YAZ++ User's Guide and Reference

Mike Taylor

Adam Dickmeiss

YAZ++ User's Guide and Reference

by Mike Taylor and Adam Dickmeiss

YAZ++ (http://www.indexdata.dk/yazplusplus/) is a set of libraries and header files that make it easier to use the popular C-language YAZ toolkit (http://www.indexdata.dk/yaz/) from C++, together with some utilities written using these libraries. It includes an implementation of the C++ binding for ZOOM (ZOOM-C++).

This manual covers version 0.9.

CVS ID: $Id: yaz++.xml.in,v 1.13 2004/04/11 12:13:32 adam Exp $

Table of Contents

Chapter 1. Introduction

YAZ++ (http://www.indexdata.dk/yazplusplus/) is a C++ layer for YAZ and implements the ANSI Z39.50 protocol for information retrieval (client and server side). While YAZ itself can be used from both C and C++ it is limited by the common denominator C.

The YAZ++ packages also features a ZOOM interface for C++ (ZOOM C++ (http://zoom.z3950.org/bind/cplusplus/)).

Later versions (0.7+) of YAZ++ also supports SRW/SRU.

Licensing

YAZ++ and ZOOM-C++ is is covered by the YAZ license.

Chapter 2. Installation

You need a C++ compiler to compile and use YAZ++. The software was implemented using GCC so we know that works well with YAZ++. From time to time the software has been compiled on Windows using Visual C++. Other compilers should work too. Let us know of portability problems, etc. with your system.

YAZ++ is built on top of the YAZ (http://indexdata.dk/yaz/) toolkit. You need to install that first. For some platforms there are binary packages for YAZ.

Building on Unix

On UNIX, the software is compiled as follows:

```
$ ./configure
$ make
$ su
# make install
```

You can supply options for the `configure` script. The most useful ones are:

`--prefix` directory

Specifies installation prefix. By default `/usr/local` is used.

`--with-yaz` directory

Specifies the location of `yaz-config`. The `yaz-config` program is generated in the source directory of YAZ as well as the binaries directory when YAZ is installed (via make install).

If you don't supply this option, `configure` will look for `yaz-config` in directories of the PATH environment - which is nearly always what you want.

For the whole list of `configure` options, refer to the help: `./configure --help`.

Configure uses GCC's C/C++ compiler if available. To specify another compiler, set `CXX`. To use other compiler flags, specify `CXXFLAGS`. To use `CC` with debugging you could use:

```
CXXFLAGS="-g" CXX=CC ./configure
```

This is what you have after successful compilation:

`src/libyazcpp.la`

The YAZ++ library. This library gets installed in your libraries directory (*`prefix`*`/lib`).

`src/libzoomcpp.la`

The ZOOM-C++ library. This library gets installed in your libraries directory (*`prefix`*`/lib`).

`include/yaz++/*.h`

Various C++ header files, which you'll need for YAZ++ development. All these are installed in your header files area (*`prefix`*`/include/yaz++`).

`yaz++-config`

A Bourne shell-script utility that returns the values of the CFLAGS and LIBS environment variables needed in order to compile your applications with the YAZ++ library. This script gets installed in your binaries directory (*`prefix`*`/bin`).

`zoom/zclient`

ZOOM C++ demonstration client that uses the ZOOM C++ classes. This client does not get installed in the system directories.

`src/yaz-my-client`

YAZ C++ demonstration client. This client does not get installed in the system directories.

`src/yaz-my-server`

YAZ C++ demonstration server. This server does not get installed in the system directories.

Building on Windows

YAZ++ is shipped with "makefiles" for the NMAKE tool that comes with Microsoft Visual Studio (http://msdn.microsoft.com/vstudio/). Version 6 and .NET has been tested. We expect that YAZ++ compiles with version 5 as well.

Start a command prompt and switch the sub directory `WIN` where the file `makefile` is located. Customize the installation by editing the `makefile` file (for example by using notepad). The following summarizes the most important settings in that file:

`DEBUG`

If set to 1, the software is compiled with debugging libraries (code generation is multi-threaded debug DLL). If set to 0, the software is compiled with release libraries (code generation is multi-threaded DLL).

