The OCLC *Z39.50 Client API* Software Library

Version 1.5

September 8, 1995

Table of Contents

Table of Contents 2

Changes from previous versions 3

High-level description 3

Availability 4

InitRequest() 5

InitResponse() 6

SearchRequest() 7

SearchResponse() 8

PresentRequest() 9

PresentResponse() 10

ScanRequest() 11

ScanResponse() 12

WhatKindOfResponse() 13

Z3950_Response() 14

IRP_Logging() 15

zapi.h 16

Example 19

Changes from previous versions

A description of our fourth version of the *Z39.50 Client API* is below. It will, of course, undergo some refinement with use and get extended to support new Z39.50 functionality as required. The changes from the fourth version include support for type-101 queries (including proximity operators in the *Client API*'s query grammer) and the removal of some unneeded parameters in the request calls. Scan was brought into compliance with the final version of the '95 standard. In addition, a number of changes were made to simplify things and to bring the *Client API* into compliance with the article that was written about it. (See the section on availability at the end of this document.)

The change from the third version is the addition of support for piggybacked Presents with the SearchRequest. This means that there is now an optional PRESENT_RESPONSE structure in the SEARCH_RESPONSE structure.

The changes from the second version are the addition of support for a simple userid and password on the InitRequest; support for full boolean queries in the SearchRequest; support for both MARC and SUTRS record syntaxes in the PresentRequest. New routines to support Scan have been added. They are in an early stage of development and will definitely be enhanced with experience. Finally, a new routine, IRP_Logging(), provides support for run-time diagnostic logging.

High-level description

The routines process IRP messages, they do not send or receive these messages; that is up to the calling routines. (IRP stands for Information Retrieval Protocol, which is the name of the protocol described by Z39.50.) An IRP Request routine is called and it produces a BER (Basic Encoding Rules, the chosen transfer syntax for ASN.1) record that can be sent to a Z39.50 target. Conversely, a BER record containing an IRP response is processed and a structure is returned. The BER records are encoded and decoded using OCLC's BER utilities. These utilities are available via anonymous ftp to ftp.rsch.oclc.org and are in the pub/BER_utilities directory.

There are two layers to the IRP response routines. The layer you choose to use will depend on coding style and application complexity. The lower layer, consisting of *InitResponse()*, *SearchResponse()*, *PresentResponse()* and *ScanResponse()*, are called directly with the Z39.50 target's response to an Init, Search, Present or Scan request. These routines fill in structures with the various parameters in the response. They are appropriate when the application (the Z39.50 origin) is relatively simple and "knows" what kind of response it is receiving. This is usually a single user application.

In more complex applications, it is difficult to know what kind of response has arrived. The next layer addresses this problem. Two routines are provided; *WhatKindOfResponse()* and *Z3950_Response()*. *WhatKindOfResponse()* returns a value indicating what kind of IRP response has arrived. *Z3950_Response()* takes that one step further, by returning a structure that indicates the type of response and a then calls the appropriate response routine, returning the response routine's structure through a pointer that can be appropriately cast.

In the descriptions of the routines, I will not discuss the meaning of the various parameters to the request routines. They are fully described in the Z39.50 standard.

In all the request routines, a long *referenceId* is provided. The value in the *referenceId* is in the structure returned by the response routines. In complex applications, we have found that a useful place to put a pointer to a structure containing information about the request that has just been processed. Doing this has allowed us to build relatively stateless applications.

The BER record that is created by the request routines can be put into a buffer provided by you or allocated by the request routine. It is often convenient to have the request routines place their BER records directly into the communications *buffer* for transmission. If you do provide a *buffer*, *len* should be set to the length of the *buffer*. If the *buffer* is not big enough to contain the BER record, NULL will be returned and *len* will contain the size of *buffer* that the record would have required. If *buffer* is NULL or *len* is 0, then a *buffer* will be allocated and returned. *Len* will be set to the size of the allocated *buffer*. The allocated *buffer* can be freed after its contents are copied or transmitted.

A copy of zapi.h can be found at the end of this document.

Availability

An article about building Z39.50 clients, which includes a detailed description of the Client API, was written for the Z39.50 Implementors Group. It is available via anonymous ftp at the Z39.50 Maintenance Agency's ftp site at ftp.loc.gov. You're on your own for finding it there.

The article will also be published by the National Institute for Standards and Technology (NIST) as part of a Z39.50 monograph. Good luck finding that too.

