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 interested 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 **resultSetName**s 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 non-

repeating 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 **resultSetName**s 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 **PresentRequest**s. 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

©1995 OCLC Online Computer Library Center, Inc., 6565 Frantz Road, Dublin, Ohio 43017-0702. OCLC is a registered trademark of OCLC Online Computer Library Center, Inc.

NOTICE TO USERS: The Z39.50 Client API ("Software") has been developed by OCLC Online Computer Library Center, Inc. Subject to the terms and conditions set forth below, OCLC grants to user a perpetual, non-exclusive, royalty-free license to use, reproduce, alter, modify, and create derivative works from Software, and to sublicense Software subject to the following terms and conditions:

SOFTWARE IS PROVIDED AS IS. OCLC MAKES NO WARRANTIES, REPRESENTATIONS, OR GUARANTEES WHETHER EXPRESS OR IMPLIED REGARDING SOFTWARE, ITS FITNESS FOR ANY PARTICULAR PURPOSE, OR THE ACCURACY OF THE INFORMATION CONTAINED THEREIN.

User agrees that :1) OCLC shall have no liability to user arising therefrom, regardless of the basis of the action, including liability for special, consequential, exemplary, or incidental damages, including lost profits, even if it has been advised of the possibility thereof; and :2) user will indemnify and hold OCLC harmless from any claims arising from the use of the Software by user's sublicensees.

User shall cause the copyright notice of OCLC to appear on all copies of Software, including derivative works made therefrom.