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 International-String, 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.