Finally, the article is available via anonymous ftp at ftp.rsch.oclc.org in the pub/SiteSearch/z39.50_client_api directory. It is available in several formats, the names of which all begin with zclient.

The OCLC ftp site also provides the source code for the Z39.50 Client API and a simple client application (zdemo), which are in the same directory, as well as the BER utilities which are in the pub/BER_utilities directory.

InitRequest()

Synopsis
#include "zapi.h"

CHAR *InitRequest(long referenceId, CHAR *options,
    long *preferredMessageSize*, long *maximumRecordSize*, CHAR **id*,
    CHAR **password*);

Function
Creates a Z39.50 InitRequest.

Returns
NULL if sufficient space could not be allocated. Otherwise, a pointer to the location of the constructed BER record is returned.

Notes
For now, set *options* to NULL. This will cause *Init_Request()* to ask for Search, Present and Scan functionality from the target. We'll add a routine later to build a proper options string. If anonymous access to the target is being requested, then *id* and *password* should be NULL. If they are provided, then they will be catenated together, with a separator of '/' and sent in the idAuthentication field of the InitRequest.

Example
See examples at the end of the document.

InitResponse()

Synopsis
#include "zapi.h"

INIT_RESPONSE **InitResponse*(CHAR **response*);

Function
Processes a Z39.50 InitResponse.

Returns
A pointer to an INIT_RESPONSE structure, or a NULL if unable to process the response. If NULL, a call to *WhatKindOfResponse()* will return a value which should indicate why *InitResponse()* failed.

Notes
In the INIT_RESPONSE structure is a variable named *IRP_context*. It must be passed to all subsequent IRP request routines when they are processing requests for the just initialized session. The variable *result* will be set to TRUE if the InitRequest was successful.

Some servers provide useful information in the *userInformation* field. If such information is provided, then the *userInformation* field will need to be free'd. The INIT_RESPONSE structure should be free'd as well.

Example
See examples at the end of the document.

SearchRequest()

Synopsis
#include "zapi.h"

CHAR **SearchRequest*(long *referenceId*, long *mediumSetPresentNumber*,
int *replaceIndicator*, char **resultSetName*, char **databaseNames*,
char **smallSetElementSetNames*, char **mediumSetElementSetNames*,
char **preferredRecordSyntax*, char **query*, long *query_type*)

Function
Creates a Z39.50 SearchRequest.

Notes
databaseNames is a string of names, separated by '\0's and terminated by two '\0's. For now, use just one database name. Only type-0 (DBI) and type-1 (described below) queries are supported.

Type-1 queries are entered as Reverse Polish Notation strings. Currently the only operators supported are AND, OR, and NOT. The USE attribute for a term can be specified by appending a '/' and then the USE attribute value as a digit string to the end of the term. (e.g. To search for dog as a subject term, use the term dog/21). A STRUCTURE attribute can be added by appending another '/' and the STRUCTURE value as a digit string. (e.g. To search for dog as a subject term with a structure of WORD, use the term dog/21/2). Terms with embedded blanks can be surrounded with double-quotes. (e.g. To search for War And Peace as a title phrase, use the term "war and peace"/4/1).

To search for documents with both dog and cat, use the *query* dog cat and.

Returns
NULL if sufficient space could not be allocated. Otherwise, a pointer to the location of the constructed BER record is returned.

Example
See examples at the end of the document.

SearchResponse()

Synopsis
#include "zapi.h"

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

Function
Processes a Z39.50 SearchResponse.

Returns
A pointer to a SEARCH_RESPONSE structure, or a NULL if unable to process the response. If NULL, a call to *WhatKindOfResponse()* will return a value which should indicate why *SearchResponse()* failed. If the search status is non-zero, there may be an *error_code* and possibly an *error_msg*. If there is an *error_msg*, it must be free'd. The structure should be free'd.

Notes
All values in the SEARCH_RESPONSE structure are as described in the Z39.50 standard.

Example
See examples at the end of the document.

PresentRequest()

Synopsis
#include "zapi.h"

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

Function
Creates a Z39.50 PresentRequest.

Returns
NULL if sufficient space could not be allocated. Otherwise, a pointer to the location of the constructed BER record is returned.

Notes
The names MARC_SYNTAX, SIMPLETEXT_SYNTAX and OCLC_BER_SYNTAX are defined in zapi.h and can be used with *preferredRecordSyntax*.

Example
See examples at the end of the document.