`YAZ_DIR`

Specifies the directory of the YAZ source.

When satisfied with the settings in the makefile, type

```
nmake
```

> **Tip:** If the `nmake` command is not found on your system you probably haven't defined the environment variables required to use that tool. To fix that, find and run the batch file `vcvars32.bat`. You need to run it from within the command prompt or set the environment variables "globally"; otherwise it doesn't work.

If you wish to recompile YAZ++ - for example if you modify settings in the `makefile` you can delete object files, etc by running.

```
nmake clean
```

The following files are generated upon successful compilation:

`bin/yazpp.dll`

YAZ++ DLL . Includes ZOOM C++ as well.

`lib/yazpp.lib`

Import library for `yazpp.dll`.

`bin/zclient.exe`

ZOOM C++ demo client. A simple WIN32 console application.

Chapter 3. ZOOM-C++

Introduction

ZOOM (http://zoom.z3950.org/) is the emerging standard API for information retrieval programming using the Z39.50 protocol. ZOOM's Abstract API (http://zoom.z3950.org/api/) specifies semantics for classes representing key IR concepts such as connections, queries, result sets and records; and there are various bindings (http://zoom.z3950.org/bind/) specifying how those concepts should be represented in various programming languages.

The YAZ++ library includes an implementation of the C++ binding (http://zoom.z3950.org/bind/cplusplus/) for ZOOM, enabling quick, easy development of client applications.

For example, here is a tiny Z39.50 client that fetches and displays the MARC record for Farlow & Brett Surman's *The Complete Dinosaur* from the Library of Congress's Z39.50 server:

```
#include <iostream>
#include <yaz++/zoom.h>

using namespace ZOOM;

int main(int argc, char **argv)
{
    connection conn("z3950.loc.gov", 7090);
    conn.option("databaseName", "Voyager");
    conn.option("preferredRecordSyntax", "USMARC");
    resultSet rs(conn, prefixQuery("@attr 1=7 0253333490"));
    const record *rec = rs.getRecord(0);
    cout << rec->render() << endl;
}
```

Note: For the sake of simplicity, this program does not check for errors: we show a more robust version of the same program later.)

YAZ++'s implementation of the C++ binding is a thin layer over YAZ's implementation of the C binding. For information on the supported options and other such details, see the ZOOM-C documentation, which can be found on-line at http://www.indexdata.dk/yaz/doc/zoom.tkl

All of the classes defined by ZOOM-C++ are in the `ZOOM` namespace. We will now consider the five main classes in turn:

- `connection`
- `query` and its subclasses `prefixQuery` and `CCLQuery`
- `resultSet`

- `record`
- `exception` and its subclasses `systemException`, `bib1Exception` and `queryException`

`ZOOM::connection`

A `ZOOM::connection` object represents an open connection to a Z39.50 server. Such a connection is forged by constructing a `connection` object.

The class has this declaration:

```
class connection {
public:
  connection (const char *hostname, int portnum);
  ~connection ();
  const char *option (const char *key) const;
  const char *option (const char *key, const char *val);
};
```

When a new `connection` is created, the hostname and port number of a Z39.50 server must be supplied, and the network connection is forged and wrapped in the new object. If the connection can't be established - perhaps because the hostname couldn't be resolved, or there is no server listening on the specified port - then an `exception` is thrown.

The only other methods on a `connection` object are for getting and setting options. Any name-value pair of strings may be set as options, and subsequently retrieved, but certain options have special meanings which are understood by the ZOOM code and affect the behaviour of the object that carries them. For example, the value of the `databaseName` option is used as the name of the database to query when a search is executed against the `connection`. For a full list of such special options, see the ZOOM abstract API and the ZOOM-C documentation (links below).

References

- Section 3.2 (Connection) of the ZOOM Abstract API (http://zoom.z3950.org/api/zoom-1.3.html#3.2)
- The Connections section of the ZOOM-C documentation (http://www.indexdata.dk/yaz/doc/zoom.tkl#zoom.connections)

`ZOOM::query` and subclasses

The `ZOOM::query` class is a virtual base class, representing a query to be submitted to a server. This class has no methods, but two (so far) concrete subclasses, each implementing a specific query notation.

