mean that we require it in Z39.50; it means we require it as a functionality of the client software. Usually this involves the client software gathering some amount of information about the item being requested (from the database record) and information about the user – both from the user and personnel database(s) – and delivering this information to a request handling system. In our present environment part of database setup is setting up what information is needed for a request and where the request is delivered and how. Presently we do not allow requests from distributed clients, since the request invokes request entry commands in our other systems. However, we will soon allow the following technique when a client program is connected to a database mounted at our server:

1. User says "I want the object described on my screen". This will cause an item order to be sent to the Z39.50 server. Technically this is an Extended Service item order package for a resultSetItem.

2. In most cases the server will just acknowledge the request and that will end the transaction (from the Z39.50 point-of-view). This works since the Z39.50 server knows who the user is (required to gain access to the database) plus which record is being requested. The Z39.50 server passes the request on to the correct request handler as set up by the database administrator.

3. When additional information is required; e.g. more billing information, permission to bill, or the size of requested item, this

information is obtained using Access Control via a PROMPT-1 access control format. PROMPT-1 allows the database provider to obtain any required information from the user. This approach is necessary because it is the database provider that knows what additional information is required with a given request, not the client designer.

This does not solve the problem of AT&T employees using distributed clients to search databases at non-AT&T information providers. In this situation we still want user requests to filter back through our request handling organizations. If we controlled the clients it would be easiest to set up our own private request format. In fact we do have a command distributed to many AT&T computers called *library* which sends electronic mail in a fixed format to request library materials and services[4]. If we could have the clients use this existing format our work is minimal. But we believe this approach has significant limitations. In our environment we need an accepted protocol for requests of material. Since we have many traditional library needs (e.g., book buying and borrowing, article photocopies), we are watching the development of ISO 10161 – the ILL protocol. In addition, that protocol is growing and developing to handle orders for diverse types of materials beyond traditional library needs.

The growth of the ILL and Z39.50 standards and their synergy[5] are of interest for other reasons. We acquire significant quantities of materials, especially books and photocopies, from external vendors. To give our customers the turnaround times and service required, we use electronic interaction with our vendors. This usually means setting up a new method of transmitting requests and information about requests with each vendor. If this could be standardized our organization, and we believe our vendors, would benefit.

5.2 Database Update

This, like item order, is a functionality we would like and are pleased will be supported in the context of Z39.50. We use a variety of technologies presently to ensure a single flow of database updates. As our computing environment becomes more complex, ensuring that only a single update process is running has become more complicated. As our environment becomes more distributed, security of data and especially of data updates is an ongoing concern.

Having a single network point for updates, in this case the Z39.50 server, simplifies issues of concurrent updates and security. In contrast with alternatives involving NFS, inter-process control (IPC), and other related network and operating system dependencies, single process control is a preferable solution. So although the availability of clients able to interoperably send database updates is not expected soon, this functionality may have internal application in the near future.

6. External Database Access

Accessing databases at other servers raises a new set of problems. To our users, at least theoretically, there should be no difference between accessing an external database and accessing an internal database. Unfortunately, the real world is not quite that simple. Our initial problems can be divided into several categories:

- Traditional issues of database loading.
- Issues of indexing and index access.
- Features of the remote Z39.50 server.
- Speed of the connection to the remote database.

It is interesting to think of the problems described below compared to issues of buying and locally mounting a database. When our organization buys information resource tapes, we investigate the best resource in terms of content for our customer needs. The assumption is made that our database administrators and systems can then make the data available in a way that will satisfy our customers. With the advent of Z39.50, acquiring a database requires answers to questions relating to whether the available Z39.50 access is sufficiently flexible to meet our requirements. This requires a new set of training and thinking in acquiring database access. The issues described have to be considered and handled before signing a contract.

6.1 Issues of Database Loading

An important advantage of Z39.50 is transparency of database access to the user. Z39.50 hides access differences, but the underlying database is still all important and different for each provider. These differences can be important, but often Z39.50 hides these as well. Issues of update frequency, completeness of loading (often all aspects of a database are not made available), quality and availability of full text, completeness of records (whether all fields loaded), and how record updates are handled are examples of important, basic issues. Librarians have long realized the importance of these factors in selecting and using a resource. However, it is clear that our end users cannot be expected to make similar judgment, especially when we have intentionally screened them from differences among database providers.

These are the first set of issues raised by the internal database administrator when we acquire and locally load a resource. It is easy to overlook these issues when acquiring Z39.50 access, particularly since most end users would not even notice these issues.

6.2 Issues of Indexing and Index Access

Indexing issues with remote servers break into two sets of major problems. The first set are issues of how the database provider indexed the records of the database, the second set concern the Z39.50 interaction.

AT&T library users have come to expect fairly complete indexing. That is, they expect indexing of most of the record content, and the ability to specify which record element is being accessed. How that data is indexed, and the depth to which it is indexed can vary considerably; librarians are trained to be aware of this factor.

SLIMMER allows considerable flexibility in indexing; our end users expect this, and our systems are designed using this functionality. For example, a "phone number" field may have content

123 456-7890

and we might index it so a user can search on "phone number"

123 456-7890 or 1234567890 or 7890 or 123.

The ability of the system to do this type of indexing is a combination of system flexibility, and equally importantly, local control. When database access is purchased indexes may be unchangeable, or at best changed via contract and interaction with the organization that makes the database available.

In addition to these issues are Z39.50 aspects of index access. The first issue often raised is coordinating the client and server to access the correct index. This can be done in three ways:

- Published lists of access points. By far the main such list is the BIB-1 attribute set which is part of the published standard.
- Out-of-band agreements on attributes. That is, the client author and the database provider can agree on values to be used for different access points.
- The Explain database to communicate the available attributes. A client using the Explain database can dynamically learn the access points for a given database and the Z39.50 method to convey using a particular access point. Explain is clearly the most flexible solution to this problem, but unfortunately there are not many existing Explain implementations.

Interpreting what is meant by an access point is a problem both in the Z39.50 environment and other search setups. The user who searches for "author" may or may not expect corporate authors or editors. These are problems in user interface but carry into the interface between the client and server developers.

6.3 Features of the Remote Z39.50 Server

Clients can be designed to compensate for some differences between remote database sources. For example, whether the server sends GRS or USMARC records is not something a user should notice. But some other server features are more difficult to hide from the user. Features (or lack thereof) we have had to cope with include:

- Whether proximity is supported. This feature is more complex, since the level of proximity, e.g., word, sentence, paragraph, is an issue as well.
- Whether Boolean operators are supported. Yes, it is possible to mount a database under Z39.50 and not support Boolean operators.
- How unspecified Use attributes are handled. This is a significant issue in our interface, since the default operation is to search all indexes. Our clients request the desired behavior, using Use attribute "any". However, many servers do not support this. For interoperability reasons our clients then switch for these servers to a Use attribute of "server choice" or send no Use attribute. The undefined behavior of the server at this point has caused some trouble and confusion.
- Whether the remote server supports named result sets. Some functions of our clients require creating and holding intermediate result sets. Without this capacity the clients

can still work but at a reduced functionality.

6.4 Speed of the Connection to the Remote Database

Z39.50 database resources are mainly available over TCP/IP networks (e.g. the Internet). When a user complains about our search response time, and the response time degradation is the result of someone doing real-time video at an unrelated site, this does not satisfy the user. The solution to this problem is the ability to purchase guaranteed band-width which is presently not available. In addition, firewalls and proxy servers can add considerably to response times – another overhead that is difficult for libraries to control.

This has been a major issue in our initial attempts to make Z39.50, as well as other Internet resources, available to our users. In the World Wide Web environment people may be willing to wait 1-2 minutes to hear the President's cat* or several minutes for an "archie" search. However, users expect external databases (which they are not even aware are external) to have response times like internally mounted databases; e.g. a few seconds. We frequently experience long delays in forming a connection, initializing the session, and retrieving records. Some of the user impact of these delays can be mitigated in the user interface and others may be avoided by caching data that the user may need to see or use again. But no complete solution that satisfies our users is presently available.

7. Workstation Issues – and the Future

Our users, and our libraries, are deploying more powerful workstations, and this raises expections about computer access in general.

7.1 User Expectations and Issues

The user expectations are starting to be met, partly through the explosion of client/server solutions coming via the World Wide Web and web browsers. Using existing (or arriving) WWW technology and clients many methods are available to present users with, and help them find, the desired information resource for their needs. These solutions range from standard HTML pages functioning as menus to searchable databases of resources which give back descriptions and pointers to relevant resources. A link on these pages can be either a pointer to text such as a president's speech, or a pointer to a database or set of databases using a Z39.50 URL. When a user selects a database search the web browser could open a Z39.50 session or invoke a companion application process to handle the link.

As appealing as the above scenario is, there are serious issues that need to be resolved. These issues have to do with who "owns" or handles the information connection.

- Who is responsible for the user's workstation as an information gathering tool? My organization is already getting calls from people who want network access, either to our services or to external information providers allowing network access. These range from people with dumb terminals to people with PCs who have never heard of nor want to hear of TCP, to people in restricted networks. Who, then, is responsible for the client software mounted on a user's computer? Although our view (and hope for client/server technology in general) is that these issues reside at the user's end, it is not clear users agree. In order to give users the greater control they desire, users presently must accept the burden of computer system administration as the overhead.
- Who is responsible for response time? Library literature has always claimed 3-5 seconds is desired response time. When our users follow an information link through the AT&T firewall out into the Internet, response time can be in minutes. Responsibility for this problem becomes murkier when the library is paying for the information access, and the user considers it unacceptable or unusable due to response time.
- Who is responsible for the functionality of the user's client software? This is further complicated since the functionality can vary depending on the remote server. There are issues of what functionality the client has (e.g., does it allow proximity searching), whether it interoperates correctly with the desired server, and whether the server supports the desired functionality. It is the author's belief that a well-designed client should present the user with

* Available at http://www.whitehouse.gov/White_House/Family/other/socks.au

functionality up to what it supports (e.g. Boolean operators, word proximity, USE attributes), and smoothly present the user with interoperability issues pertaining to a given server (e.g. the server fails the word proximity search request).

- Who is the user's point of contact? This issue intermixes with how databases are presented to the user: e.g. whose name and number appears on the screens, and who the user perceives is providing the information service. Setups where the user is presented with the information resource as coming from an internal provider as well as setups where the user is fully aware that a remote provider is involved are both used in the non-client/server world. Which setup is used is based on what the information provider and the intermediaries perceive as their role and relationship to the user.
- Who is responsible for the functionality and contract with the database server? This will be an early issue that needs to be resolved by my organization. If we purchase access to databases for a customer, we want the customer to help pay for the access and to be aware of the library's role, and we want user requests to be in our control and user feedback to come back to our organization. Whether we will be able to keep this degree of control in the new environment is uncertain – but the money issues are the bottom line. A number of solutions to this problem exist, but many details need to be resolved.

7.2 Library Expectations and Requirements

Our libraries have two points of interaction with databases; for searching and for systems access. In the case of searching, the staff have the issues (and desires) of users – they want a single integrated search environment. Library user wants are similar, but a single, consistent, user adaptable environment is most important since diverse users use the same system. Presently this goal has not been reached, as users are confronted with different interfaces for every CD-ROM product, laser-disk system, locally mounted OPAC-accessible databases, and, increasingly, OPAC-accessible externally mounted databases. Our vision is of a library OPAC that consists of client software accessing all the resources the library makes available through one common interface. Although we have attempted to achieve this in the past, without a common protocol and buy-in by the database providers this goal is basically unachievable.

Our library staff also accesses databases for all the basic functions of a library (e.g. for circulation, entering and tracking user photocopy requests, and for checking book status). It is not clear that workstation client/server access is an improvement for this functionality. However, if we decide to move in this direction, our environment combined with Z39.50 may make this a less painful move than might be expected. At a low level (below user interface) we should be able to take our present environment built on high level language scripts driving database access and updates and port it. That is, in theory we should be able to purchase a PC version of Perl, Awk, and shell and use this to provide our present functionality. Although we have little implementation experience at this level, we have reason to believe our environment has the desired flexibility and functionality to achieve a move of this scope.

8. Conclusion

The AT&T library organization developed an interest in Z39.50 for a number of diverse reasons. We hope that eventually Z39.50 will help with or solve several classes of problems ranging from behind-the-scenes issues resulting from distributed computing architectures to diversity of user interfaces. In addition to helping with known problems, we hope that Z39.50 will give us the flexibility we require for a constantly evolving library organization in an international corporate environment.

For Z39.50 to meet our needs, the main requirement is that the protocol itself incorporate all the functionality of our existing information retrieval environment. Version 2 of the standard was a major start, but version 3 comes much closer to incorporating existing functionality. In particular, Explain, generic record syntax (GRS), search restriction by attribute, new search information formats, and new access formats make the protocol viable in our environment with little or no non-conforming extensions to the protocol.

The next major requirement is proven interoperability and transparency of database providers to our users. This functionality is coming, though somewhat slower than hoped. Issues of indexing style, default operations and ways to override defaults, database coverage and loading characteristics become even more apparent in Z39.50 than in the traditional online world. However, we will soon be able to buy database access rather than mount tapes, with no loss of functionality or noticeable changes in access for our customers.

Our final requirement is that our end users, like users everywhere, are expecting interfaces integrated into their regular computing environment. Developing user interfaces for diverse user environments takes significant resources. Our users also expect the search tools to work the same, whether against internal or external resources. The clear solution to this problem is a well-accepted search and retrieval protocol. Z39.50 is well-positioned to become this protocol, and in this belief we have focused our attention on developing a high-quality server for our internal resources.

After three years of involvement in Z39.50, it appears that Z39.50 was the correct choice for flexible future growth of our organization. Z39.50 continues to gain acceptance and Z39.50 implementations continue to become more available. Acceptance by ISO is an important milestone. Increasingly, database suppliers (including CD-ROM vendors) offer Z39.50 access, making this technology an increasingly attractive alternative to mounting databases internally. The federal Government Information Locator Service (GILS) initiative makes access to government information, which is important at a corporation such as AT&T, a desired benefit. Finally, the growing interest in Z39.50 in the Internet community as demonstrated by development of Z39.50 URLs makes it likely that the less formal information resources of the Internet will also be available and searchable by a common protocol.

REFERENCES

1. Waldstein, Robert K. SLIMMER - a UNIX system based information retrieval system. *Reference Services Review* Vol 16, No 1-2, pp 69-76, 1988.

2. Wall, Larry and Schwartz, Randal L. *Programming perl*. O'Reilly & Associates, Inc. 1991.

3. Aho, Alfred V., Kernighan, Brian W, and Weinberger, Peter J. *The AWK Programming Language*. Addison-Wesley Publishing Company, 1988.

4. Waldstein, Robert K. *Library* an Electronic Ordering System. *Information Processing and Management* Vol. 22, No 1, pp. 39-44, 1986.

5. Turner,Fay *The ILL protocol and Z39.50*. Available via anonymous ftp from ftp.nlc.ca; in postscript (zillart.ps) or as ascii text (zillart.txt).

The Implementation of Z39.50 in the National Library of Canada's AMICUS System

J. C. Zeeman
Software Kinetics Limited.
Stittsville, Ontario
zeeman@sofkin.ca

Abstract

AMICUS is the National Library's new integrated bibliographic system. The initial phase of development, released in the second quarter of 1995, supports cataloguing and catalogue products, bibliographic searching and customer information management. The search module is implemented as a Z39.50 server that accesses the two database engines integrated into AMICUS: a relational database for bibliographic data management and a full text database for keyword searching and future full text access. An overview of the AMICUS applications is followed by a brief introduction to the modelling of AMICUS bibliographic information in the relational database. A more detailed description of the architecture of the AMICUS search engine is given, describing components, internal messaging, query analysis, optimization, semantic mapping and record conversion. A description of the three AMICUS clients concludes the paper.

Background

The National Library of Canada is the national copyright deposit library in Canada. It serves as the primary cataloguing agency for Canadian published materials and as the national agency for assignment of International Standard Bibliographic Numbers (ISBNs) and International Standard Serial Numbers (ISSNs). It makes catalogue records available to a large number of Canadian libraries and exchanges national level records with both the Library of Congress and the British Library to make their records available as source records in Canada. The NLC maintains a large union catalogue of Canadian holdings and hosts the catalogues of several other federal agencies in Ottawa (the "full-service libraries"), notably the catalogue of the Canada Institute for Scientific and Technical Information (CISTI) — the library of the National Research Council of Canada. Because Canada is a bilingual country, cataloguing is done in both English and French, and the NLC offers its products and services in both languages.

The NLC began preparations for the replacement of its bibliographic system, DOBIS, in the late 1980s. A fundamental conclusion arising from extensive investigation was that no vendor could provide an off-the-shelf product that would meet the complex requirements of the National Library in terms of support for multiple languages, standard number assignment, and multiple overlapping databases to support:

- Canadiana cataloguing,
- multiple sets of source records for both bibliographic items and authorities,
- the Canadian union catalogue incorporating records of highly variable quality, and
- the individual library catalogues of the National Library and its full-service partners.

In 1992 a Canadian systems integrator, Groupe CGI, was competitively awarded the contract to develop a new information system for the National Library. Software Kinetics Limited teamed with CGI to develop the winning proposal and acted as subcontractor during design and implementation of the system, with particular involvement in the design and development of the bibliographic search engine.