PresentResponse()

Synopsis
#include "zapi.h"

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

Function
Processes a Z39.50 PresentResponse.

Returns
A pointer to a PRESENT_RESPONSE structure, or a NULL if unable to process the response. If NULL, a call to *WhatKindOfResponse()* will return a value which should indicate why *PresentResponse()* failed. The structure should be free'd.

Notes
All values in the PRESENT_RESPONSE structure are as described in the Z39.50 standard. A structure to hold the returned records (named *records*) is also in the PRESENT_RESPONSE. It is an array of pointers to structures that each contain a pointer to a single record. You should be sure to free each of the records and the *records* structure when done with the PRESENT_RESPONSE.

Example
See examples at the end of the document.

ScanRequest()

Synopsis
#include "zapi.h"

CHAR **ScanRequest*(long *referenceId*, char **databaseName*, char **term*,
    int *stepSize*, int *numberOfTermsRequested*,
    int *referredPositionInResponse*);

Function
Creates a Z39.50 ScanRequest.

Returns
NULL if sufficient space could not be allocated. Otherwise, a pointer to the location of the constructed BER record is returned.

Notes
The *term* is constructed the same as a term in a *query* in a SearchRequest. It can have both a USE attribute and a STRUCTURE attribute.

Example
See examples at the end of the document.

ScanResponse()

Synopsis
#include "zapi.h"

SCAN_RESPONSE **ScanResponse*(CHAR **response*);

Function
Processes a Z39.50 ScanResponse.

Returns
A pointer to a SCAN_RESPONSE structure, or a NULL if unable to process the response. If NULL, a call to *WhatKindOfResponse()* will return a value which should indicate why *ScanResponse()* failed. The structure should be free'd.

Notes
All values in the SCAN_RESPONSE structure are as described in the Z39.50 standard. A structure to hold pointers to the returned terms (named *terms*) is also in the SCAN_RESPONSE. Parallel to the *terms* array is a *postings* array. It contains the count of documents associated with each term in the *terms* array. You should be sure to free *terms* and *postings* arrays and each of the terms pointed to in the *terms* structure, when done with the SCAN_RESPONSE.

Example
See examples at the end of the document.

WhatKindOfResponse()

Synopsis
#include "zapi.h"

IRP_type *WhatKindOfResponse*(CHAR **response*);

Function
Identifies the type of a Z39.50 response.

Returns
The type of the IRP response.

Notes
Besides identifying known types of IRP responses, *WhatKindOfResponse()* will also indicate if the response is not a BER record or an unknown IRP response type.

Example
See examples at the end of the document.

Z3950_Response()

Synopsis
#include "zapi.h"

Z3950_RESPONSE **Z3950_Response*(CHAR **response*);

Function
Processes a Z39.50 response.

Returns
A pointer to an Z3950_RESPONSE structure. The structure should be free'd.

Notes
If the type of the response if not BER or an unknown IRP response, then the return value will be NULL. Otherwise, the return value can be cast to a pointer to the appropriate type of response structure.

Example
See examples at the end of the document.

IRP_Logging()

Synopsis
#include "zapi.h"

void **IRP_Logging*(int *flag*);

Function
Enable and disable run-time diagnostics.

Returns
Nothing.

Notes
This routine opens a file named *irp.log* in the current directory. A *flag* of 1 (TRUE) enables logging and a *flag* of 0 (FALSE) disables logging. Logging can be repeatedly enabled and disabled. A special value for *flag* of 2 is the signal that hex_dir() style formatted dumps of the input and output messages are requested. A special value for *flag* of 666 is the signal that the logging file should be closed. You should call *IRP_Logging(666)* before stopping your program, if you have ever enabled logging.

Example
See examples at the end of the document.