`ZOOM::prefixQuery`

```
class prefixQuery : public query {
public:
  prefixQuery (const char *pqn);
  ~prefixQuery ();
};
```

This class enables a query to be created by compiling YAZ's cryptic but powerful Prefix Query Notation (PQN) (http://www.indexdata.dk/yaz/doc/tools.tkl#PQF).

`ZOOM::CCLQuery`

```
class CCLQuery : public query {
public:
  CCLQuery (const char *ccl, void *qualset);
  ~CCLQuery ();
};
```

This class enables a query to be created using the simpler but less expressive Common Command Language (CCL) (http://www.indexdata.dk/yaz/doc/tools.tkl#CCL). The qualifiers recognised by the CCL parser are specified in an external configuration file in the format described by the YAZ documentation.

If query construction fails for either type of `query` object - typically because the query string itself is not valid PQN or CCL - then an `exception` is thrown.

Discussion

It will be readily recognised that these objects have no methods other than their constructors: their only role in life is to be used in searching, by being passed to the `resultSet` class's constructor.

Given a suitable set of CCL qualifiers, the following pairs of queries are equivalent:

```
prefixQuery("dinosaur");
CCLQuery("dinosaur");

prefixQuery("@and complete dinosaur");
CCLQuery("complete and dinosaur");

prefixQuery("@and complete @or dinosaur pterosaur");
CCLQuery("complete and (dinosaur or pterosaur)");

prefixQuery("@attr 1=7 0253333490");
CCLQuery("isbn=0253333490");
```

References

- Section 3.3 (Query) of the ZOOM Abstract API (http://zoom.z3950.org/api/zoom-1.3.html#3.3)
- The Queries section of the ZOOM-C documentation (http://www.indexdata.dk/yaz/doc/zoom.query.tkl)

`ZOOM::resultSet`

A `ZOOM::resultSet` object represents a set of records identified by a query that has been executed against a particular connection. The sole purpose of both `connection` and `query` objects is that they can be used to create new `resultSets` - that is, to perform a search on the server on the remote end of the connection.

The class has this declaration:

```
class resultSet {
public:
  resultSet (connection &c, const query &q);
  ~resultSet ();
  const char *option (const char *key) const;
  const char *option (const char *key, const char *val);
  size_t size () const;
  const record *getRecord (size_t i) const;
};
```

New `resultSets` are created by the constructor, which is passed a `connection`, indicating the server on which the search is to be performed, and a `query`, indicating what search to perform. If the search fails - for example, because the query uses attributes that the server doesn't implement - then an `exception` is thrown.

Like `connections`, `resultSet` objects can carry name-value options. The special options which affect ZOOM-C++'s behaviour are the same as those for ZOOM-C and are described in its documentation (link below). In particular, the `preferredRecordSyntax` option may be set to a string such as "USMARC", "SUTRS" etc. to indicate what the format in which records should be retrieved; and the `elementSetName` option indicates whether brief records ("B"), full records ("F") or some other composition should be used.

The `size()` method returns the number of records in the result set. Zero is a legitimate value: a search that finds no records is not the same as a search that fails.

Finally, the `getRecord` method returns the *`i`*th record from the result set, where *`i`* is zero-based: that is, legitmate values range from zero up to one less than the result-set size. If the method fails, for example because the requested record is out of range, it `throws` an `exception`.

References

- Section 3.4 (Result Set) of the ZOOM Abstract API (http://zoom.z3950.org/api/zoom-1.3.html#3.4)
- The Result Sets section of the ZOOM-C documentation (http://www.indexdata.dk/yaz/doc/zoom.resultsets.tkl)

ZOOM::record

A `ZOOM::record` object represents a chunk of data from a `resultSet` returned from a server.

The class has this declaration:

```
    class record {
    public:
      ~record ();
      enum syntax {
UNKNOWN, GRS1, SUTRS, USMARC, UKMARC, XML
      };
      record *clone () const;
      syntax recsyn () const;
      const char *render () const;
      const char *rawdata () const;
    };
```

Records returned from Z39.50 servers are encoded using a record syntax: the various national MARC formats are commonly used for bibliographic data, GRS-1 or XML for complex structured data, SUTRS for simple human-readable text, etc. The `record::syntax` enumeration specifies constants representing common record syntaxes, and the `recsyn()` method returns the value corresponding to the record-syntax of the record on which it is invoked.