Amongst other requirements the RFC specified that the system must:

- be based on a relational database system(RDBMS) for management of bibliographic data;
- support keyword searching;
- meet stringent performance requirements for both cataloguing and searching;
- support Z39.50 access; and

- support an initial database in excess of 10 million records, rising to 20 million over the life of the system.

The winning bid proposed the use of Digital Equipment's VAX hardware platform with the VMS operating system, the Ingres RDBMS and Fulcrum's Ful/Text engine for keyword access. Catalogue access would be provided for a Windows graphical user interface (GUI) client and for a host-based terminal client. Analysis, design and as much of implementation as possible would be done using CASE tools.

Work on the project plan began in July 1992. Development was largely completed by April 1995 and the system went into production on 12 June 1995.

AMICUS applications

Cataloguing

Cataloguing is implemented as a Microsoft Windows-based client-server application developed using the Ingres OpenRoad application development tool. Cataloguing is intimately connected with searching: the cataloguing functions are available to authorized users as menu options from the search windows and vice-versa. The main cataloguing window appears as a MARC-based worksheet. Cataloguers can copy-catalogue from the large set of source records; edit previously created records either to correct errors or to create new records for similar items; or add new records.

The basic worksheet is a blank form on which the cataloguer can create a MARC-tagged record. The same form is used for all record types, including authorities. The cataloguer can input tags, indicators and codes directly, from memory, or can choose appropriate values from labelled pop-ups. All input is validated and updates the database immediately.

The cataloguing application matches cataloguer input for controlled headings, including names, titles, subject headings, control numbers, classifications and call numbers with data elements existing in the database. If the cataloguer's input is matched, the access point table for the appropriate heading is automatically updated. If the input is not matched, the cataloguer may browse existing values in the database to find the appropriate term. If a term is not found, the cataloguer may choose to add the term to the database, in which case he or she is presented with a worksheet on which to enter the required control information. When this information is supplied, the original workform is automatically updated with the new data element.

Standard Number management

The National Library acts as the Canadian numbering authority for International Standard Book Numbers (ISBNs) and for International Standard Serial Numbers (ISSNs). To facilitate its management obligations, AMICUS incorporates applications that allow National Library staff to manage the assignment of number blocks to publishers and trace number assignment to individual items.

Bibliographic Searching

Bibliographic searching is supported for users of the AMICUS Windows-based client and for users of terminals connected to the host VAX computer via telnet or Datapac, the Canadian public X.25 network. For these latter users, two host-based search interfaces have been implemented: a command-based search interface for experienced searchers, modelled on the NISO Z39.58 Common Command Language; and a form-based interface for patrons of the National Library and CISTI reading rooms and reference services. All these interfaces use the NISO Z39.50 Information Retrieval protocol to communicate with the bibliographic search engine located on the VAX computer.

The bibliographic database is maintained as a fully normalized Ingres relational database, with keyword indexes to bibliographic records maintained in the Ful/Text component of the system. The bibliographic search engine manages concurrent searching of both database systems transparently from the user's point of view.

The AMICUS system allows the user to search a number of databases. At present the following databases are available:

- Canadiana bibliographic records,

- Canadiana authority records,
- Library of Congress Cataloguing Distribution Service bibliographic records,
- the catalogue of the National Library's collections,
- the catalogue of the CISTI collections,
- the catalogue of the AMICUS Full-Service libraries,
- the Canadian Union Catalogue.

In addition the user can search the combination of all the bibliographic databases as "Any AMICUS database".

These databases are implemented as logical subsets of the single AMICUS physical database. The same physical database record can be associated with multiple logical databases and different, possibly conflicting, data elements can be part of the record in different databases.

User/Supplier Information

AMICUS includes applications to manage information pertaining to the users of National Library systems and information and also to the suppliers of information to the National Library, including publishers participating the cataloguing-in-publication, ISBN and ISSN programs and libraries contributing to the Union Catalogue.

Products

The existing set of National Library bibliographic products, including *Canadiana*, the national bibliography, will be produced from AMICUS. The system includes applications to generate and manage these products.

Billing

The National Library has an obligation to recover a portion of its costs and most of the bibliographic services provided by the National Library to other libraries are therefore charged services. AMICUS includes applications to monitor usage and generate billing information for the production of invoices.

Functions Not Implemented

The current release of AMICUS does not support the following functions, among others:

- circulation
- acquisitions and serials control
- ILL messaging
- financial management
- publishing and ad-hoc products

These functions are currently provided by legacy systems or by the Dynix integrated library system acquired by the National Library to serve as an interim measure until the functions can be integrated into AMICUS.

The Relational Database

AMICUS is fundamentally a relational database. This has a significant impact on how searches are performed in the system.

A greatly simplified version of a portion of the logical data model used for AMICUS bibliographic information is shown in Figure 1 below. Arrows in the diagram point in the direction of the "many" aspect of a one-to-many relationship. The diagram shows how some of the entities in the system are related to each other. To keep the diagram simple many entities, such as subject headings, notes, physical descriptions, location information, etc. have been omitted. It should be noted how names, titles, etc. are related to bibliographic items and to authorities and also how bibliographic items are related to physical copies and to each other. There is a separate access point entity for each entity that has a many-to-many relation with the bibliographic item. Most of the relationships shown in the diagram are optional in the sense that not every instance of the entity will have a corresponding access point: for instance, some bibliographic items will have no name heading related to them.

Each of the entities has a number of attributes that specify both the data elements that form the entity and the entity's relationships with other entities. The principal attributes for the three major types of AMICUS entities are shown below.

Principal attributes of a headings entity:

- heading number
- heading type (e.g. "personal name inverted order" for a name, etc.)
- heading display text
- heading searchable sort form
- heading inverse sort form
- control information (e.g. language, verification level, etc.)

The "heading searchable sort form" is a normalized form of the heading to allow the headings in the table to be sorted in lexical order, for production of scan lists, and to enable a heading be matched against a searcher's input term without having to worry about variations in capitalization and punctuation. This normalized form is stored as a distinct data element, and is in fact the principal element used for searching. The exact nature of the normalization performed depends on the heading type. Typically punctuation and MARC subfield codes are removed, all text is converted to upper case, non-ASCII characters are mapped to ASCII equivalents, etc.

The "heading inverse sort form" is present to allow backwards movement in a scan list. ISO standard SQL does not support moving backwards in a table, so the inverse form contains a simple binary inversion of the searchable form, which has the effect that moving forwards in the order of the inverse sort form actually moves backwards in the order of the searchable sort form.

Principal attributes of an access point entity:

- heading number (to provide access to and from rows in the headings tables)
- bibliographic item number (to provide access to and from rows in the items table)
- heading function (e.g. "added entry", etc.)
- control information

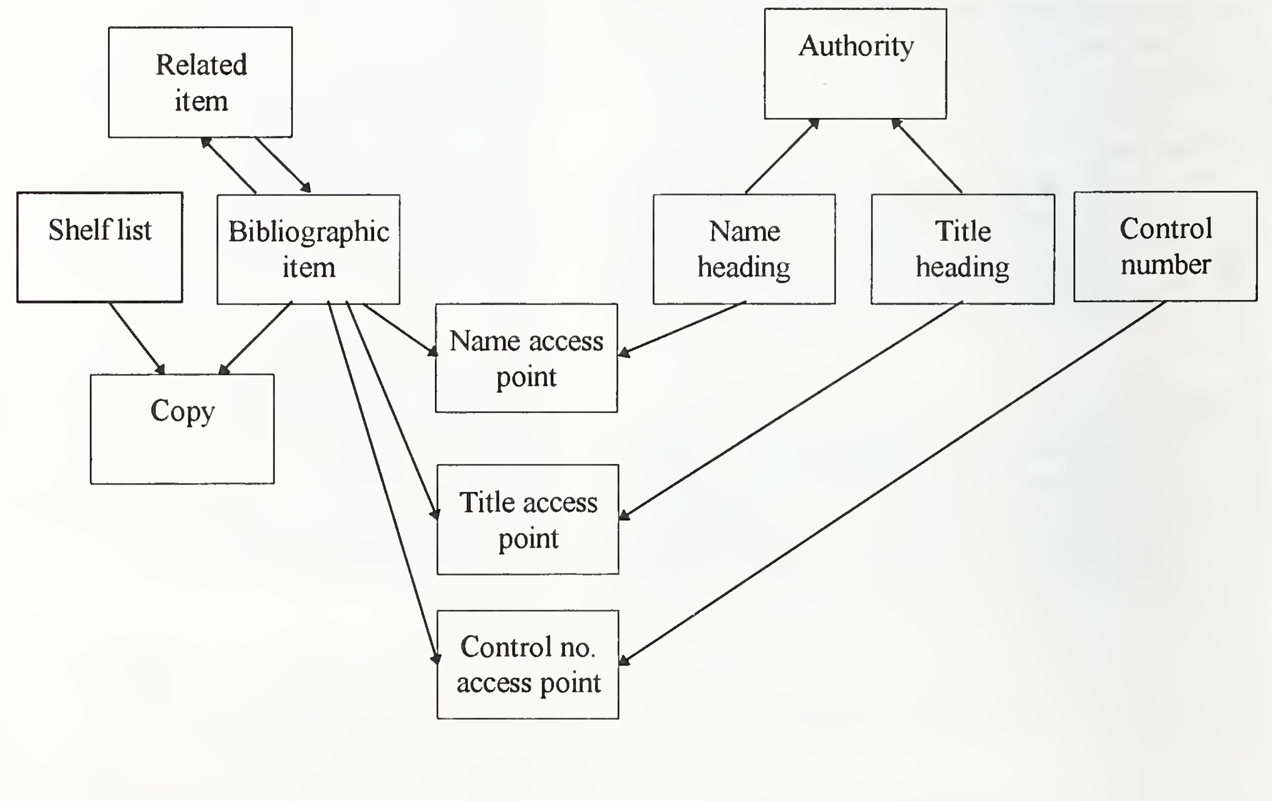


Figure 1 - Portion of AMICUS Logical Data Model

The heading number and bibliographic item number attributes allow instances of the bibliographic item entity that are associated with a given heading to be selected and vice versa. The heading function element allows appropriate displays to be constructed on the basis of the heading function and also allows selections to be made on the basis of the type of relationship between the heading and the bibliographic item.

Principal attributes of the bibliographic item entity:

- bibliographic item number
- various coded elements (e.g. bibliographic level, record type, country of publication, etc.)
- date of entry on file and of last transaction
- dates of publication
- control information.

It should be noted that the bibliographic item entity contains none of the descriptive information or headings normally considered to be part of the "bibliographic" information.

This data model is used as the basis for the physical design of the database. The database consists of a series of tables, each corresponding to a single entity in the logical model. The attributes of the entity become the columns in the table and the data records become rows. Any individual element in the database can therefore be identified as the intersection of a row and column in a particular table. The internationally standardized Structure Query Language (SQL) is used to create the database and to access and manipulate all the information in the database, and forms the only valid means of accessing the data in the database.

An SQL statement to retrieve bibliographic records related to a name would be as follows:

```
SELECT
        itemNumber
FROM
        nameHeading,
        nameAccessPoint
WHERE
        nameHeadingSearchForm IS LIKE
            'JOHN SMITH %'
        AND
        nameHeading.nameHeadingNumber =
            nameAccessPoint.nameHeading-
            Number
```

This query selects values of the item-number column in rows in the name-access-points table that are identified by having the same value in the name-heading-number column as rows in the name-headings table that are identified by having a value in the name-heading-sort-form column beginning with the string "JOHN SMITH ". It should be noted that it is not necessary to access the bibliographic items table in order to create a set of bibliographic item numbers.

An SQL statement like that shown above is readable by a human operator. It represents, however, a very simple case. When additional qualifiers are added, such as heading types and heading functions, SQL statements quickly become too long to be easily input or even understood by a human operator. System users cannot be expected to routinely use SQL to query the database and SQL is in fact normally generated by an application that presents a more intuitive interface to the user and shelters him or her from the physical schema of the database and from the complexity of the SQL syntax.

Although a large number of tables can theoretically be joined in one query, experience has shown that performance begins to degrade rapidly when an SQL Select statement requires accessing more than three tables. The AMICUS data model allows single-term searches to be executed with only two-table accesses in all but a very few instances; for these, three-table accesses are required. Four-table or more accesses are never used in the AMICUS bibliographic search engine.

The Full Text Engine

Keyword access to AMICUS bibliographic data is provided by use of a separate database system — independent from the relational DBMS — the Ful/Text DBMS. This system maintains keyword indexes of all names, titles, subject headings and bibliographic notes. Searching is performed via a proprietary Application Programming Interface (API) that supports Boolean operations, proximity searching, thesaurus look-up and substitution, and relevance ranking of results. The last two features are not used in the current AMICUS implementation, although they will offer considerable power in later phases of development when full text and other data is added to AMICUS.

A Ful/Text database is modelled as a set of documents that are searchable via the engine. To allow specificity of searching, a document can be divided into a number of "zones" that can be searched individually or in combination. For AMICUS records the following zones have been defined, "author", "title", "subject", "publisher" and "notes". Zones in a Ful/Text query are specified using numeric zone identifiers.

Ful/Text does not require the documents themselves to be stored as part of its database. Documents can be external entities to which Ful/Text maintains searchable indexes together with a catalogue of pointers back to the documents themselves. AMICUS thus maintains no bibliographic data in the Ful/Text database. Instead, the Ful/Text indexes refer to the bibliographic records in the Ingres database.

The simplest Ful/Text query consists of a single word to be found: e.g., "SMITH". This will build a result set of documents containing the word "SMITH" anywhere in the document. Searches can be restricted to specific parts of documents by using a "zone operator": to find a word used as part of an author's name, the search string would be "\C40s SMITH", where "\C" represents an escape sequence, "s" is the zone operator and "40" the parameter indicating the zone value to which the search is to be restricted. More complex queries are created by adding Boolean, proximity and other operators.

In addition to the zone operator, Ful/text supports a number of other operators that act on multiple words. The syntax of these operators requires a leading escape (represented by "\C"), one or more optional parameters, the operator identifier, the words the operator is to be applied to and a final escape following the term plus a closing brace ("\C}") to indicate the limit of the scope of the operator. The proximity operator, for example is "\C<distance>p ... \C}". A phrase is searched as two adjacent words by using the proximity operator with a distance parameter of zero, so the Ful/Text query for a phrase used in an author's name is: "\C40s \C0p SMITH JOHN \C}". Operators can be nested and can occur in any order.

The basic search unit of Ful/Text is the word, and all operators other than proximity apply to one or more words. The only distance unit supported for proximity is character distance. AMICUS therefore, also supports only the character unit for proximity distance.

The result of a Ful/Text search is a result set of document ids maintained by the Ful/Text engine. The bibliographic search engine obtains these ids from a Ful/Text search to use as either an interim result set or as the final result set of the search as appropriate.

The Bibliographic Search Engine

The bibliographic search engine is at the heart of the AMICUS bibliographic information management and retrieval functions.

The principal requirements to be met by the AMICUS search engine were as follows:

- to integrate search access of the relational and the full text databases;
- to integrate with the cataloguing application being developed using Ingres Windows-4GL;
- to support a very large database;
- to provide a Z39.50 version 2 target;
- to support up to 250 simultaneous users;
- to meet stringent performance requirements.

The design approach was to modularize searching as much as possible into separate components, each dedicated to a specific role in executing the Z39.50 query and each operating independently and simultaneously. The general architecture of the search engine is shown in Figure 2 below (acronyms are explained below).

The search engine thus consists of a number of processes that together implement the search functionality supported by AMICUS. Each of the processes in the search engine operates independently of all others, and uses asynchronous messaging to communicate with the other processes as needed. Each process maintains its own message queue and deals with each message in the queue in turn. While the message queue is currently implemented using the VMS mailbox utility that provides efficient low-level support for interprocess communication, message handling is sufficiently isolated that a different messaging mechanism, such as RPC, could be implemented without major disruption should the engine be transferred to a different platform.

Each of the processes is dedicated to a specific task in responding to a Z39.50 protocol message.

EIT The *External IR Target* implements the Z39.50 protocol machine for non-AMICUS users. It is based on the IR Toolkit software developed by Software Kinetics Ltd. for the National Library of Canada. It implements version 2 of the protocol as specified in the 1992 standard. All services of version 2 are currently supported with the exception of AccessControl, DeleteResultSet, and ResourceReport.

The EIT polls for incoming Z39.50 Application Protocol Data Units (APDUs). When an InitRequest APDU is received, it decodes the APDU and issues a message to the MISR process (see below) requesting validation of the user's authentication information and the supply of session information such as the various resource limits to apply to searches and the set of databases the user is allowed to search. The protocol machine uses the response from the MISR to formulate the InitResponse APDU.

APM The *AMICUS Protocol Manager* process implements a proprietary protocol developed for use between AMICUS clients and the bibliographic search engine. This protocol acts as a wrapper around Z39.50 protocol messages and other messages used by AMICUS clients. AMICUS clients offer additional search services not available to external Z39.50 clients, such as saving queries and result sets. These services were not available in the 1992 text of Z39.50 and what became the 1995 specifications were not sufficiently stable when the AMICUS design was finalized in early 1993 to allow them to be implemented. Therefore, a proprietary protocol has been used.