zapi.h

```
/*************************************************************************
*                          OCLC, Inc.
*
*   OCLC proprietry information: the enclosed materials contain
*   proprietry information of OCLC Online Computer Library Center, Inc.
*   and shall not be disclosed in whole or in part to any third party
*   or used by any person for any purpose, without written consent of
*   OCLC, Inc. Duplication of any portion of these materials shall
*   include this legend.
*************************************************************************/

#ifndef ZAPI_H
#define ZAPI_H

#ifndef TRUE
#define TRUE 1
#endif

#ifndef FALSE
#define FALSE 0
#endif

#define MARC_SYNTAX "1.2.840.10003.5.10"
#define SIMPLETEXT_SYNTAX "1.2.840.10003.5.101"
#define OPAC_SYNTAX "1.2.840.10003.5.102"
#define OCLC_BER_SYNTAX "1.2.840.10003.5.1000.17.1"
#define OCLC_USERINFORMATION_1 "1.2.840.10003.10.1000.17.1"

typedef struct
{
    long referenceId;
    char *options;
    long preferredMessageSize;
    long maximumRecordSize;
    int  result;
   char *MessageOfTheDay;
   int  NumDBs;
   char **DBList;
   int  failureCode;
} INIT_RESPONSE;

typedef struct
{
        long len;
        CHAR *record;
} IRP_RECORD;
```

```
typedef struct
{
        long referenceId;
        int numberOfRecordsReturned;
        long nextResultSetPosition;
        long presentStatus;
        long error_code;
        char *error_msg;
        char recordSyntax[50];
        IRP_RECORD *records;
} PRESENT_RESPONSE;


typedef struct
{
        long referenceId;
        long resultCount;
        int  searchStatus;
        long resultSetStatus;
        long error_code;
        char *error_msg;
       PRESENT_RESPONSE PresentResponse;
} SEARCH_RESPONSE;

typedef struct
{
        long referenceId;
        long numberOfEntriesReturned;
        long scanStatus;
        long error_code;
        char *error_msg;
        char **terms;
        long *postings;
} SCAN_RESPONSE;

enum IRP_type
{
        not_BER,
        IRP_unknown,
        IRP_init,
        IRP_search,
        IRP_present,
        IRP_scan,
           IRP_docorder_price,
           IRP_docorder
};
```

```
typedef struct
{
        enum IRP_type irp_type;
        CHAR     *Z3950_Response;  /* cast to appropriate struct */
} Z3950_RESPONSE;              /* depending on irp_type */

void             IRP_Logging(int flag);
CHAR              *InitRequest(long referenceId, char *options,
                  long preferredMessageSize, long maximumRecordSize,
                  char *id, char *password, char *newpassword,
                  DATA_DIR *userInformationField);
INIT_RESPONSE    *InitResponse(CHAR *response);
INIT_RESPONSE    *InitResponse2(DATA_DIR *dir);
CHAR              *SearchRequest(long referenceId, long smallSetUpperBound,
                  long largeSetLowerBound, long mediumSetPresentNumber,
                  int replaceIndicator, char *resultSetName,
                  char *databaseNames, char *smallSetElementSetNames,
                  char *mediumSetElementSetNames, char
*preferredRecordSyntax,
                  char *query, long query_type);
SEARCH_RESPONSE  *SearchResponse(CHAR *response);
SEARCH_RESPONSE  *SearchResponse2(DATA_DIR *dir);
CHAR              *PresentRequest(long referenceId, char *resultSetName,
                  long resultSetStartPoint, long numberOfRecordsRequested,
                  char *ElementSetNames, char *preferredRecordSyntax);
PRESENT_RESPONSE *PresentResponse(CHAR *response);
PRESENT_RESPONSE *PresentResponse2(DATA_DIR *dir);
CHAR              *ScanRequest(long referenceId, char *databaseName,
                  char *term, int stepSize, int numberOfTermsRequested,
                  int preferredPositionInResponse);
SCAN_RESPONSE    *ScanResponse(CHAR *response);
SCAN_RESPONSE    *ScanResponse2(DATA_DIR *dir);
enum IRP_type     WhatKindOfResponse(CHAR *response);
Z3950_RESPONSE     *Z3950_Response(CHAR *response);
Z3950_RESPONSE     *Z3950_Response2(DATA_DIR *dir);

#endif
```

Example

This example assumes the existance of some user provided routines. *IRP_Connect()* and *IRP_Disconnect()* are used to make a break the tcp/ip connect to the remote server. *doirp()* sends a BER encoded request, frees the encoded request passed to it and waits for the BER encoded response.