> **Note:** Because this interface uses an enumeration, it is difficult to extend to other record syntaxes - for example, DANMARC, the MARC variant widely used in Denmark. We might either grow the enumeration substantially, or change the interface to return either an integer or a string.

The simplest thing to do with a retrieved record is simply to `render()` it. This returns a human-readable, but not necessarily very pretty, representation of the contents of the record. This is useful primarily for testing and debugging, since the application has no control over how the record appears. (The application must *not* `delete` the returned string - it is “owned” by the record object.)

More sophisticated applications will want to deal with the raw data themselves: the `rawdata()` method returns it. Its format will vary depending on the record syntax: SUTRS, MARC and XML records are returned “as is”, and GRS-1 records as a pointer to their top-level node, which is a `Z_GenericRecord` structure as defined in the `<yaz/z-grs.h>` header file. (The application must *not* `delete` the returned data - it is “owned” by the record object.)

Perceptive readers will notice that there are no methods for access to individual fields within a record. That's because the different record syntaxes are so different that there is no even a uniform notion of what a field is across them all, let alone a sensible way to implement such a function. Fetch the raw data instead, and pick it apart "by hand".

Memory Management

The `record` objects returned from `resultSet::getRecord()` are "owned" by the result set object: that means that the application is not responsible for `delete`ing them - each `record` is automatically deallocated when the `resultSet` that owns it is `delete`d.

Usually that's what you want: it means that you can easily fetch a record, use it and forget all about it, like this:

```
resultSet rs(conn, query);
cout << rs.getRecord(0)->render();
```

But sometimes you want a `record` to live on past the lifetime of the `resultSet` from which it was fetched. In this case, the `clone(f)` method can be used to make an autonomous copy. The application must `delete` it when it doesn't need it any longer:

```
record *rec;
{
    resultSet rs(conn, query);
    rec = rs.getRecord(0)->clone();
    // 'rs' goes out of scope here, and is deleted
}
cout << rec->render();
delete rec;
```

References

- Section 3.5 (Record) of the ZOOM Abstract API (http://zoom.z3950.org/api/zoom-1.3.html#3.5)
- The Records section of the ZOOM-C documentation (http://www.indexdata.dk/yaz/doc/zoom.records.tkl)

`ZOOM::exception` and subclasses

The `ZOOM::exception` class is a virtual base class, representing a diagnostic generated by the ZOOM-C++ library or returned from a server. Its subclasses represent particular kinds of error.

When any of the ZOOM methods fail, they respond by `throw`ing an object of type `exception` or one of its subclasses. This most usually happens with the `connection` constructor, the various query

constructors, the `resultSet` constructor (which is actually the searching method) and `resultSet::getRecord()`.

The base class has this declaration:

```
class exception {
public:
  exception (int code);
  int errcode () const;
  const char *errmsg () const;
};
```

It has three concrete subclasses:

ZOOM::systemException

```
class systemException: public exception {
public:
  systemException ();
  int errcode () const;
  const char *errmsg () const;
};
```

Represents a “system error”, typically indicating that a system call failed - often in the low-level networking code that underlies Z39.50. `errcode()` returns the value that the system variable `errno` had at the time the exception was constructed; and `errmsg()` returns a human-readable error-message corresponidng to that error code.

ZOOM::bib1Exception

```
class bib1Exception: public exception {
public:
  bib1Exception (int errcode, const char *addinfo);
  int errcode () const;
  const char *errmsg () const;
  const char *addinfo () const;
};
```

Represents an error condition communicated by a Z39.50 server. `errcode()` returns the BIB-1 diagnostic code of the error, and `errmsg()` a human-readable error message corresponding to that code. `addinfo()` returns any additional information associated with the error.

For example, if a ZOOM application tries to search in the “Voyager” database of a server that does not have a database of that name, a `bib1Exception` will be thrown in which `errcode()` returns 109, `errmsg()` returns the corresponding error message “Database unavailable” and `addinfo()` returns the name of the requested, but unavailable, database.