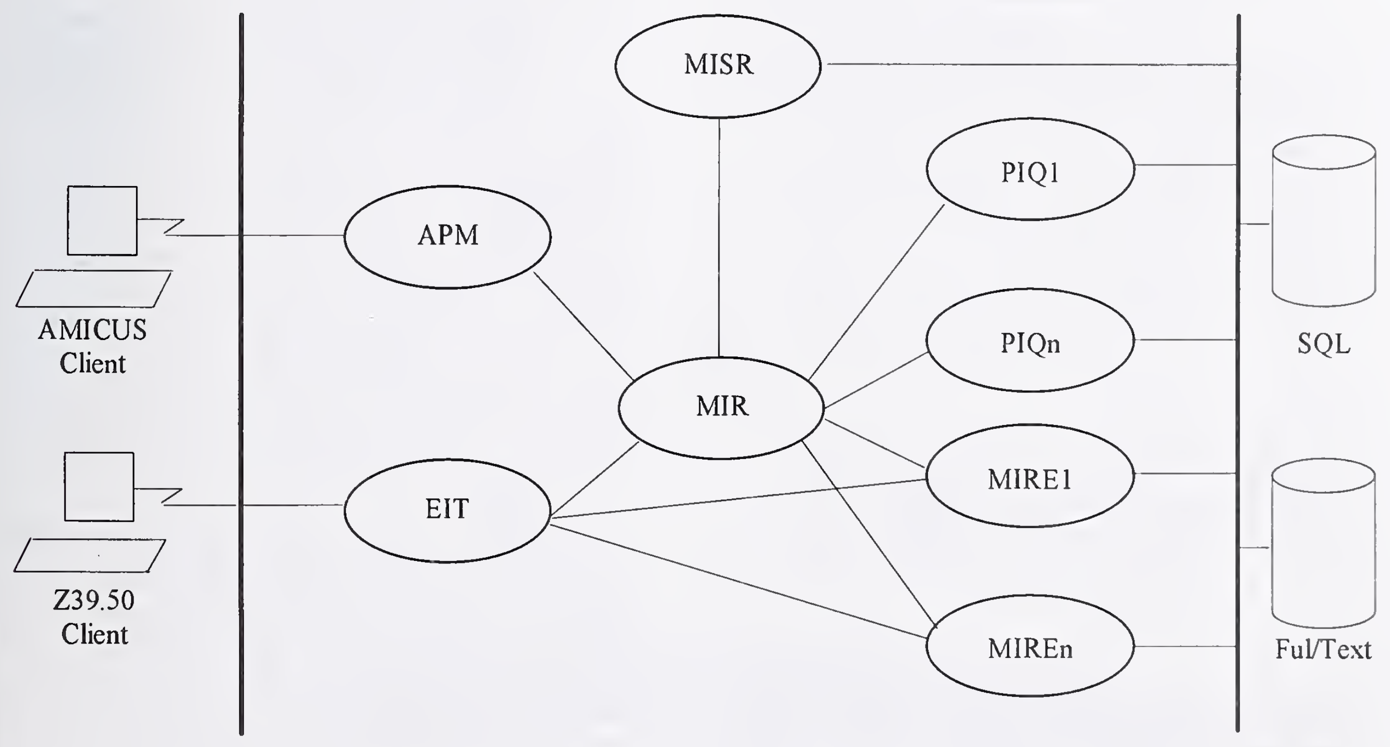


Figure 2: Bibliographic Search Engine Architecture

The AMICUS protocol is also used by AMICUS clients to request that a MARC record image be placed in an Ingres table for subsequent use by the client. The AMICUS protocol is furthermore designed to permit the transfer of Z39.50 messages to and from a remote Z39.50 host via a client gateway on the AMICUS server. None of these services are available to non-AMICUS Z39.50 clients.

The APM includes the same Z39.50 protocol machine as the EIT and generally behaves in the same way. Principal differences lie in the internal naming of result sets and in the creation of records. AMICUS clients do not use Z39.50 to obtain records for display, but instead manage their own displays directly from the Ingres database.

MISR The *Manage IR Security and Resources* process finds the message in its queue and executes the appropriate SQL statements to obtain authorization and session information. If the user has supplied a valid user id and password the MISR obtains the necessary session information, such as resource limits that apply to the user and databases the user is allowed to search, and makes it available to other processes for use as required.

MIR Query analysis, optimization and execution is managed by the *Manage Information Retrieval* process. There is a single MIR process that continuously loops through all outstanding searches, analyzing queries and dealing with messages from other processes as necessary.

The *query analyzer* receives a decoded query from the protocol engine (APM or EIT) and builds a *query execution tree* based on the logic of the query. Figure 3 below illustrates the query tree that would be built from the common command language query “find TW cats or dogs and SU pets or animals and DA > 1993” (find records with title word “cats” or “dogs” and with subject heading “pets” or “animals” and with publication date greater than 1993). As illustrated in Figure 3 below, each term in the query becomes a leaf of the tree and leaves are joined into branches with the operators in the query. Execution of the query begins with the bottom left-hand leaf and proceeds upwards and to the right. Branches of the tree may be optimized to make most effective use of the database engines' native query optimizers while minimiz-

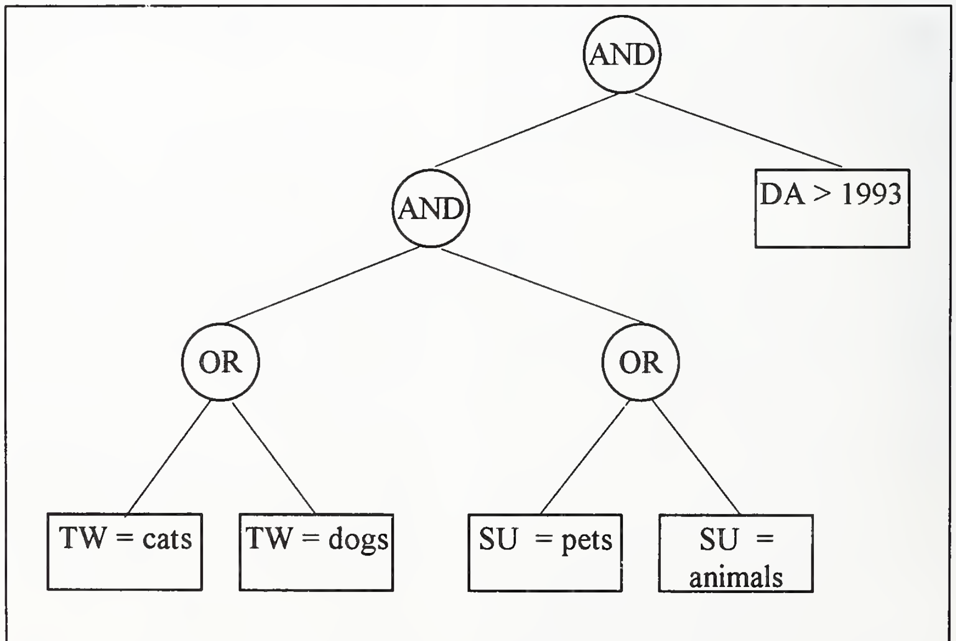


Figure 3 - Query Execution Tree

ing query execution time. In this query, the title-word terms are passed to the Ful/Text engine as a single subquery, and the subject-heading terms are passed to the Ingres engine as a single subquery using the SQL "union" operator. The result of each search is an Ingres table holding the intermediate result. These two subquery results are next joined into a single intermediate result and, since date of publication is not indexed in the Ingres database, the intermediate result will finally be joined with the date attribute in the bibliographic items table to form the final result set.

Query execution is implemented as a finite state machine that makes recursive passes through the query tree, optimizing where possible, dispatching subqueries to a database interface process (a PIQ, see below), interpreting results and changing the state of the various nodes as appropriate. Each query node may be in any one of the following states: "incomplete", "wait for dependent", "wait for PIQ", "wait for result", "complete". The query is reprocessed until the top-most ("root") node reaches the "complete" state at which point the query has been fully processed and the result set (if any) has been built.

When the MIR receives a message from a PIQ that a subquery has been processed, it finds the node associated with that PIQ and changes the state of the node as appropriate. It then reanalyzes the query, changing the state of other nodes as necessary. For instance, if the dependents of a query node have all reached the "complete" state, the node can then be changed from "wait for dependent" to "wait for PIQ", at which point the subquery specified by the node will be executed.

MIRE The *Manage IR External* process manages the creation of MARC records for return to an external Z39.50 user. There are multiple MIRE processes active simultaneously; the number is specified at search engine startup time, allowing system managers to tune this number for optimal use of system resources and performance. If the SearchRequest contains a "piggy-backed" present, the MIR will ask the next idle MIRE to create one record required for the response. If multiple records have been requested, multiple requests will be passed to one or more MIRE processes. If the EIT receives a PresentRequest it will repeatedly ask the next idle MIRE to create the next record required for the response. To create the record the MIRE executes a database procedure that extracts the required data from the various database tables and then assembles this data into the MARC exchange format. At present only the CanMARC format is supported for output.

PIQ All search interaction with the two database engines is handled by a *Process IR Query* process. As with the MIREs, there are multiple PIQ processes active simultaneously, with their use managed by the MIR. The number of simultaneous PIQs is specified when the search engine is started. Each PIQ can interact with both the RDBMS and the full text engine. Since execution of SQL statements by the RDBMS is synchronous (i.e. the process blocks until the query completes), using multiple PIQs allows multiple subqueries to be executed simultaneously.

The PIQ executes the database interaction and builds either an intermediate or a final result set. These sets are placed in RDBMS tables. Each PIQ processes a single subquery, which may be an SQL select statement to evaluate a single term, an optimized query that contains several terms, an SQL statement to perform a Boolean operation on the results of previous terms, or'a full-text query.

Full-text queries are always optimized as much as possible. The result of a full-text query is a list of matching bibliographic item keys, which is copied into an Ingres table as the final or an intermediate result set.

Semantics Tables

The query analyzer has no built-in knowledge of the semantics of the AMICUS databases or of an AMICUS search. All the semantic knowledge relating to the physical data model of Ingres and the Ful/Text database structure resides in separately maintained *semantics tables* used by the query analyzer to generate subqueries for execution by a PIQ. There is one semantics table for each record type in the system that requires different semantic processing. In AMICUS at present there are separate semantics tables for authority records and for bibliographic items. Adding search support for other record types may involve as little work as defining a new semantics table.

The query analyzer has a only basic understanding of how to construct an SQL statement that involves a two- or three-table join and how to construct a Ful/Text query. It also incorporates a growing set of deterministic optimization cases based on comparison of tables names and other data from the semantics tables.

The semantics tables are held as Ingres tables for ease of maintenance, but for performance reasons are stored in memory while the search engine is running. This permits the search semantics to be altered in the database tables as required, without interfering with the operation of the search engine. Alterations will take effect the next time the engine is restarted (normally daily).

The semantics table for a database lists every Z39.50 Bib-1 attribute combination supported for that database, and, for each, specifies the semantics of the subquery that corresponds to the attribute combination. Some of these combinations specify full-text searches, others specify SQL searches of the Ingres database. For every term in a query the attribute combination is looked up in the table. If a row is not found, an "unsupported attribute combination" diagnostic is returned and the search is failed. If the row is found the query elements are placed in a memory structure used by the MIR to construct a subquery that is passed to a PIQ for execution.

Each row in a semantics table contains the following information:

- the attribute combination,
- parameters used in generating the searchable form of the term,
- a flag to indicate whether the attributes specify an indexed or unindexed database element, and either
- a skeleton Ful/Text query or
- the components of the SQL query.

The elements of the last item are used to create an SQL select statement for the subquery and may be used to optimize several subqueries into a single more complex select.

For example, the attribute combination:

use = 4 (title),
position = 1 (first-in-field),
relation = 3 (equality),
structure = 1 (phrase),
truncation = 100 (do not truncate),
completeness = 1 (incomplete subfield)

retrieves the following parameters from the semantics table:

primary search table = titleHeading
first join table = titleAccessPoint
second join table = null
key of primary table = titleHeadingNumber
join table key for first join = titleHeadingNumber
join table key for second join = null
element name from which to select record id = bibliographicItemNumber
element name in primary table in which to match term = titleHeadingSearchForm
SQL operator = LIKE
variable string containing the search term = '%s %%'
string constant to add to the SQL statement = null

These parameters lead to the creation of the following SQL statement (assuming the user's search term is "Rape of the lock":

```
SELECT bibliographicItemNumber
FROM titleHeading, titleAccessPoint
WHERE
        titleHeading.titleHeadingNumber =
titleAccessPoint.titleHeadingNumber
AND
        titleHeading.titleHeadingSearchForm
IS LIKE 'RAPE OF THE LOCK %'
```

Note that the C-language "sprintf()" function is used to place the term into the SQL statement. This function replaces a "%s" in the input string with a variable (in this case the search term). To use "%" as a literal character in the output string requires that it be repeated in the input string, thus " '%s %%' ". The single quotes are literal characters in the output string that are required by the SQL syntax.

Most of the data in a bibliographic record (in particular, almost all coded data elements in a MARC record) can be searched using the search engine.. To support searching these coded elements, some 240 local use attributes have been added to the basic Bib-1 attribute set. Not all these elements are indexed, however. If these unindexed elements were searched on their own, complete Ingres table reads would be required to satisfy the query, a process which would take hours, if not days, in a database of 20 million bibliographic records. The search engine therefore enforces rules as to the combination of indexed and unindexed terms that may be searched. Any unindexed term may be searched in combination with a result set, with an indexed term or with an intermediate result. Otherwise, if query optimization would result in an unindexed term on its own (or a combination of unindexed terms) forming an SQL statement, a user-authentication parameter obtained during initialization is examined to determine whether the user has the privilege to initiate searches of unindexed terms (very few users will have such permission). If the user does not have permission, the entire search is failed and an appropriate diagnostic is returned.

To allow users to search for multiple values of the same unindexed element (e.g. language English or French or German) in a single query without having to input a query with complex nesting of parentheses and repetition of terms, the search engine allows lists of such values to be sent as a single term, which the engine processes as an SQL "... IN (value,...,value)" statement, e.g., code-language = eng, fre, ger. Queries of this form are handled very efficiently by the RDBMS.

AMICUS Clients

Three separate user interfaces have been created for the AMICUS system. One is a Microsoft Windows interface intended for use by cataloguing and other technical services staff; one, *Access AMICUS*, is a VMS host-based interface using a command-driven paradigm, intended for use by subscribers to the NLC's MARC record distribution, union catalogue and ILL services; and the last, *ISAAC*, developed for CISTI, is also a VMS host-based interface but using a menu and form-driven paradigm, and intended for use by patrons of the NLC's and CISTI's reading rooms and reference services. There are French and English language versions of each of these interfaces, and users can switch freely between the two languages.

The Windows interface provides access to all AMICUS functionality, including bibliographic searching and cataloguing. Access AMICUS supports bibliographic searching, ILL requesting and notification of holdings. ISAAC supports bibliographic searching and the creation of requests for the NLC's ILL system and CISTI's document delivery system.

All the clients support complex searching using nested Boolean operators, keyword searching and proximity searching. They all support the use of at least the previous result set as an operator in the query. They all support index scanning and generation of searches from index terms and offer multiple record display formats that are user selectable.

Each of these interfaces uses the same Z39.50 origin software, based on the IR Toolkit developed by Software Kinetics Ltd. The use of the Z39.50 protocol is, however, invisible to users of the AMICUS interfaces.

The interfaces initialize a Z39.50 session when they are started. Subsequently they use the Search service only. The Z39.50 client portion of each interface receives a Z39.58 Common Command Language (CCL) query string from the user interface, together with other search parameters (such as the name of the database to search), parses the query into a Z39.50 type 1 or 101 query, and generates a SearchRequest message which is wrapped inside an AMICUS Protocol message and sent to the search engine. The CCL query string may be either input directly by the user or generated by the user interface in response to mouse clicks or form filling.

None of the clients currently request records to be returned from the Z39.50 search. Instead they access the result set in the Ingres database directly to obtain display information, using the proprietary IngresNet protocol. If a search results in a single record, it is displayed to the user in the session default display format. Numerous predetermined display formats are currently supported by AMICUS, including brief and full labelled formatted displays for specific uses such as interlibrary loan or reference services, and two CanMARC displays, one oriented toward descriptive cataloguers and one for subject cataloguers. If a search results in multiple records, a tabular display of result set records is presented, from which the user can select one or more records for a fuller display.

Similarly, index scanning and result set sorting are implemented through IngresNet access to the Ingres database. When the architectural model for the client was prepared, the 1995 Z39.50 Scan and Sort services were insufficiently stable to permit their use in the AMICUS clients. Future versions of the client may implement the Scan and Sort services.