This example also uses OCLC's BER utilities. These utilities are available via anonymous ftp at ftp.rsch.oclc.org and are in the pub/BER_utilities directory.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "localtcp.h"  /* this is where IRP_Connect(), IRP_Disconnect() and
                doirp() are defined */
#include "berutil.h"
#include "zapi.h"

void main()
{
    CHAR                *irp_request, *Z3950_Response;
    DATA_DIR            *dir, *irp_resp;
    INIT_RESPONSE       *init_response;
    int                 CanDoScan=FALSE, I, MustUseDefault=FALSE;
    long                len, nRecs;
    PRESENT_RESPONSE    *present_response;
    SCAN_RESPONSE       *scan_response;
    SEARCH_RESPONSE     *search_response;

    if(!IRP_Connect("tikal.dev.oclc.org", 210))
    {
        printf("unable to connect to server\n");
        exit(1);
    }

    /* send and process the InitRequest */
    irp_request=InitRequest(0, "yynnnnny", 32767, 32767, NULL,
        NULL, NULL, &len);
    Z3950_Response = do_irp(irp_request);
    if(!Z3950_Response)
    {
        printf("unable to send init request\n");
        exit(2);
    }
    init_response=InitResponse(Z3950_Response);
    if(!init_response || !init_response->result)
    {
        printf("init failed\n");
        hex_dir(irp_resp, 0);
        exit(3);
    }
    if(init_response->userInformation)
        puts(init_response->userInformation);
    if(strlen(init_response->options)>7 && init_response->options[7]=='y')
        CanDoScan=TRUE;
    else
        printf("This server doesn't support the Scan Facility\n");
    free((char*)init_response);
    free((char*)Z3950_Response);
```

```
/* send and process a ScanRequest  on the term "dog"*/
if(CanDoScan)
{
    irp_request=ScanRequest(0, "worldcat", "dog", 1, 20, 10);
    Z3950_Response = do_irp(irp_request);
    scan_response=ScanResponse(Z3950_Response);
    if(!scan_response)
    {
        printf("Did not get an IRP_scanResponse!\n");
        exit(4);
    }
    if(scan_response->scanStatus==0)
        printf("Scan Successful! :-)\n");
    else
        printf("Scan Failed! :-(\nstatus=%ld\n",
            scan_response->scanStatus);
    for(i=0; i<scan_response->numberOfEntriesReturned; i++)
    {
        printf("'%s' %ld\n", scan_response->terms[i],
            scan_response->postings[i]);
        free(scan_response->terms[i]);
    }
    free((char*)scan_response->terms);
    free((char*)scan_response->postings);
    free((char*)scan_response);
    free((char*)Z3950_Response);
}

/* send and process a SearchRequest */
irp_request=SearchRequest(0, TRUE,
    MustUseDefault ? "default" : "Search 1", "worldcat", "dog/21", 1);
Z3950_Response = do_irp(irp_request);
search_response=SearchResponse(Z3950_Response);
if(!search_response)
{
    printf("Did not get an IRP_searchResponse!\n");
    exit(5);
}
printf("%ld records found.\n", search_response->resultCount);
if(search_response->searchStatus)
    printf("Search Successful! :-)\n");
else
{
    printf("Search Failed! :-(\n");
    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_msg)
        free(search_response->error_msg);
    if(search_response->error_code==22)
    {
        puts("Must use ResultSetName of \"default\"");
        puts("Resetting internal flags; please try again");
        MustUseDefault=TRUE;
    }
}
free((char*)search_response);
free((char*)Z3950_Response);

/* send and process a PresentRequest  for 5 full SUTRS records */
irp_request=PresentRequest(0, "Search 1", 1, 5, "f",
    SIMPLETEXT_SYNTAX);
Z3950_Response = do_irp(irp_request);
present_response=PresentResponse(Z3950_Response);
if(!present_response)
{
    printf("Did not get an IRP_presentResponse!\n");
    exit(6);
}

nRecs=present_response->numberOfRecordsReturned;
printf("%ld records returned\n", nRecs);
switch(present_response->presentStatus)
{
    case 0:
        printf("Present successful\n");
        break;
    case 1:
    case 2:
    case 3:
    case 4:
        printf("Partial results returned\n");
        break;
    case 5:
        printf("Present failed\n");
        break;
}
printf("recordSyntax='%s'\n", present_response->recordSyntax);
for(i=0; i<nRecs; i++)
    if(present_response->records[i].record)
    {
        char *end, *ptr, *t;
        DATA_DIR *temp=dalloc(3);
```

```
            bld_dir(present_response->records[i].record, temp);
            ptr=temp->ptr.data;
            end=ptr+temp->count;
            dfree(temp);
            while(ptr<end)
            {
                for(t=ptr; t<end; t++)
                    if(*t=='\n')
                    {
                        *t='\0';
                        break;
                    }
                puts(ptr);
                ptr=t+1;
            }
            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((char*)present_response);
    free((char*)Z3950_Response);
}
```