ZOOM::queryException

```
class queryException: public exception {
public:
  static const int PREFIX = 1;
  static const int CCL = 2;
  queryException (int qtype, const char *source);
  int errcode () const;
  const char *errmsg () const;
  const char *addinfo () const;
};
```

This class represents an error in parsing a query into a form that a Z39.50 can understand. It must be created with the `qtype` parameter equal to one of the query-type constants, which can be retrieved via the `errcode()` method; `errmsg()` returns an error-message specifying which kind of query was malformed; and `addinfo()` returns a copy of the query itself (that is, the value of `source` with which the exception object was created.)

Revised Sample Program

Now we can revise the sample program from the introduction to catch exceptions and report any errors:

```
/* g++ -o zoom-c++-hw zoom-c++-hw.cpp -lyaz++ -lyaz */

#include <iostream>
#include <yaz++/zoom.h>

using namespace ZOOM;

int main(int argc, char **argv)
{
    try {
        connection conn("z3950.loc.gov", 7090);
        conn.option("databaseName", "Voyager");
        conn.option("preferredRecordSyntax", "USMARC");
        resultSet rs(conn, prefixQuery("@attr 1=7 0253333490"));
        const record *rec = rs.getRecord(0);
        cout << rec->render() << endl;
    } catch (systemException &e) {
        cerr << "System error " <<
            e.errcode() << " (" << e.errmsg() << ")" << endl;
    } catch (bib1Exception &e) {
        cerr << "BIB-1 error " <<
            e.errcode() << " (" << e.errmsg() << "): " << e.addinfo() << endl;
    } catch (queryException &e) {
        cerr << "Query error " <<
            e.errcode() << " (" << e.errmsg() << "): " << e.addinfo() << endl;
    } catch (exception &e) {
        cerr << "Error " <<
            e.errcode() << " (" << e.errmsg() << ")" << endl;
```

```
        }
    }
```

The heart of this program is the same as in the original version, but it's now wrapped in a `try` block followed by several `catch` blocks which try to give helpful diagnostics if something goes wrong.

The first such block diagnoses system-level errors such as memory exhaustion or a network connection being broken by a server's untimely death; the second catches errors at the Z39.50 level, such as a server's report that it can't provide records in USMARC syntax; the third is there in case there's something wrong with the syntax of the query (although in this case it's correct); and finally, the last `catch` block is a belt-and-braces measure to be sure that nothing escapes us.

References

- Section 3.7 (Exception) of the ZOOM Abstract API (http://zoom.z3950.org/api/zoom-1.3.html#3.7)
- Bib-1 Diagnostics (http://lcweb.loc.gov/z3950/agency/defns/bib1diag.html) on the Z39.50 Maintenance Agency (http://lcweb.loc.gov/z3950/agency/) site.

Because C does not support exceptions, ZOOM-C has no API element that corresponds directly with ZOOM-C++'s `exception` class and its subclasses. The closest thing is the `ZOOM_connection_error` function described in The Connections section (http://www.indexdata.dk/yaz/doc/zoom.tkl#zoom.connections) of the documentation.

Chapter 4. YAZ C++ API

The YAZ C++ API is an client - and server API that exposes all YAZ features. The API doesn't hide YAZ C data structures, but provides a set of useful high-level objects for creating clients - and servers.

The following sections include a short description of the interfaces and implementations (concrete classes).

In order to understand the structure, you should look at the example client `yaz-my-client.cpp` and the example server `yaz-my-server.cpp`. If that is too easy, you can always turn to the implementation of the proxy itself and send us a patch if you implement a new useful feature.

Note: The documentation here is very limited. We plan to enhance it - provided there is interest for it.