Each of the clients contains a query parser that generates Z39.50 queries from user input. The same parser is implemented on all clients. This parser accepts a Z39.58 Common Command Language (CCL) string as input and generates as output a Z39.50 version 2 query of type 1 (or 101 if the query contains a proximity operator). The parser uses a table to control the mapping of CCL index names to Z39.50 attribute combinations. This table specifies an attribute value combination for every index name recognized by the parser. Thus, for the Title Keyword index, ("TW") the table specifies the following attribute values: Use = 4, Position = 3, Completeness = 1. Values of other attributes are shown as 0, meaning that the parser should calculate correct values to send on the basis of the query term. If the term contains multiple words, the parser will set the Structure attribute to 1, for "phrase"; if the term contains a final "?", the parser will set the Truncation attribute to 1 for "right truncated", etc. A calculation is not specified for every case; if there is no calculation, the query omits the attribute type completely, allowing the server to use its default value for that attribute. If the input term contains no explicit truncation mark, for instance, no truncation attribute will be contained in the Z39.50 query for that term and the default value of the server will be used. For the AMICUS server the default value for truncation is 100, "do not truncate".

The index table is maintained as an Ingres table, from which configuration files are extracted for use by the clients. For speed at startup, the index table is maintained as two files that are read when the user interface is initialized. The clients implement a paradigm of *indexes* and *limiters*, with "indexes" referring to those searchable elements for which physical indexes are maintained, and "limiters" referring to those elements that are not indexed in the database and are intended to be used only to modify sets created by a search of a primary index (although a few users have permission to search directly on limiters, as described above). There is an index name corresponding to almost every distinct element in the AMICUS data model.

Query strings of almost unlimited complexity can be built through the use of parentheses in the CCL query string, with the limitation that a single query may not exceed 1000 characters and a single term may not exceed 200 characters. Multiple index names can be applied to a single term (e.g. "tw sw nuclear physics"), which the parser interprets as a Boolean OR of the single term applied to the each of the index names ("tw nuclear physics OR sw nuclear physics").

Proximity searching is fully supported by the clients, using the CCL "within" operator for ordered proximity and the CCL "near" operator for unordered proximity. Searchers may use the client's default value for distance, may change the default for a session or may supply a specific value on a case-by-case basis. As mentioned earlier, only the "character" unit is supported for distance.

The Windows client is designed to support searching of both the AMICUS databases and databases at remote servers via a Z39.50 client gateway on the AMICUS server. To support this, the user can choose which system to connect to, and which database at the system to search. The index tables support this model by maintaining different attribute combinations for each index name depending on the system and the database being searched. Only searching of the AMICUS databases will be supported for the initial release of the AMICUS clients, however.

The Windows Interface

The Windows interface permits the searcher to construct queries either by typing in a CCL query string directly or by choosing the various elements of the query from selection lists. It is expected that frequent users of the interface will normally prefer to enter their queries directly, while occasional users will prefer the point-and-click approach. When the searcher uses a selection list to choose an index that requires system-controlled term values, such as a language code or physical description code, a list of the allowed values is presented to the searcher. Selecting a value inserts the index name and value in the query. The searcher can type in operator names directly or click on buttons to select them.

Every AMICUS user has a default database name to which all searches are applied. The searcher can select from the list of AMICUS databases a different database to search. This selection applies for the remainder of the session, or until changed again.

To simplify the input of complex repeated queries, a user can save all or portions of queries for subsequent insertion into a query string. The searcher selects from a list of these "Search qualifiers" and the user interface includes functions that allow a user to manage his or her personal list.

The interface maintains a log of searches that have been performed in the current session. The searcher can examine the log; display records from any previous search; select a previous result set to use as an operand in a subsequent search. The 25 most recent queries are maintained by the log, and the searcher can delete individual queries from the log as desired.

The Access AMICUS Interface

The Access AMICUS interface presents a host-based command-oriented search interface to the National Library's Search Service subscribers across Canada. This interface is deliberately designed to resemble the DOBIS search interface, to minimize the amount of retraining required by the 500-plus Search Service users. The query language is uses the CCL Find and Scan commands, with the exception that single-character commands, "F" for "find" and "S" for "scan", are required, as opposed to the full name and three-character acronyms specified by CCL.

Users of Access AMICUS enter CCL commands to scan indexes and search the AMICUS databases. The interface also allows users to download CanMARC records for cataloguing purposes, notify the National Library of local holdings and make ILL requests for items in the National Library's and CISTI's collections.

The interface does not support a number of the features of the Windows interface: search logs are not maintained; search qualifiers are not available; pop-up lists of term values are limited to only the most commonly used indexes and have limited sets of values. The interface is not currently designed to allow searching of systems other than AMICUS.

The ISAAC Interface

The ISAAC interface is designed for use by untrained end-users of the CISTI and National Library collections. The interface is implemented as a set of menus that lead to search templates, with each template presenting the user an input form that, when completed, specifies a search to be performed. There are templates for performing a "quick search" (name, title and/or distinguishing number such as ISBN or AMICUS number); for searching for monographs, serials, and a number of special types of material such as technical reports, theses, newspapers, audio-visual material, music; and for doing subject searching. Each of these templates uses a dialogue tailored for the specific type of material identified by the template. The interface uses the template and data supplied by the searcher to construct a CCL query string which is passed to the client query parser for generation of a Z39.50 SearchRequest.

When records have been found the user can initiate a dialogue to narrow the search, broaden the search or to "find more like this".

ISAAC offers access to only the National Library collections, CISTI collections and Union Catalogue databases. Only a limited subset of AMICUS indexes is used.

Since the National Library and CISTI are both closed-stack libraries, ISAAC allows the user to issue a request for delivery of an item to a reading room or (for CISTI staff) to an office; it allows the user to issue ILL and document delivery requests to CISTI or the National Library.

Conclusion

This paper has described how Z39.50 provides the core functionality of the AMICUS bibliographic search engine. The server has been designed to provide maximum efficiency in searching and offers sub-second responses in building result sets for common queries. It is designed to be highly flexible and can be adapted to different data models with relative ease. The underlying database technology can be changed without requiring major alterations to the engine, and the system can be ported to different hardware/software platforms with little difficulty.

Z39.50 is used to provide transparent search access to heterogeneous database systems and also to provide access to the National Library of Canada's core information to the widest possible range of clients. AMICUS demonstrates that Z39.50 can successfully be used to provide a public search interface to SQL databases.

One positive effect of the client-server technology used has been the feasibility of creating multiple user interfaces for different purposes, as shown by the three interfaces created for AMICUS.

AMICUS is a significant new bibliographic system, designed from the ground up to meet the needs of two large research institutions, one of which has the additional requirements of a national library.

Developing a Multi-Platform Z39.50 Service

Terry Sullivan and Mark Hinnebusch
The Florida Center for Library Automation
Gainesville, Florida

Abstract

It is possible to develop a Z39.50 service that is independent of the underlying computer hardware and operating system and that will interoperate using many different transport mechanisms, such as TCP/IP, OSI, SNA, and Named Pipes. This article discusses the design and implementation of one such service, developed at the Florida Center for Library Automation, with special regard to independence issues.

A System Model for Independence

Z39.50 has been crafted with a single overriding goal, which is to provide a "lingua franca" for search and retrieval between disparate systems, usually geographically dispersed and provided by different vendors. The primary function of the Z39.50 Implementors' Group (ZIG) has been the crafting, through intense negotiation among implementors, of mechanisms that generalize the services provided by, or envisioned by, various system vendors, with interoperability between these various systems as the principal goal.

It has been clear to a number of implementors that Z39.50 not only provides a mechanism for interoperability with others' systems but also offers a powerful model for the distribution of services within a single system or system complex, or between various products offered by the same vendor.

The Florida Center for Library Automation was an early implementor of Z39.50. At the time we began work on Z39.50, we had little reason to believe that we understood how the protocol and the software that we were developing would be used. This meant, in turn, that we could make no assumptions about the underlying hardware and operating system services upon which we would need to build our Z39.50 software.

We posited a distributed model in which Z39.50 agents could communicate with corresponding agents in remote systems as well as with those located in the same system. This is best shown by a diagram. In Figure 1, the relationship of the Z39.50 service providers and service users is diagrammed. Service providers are programs that implement the Z39.50 protocol. Service users are system components that utilize the services of the service providers. In this figure, all of the four elements reside within a single system, but there is no requirement that this be so. Each could be in a separate system, or any two or more could be co-resident.

A traditional database server/requester system, implemented monolithically, can be viewed as the origin and target service users, as in Figure 2.

But suppose you want to serve multiple service users. You can replicate the Z39.50 origin and/or target service providers and tightly couple them to the various service users. Alternatively, you can have a single Z39.50 service provider with the intelligence to support multiple service users. Or, you can interconnect multiple service providers and service users in a distributed network, as shown in Figure 3.

In such a configuration it is imperative that the Z39.50 software be available on a variety of platforms so that the best machine can be used for each particular situation.

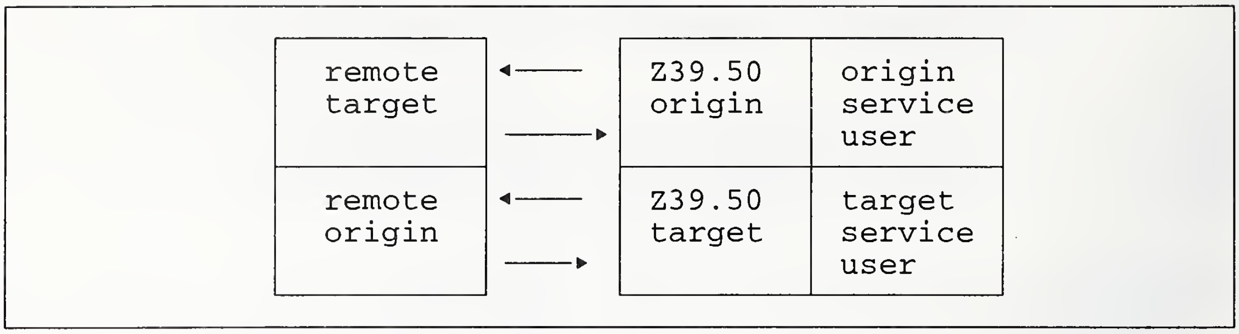


Figure 1

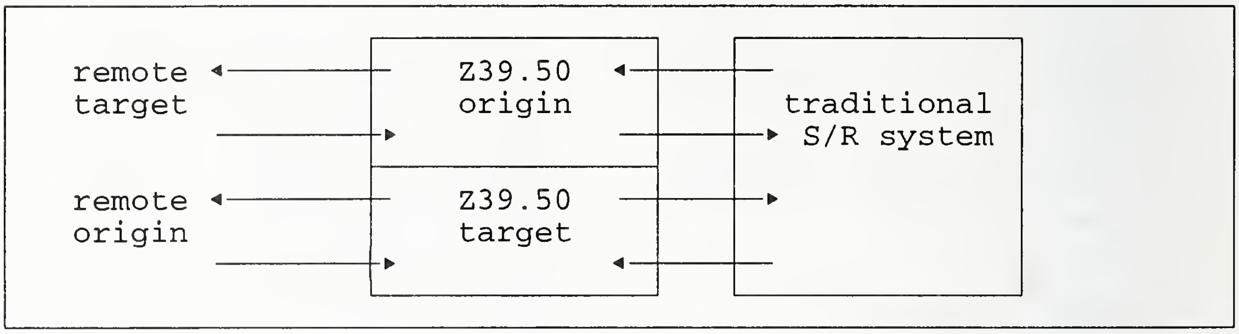


Figure 2

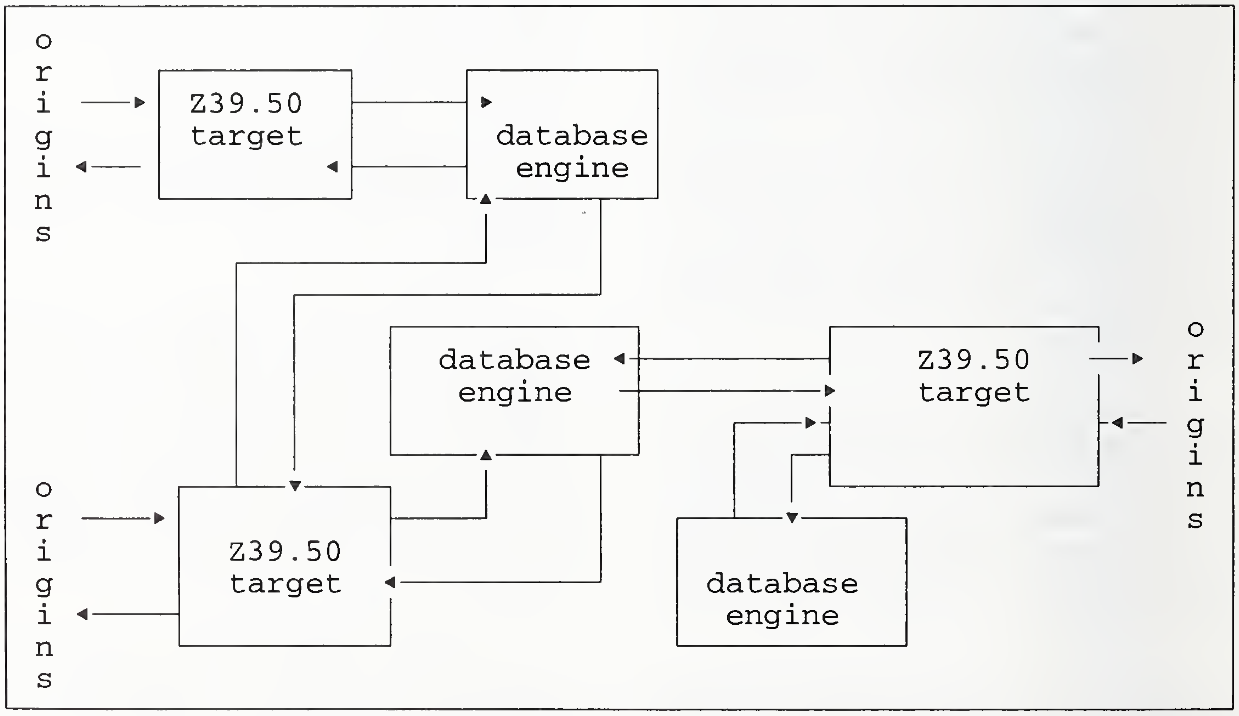


Figure 3

When FCLA first became active with Z39.50, there were strong indications that OSI would be mandated for use in federally acquired systems. The US GOSIP was being promulgated and the State of Florida had followed suit, producing a Florida GOSIP, aligned with the US GOSIP. Because of this, our original plans were based on OSI. However, we discovered, as the ZIG was formed, that other implementors were more interested in offering Z39.50 over the more popular and more widely implemented TCP/IP protocols. FCLA shifted its focus to developing a combined OSI/TCP model. Over time, we have recognized the value of running Z39.50 over LANs, hence the need for NetBios and Named Pipe support.

FCLA developed Z39.50 code capable of supporting the model shown in Figure 3. The code is named AccessFlorida, and is running on an IBM 3090 mainframe as a separate VTAM application under the MVS operating system, on an IBM RS/6000 running AIX 3.2, and on an IBM 486 DX class PC running OS/2. We are in the process of moving the AIX version to a new IBM SP2 PowerParallel multiprocessor and we know of at least one instance of porting AccessFlorida to Microsoft NT.

While AccessFlorida is designed to provide Z39.50 origin and target services, its design allows for its use with many different connection oriented protocols. This feature will be exploited in the future. We are currently providing a prototype Z39.50/HTML gateway using AccessFlorida, and we have long term plans to support X12 (business transactions) messaging using this versatile code base.

The remainder of this paper discusses the design of AccessFlorida as it pertains to independence from hardware and operating systems.

Figure 4 represents the conceptual layering of AccessFlorida onto each system upon which it is implemented. This layering helps to identify and isolate system dependent and network dependent fragments within AccessFlorida. Each piece of the diagram was examined and this analysis affected the design at the very start of the project. The next two sections detail the research that went into the review of the different portions of the diagram, the outcome of this research, and its effect on both the design and implementation of AccessFlorida.

Operating System Independence

One of the primary design goals of AccessFlorida is to isolate the system from all dependencies based on a particular operating system and network interface. This requirement is met in two ways. The first is in choosing a programming language that is supported by all operating systems we plan to use. The second is to isolate all system calls in separate modules and add a thin layer of code to reduce the dependencies on these calls.

In considering the programming language, we examined each targeted platform to determine which languages were supported. On each platform, the network interface was also taken into consideration, since the API presented by these interfaces would also have an influence on the choice of language. The outcome of this research was the selection of C, using the IBM supplied compilers for each platform. On MVS the compiler is C/370, on OS2 the compiler was originally IBM C/2 and later the C Set++ compiler, and on AIX the native C compiler is used.

Even in choosing a common language and using compilers supplied by the same company, care is needed during design and implementation. Each platform has specific requirements that bind both compiler and linker. For example, on MVS, all external names are limited to eight characters; on MVS and OS2 (using the FAT file system), file names are limited to eight characters; and on AIX, file names are case sensitive. Each implementation of the compiler has extensions that are not supported on all platforms. Furthermore, MVS and the other platforms have different file systems.

These restrictions resulted in four design decisions. The first is to limit all filename references to eight lower case characters.

