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/yaz++/) 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++) and a powerful, general-purpose Z39.50 proxy.

This manual covers version 0.5.

Table of Contents

Chapter 1. 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-yazconfig` 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`.

This is what you have after successful compilation:

`src/yaz-proxy`

: The YAZ Z39.50 Proxy utility. This program gets installed in your binaries directory (*prefix*`/bin`).

`lib/libyaz++.la`

: The YAZ++ library, including the ZOOM-C++ classes. 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. 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

You'll find Visual Studio project files in sub directory `win`. Open workspace `yazxx.dsw` which includes the following projects:

`yazxx.dsp`

Builds the `yazxx.dll`.

`yazclient.dsp`

Builds the sample client `yazmyclient.exe`.

`yazserver.dsp`

Builds the sample server `yazmyserver.exe`.

`yazserver.dsp`

Builds the proxy `yazproxy.exe`.

`zoomxxclient.dsp`

Builds the ZOOM C++ demo client `zoomxxclient.exe`.

By default, the include path and library path for the projects assumes that YAZ is located in `..\yaz` (i.e. same prefix as the YAZ++ source). If YAZ is in a different directory you'll have to modify the include path in `Project | Settings | C/C++ | Preprocessor | Additional include` `directories` and library path in `Project | Settings | Link | Input | Additional library path`.

Chapter 2. 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.php

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.php#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.php#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.php#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.php)

`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.php)

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` obejcts 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.php)

`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.php#zoom.connections) of the documentation.

Chapter 3. The YAZ Proxy

The YAZ proxy is a transparent Z39.50-to-Z39.50 gateway. That is, it is a Z39.50 server which has as its back-end a Z39.50 client that forwards requests on to another server (known as the *backend target*.)

The YAZ Proxy is useful for debugging Z39.50 software, logging APDUs, redirecting Z39.50 packages through firewalls, etc. Furthermore, it offers facilities that often boost performance for connectionless Z39.50 clients such as web gateways.

Unlike most other server software, the proxy runs single-threaded, single-process. Every I/O operation is non-blocking so it is very lightweight and extremely fast. It does not store any state information on the hard drive, except any log files you ask for.

Example: Using the Proxy to Log APDUs

Suppose you use a commercial Z39.50 client for which you do not have source code, and it's not behaving how you think it should when running against some specific server that you have no control over. One way to diagnose the problem is to find out what packets (APDUs) are being sent and received, but not all client applications have facilities to do APDU logging.

No problem. Run the proxy on a friendly machine, get it to log APDUs, and point the errant client at the proxy instead of directly at the server that's causing it problems.

Suppose the server is running on `foo.bar.com`, port 18398. Run the proxy on the machine of your choice, say `your.company.com` like this:

```
yaz-proxy -a - -t tcp:foo.bar.com:18398 tcp:@:9000
```

(The `-a -` option requests APDU logging on standard output, `-t tcp:foo.bar.com:18398` specifies where the backend target is, and `tcp:@:9000` tells the proxy to listen on port 9000 and accept connections from any machine.)

Now change your client application's configuration so that instead of connecting to `foo.bar.com` port 18398, it connects to `your.company.com` port 9000, and start it up. It will work exactly as usual, but all the packets will be sent via the proxy, which will generate a log like this:

```
decode choice
initRequest {
    referenceId OCTETSTRING(len=4) 69 6E 69 74
    protocolVersion BITSTRING(len=1)
    options BITSTRING(len=2)
    preferredMessageSize 1048576
    maximumRecordSize 1048576
    implementationId 'Mike Taylor (id=169)'
    implementationName 'Net::Z3950.pm (Perl)'
    implementationVersion '0.31'
}
encode choice
initResponse {
    referenceId OCTETSTRING(len=4) 69 6E 69 74
```

```
    protocolVersion BITSTRING(len=1)
    options BITSTRING(len=2)
    preferredMessageSize 1048576
    maximumRecordSize 1048576
    result TRUE
    implementationId ’81’
    implementationName ’GFS/YAZ / Zebra Information Server’
    implementationVersion ’YAZ 1.9.1 / Zebra 1.3.3’
}
decode choice
searchRequest {
    referenceId OCTETSTRING(len=1) 30
    smallSetUpperBound 0
    largeSetLowerBound 1
    mediumSetPresentNumber 0
    replaceIndicator TRUE
    resultSetName ’default’
    databaseNames {
        ’gils’
    }
    {
        smallSetElementSetNames choice
        generic ’F’
    }
    {
        mediumSetElementSetNames choice
        generic ’B’
    }
    preferredRecordSyntax OID: 1 2 840 10003 5 10
    {
        query choice
        type_1 {
            attributeSetId OID: 1 2 840 10003 3 1
            RPNStructure choice
            {
                simple choice
                attributesPlusTerm {
                    attributes {
                    }
                    term choice
                    general OCTETSTRING(len=7) 6D 69 6E 65 72 61 6C
                }
            }
        }
    }
}
```

Specifying the Backend Target

When the proxy accepts a Z39.50 client session, it determines the backend target by the following rules:

1. If the `InitializeRequest` PDU from the client includes an `otherInfo` element with OID `1.2.840.10003.10.1000.81.1`, then the contents of that element specify the target to be used, in the usual YAZ address format (typically `tcp:`*`hostname`*`:`*`port`*) as described in the Addresses section of the YAZ manual (http://www.indexdata.dk/yaz/doc/comstack.addresses.php).
2. Otherwise, the Proxy uses the default target, if one was specified on the command-line with the `-t` option.
3. Otherwise, the proxy closes the connection with the client.