Interfaces

IYazSocketObservable

This interface is capable of observing sockets. When a socket even occurs it invokes an object implementing the IYazSocketObserver interface.

```
#include <yaz++/socket-observer.h>

class my_socketobservable : public IYazSocketObservable {
   // Add an observer interested in socket fd
   virtual void addObserver(int fd, IYazSocketObserver *observer) = 0;
   // Delete an observer
   virtual void deleteObserver(IYazSocketObserver *observer) = 0;
   // Delete all observers
   virtual void deleteObservers() = 0;
   // Specify the events that the observer is interested in.
   virtual void maskObserver(IYazSocketObserver *observer,
                             int mask) = 0;
   // Specify timeout
   virtual void timeoutObserver(IYazSocketObserver *observer,
                                unsigned timeout)=0;
};
```

IYazSocketObserver

This interface is interested in socket events supporting the IYazSocketObservable interface.

```
#include <yaz++/socket-observer.h>

class my_socketobserver : public IYazSocketObserver {
```

```
    public:
     // Notify the observer that something happened to socket
     virtual void socketNotify(int event) = 0;
}
```

IYaz_PDU_Observable

This interface is is responsible for sending - and receiving PDUs over the network (YAZ COMSTACK). When events occur, an instance implementing IYaz_PDU_Observer is notified.

```
#include <yaz++/pdu-observer.h>

class my_pduobservable : public IYaz_PDU_Observable {
  public:
    // Send encoded PDU buffer of specified length
    virtual int send_PDU(const char *buf, int len) = 0;
    // Connect with server specified by addr.
    virtual void connect(IYaz_PDU_Observer *observer,
                const char *addr) = 0;
    // Listen on address addr.
    virtual void listen(IYaz_PDU_Observer *observer, const char *addr)=0;
    // Close connection
    virtual void close() = 0;
    // Make clone of this object using this interface
    virtual IYaz_PDU_Observable *clone() = 0;
    // Destroy completely
    virtual void destroy() = 0;
    // Set Idle Time
    virtual void idleTime (int timeout) = 0;
};
```

IYaz_PDU_Observer

This interface is interested in PDUs and using an object implementing IYaz_PDU_Observable.

```
#include <yaz++/pdu-observer.h>

class my_pduobserver : public IYaz_PDU_Observer {
  public:
    // A PDU has been received
    virtual void recv_PDU(const char *buf, int len) = 0;
    // Called when Iyaz_PDU_Observable::connect was successful.
    virtual void connectNotify() = 0;
    // Called whenever the connection was closed
    virtual void failNotify() = 0;
    // Called whenever there is a timeout
    virtual void timeoutNotify() = 0;
```

```
    // Make clone of observer using IYaz_PDU_Observable interface
    virtual IYaz_PDU_Observer *sessionNotify(
    IYaz_PDU_Observable *the_PDU_Observable, int fd) = 0;
};
```

Yaz_Query

Abstract query.

```
#include <yaz++/query.h>
class my_query : public Yaz_Query {
  public:
    // Print query in buffer described by str and len
    virtual void print (char *str, int len) = 0;
};
```

Implementations

Yaz_SocketManager

This class implements the IYazSocketObservable interface and is a portable socket wrapper around the select call. This implementation is useful for daemons, command line clients, etc.

```
#include <yaz++/socket-manager.h>

class Yaz_SocketManager : public IYazSocketObservable {
  public:
    // Add an observer
    virtual void addObserver(int fd, IYazSocketObserver *observer);
    // Delete an observer
    virtual void deleteObserver(IYazSocketObserver *observer);
    // Delete all observers
    virtual void deleteObservers();
    // Set event mask for observer
    virtual void maskObserver(IYazSocketObserver *observer, int mask);
    // Set timeout
    virtual void timeoutObserver(IYazSocketObserver *observer,
                                 unsigned timeout);
    // Process one event. return > 0 if event could be processed;
    int processEvent();
    Yaz_SocketManager();
    virtual ~Yaz_SocketManager();
};
```