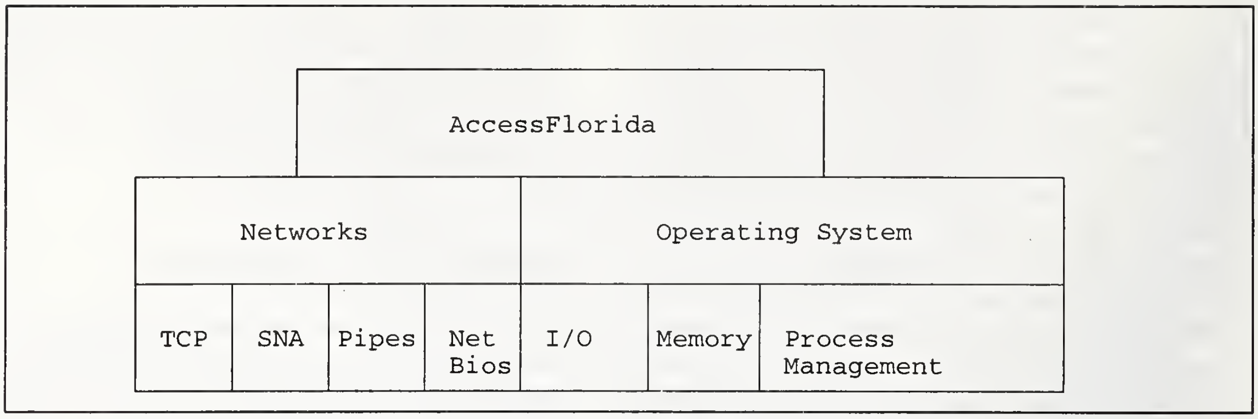


Figure 4

The second is to include a header file in each software module that isolates all system dependent requirements. This header file is tailored to each operating system. It initially redefined every external function name to eight characters for the MVS system; later it was used to easily switch the system dependent header files, again by redefining these names; and finally, it is used to redefine system wide parameters, such as buffer sizes and default parameters. The third design decision is that AccessFlorida will not make any assumptions concerning the location and names of the files it uses for processing. We decided that this will be controlled by setting up the appropriate environment on each system. The fourth decision is to completely base the code on ANSI C and to not use any extensions supplied by the various compilers.

Another area of concern is how each operating system handles process management. On AIX, a process can only spawn duplicate images of itself; on OS2, a process can be multithreaded or spawn copies of itself, and on MVS one process can create a new process, using ATTACH, which can be an independent process or a thread. Each of these methods, although similar in nature, produces different effects on a software product and adds operating system overhead in the management of these newly created resources. Since the main resources that AccessFlorida manages are the Z39.50 connections between remote clients and remote servers, we decided to design a synchronous system that maintains it own resources in an identical fashion on all platforms. This leads to another design decision to not multithread or spawn, but rather to enhance the management of resources in AccessFlorida in such a way as to provide an efficient scheduler where all resources are given appropriate time to complete processing.

In effect, AccessFlorida becomes a dispatcher of its own threads and resources. The management of these resources and their execution path is implemented in the concept of the Connection Description Block (CDB), and the execution of a state machine. The CDB contains all of the information that must be maintained during a context switch, and it contains only the necessary information. Because of this, internal context switching is far more efficient than operating system context switching as implemented through a process or a thread switch, where the process or thread contexts must be maintained by the operating system.

The last major area of concern lay in the management of memory on the various platforms. The C language enables AccessFlorida to use memory in a consistent manner regardless of operating system. However, since the design required that AccessFlorida handle its resources in a highly efficient manner, we decided that to improve performance, once memory is obtained by AccessFlorida it remains under its control. In short, the software manages its own heap and only calls the operating system when it needs to expand its heap size. This management of memory is further refined into two categories, buffer management and dynamic management.

Buffer management is responsible for obtaining memory for the buffers that are used to send and receive data over the communications networks. The management of these buffers consists of obtaining memory in relatively large chunks and ensuring that this memory remains persistent for as long as the buffer remains active, that is, contains valid data. Each buffer is indirectly connected to a CDB and has enough side information to identify the amount of valid data that it contains, the network that it is associated with, and the connection within that network that is using it. Two identical types of buffers are used: receive buffers and send buffers. These buffers are implemented in a separate module where they are managed independently of the rest of the software. This module's API presents functions necessary to retrieve individual buffers, to clear the memory associated with the buffers, and to provide the ability to reuse every buffer.

Dynamic memory management is used during the encoding and decoding process. As with the use of buffers, this processing has to proceed in an efficient manner. Unlike buffer management, where large chunks of memory are used all at once, this dynamic memory manager is responsible for retrieving relatively small chunks of memory from a larger memory pool. Again, the goal is to reuse memory once allocated, thereby reducing calls to the operating system. The management of this memory resides in a separate software module and is independent of the rest of AccessFlorida. This module allocates a large chunk of memory once, during initialization, and reuses this memory throughout the execution of AccessFlorida. Dynamic memory management is highly efficient and aids greatly in the decoding and encoding of data from and to the network buffers.

Network Independence

Another major area of concern in the design of AccessFlorida deals with the various APIs presented by the targeted networks, and the implementation of these APIs on the various platforms. From the beginning, the design provided for a layered approach that helped to reduce the cost of supporting multiple networks. This layering is conceptually represented in Figure 4, where each segment of the network layer is translated into software modules.

AccessFlorida accesses each network through one set of functions. Access to these functions depends on whether information is being received or sent. The internal API to connect and send data over the networks is implemented in the connection manager, while the functions to receive and accept connections are implemented as part of the main processing of the control program. All of these calls are used at a layer above the network layer to isolate the network specific dependencies from the main processing in AccessFlorida. The overall operation of this interface is depicted in Figure 5.

The actual network interface is implemented in separate software modules, all exposing the same external API to the rest of AccessFlorida. There are currently three such modules in AccessFlorida, one for the TCP/IP network, one for the SNA network, and another for transmission of data via Named Pipes. The OSI support in AccessFlorida is now obsolete and would require some modification to be re-enabled. The network interface modules are used to present the same API regardless of operating system. The implementation of the API is accomplished through these software modules, plus one additional module that actually performs the specific call to the network API. This isolation has greatly helped in the porting of AccessFlorida from the original MVS implementation to both the AIX and OS2 operating systems by isolating the specific implementations in lower layered software modules. The original networks used in AccessFlorida were a pure OSI stack, as provided by the IBM OSI/CS product, and IBM's SNA network. Both of these networks are message driven, delivering information in well defined packets or buffers.
These message based protocols fit neatly with the bit stream generated by the BER encoding of

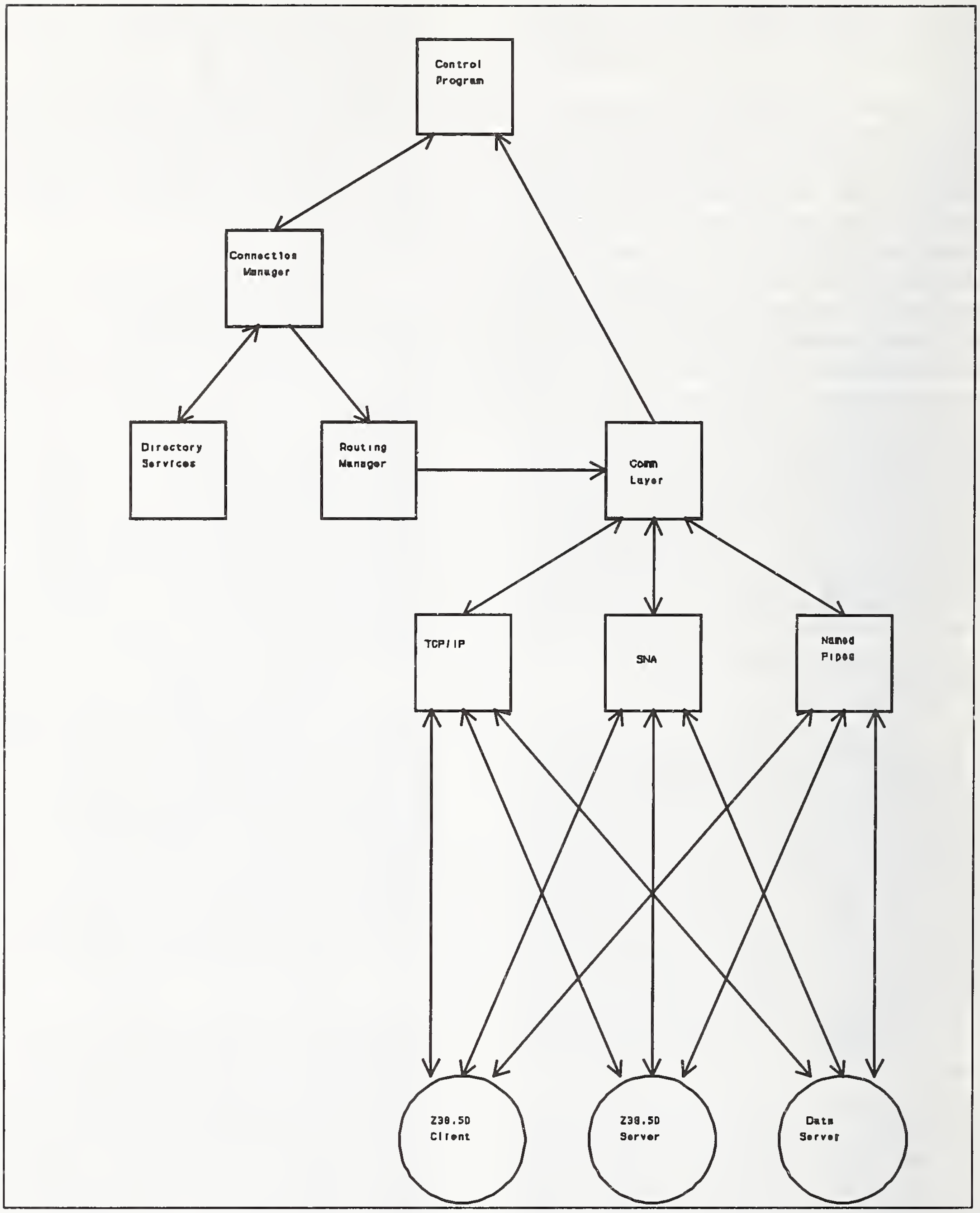


Figure 5

the ASN.1 of Z39.50. This encoding generates well defined delineations within the bit stream sent through the networks. When we dropped OSI/CS, we migrated our BER support to SNA-CC, originally developed by Michael Sample at the University of British Columbia. With the introduction of the SNACC generated code from the ASN.1 of Z39.50 Version 2, and the SNACC runtime libraries, these message driven networks worked fine, since one entire APDU could be received or sent with one call to the network. With the introduction of TCP/IP, which uses a stream based protocol, a layer is required to provide this packetized functionality over the stream-based socket interface of TCP/IP. When remote access was provided by using Named Pipes, a message based scheme was chosen to retain the sending and receiving of complete APDUs.

OSI

The IBM OSI/CS implementation had problems at the Presentation layer. To circumvent these problems, access to this network was accomplished at the session layer. The Presentation layer was then replaced with code generated by the public domain product, SNACC. This was the implementation that AccessFlorida first used when it was released into the public domain as a production system. There was still a connectivity problem with the OSI network. Connectivity is one of the first requirements to interoperate in an open environment. In the United States, OSI support usually requires X.25 as the lowest three layers. While there are several X.25 networks, we were unable to connect with other Z39.50 OSI implementations because there were no adequate bridges between the various X.25 networks and there was no other OSI based Z39.50 service in our X.25 domain. This lack of connectivity led to the introduction of TCP/IP into the network layer used by AccessFlorida.

The OSI network was never ported to the other operating system platforms; its implementation remains solely on the MVS system. However, a by-product of AccessFlorida is the implementation of Tosi, a thin implementation of the upper layers of OSI, that is also transport independent, that has been placed in the public domain.

SNA

AccessFlorida must be able to communicate via SNA since our primary database server is implemented on the IBM CICS/MVS platform, SNA support is excellent in CICS, and other communication protocols were not supported in CICS when we began this project. The implementation of the SNA network is accomplished by designating AccessFlorida as an LU6.2 application, capable of managing its own communication and resources on an SNA network. The specific protocol used for communication is Advanced Peer to Peer Communication, more commonly known as APPC. LU6.2 is an IBM proprietary protocol represented by two publicly available APIs: CPI/C and APPC. APPC is a message based communication protocol, sending data in easily identified packets over a highly delineated and secure communication line. One LU6.2 application can open several sessions, or links, with one or more LU6.2 applications. Sessions may be parallel, i.e., capable of running simultaneous conversations between the applications. In IBM terminology a session is established between the two LU6.2 applications and then conversations are established between specific code fragments, or transactions, of the two applications. Thus the applications are linked at two levels with security provided at each level. Communication is actually at the conversation level and in most instances is two way (i.e., duplex) although one way conversations are also common.

APPC is a highly reliable and secure form of communication, but unfortunately specific implementations of APPC differ greatly. On MVS, access to APPC is accomplished through an assembler interface that is linked into the rest of AccessFlorida. The interface on the other platforms is accomplished through C function calls, but still differ enough that each platform requires its own specific APPC module. These system dependencies are isolated by providing an internal API for this network that is used by the rest of AccessFlorida. All that was required was the rewrite of one module to interface to the specific implementation of APPC on each operating system.

As stated earlier, each pair of LU6.2 applications taking part in APPC communication is required to open one or more sessions between them. AccessFlorida applies a special semantic meaning to establishment of these sessions. Once a session is created between AccessFlorida and another LU6.2 application, the two applications are said to be in a connected state with respect to the network, and a closed state with respect to Z39.50. In OSI terminology, the creation and establishment of these sessions is the same as establishing both a presentation connection and an application association.

Conversations are used by AccessFlorida to send APDUs to the partner application by special convention. Each conversation sends or receives data only one time. Thus a conversation is allocated, a PDU sent (or received) over the conversation, and then the conversation is deallocated. There is never any direct notification that a message is received; only indirectly, by the later arrival of a reply, does the sender know that the message was processed at the destination. In order to associate one message with another within the sessions connecting two Z39.50 applications, the reference id is used. Each Z39.50 application protocol data unit contains a reference id. The Z39.50 standard specifies that this field will be echoed by the target if sent by the origin and supplies no additional meaning to this field. With version 3 of the standard and the introduction of concurrent operations, the reference id is used to give the origin the ability to interleave operations within a connection. This is identical to the mechanism used in AccessFlorida to implement Z39.50 over APPC.

TCP/IP

We added the TCP/IP network to AccessFlorida after the OSI and APPC networks were implemented. TCP/IP gives AccessFlorida the greatest connectivity with other Z39.50 applications. The TCP/IP socket implementation and API are straightforward and fit neatly into the internal network API of AccessFlorida. In connecting AccessFlorida to this network, three aspects of the socket protocol were considered.

The first is the fact that a socket connection within TCP/IP is stream-based, while AccessFlorida expects to send and receive information in chunks, one protocol data unit at a time. To accommodate this difference, two approaches were examined. In the first approach, the PDU is decoded or encoded while the PDU is being received or sent over a connected socket. This approach is more commonly referred to as decoding/encoding directly over the socket. The second approach entails decoding only enough information to guarantee that an entire PDU is received, and encoding only enough information to guarantee that an entire PDU is sent. Each approach has benefits. The first might be faster since data would be moved to and from the "line" quicker. The second might provide better troubleshooting and migration paths since data is received and sent in logical messages specified by the application. Since AccessFlorida expects its data to be delivered in complete PDUs, the second approach was adopted.

In adopting this second approach, the C function to read information from the socket was constructed to determine the amount of data needed to complete the PDU by examining the tag and length specification of the encoding. When BER is applied to an ASN.1 specification, the resulting encoding is said to consist of a Tag, Length, and Value; this is commonly called a TLV encoding. For tags that are defined to be constructed, the length of the constructed value may be specified either directly (the definite form) or indirectly (the indefinite form). When the indefinite form is used the value terminates when two null octets (known as the end-of-content octets) are encountered. Whenever a constructed tag is encountered, the value for this encoding consists of at least one more TLV encoding.

With this information, the socket reading function performs a peek on the socket to look at the first ten bytes. It then examines the tag value to calculate the length of the tag. If two null octets are encountered it decrements a counter for the end-of-contents octets and sets the number of octets to be read to two, otherwise it sets the number of octets to be read to the length of the tag. If no end-of-contents octets are encountered, the function then examines the length value. If a definite length is encountered, it sets the number of octets to read to the number of bytes of the length field plus the number specified by the length field. If an indefinite length is encountered, it increments the number of end-of-contents octets to be read and sets the number of bytes to be read to one since an indirect length takes only one byte. After both the tag and length fields are examined, the function then reads the number of bytes calculated. This logic is repeated until the entire PDU is read.

The second consideration dealt with the mechanism AccessFlorida uses when accessing the TCP/IP network. Recall that AccessFlorida operates as its own dispatcher; no threading or spawning is involved in its processing. Since AccessFlorida needs to accommodate multiple connections, the TCP/IP network is accessed in an asychronous mode. Thus, complete PDUs might not be sent or received entirely in one call to the network. In that case, rather than blocking on the socket, AccessFlorida saves enough information to complete the operation later. When data is being received and a PDU has not been completed, the amount of data received, the number of end-of-contents octets to read, and the number of bytes to read to complete a TLV encoding are saved and the process of receiving data is terminated temporarily. The process later resumes tests to see if there is any data on any active socket ready to be received. If, after processing the other active sockets, this socket has more data, the process of reading the PDU is resumed. A similar mechanism is used when sending data.

