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, de-

fault 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.