Yaz_PDU_Assoc

This class implements the interfaces IYaz_PDU_Observable and IYazSocketObserver. This object implements a non-blocking client/server channel that transmits BER encoded PDUs (or those offered by YAZ COMSTACK).

```
#include <yaz++/pdu-assoc.h>

class Yaz_PDU_Assoc : public IYaz_PDU_Observable,
                              IYazSocketObserver {

  public:
    COMSTACK comstack(const char *type_and_host, void **vp);
    // Create object using specified socketObservable
    Yaz_PDU_Assoc(IYazSocketObservable *socketObservable);
    // Create Object using existing comstack
    Yaz_PDU_Assoc(IYazSocketObservable *socketObservable,
             COMSTACK cs);
    // Close socket and destroy object.
    virtual ~Yaz_PDU_Assoc();
    // Clone the object
    IYaz_PDU_Observable *clone();
    // Send PDU
    int send_PDU(const char *buf, int len);
    // connect to server (client role)
    void connect(IYaz_PDU_Observer *observer, const char *addr);
    // listen for clients (server role)
    void listen(IYaz_PDU_Observer *observer, const char *addr);
    // Socket notification
    void socketNotify(int event);
    // Close socket
    void close();
    // Close and destroy
    void destroy();
    // Set Idle Time
    void idleTime (int timeout);
    // Child start...
    virtual void childNotify(COMSTACK cs);
};
```

Yaz_Z_Assoc

This class implements the interface IYaz_PDU_Obserer. This object implements a Z39.50 client/server channel AKA Z-Association.

```
#include <yaz++/z-assoc.h>

class Yaz_Z_Assoc : public IYaz_PDU_Observer {
  public:
    // Create object using the PDU Observer specified
```

```
Yaz_Z_Assoc(IYaz_PDU_Observable *the_PDU_Observable);
// Destroy association and close PDU Observer
virtual ~Yaz_Z_Assoc();
// Receive PDU
void recv_PDU(const char *buf, int len);
// Connect notification
virtual void connectNotify() = 0;
// Failure notification
virtual void failNotify() = 0;
// Timeout notification
virtual void timeoutNotify() = 0;
// Timeout specify
void timeout(int timeout);
// Begin Z39.50 client role
void client(const char *addr);
// Begin Z39.50 server role
void server(const char *addr);
// Close connection
void close();

// Decode Z39.50 PDU.
Z_APDU *decode_Z_PDU(const char *buf, int len);
// Encode Z39.50 PDU.
int encode_Z_PDU(Z_APDU *apdu, char **buf, int *len);
// Send Z39.50 PDU
int send_Z_PDU(Z_APDU *apdu);
// Receive Z39.50 PDU
virtual void recv_Z_PDU(Z_APDU *apdu) = 0;
// Create Z39.50 PDU with reasonable defaults
Z_APDU *create_Z_PDU(int type);
// Request Alloc
ODR odr_encode ();
ODR odr_decode ();
ODR odr_print ();
void set_APDU_log(const char *fname);
const char *get_APDU_log();

// OtherInformation
void get_otherInfoAPDU(Z_APDU *apdu, Z_OtherInformation ***oip);
Z_OtherInformationUnit *update_otherInformation (
      Z_OtherInformation **otherInformationP, int createFlag,
      int *oid, int categoryValue, int deleteFlag);
void set_otherInformationString (
      Z_OtherInformation **otherInformationP,
      int *oid, int categoryValue,
      const char *str);
void set_otherInformationString (
      Z_OtherInformation **otherInformation,
      int oidval, int categoryValue,
      const char *str);
void set_otherInformationString (
      Z_APDU *apdu,
      int oidval, int categoryValue,
```

```
            const char *str);

    Z_ReferenceId *getRefID(char* str);
    Z_ReferenceId **get_referenceIdP(Z_APDU *apdu);
    void transfer_referenceId(Z_APDU *from, Z_APDU *to);

    const char *get_hostname();
};
```

Yaz_IR_Assoc

This object is just a specialization of Yaz_Z_Assoc and provides more facilities for the Z39.50 client role.

```
#include <yaz++/ir-assoc.h>

class Yaz_IR_Assoc : public Yaz_Z_Assoc {
  ...
};
```

The example client, `yaz-my-client.cpp`, uses this class.

Yaz_Z_Server

This object is just a specialization of Yaz_Z_Assoc and provides more facilities for the Z39.50 server role.

```
#include <yaz++/z-server.h>

class Yaz_Z_Server : public Yaz_Z_Server {
  ...
};
```

The example server, `yaz-my-server.cpp`, uses this class.

Appendix A. License

YAZ License