Since AccessFlorida may be servicing many connections, care is taken to give every active connection an equal opportunity to be processed. This is accomplished by keeping track of both the active sockets and those sockets having data to be read or sent. AccessFlorida uses two TCP/IP supplied structures and a counter for this purpose. The structures are the TCP/IP fd_set structures, which are used to test the availability of sockets for reading and writing. The first fd_set structure is populated with the appropriate socket once a socket connection is made. The socket is only removed after it has been disconnected. The second structure is populated with active sockets that have data to be read or are ready for writing. A counter is initialized to the number of sockets in this second structure. Once the second structure has been populated, the first available socket is removed, the counter is decremented, and the socket is processed. All sockets in this second structure are processed before it is reset using the first structure. Using this method, all sockets that are ready for processing are guaranteed to be processed before any socket is reprocessed.

The third concern dealt with detection of a closed socket. A closed socket is used in version 2 of Z39.50 to abort or gracefully close a connection. AccessFlorida not only needs to detect when a socket is closed, but also to translate the close into either an abort or simple close. The detection of a closed socket is handled by the return code from any TCP/IP system function. Depending on which function is called, AccessFlorida translates the TCP/IP error into either an ABORT or a RELEASE RECEIVED.

Named Pipes

The last network layer to be added to AccessFlorida was using Named Pipes. This implementation is similar to the SNA network implementation. The addition of this network enables database servers on local area networks to be accessed via Z39.50.

AccessFlorida Components

The various components making up the AccessFlorida system are classified according to the tasks they perform. Each component is briefly described in this section.

The Initialization component consists of the C functions necessary to establish the operational environment and call the communications protocol interfaces to establish the various network environments. This component calls each of the software managers via their respective initialization functions thereby completing the initialization of AccessFlorida.

The Control Program is the primary control process and the heart of the system. It is essentially an infinite loop which calls the network components as needed to perform the main work of the system.

The OSI component is a set of programs that call the necessary OSI/CS subroutines to initialize the OSI environment, accept new origin associations, initiate new target associations, listen for and receive APDUs from end systems, send APDUs to end systems, terminate associations, and terminate the OSI environment.

The TCP/IP component is a set of programs that call the necessary subroutines to initialize the TCP environment, accept new origin associations, initiate new target associations, listen for and receive APDUs from end systems, send APDUs to end systems, terminate associations, and terminate the TCP environment.

The APPC component is a set of programs that call the necessary APPC subroutines to initialize the APPC environment, build new logical associations, listen for and receive messages from end systems, send messages to end systems, terminate logical associations, and terminate the APPC environment. Some APPC services are available from the underlying operating system or there may be basic APPC support subroutines as part of the interface if the underlying system provides only partial APPC support.

The connection manager is implemented in several modules and operates on a connection description block (CDB). It is responsible for keeping track of each connection, the state of the connection, the partners involved in the connection, the messages received and sent on the connection, and the networks involved in the connection.

The buffer manager is responsible for managing all of the receive and send buffer structures in AccessFlorida. It consists of C functions to create and initialize the buffers, to identify and return the buffers, and to release or free any memory associated with the buffers.

The state machine verifies that incoming Z39.50 APDUs and APPC messages are valid for the state of the association, modifies the state to reflect the incoming APDU or message, and defines the actions to be taken in response to the incoming APDU or message given the state of the association. The state machine is implemented in its own module and uses the functions that are made available through the CDB manager.

The Z39.50 protocol manager generates Z39.50 APDUs to send to end systems, using information maintained in memory and associated with the CDB. It also sets information in memory based on the content of APDUs received from end systems. The Z39.50 protocol manager is the only component of the AccessFlorida system which must be aware of the structure of Z39.50 APDUs.

The protocol identifier module identifies the appropriate protocol and is only aware of the structure of the protocols needed to uniquely identify the message and discover which CDB it is associated with. This allows AccessFlorida to operate ultimately as a multiprotocol gateway.

AccessFlorida encodes and decodes all messages through one module that interacts with the SNACC generated decode/encode functions and the SNACC runtime library. The translator converts Z39.50 PDU contents to an internal form and attaches the resulting structures to the CDB. The translator is logically imbedded in the modules used for encoding and decoding messages and is a logical construct. The use of the translator makes integration of new protocols easier and less confusing. By mapping all messages to an internal structure, the problem of supporting more than two protocols is overcome.

The attribute mapper performs Z39.50 attribute mapping to enhance the interoperability of the two connected systems. It operates on the internally defined structure representing a search request.

The routing manager ascertains the target system, communications protocol, and application protocol for a database specified by an origin in a Z39.50 SEARCH APDU. The routing manager logically associates the database with the target and the protocols needed to reach that target.

To identify the target system the routing manager uses a database directory. The routing manager contains the necessary functions to ascertain the target which owns the database being requested and returns both the application protocol and communication protocol used by this target.

The function of identifying the application protocol being received by a remote origin is part of the protocol identifier. The communication protocol is handled by the routing manager which does the actual routing of the messages to the appropriate communication network.

Termination routines perform the housekeeping functions necessary to cleanly terminate processing. They also call the communications protocol interfaces to terminate the OSI, TCP/IP and SNA APPC environments.

Program Logic Flow

The initial entry point of the AccessFlorida system is the main function. This program is started by the operating system. The first function call is to the initialization manager that calls the appropriate functions to initialize all control structures, the CDB chains, and the state machine. It then calls the functions to initialize the OSI, TCP/IP and APPC interfaces to establish the necessary communications environments. Once initialized, AccessFlorida sets a timer and enters a loop that terminates once the timer has expired or AccessFlorida encounters an unresolvable error condition. In this loop AccessFlorida processes all incoming messages on the networks. For each iteration of the loop, the support manager is called to generate statistics and other related information. Eventually the timer expires, and control passes to the termination routine which performs orderly shutdown.

The control program calls the OSI, TCP, APPC or Named Pipe interfaces to accept new associations and to receive Z39.50 APDUs or other messages.

When a message is received by the OSI, TCP/IP, APPC, or pipe interface, control is passed to the connection manager so that it can either initialize a new CDB or obtain the CDB associated with the association upon which the APDU is received. Control then passes to the protocol manager which identifies the application protocol. Using the information returned by the protocol manager, the actual CDB is identified, and the input for this CDB is updated. This CDB is then passed to the state machine which uses an action table to process the CDB and send the appropriate messages to either the remote origin or remote target. This sequence of events is similar for each network.

The State Machine, using the information in the CDB, validates the APDU and identifies the appropriate actions to be taken in response to the APDU. The State Machine extracts from its internal state table the identity of a subroutine to be called to handle the input, given the current states of the two sides of the conversation. The subroutine performs an action and sets a return code reflecting the results of that action. The State Machine uses the subroutine identifier, the current conversation states, and the return code from the subroutine to enter the action table to extract a new set of conversation states which it stores in the CDB. If the "continue flag" is on in the action table entry, the State Machine then reenters the state table to ascertain additional actions to be taken. The continue flag is used to force AccessFlorida to switch from one service provider to another, or to continue processing as the current service provider.

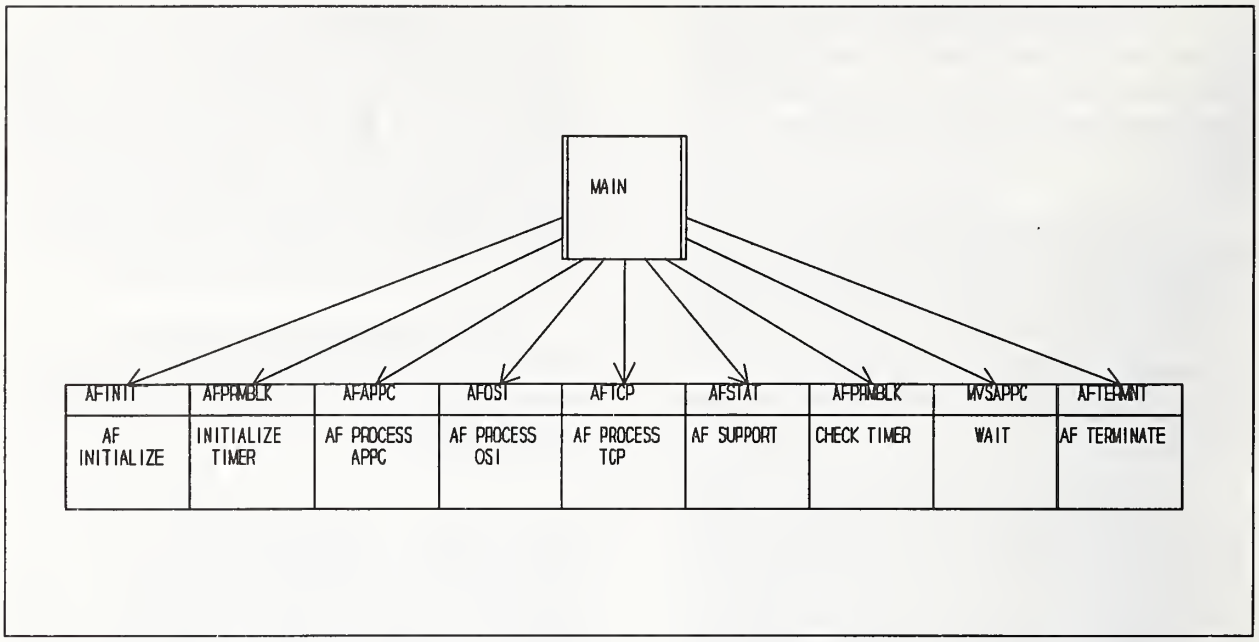


Figure 6

This process continues until the action table entry continue flag is off, at which time the State Machine returns to the calling function in one of the network modules.

If the input is a SEARCH APDU, one of the routines called will be the Query Translator which will perform the necessary translation, calling the attribute mapper to complete that task.

The Z39.50 protocol manager is called to create any Z39.50 APDUs that must be generated and the appropriate communications interface is called to send the generated APDUs.

Conclusion

FCLA expended a significant effort during the design and implementation of its Z39.50 software, AccessFlorida, to ensure future portability across hardware platforms, software bases and communications protocols. This effort has proven worthwhile by the ease with which AccessFlorida has been ported from its original platform, IBM's mainframe workhorse, the 3090 running MVS, to an RS/6000 running the AIX flavor of Unix, and to a PC running OS/2. Work has been done to support communications over TCP/IP, OSI, SNA, and Named Pipes.

The modular approach taken in the design and implementation of AccessFlorida ensures portability to future hardware and operating system options. We know others are interested in porting our software to the OS/400 operating system running on the IBM AS/400 family and we are aware of a working port to Windows NT.

AccessFlorida is designed as a generalized protocol manager and state machine, capable of providing support for, and translation between, multiple protocols, at both the transport and application layers. At the transport layer, AccessFlorida already gateways between TCP/IP, APPC, and Named Pipes, and OSI support could be re-established with minimal effort, should the need arise. At the application level, AccessFlorida currently translates between Z39.50 and internal search engine messages. We have a prototype Z39.50/HTML/HTTP gateway running and have long term plans to perform X12 transactions via AccessFlorida.

Use of Z39.50 for the Delivery of Current Awareness Products

Peter N. Ryall, LEXIS-NEXIS (email: peterr@lexis-nexis.com)

ABSTRACT

In the context of LEXIS-NEXIS, a Current Awareness Product (CAP) is an information product which provides concise, relevant data to a customer, related to a particular topic (or topics) of interest to that specific customer, or to customers within a broader market or industry group. This paper will discuss how Z39.50 is being used to deliver Current Awareness products to a variety of customers in a wide range of user environments. One essential objective of this delivery facility is to provide the necessary relevant information, proactively, to each customer in their own native environment, with minimal deviation from their standard methods for interacting with their groupware and/or individual workstation applications.

An assortment of Z39.50 services is used in this facility to provide a flexible delivery platform that supports such unconventional features as: information alerts, uploading record usage information for accounting and billing purposes, initiating subscriptions to specific Current Awareness products, establishing information filtering profiles, and providing access to (and delivery of) products consisting of 'webs' of pre-fabricated, related result sets.

Z39.50 Conformance Caveats

The application discussed in this paper was implemented and initially deployed in mid-1994, so that it was not able to take advantage of more recent Z39.50 Implementor's Group (ZIG) thinking and decisions with regards to particular Z39.50 services. In addition, some of the required features of the CAP implementation were driven by the need to accomplish all CAP access and delivery interactions within the same protocol. As Z39.50 was chosen as this single unified protocol, some compromises were required to accomplish this objective. The following is a brief statement of the intentional deviations from conformance to the Z39.50 Version 3 standard, along with a set of predictions as to how and when these deviations will be brought in line with the standard.

All of the non-conformant features have been implemented within private groupings of Origin and Target systems. They are not being recommended for use in public Z39.50 Target or Origin implementations, nor are any of them being proposed as enhancements to the Z39.50 standard at this time.

The first CAP feature that fits into this category is the use of Resource Control to deliver 'alerts' from the Target to the Origin system. The decision to use Resource Control in this non-standard way was strongly driven by the need to deliver all product functionality using the Z39.50 protocol. Although alerts could more naturally be sent using an offline delivery vehicle such as FAX or email, the use of Resource Control reports to convey alerts allowed us to provide 'real-time' reporting using long-running Z39.50 Associations. In future releases, delivery of these alerts will be migrated to other vehicles (FAX and email) in environments where real-time notification is not required.

The second non-conformant CAP feature is the use of a 'virtual Record 0' within a Result Set to provide access to set-level metadata. Record 0 is not defined by the Z39.50 standard as a valid record for retrieval, so this usage of Record 0 does not fit the model of "each record of the result set contains either a database record, a diagnostic record, or a surrogate diagnostic record". To bring this feature into conformance, a possible short-term (Version 3) solution would be to issue a search using the Result Set as the Operand, and requesting that the metadata be returned in additionalSearchInfo. A more flexible solution will be proposed for Version 4, possibly using Record 0 as defined here, or by defining a set model which includes set-level metadata as an extension to each record in the Result Set, thereby allowing it to be retrieved by simply Presenting set-level meta-elements from any record in the set.

The third non-conformant CAP feature is the usage of the implementationID, implementationName, and implementationVersion Init parameters to allow an Origin to connect to a specific set of back-end search and retrieval services. This is used mostly for testing and backwards compatibility purposes, and is not planned for use in public Z39.50 service implementations (as described in Section 4.0).

The final non-conformant CAP feature is the unconventional usage of Persistent and Transient Result Sets. Each of the resultSetIds conveyed in an Alert represents an identifier for a Transient Result Set (one which only endures for the lifetime of the Association). However, in this implementation of CGTI, the Target performs an automatic (and transparent) service for the Origin. It maintains a Persistent Result Set Task Package corresponding to each of the Transient Result Sets it reports to the Origin in an Alert.

Then, when the Origin issues a Present against the Result Set (using the Transient Result Set ID), the Target automatically asks the back-end retrieval service to fetch the essential elements of the Persistent Set and create a Transient Set. This Transient Set is created with a front-end mapping filter which allows it to be accessed via Present using the Persistent Result Set Package ID. Thus the Target and its associated back-end services present a 'virtual Persistent Result Set' image to the accessing Origin. Note: in subsequent Associations, the Origin still uses the same Transient Result Set ID to access records in the 'virtual Persistent Result Set'.

In a future release, the usage of Persistent and Transient Result Sets will be brought into line with the Z39.50 standard by requiring the Origin to first issue a Present using the Persistent Result Set Package ID, in order to obtain a Transient Result Set ID to be used within the Association.

It is also important to note here that, while the CAP facility does not require the use of the Z39.50 Search service, the Target which supports CAP functionality also supports search services. Thus while the Target is compliant with the Z39.50 requirement to support Search, the Search service is not used by an Origin in accessing and retrieving information from CAPs.

1.0 Overview of CGTI and Current Awareness Products

Traditionally, LEXIS-NEXIS has provided extensive search and retrieval information services using interfaces and formats defined in-house. After free-text searching across our large information databases, users would then determine which result sets are relevant to their search. LEXIS-NEXIS Current Awareness Products (CAPs) address an audience with broad or focused needs for information about specific industries and markets. CAPs are designed as a value-added product consisting of predefined sets of information organized by subject areas. External user environments access CAPs and other search and retrieval services through our Coarse Grain Transaction Interface (CGTI).

The "coarse grain" nature of CGTI reflects a more loosely-coupled interface which allows access to a broader, more flexible family of services than the traditional LEXIS-NEXIS interface. The CGTI is based on the ANSI/NISO (National Information Standards Organization) Z39.50 search and retrieval standard. The Z39.50 standard includes protocol specifications for search and retrieval of information stored in machine-readable databases.

1.1 Definitions and Description

This paper introduces and describes the CGTI, by which a broad range of Current Awareness Products (CAPs) may be accessed via an external system. It presents the set of message-based requests that may be constructed and sent to the Information Service (Z39.50 Target) by an external client (Z39.50 Origin). Each Origin request (and associated Target response) is described in detail, and examples are presented showing how each CAP capability is accessed.

The interface used to access CAPs (as well as other search and retrieval services) is based on Version 3 of the Z39.50 standard, which is specifically designed to meet the needs of client environments desiring access to information search and retrieval services such as those provided by LEXIS-NEXIS.