Keep-alive Facility for Stateless Clients

Stateless clients such as web gateways may generate a cookie for a Z39.50 session which is sent to the proxy as part of PDUs. In this case, the proxy will keep alive its Z39.50 session to the backend target even when the connection from the client to the proxy is closed. When the client contacts the proxy again, and re-issues the same cookie, the proxy reuses the Z39.50 connection with the backend target.

There is no guarantee that the Z39.50 connection to the backend target is kept forever: the proxy will shut it down after certain idle time. So in effect, the connection from the client's point of view should be considered stateless, and the keep-alive facility should be treated only as a performance booster.

Cookies may be passed in an `otherInfo` element with OID `1.2.840.10003.10.1000.81.2`.

Query Caching

Simple stateless clients often send identical Z39.50 searches in a relatively short period of time (e.g. in order to produce a results-list page, the next page, a single full-record, etc). And for many targets, it's much more expensive to produce a new result set than to reuse an existing one.

The proxy tries to solve that by remembering the last query for each backend target, so that if an identical query is received next, it is turned into Present Requests rather than new Search Requests.

> **Note:** In a future we release will will probably allows for an arbitrary-sized cache for targets supporting named result sets.

You can enable/disable query caching using option -o.

Other Optimizations

We've had some plans to support caching of result set records, but this has not yet been implemented.

Proxy Usage

yaz-proxy

Name

`yaz-proxy` — The YAZ toolkit's transparent Z39.50 proxy

Synopsis

yaz-proxy [-a *filename*] [-c *num*] [-v *level*] [-t *target*] [-u *auth*] [-o *level*] [-i *seconds*] {*host*:*port*}

DESCRIPTION

The proxy runs stand-alone (not from `inetd`). The *host*:*port* argument specifies host address to listen to, and the port to listen on. Use the host `@` to listen for connections coming from any address.

OPTIONS

-a *filename*

Specifies the name of a file to which to write a log of the APDUs (protocol packets) that pass through the proxy. The special filename `-` may be used to indicate standard output.

-c *num*

Specifies the maximum number of connections to be cached [default 50].

-v *level*

Sets the logging level. *level* is a comma-separated list of members of the set {`fatal`,`debug`,`warn`,`log`,`malloc`,`all`,`none`}.

-t *target*

Specifies the default backend target to use when a client connects that does not explicitly specify a target in its `initRequest`.

-u *auth*

Specifies authentication info to be sent to the backend target. This is useful if you happen to have an internal target that requires authentication, or if the client software does not allow you to set it.

-o *level*

Sets level for optimization. Use zero to disable; non-zero to enable. Handling for this is not fully implemented; we will probably use a bit mask to enable/disable specific features. By default optimization is enabled (value 1).

-i *seconds*

Specifies in seconds the idle time for communication for proxy. If a connection is inactive for this long it willl be closed. Default: 600 seconds (10 minutes).

EXAMPLES

The following command starts the proxy, listening on port 9000, with its default backend target set to the Library of Congress bibliographic server:

```
$ yaz-proxy -t z3950.loc.gov:7090 @:9000
```

The LOC target is sometimes very slow. You can connect to it using yaz-client as follows:

```
$ yaz-client localhost:9000/voyager
Connecting...Ok.
Sent initrequest.
Connection accepted by target.
ID     : 34
Name   : Voyager LMS - Z39.50 Server
Version: 1.13
Options: search present
Elapsed: 7.131197
Z> f computer
Sent searchRequest.
Received SearchResponse.
Search was a success.
Number of hits: 10000
records returned: 0
Elapsed: 6.695174
Z> f computer
Sent searchRequest.
Received SearchResponse.
Search was a success.
Number of hits: 10000
records returned: 0
Elapsed: 0.001417
```

In this test, the second search was more than 4000 times faster than the first, because the proxy cached the result of the first search and noticed that the second was the same.

The YAZ command-line client, `yaz-client`, allows you to set the proxy target as part of the Initialize Request using option `-p`. For example, to connect to Index Data's target you could use:

```
yaz-client -p indexdata.dk localhost:9000/gils
```

OtherInformation Encoding

The proxy uses the OtherInformation definition to carry information about the target address and cookie.

```
  OtherInformation   ::= [201] IMPLICIT SEQUENCE OF SEQUENCE{
    category           [1]   IMPLICIT InfoCategory OPTIONAL,
    information        CHOICE{
      characterInfo            [2]  IMPLICIT InternationalString,
      binaryInfo               [3]  IMPLICIT OCTET STRING,
      externallyDefinedInfo    [4]  IMPLICIT EXTERNAL,
      oid                      [5]  IMPLICIT OBJECT IDENTIFIER}}
--
  InfoCategory ::= SEQUENCE{
      categoryTypeId   [1]   IMPLICIT OBJECT IDENTIFIER OPTIONAL,
      categoryValue    [2]   IMPLICIT INTEGER}
```

The `categoryTypeId` is either OID 1.2.840.10003.10.1000.81.1, 1.2.840.10003.10.1000.81.2 for proxy target and proxy cookie respectively. The integer element `category` is set to 0. The value proxy and cookie is stored in element `characterInfo` of the `information` choice.

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.

Yaz_Proxy

This object is a specialization of Yaz_Z_Assoc and implements the YAZ proxy.

```
#include <yaz++/proxy.h>

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

The proxy server, `yaz-proxy-main.cpp`, uses this class.

Appendix A. License