In this paper, the LEXIS-NEXIS implementation of the Z39.50 search and retrieval standard is referred to as the CGTI. Because CGTI is less tightly-coupled to the internal environment of the search and retrieval system, a more open and consistent interface is provided, allowing both the information provider (i.e., LEXIS-NEXIS) and external system and workstation developers to mature and enhance their systems independently of one another.

1.2 CGTI

The CGTI model consists of three distinct components:

- Third-party platform domain, consisting of the Z39.50 Origin and local user applications residing on a specific platform;
- Delivery domain, consisting of the lower layer communications protocol components;
- LEXIS-NEXIS domain, consisting of the Z39.50 Target and back-end information products (CAPs) and services.

The third-party platform domain consists of the Z39.50 Origin, a set of APIs, and local user applications residing on a user-specific platform. A set of environment-independent interfaces (APIs) allows local applications to initiate transfer activity through the CGTI to the Z39.50 Target. This transfer activity takes the form of an exchange of message requests and responses. The developer can write code using Z39.50 protocol data units (PDUs) or use a library of functions defined in specialized toolkits to communicate across the CGTI.

A single toolkit function may call one or more PDUs to execute multiple Z39.50 services. LEXIS-NEXIS provides a toolkit (called the "Origin Adapter Toolkit") for this purpose.

This flexible arrangement allows the end user access to CAPs from a variety of environments such as:

- Information desktop workstations/services such as Folio and Lotus Notes
- Message handling systems such as AT&T Personalink and other value-added distributor systems

- Mass market distributor systems like Prodigy
- Integrated corporate and third-party information systems

The delivery domain provides the logical and physical bridge that links the Target and the Origin. A physical connection must be made between the Origin and the remote Target system. This connection can be established over a variety of physical communication mechanisms, such as: leased lines, TCP/IP, or X.25. A logical connection, or Z39.50 association, must also be established between the Origin and Target. Messages from the Origin and Target are translated into Z39.50 requests/responses before transmission through the delivery domain.

Figure 1-1 illustrates the CGTI model.

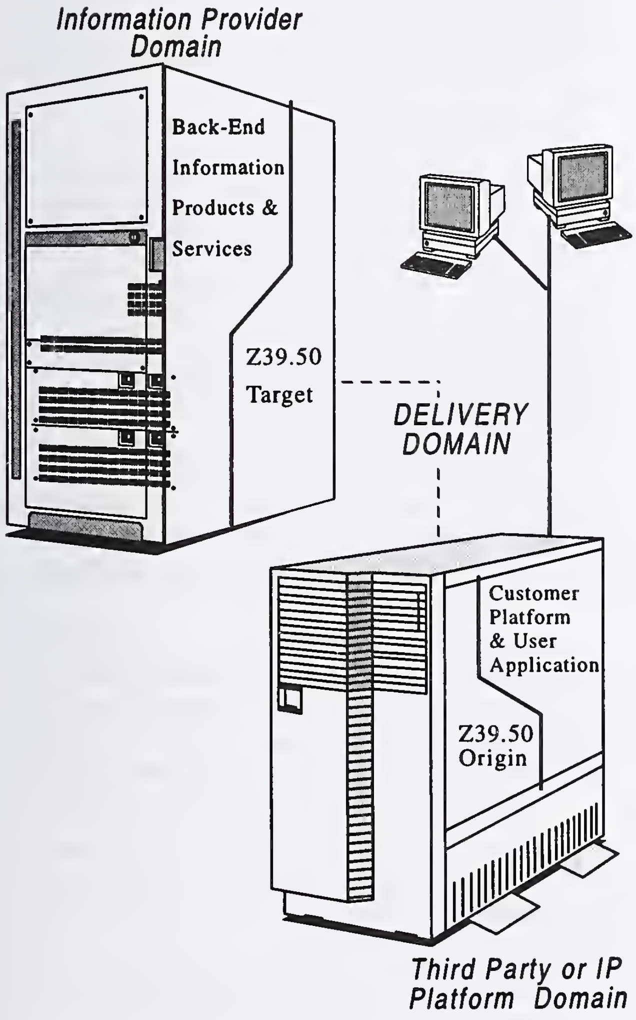

Information Provider Domain
Back-End Information Products & Services
Z39.50 Target
DELIVERY DOMAIN
Customer Platform & User Application
Z39.50 Origin
Third Party or IP Platform Domain

Figure 1-1 CGTI Model

The LEXIS-NEXIS domain consists of the CGTI Target and back-end information products and services. The CGTI Target translates:

- Incoming CGTI requests/responses into function/procedure calls;
- Function call responses/callbacks into the corresponding outgoing CGTI requests/responses

Some situations require that back-end services be called to execute request processing and/or data manipulation.

The CGTI "Origin Adapter Toolkit" provides the necessary functions to allow a local application to access a CAP without the complications of building an Origin supporting the CGTI Z39.50 profile (private to LEXIS-NEXIS).

1.3 Current Awareness Products Conceptual Model

The basic conceptual model of Current Awareness Products is as follows:

- LEXIS-NEXIS, through its advanced authoring facilities (combining human Subject Experts and automated processes), creates value-added relevancy-based information products which are designed for use in specific industries and markets, and for access by users who have either broad or focused needs for relevant information related to specific subject areas;

Note: These products are referred to as Current Awareness Products to distinguish them from standard search and retrieval services, which provide free-text searching across large subsets of the information warehouse but do not guarantee the return of highly-focused, relevant answer sets.

- LEXIS-NEXIS makes these products available via the CGTI, using a minimal subset of the Z39.50 standard requests to provide the following capabilities:
  - access to, and retrieval of, one or more directories of CAPs that are potentially available to end users attached to the client (Z39.50 Origin) system;

  Note: These directories may be presented in an 'active' way to the end user - e.g., as forms which allow the user to subscribe to an individual CAP; once subscribed, the user could receive new information (documents/articles) automatically whenever the CAP is updated.

The CGTI server will notify the CGTI client when new material is available; the degree of automation in obtaining results is determined by the client system implementation, and can vary from real-time alerts at the user's workstation, to notifications sent through a store/forward mechanism (e.g., E-Mail system), to requiring the user to check a shared folder or database where new results are placed as they are received.

In the LEXIS-NEXIS CAP implementation, one option is for the server to send notifications (alerts) to the client via a non-standard mechanism which uses the Z39.50 Resource Control request to convey the alert data. The specifics of these alerts and how they are encoded is described later in this paper.

- the ability for the CGTI Origin to establish a persistent (long-running) Alert Specification for a particular CAP. This Specification instructs the CAP Provider to notify the Origin system whenever the specified CAP is updated with new information;
- once an Alert Specification is established, and the Origin system is notified of new information in a topic/subject category, the Origin may then retrieve one or more of the documents in the updated category;
- the ability to access the directory (table of contents) of a CAP on an episodic basis; this involves retrieving the desired directory from the Target, presenting it to the user, and then allowing the user to select and navigate through the hierarchy of the directory;
- also on an episodic basis, the ability to request retrieval of one or more of the available forms of the documents/articles in a specific topic/subject category of the CAP directory (examples of possible forms are 'cite-list', 'preview', 'full-text', and 'full-text with graphic');
- the ability to access the descriptive and hierarchy (parent/child) information related to a specific topic/subject category.

• A future capability to be supported for access to CAPs is the capability to search through a particular product directory (or the master directory of all products accessible to an Origin system). The Origin will be allowed to issue a Z39.50 Search request that searches across a product (or all products) for a specified topic/subject of interest, or a set of terms. The Origin will be able to specify whether it wants the search to be:
  - restricted to the Topic names and descriptions of the various categories;
  - restricted to the abstract/preview portion of the documents in each topic category, or;
  - free to search across the entire full-text of the documents in each category.

1.4 CGTI User Model

There are two classes of users who gain access to LEXIS-NEXIS services via the CGTI: ***direct*** and ***indirect***. This classification of users is important to the model of CGTI CAP services outlined in this paper, so it is described in some detail below.

A ***direct*** (end) user is an individual user who accesses the system via a direct CGTI connection, either through a workstation which contains a Z39.50 Origin, or via a multi-user (server) system which executes the Origin within the server, but 'passes through' any information from the user dealing with ID, password, and/or token authentication, so that the user is still directly attached to the system. The attributes of a ***direct*** user are:

• data about the end user is contained in the customer information database; it is entered by an administrator at the time the user signs up for the service, and it is modified whenever changes are required;
• in order to access LEXIS-NEXIS services, a user must first go through an individual sign-on process and provide identifying information to the Authentication Service, which will return a ticket to the user's system, granting permission to access authorized services and products;
• records are logged for billable events initiated by the user, specifically identifying the user as the billable party;
• typically, invoices for charges incurred by the user for usage of services will be delivered directly to the user (or the user's firm with itemization by user);
• when users experience difficulty connecting to the system or using a particular service, they will contact Customer Services (i.e, the CAP

Provider supplies the first line of Customer support).

An ***indirect*** user, on the other hand, is one who accesses the system via some intermediate (or agent) system, which may or may not be owned, serviced, and/or managed by the Provider. The intermediate system is (logically) the ***direct*** user of the system, as it contains the knowledge of the user ID(s) and password(s) needed to access Provider services, and it is responsible for managing the CGTI interactions with the Target system. The customer's system must provide the access points to its end users, including local authentication and authorization, communications connectivity, customer support, customer sign-up and subscription services, individual user billing and invoicing services (generally via a charge back system of some type) and anything else which requires identification and tracking of individual end users and their activities. The unique attributes of an ***indirect*** versus a ***direct*** user are:

- the Information Provider (IP) maintains no persistent information about ***indirect*** end users - they are known only within the administrative domain of the customer firm; the IP also has no involvement in signing up or subscribing ***indirect*** users to specific services;
- a system within the customer's firm is responsible for 'signing on' to the Provider's service; the end user interacts directly only with his/her local system to login and enter a password (if necessary);
- events visible to the IP will be logged (for billable events initiated by any user attached to the customer system), but no data is logged specifically identifying the end user;
- typically, invoices for charges incurred by users for usage of Provider services will be delivered to the customer firm or the third party agent, who is responsible for any charge back billing or invoicing to individual end users within the firm, or under the administrative control of the agent;
- when users experience difficulty connecting to the system or using a particular service, they will contact their local administrator or help desk (i.e, the IP provides only the second line of Customer support).

For those CGTI services which require that the authenticity of the Origin system's identity needs to be verified, the IP will return a ticket to the Origin, which must be attached to the Origin's request in order to access the service (for both ***direct*** and ***indirect*** users).

Multiple User ID's may be permitted from a single Origin system; depending on the classification of the end user according to the descriptions above:

- for ***indirect*** users, the Origin system manages User IDs to grant different spans of authority to different users and/or local applications. The IP will follow a standard Authorization process for these IDs. As a part of this process, the Origin system will be given tickets for each of these IDs, which will permit access to different groups of Provider services and products.
- for ***direct*** users, the end user's workstation manages User IDs to allow multiple users to use the same workstation.

2.0 How CGTI Uses the Z39.50 Standard

The CGTI is one LEXIS-NEXIS implementation of the Z39.50 search and retrieval standard. At present, CAPs are delivered using only the following Z39.50 facilities of CGTI:

- Initialization
- Extended Services
- Resource Control
- Retrieval
- Termination

CGTI also supports the Z39.50 Search facility, but as it is not required for delivery of Current Awareness products, it is not discussed in this document. The following sections provide brief descriptions of the five facilities listed above, and tell how the CGTI utilizes these facilities to provide access to Current Awareness Products.

2.1 Initialization Facility

The CGTI Target supports the Z39.50 Initialization facility exactly as defined in Version 3 of the standard. The Origin sends an Init request to the CGTI Target, including setup information such as operations that should be supported, user authentication, and message size.

An "accept" result from the Init response indicates that the association is established and the Origin can proceed to access CAPs for which it is authorized. If the association is unsuccessful, the Origin can attempt another initialization.

2.2 Extended Services Facility

The Extended Services facility implemented in the CGTI system allows the Origin to:

- Set up a delivery notification mechanism (notifying the user of additions and updates to a CAP) by issuing an Alert Specification Extended Service;
- Send confirmation of billable information delivery to the user via a Final Delivery Notice Extended Service;
- Send a Usage Accounting Report Extended Service to the Target indicating CAP usage within the confines of the external delivery system provider;
- Send a Subscription Accounting Report Extended Service to the Target containing end user subscription requests.

2.3 Resource Control Facility

As mentioned earlier, Resource Control is used in a non-standard way by the CGTI, and as such it is only discussed here to provide completeness to the overall CAP delivery system description. In a future release, there is a plan to migrate this feature over so that it uses more standard Z39.50 facilities (e.g., Search and Present against a standard database containing 'product update' records, as well as discontinuing the use of Resource Control for delivery of alerts, in favor of email and FAX alert delivery).

The Resource Control request is issued by the CGTI Target to notify the Origin of the availability of new information topics or additional documents for a CAP. In response, the Origin sends a Resource Control response which indicates to the Target that the Origin is ready to receive another alert. The Origin can send a request to retrieve the information after the Resource Control response is sent to the Target.

In order to start this CAP notification process, the Origin must first establish an 'Alert Profile' by sending an 'Information Alert Specification' Extended Service request to the Target (see Section 5.1 "Creating and Sending an Alert Specification").

2.4 Retrieval Facility

The CGTI Target supports the Present and Segment services. As with any standard Target, these services define how result records appear when retrieved from the appropriate database.

Information retrieved from a CAP is contained in a result set maintained by the Target. The result set is a data structure with a pointer indicating where a record is located within the appropriate database; therefore, records are referenced by their position within the result set.

The Present service allows the Origin to request the retrieval of records from a specified result set. The Origin issues a Present request specifying a range of records that should be retrieved. The request message can also specify subsets of the records that define the "view" of what the user sees. These "views" include Cite, Preview, and Full.

Full database records may consist of document text or metadata. See Section 6.1 for an explanation of the various classes of metadata.

The CGTI Target supports Level 1 Segmentation, which allows large documents (records) to be broken down into manageable "fragments" for transfer to the Origin.

2.5 Termination Facility

In the CGTI system, the Close service operates according to the Version 3 standard, allowing either the Origin or Target to terminate a Z39.50 association. Reasons for termination include system problems, security violations, protocol errors, lack of activity, and completion of a 'user session'. The Close request terminates a single Z39.50 association between the Origin and Target. The recipient of the Close request responds with a Close response confirming the termination.

3.0 How to Access Current Awareness Products via Z39.50

To retrieve information from one or more Current Awareness Products, a Z39.50 Origin initiates a sequence of Z39.50 request/response exchanges with the Target. The following list defines the functions which must be implemented when developing a CGTI Origin:

- Establish an Association
- Identify new information
- Retrieve information
- Provide notification, usage, and subscription information
- Provide diagnostic and error message information
- Terminate an association

In order to start a Z39.50 Association, the Origin will first need to establish a physical connection with the Target system, and then send a Z39.50 Init request to establish an (application level) Association. Once an Association has been established, the Origin will typically set up a delivery notification mechanism for new CAP information. Given the fact that CAPs are regularly updated, a client will most likely want the Target to create an alert which notifies the Origin/client when new information is available.

Once the Target sends notification of new CAP information, the Origin can retrieve that information from the CAP Provider. Documents (records) can be retrieved at different hierarchial levels (topics/subtopics) and can be delivered in different "views" (e.g., cite lists, previews, full document text), by specifying different Element Set Names in the Z39.50 Present request.

After setting up retrieval options, the Origin should specify how it wants the Information Provider to track CAP usage and activity for billing and accounting purposes. In many cases, it will be desirable to allow the Origin to create and send notices/reports that provide notification, subscription, and usage information to the Target for processing.

4.0 Establishing a CGTI Association

Before attempting to access Current Awareness Products, an Origin must first establish a Z39.50 connection with a CGTI Target, just as it would with any other Target. This process consists of these steps:

- Making a physical connection between the Target and Origin environments using TCP/IP or X.25 (using either a leased line or a dial-up connection)
- Establishing a Z39.50 association between the Target and Origin

In order to conform to the CGTI model, the client must set the proper Init Option flags to indicate that it supports Present, Resource Control, Extended Services, and Close.

In order to distinguish among LEXIS-NEXIS Targets which support different functionality, the three Init parameters implementationId, implementationName, and implementationVersion may be used to specify a particular LEXIS-NEXIS Target implementation with which the Origin desires to interact. This is only used for testing out new implementations of Target functionality, such as a beta release of CAP capabilities.

For backwards compatibility, new values may be used to allow new CAP features to be provided to Origin systems wishing to take advantage of the new features. For Origin systems not wanting to implement or take advantage of new features, the Target continues to support the previous Target implementation, which the Origin indicates by specifying null (default) values in these three parameters.

5.0 Identifying New Information

Once the Origin has established an association with the Target, the next step is to set up an automated notification facility that alerts users of changes to the CAP(s). The effectiveness of a CAP is degraded if the product information is inaccurate and/or out-of-date. A CGTI product domain can update CAPs on a daily, weekly, or monthly basis.

CAPs may change due to the availability of new source information, the roll-out of new CAPs, and scheduled promotions. Since a key component of a product's value is determined by the CGTI server's ability to deliver that product in a timely manner, CGTI provides an automated notification capability via the Z39.50 interface.

This process of identifying new information consists of these steps:

- Creating and sending an Alert Specification to the Target using Extended Services;
- Receiving one or more alerts from the Target using Z39.50 Resource Reports.

The following sections describe the alert notification cycle and other related functions.

5.1 Creating and Sending an Information Alert Specification

The Present service allows product information records/documents to be retrieved via a synchronous request from the Origin. Product Alerts (usually delivered via Resource Control reports in the current release) provide an asynchronous notification mechanism to an Origin system.

An Origin system must send an Alert Specification ('profile') to initially activate delivery of Product Alerts from the CAP Provider system to the Origin system. The Alert Specification instructs an agent within the CAP Provider domain to search for new information in a specified subject area (or attributed to a particular CAP) and to notify the Origin whenever any relevant information is found.

The Origin specifies how alerts are to be managed by the Target by the setting of the Action parameter in the Alert Specification Extended Service. When this parameter is set to 'queued', the Target is instructed to send all queued alerts related to the specified product and topic immediately upon completion of the Extended Service exchange.

The functional flow of the Product Alert capability is as follows:

- First, the Origin sends an Extended Services request to establish an Alert Specification at the CGTI Target system,
  - as a part of the request, the Origin specifies a set of parameters, to be used by the Target in delivering notifications of new, relevant information:
    - The Alert Delivery Vehicle, which is generally set to 'Z39.50 Resource Control' for the first release of CGTI CAP services;
    - the Product Name, which specifies the name of the CAP to be tracked;
    - an Action, which contains either 'queued' (meaning that all the alerts on the 'new information' queue at the Target are delivered to the Origin, but once the queue is cleaned out, additional alerts will not be sent until another Alert Specification is sent), 'realtime' (meaning that alerts can be delivered to the Origin at any time after the Alert Specification is created, and will continue to be sent until the Alert Specification is deleted), or 'refresh' (meaning re-send or 'refresh' the entire CAP, including Topics and documents previously delivered to the Origin);
    - a flag (alertCombinations Desired) specifying whether the Origin is prepared to handle multiple topics and/or multiple topics for multiple products in a single alert, or whether it will only handle one topic per alert (a topic is a single category of information in a CAP which covers a single subject area - e.g., baseball scores under the Sports CAP);
    - if alertCombinationsDesired specifies "multipleTopicsPer Alert", the maxTopicsPerAlert parameter indicates the maximum number of Topics which are allowed to be packaged into a single Resource Control 'Alert'.
- the Target will next save the Alert Specification (if valid) and reply with an Extended Services response specifying that the request is valid and has been processed (i.e, the Alert Specification has been created);
- once the Specification is created, the Target then proceeds to send alerts to the requesting Origin system (based on the setting of the action parameter) whenever new documents are received for the specified product(s), using Z39.50 Resource Control 'Alert' reports.

5.2 Sending Information (Product) Alerts

Once an Alert Specification is created, the Target is responsible for sending alerts to the Origin whenever new topics are added to a CAP, or a CAP is updated with new records/documents.

Resource Control requests, each containing one or more Alerts, are issued by the Target to notify the Origin of the availability of new CAP topics, deleted topics, or additional documents added to an existing CAP topic. In response, the Origin sends a Resource Control response telling the Target whether or not it wants to continue receiving Alerts. The Origin can then send a Present request to retrieve the new records, after sending the Resource Control response to the Target.

As stated earlier in this paper, this use of Resource Control for sending alerts is not conformant with the use of Resource Control as defined within the Z39.50 standard. These Resource Control 'alerts' do not correlate with any specific request (as required by the Z39.50 state tables) nor do they relate to a Resource Report pertaining to the entire association. In future releases of the CAP delivery system, notification mechanisms (such as FAX and E-mail) will be used for delivery of alerts, thus deprecating Resource Control as an alert delivery mechanism.

The Target will issue a request with a referenceId, a resourceReport ('Alert', see definition in 5.3 below), and a responseRequired flag set to 'ON'. Within each alert is an alertAction parameter; it can be set to either "new", "update", or "remove". If it is set to "new", this is a new Topic (i.e, it has not been retrieved by this Origin previously). If alertAction is set to "update", this is an existing Topic which has been updated with new documents. "Remove" indicates that the designated TopicID has been removed from the Product; if the Topic is a hierarchy node, it indicates that all the 'child' nodes have been removed from the Product as well.

ResultSetID, of course, indicates the ID of the Result Set which corresponds to the latest version of the Topic 'set'. The Origin uses this as the Result Set ID when it builds the Present request to retrieve records from the Result Set (see the Note on 'virtual Persistent Result Sets' below). NumItems specifies the total number of records in the 'Topic' result set.

AlertsQueue is a parameter which indicates how many remaining alerts are waiting to be sent relative to this Product Alert Specification. If alertCombinationsDesired is set to "multipleTopicsPerAlert" or "multipleProductsPerAlert" in the Alert Spec, then the Topics structure will carry a sequence of potentially multiple pairs of topicPath, resultSetId parameters. The Origin will send a response by returning the referenceId received during the request, in addition to a continueFlag set to "ON".

Note: Each of the resultSetIds conveyed in an Alert represents an identifier for a Transient Result Set. However, in this implementation of CGTI, the Target performs an automatic service for the Origin. It maintains a Persistent Result Set Task Package for each of the Transient Result Sets it reports to the Origin in an Alert. Then, when the Origin issues a Present against the Result Set (using the Transient Result Set ID), the Target calls the back-end retrieval service to create a Transient Set from the Persistent Set. This Transient Set may then be accessed via Present using a result set name that is actually associated with the Persistent Result Set Task Package. Thus the Target and its associated back-end services present a 'virtual Persistent Result Set' image to the accessing Origin.

The SetType parameter indicates whether this set (which is the subject of this alert) is a hierarchy 'node' or a leaf 'node' in the Product structure (see "Understanding CAP Hierarchy" in Section 6.1 below). In simple terms, a leaf node set always contains only records/documents (or pointers to records), whereas a hierarchy node set may contain records, but also contains pointers to sets subordinate to itself in the Product hierarchy.

5.3 LN-RR-1 Resource Report Definition

The following is the LN-RR-1 Resource Report definition, which is a privately-registered Resource Report (RR) type to be used by LEXIS-NEXIS for asynchronous notification of events (in this case, Information Alerts).

```
RR (1.2.840.10003.7.1000.14.3) DEFINI-
TIONS::= BEGIN

LNResourceReport ::= IMPLICIT SEQUENCE {
    resourceReportId IMPLICIT OBJECT
                              IDENTIFIER,
    -- specifies an OID to identify this Resource
    -- Report type. The following is the structure
    -- of the CAP 'Information Alert', which
    -- is used to notify the Origin of newly-received
    -- relevant CAP information.
    operation [0] IMPLICIT VisibleString,
    --set to "alert" for this RR class
    alertTopicsQueue [1] IMPLICIT INTEGER,
    numTopicsThisAlert [2] IMPLICIT INTEGER,
    topicNodeRecord [3] IMPLICIT SEQUENCE
                              OF SEQUENCE {
        topics [1] IMPLICIT SEQUENCE OF
        SEQUENCE {
            topicId  [0] IMPLICIT VisibleString,
            resultSetId [1] IMPLICIT VisibleString,
            numItems [2] IMPLICIT INTEGER
            alertAction [3] IMPLICIT INTEGER {
                                        new     (1),
                                        update  (2),
                                        remove (3) },
            setType [4] IMPLICIT INTEGER {
                                        hierarchy (1),
                                        leaf (2) }
    OPTIONAL } },
    }
    -- End of LN-RR-1 Resource Report definition
    END
```

6.0 Retrieving Information

Retrieving CAP information is the primary objective of an Origin system which is accessing Current Awareness Products. Each Origin has a defined set of CAPs that it is authorized to access. This is verified through the user ID of the Origin system (see discussion of CGTI User Model in Section 1.4). Each set of user CAPs is known as a root product set. Each root product set consists of:

- Descriptive information about the root product (result) set
- Product (result) sets accessible to the end user and the topics associated with each product set

You can retrieve information from these product sets with or without alert notification. The procedure for retrieving information is similar; however, you must understand the hierarchal nature of a CAP before attempting to retrieve CAP information.

6.1 Understanding CAP Hierarchy

A CAP is designed as a hierarchical product with a tree-like structure. The top level of the tree is a topic or information category associated with the product. Subsequent levels of the tree consist of subtopics associated with the topic or subtopic at the next higher level. Each topic that contains one or more subtopics is considered a hierarchy node. The lowest level of a hierarchal system consists of leaf nodes. There are no subtopic levels associated with leaf nodes.

A CAP can also be a flat product with no underlying tree structure (no hierarchy nodes).

From a Z39.50 perspective, each version (e.g., daily update) of a CAP topic corresponds to a unique result set and has a corresponding result set ID. Each version of a subtopic at a hierarchical level also corresponds to a result set. An Origin can request different 'views' or subsets of the records in a CAP result set, depending on what 'views' it wants to see. Records in a result set can contain the following types of information:

- Set metadata information
- Documents relating to a topic
- If available, the parent of the topic and any other child subtopics

As stated previously, each CAP Topic/subtopic set contains set-level metadata and document data. To access the set-level metadata, the Origin issues a Present request against record (document) 0 of a valid CAP result set.

Note: The use of Record 0 for access to and retrieval of set metadata is not conformant with the Result Set model specified in the Z39.50 standard. However, it is being used as a convenient mechanism for access to this metadata. It is foreseen that the set metadata will be accessed via more conventional (and standard) Z39.50 facilities in future releases of CGTI. For instance, one possible solution would be to allow the set-level metadata to be retrieved from any valid record in a set (1-N), simply by specifying the correct set meta-element names/tags.

At a hierarchy node, the Origin can request the following elements of metadata describing a CAP topic or subtopic:

- Set class indicating whether the CAP is flat or hierarchal
- Topic or subject name of the CAP
- Result set name or ID
- Description of the result set or topic
- Existing child subtopics
- the root ancestor of this topic (the Product Name/ID) [future]
- an (optional) list of related/associated topics/subjects [future]

A hierarchy node (result) set contains only "directory" information such as the data structure with a pointer indicating where a record is located in the CAP Provider database. Records are referenced by their relative position within the result set.

At a leaf node, you can request different views of a document such as a:

- Cite list
- Preview
- Full Document text

A leaf node result set is made up of the individual records (documents) in the set. Each record contains the full document content and descriptive information such as author, title, document publication date, and so on. The format of the document can be in ASCII text, SGML, or in any other type supported by the CGTI system.

Figure 6-1 illustrates the CAP hierarchy concept.

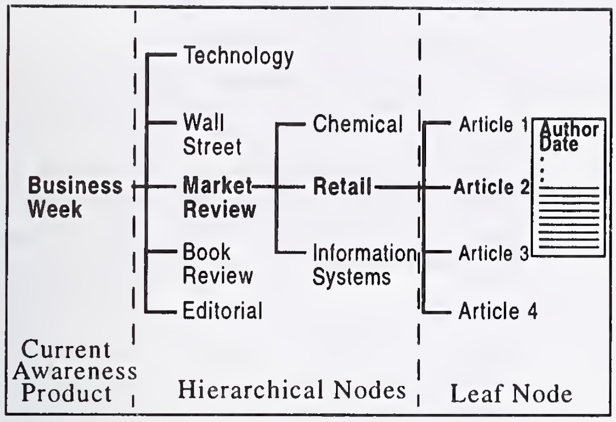


Figure 6-1 CAP Hierarchy

6.2 Retrieving Data Information

The CGTI Target allows an Origin to initiate data retrieval using the Present and Segment services. When the Origin issues a Present request, it uses designated Present elements to define the view of the retrieved information. A typical flow looks like this:

- Suppose you want to know what topics are associated with a product. The Origin sends a Present request specifying the result set ID of the product;
- The Target generates internal retrieval responses and returns an aggregate Present response, i.e, zero or more Segment responses followed by a Present response. A single document (within a CAP topic category) is sent to the Origin in each response.
- You can choose to view any level of the product hierarchy, or documents associated with a topic, by issuing additional Present requests.

  Note: Level 1 Segmentation is used by CGTI services to break down large result sets into segments (records) which fit into the specified maximum message and record sizes.

6.3 Retrieving New or Updated Information

Suppose the Origin receives an alert triggered by the availability of new CAP topics or the addition of new documents to a CAP. The flow for retrieving alert information is similar to retrieving data information. You still need to send the Present request; however, you need only the result set ID for the new topic alert or new document alert (see the Note in Section 5.3 on the use of 'virtual Persistent Result Sets' within the current CAP implementation). Like the data information retrieval process, you can select different views of the new or updated information.

The Origin initiates the Z39.50 Present service by sending a Present request to the Target. Typically, the Origin returns to the client application when the Present response is received in full.

The preferredRecordSyntax parameter in the Present request is not currently used by the Target. The current release of CGTI supports only a 'default' record syntax (using a LEXIS-NEXIS private OID) which does not provide encapsulation of the record contents; however, in future releases, both SUTRS and GRS-1 will be supported.

The presentElements parameter is used to specify the desired subset/view of the records expected in the present response. For Release 1, this will simply consist of an elementSetName, which will be structured to contain the desired document/record 'view', in addition to the desired document format.

In future releases, the eSpec-1 structure will be used in place of elementSetName to describe more complex composition specifications by which to retrieve the records.

The following are the document 'views' supported in the current release of CGTI:

- CITE - retrieves "headline" information about a topic
- PREVIEW - retrieves an abstract of a document
- FULL - retrieves entire document contents
- SUBINFO - retrieves subtopic information
- TOPICINFO - retrieves set metadata information

The Origin can also specify the text format of the retrieved documents using these values as part of the Element Set Name:

- FASCII - formatted ASCII text
- UASCII - unformatted ASCII text
- GSGML - text tagged with SGML tags using the 'generalized CAP DTD'

So, using the legal values above, an example of an ElementSetName is 'CITE;FASCII', which would be used to request retrieval of headline information in formatted ASCII.

7.0 Delivery Notification, Usage, and Subscription Information

Delivery and document usage activities are tracked by LEXIS-NEXIS to provide usage, billing, and subscription information. These activities provide revenue based on price schedules for documents, document usage, delivery services, and subscription services. Various notices and reports are generated by the Origin to provide tracking information. The Extended Services facility allows the Origin to send this information to the Target and to generate a Final Delivery Notice, Usage Accounting Report, and Subscription Accounting Report.

7.1 Providing Delivery Notification

The Final Delivery Notice (FDN) is an Extended Services task that allows the Origin to send confirmation to the Target that a document has been delivered. This confirmation contains the following types of information:

- The delivery status
- The date and time the document was delivered
- The retail price, in cents, charged to the end user
- The suggested wholesale price

The FDN is used in situations where the Origin initiates a Present request to retrieve a document and passes the document directly to one or more end users.

The process of creating and sending an FDN to the CGTI Target is similar to the alert specification process. The Origin builds an FDN Extended Services request, which creates an FDN parameter package at the CGTI server that captures the report. After processing, the Target returns an Extended Services response indicating whether or not the notice was delivered.

7.2 Providing Document Usage Information

Mostly on behalf of *Indirect* end users (see Section 1.4), external delivery systems within the Origin domain are required to capture data about CAP document usage. They then send this data to the Target in the form of a Usage Accounting Report (UAR). The UAR contains these types of information:

- The number of copies of a document that were delivered to the Origin
- The date and time the document was delivered
- The retail price, in cents, charged to the end user
- The suggested wholesale price

The UAR is used in situations where the Origin initiates a Present request to retrieve a document and stores the results (document) locally. The Origin then delivers these documents to end users directly from local storage, rather than from the Target. The Origin periodically sends UARs to the Target indicating what and how many documents have been delivered.

Similarly to the FDN, the Origin builds a UAR Extended Services request, which creates a UAR parameter package at the CGTI server that captures the report. After processing, the Target returns an Extended Services response indicating whether or not the report was delivered.

7.3 Providing Subscription Information

End users have the ability to request subscriptions to specific CAPs. Mostly on behalf of *Indirect* end users (see Section 1.4), external delivery systems within the Origin domain capture subscription request data and send the data to the Target in the form of a Subscription Activity Report (SAR). The SAR tracks subscription information such as:

- The number of requested subscriptions
- The length of the subscription in months
- The retail price, in cents, charged to the end user
- The suggested wholesale price

Similarly to the FDN and the UAR, the Origin builds an SAR Extended Services request, which creates an SAR parameter package at the CGTI server that captures the report. After processing, the Target returns an Extended Services response indicating whether or not the report was delivered.

8.0 Terminating a CGTI Association

The preceding sections have described the CGTI Z39.50 profile which allows an Origin system to establish an association, identify new information, retrieve information, and create administrative reports. As with the Initialization of a Z39.50 association, termination of the association conforms to the Z39.50 Version 3 specification. Reasons for termination range from a security violation to internal system errors at the Origin or Target.

The termination process consists of these steps:

- Terminating the Z39.50 association between the Target and Origin using the Close request
- Terminating the TCP/IP (or X.25) connection between the Target and Origin environments

A Close request can be issued from either the Origin or Target. The reason for termination is specified within the Close request. After processing the request, the Origin or Target returns a Close response.

*U.S. GOVERNMENT PRINTING OFFICE: 1995-386-627/47911