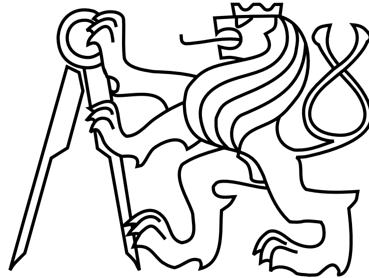
Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Computer Science and Engineering



Master's Thesis

Client-server Communication Protocol for CellStore Database Engine

Klient-server komunikační protokol pro databázový stroj CellStore

Bc. Martin Plicka

Supervisor: Ing. Jan Vraný

Study Programme: Electrical Engineering and Information Technology

Field of Study: Computer Science and Engineering

May 3, 2009

Aknowledgements

Fenk juu all

Declaration

I hereby declare that I have completed this thesis independently and that I have listed all the literature and publications used.
I have no objection to usage of this work in compliance with the act §60 Zákon č. 121/2000Sb. (copyright law), and with the rights connected with the copyright act including the changes in the act.

Prague, May 3, 2009 ....

Abstract

The aim of this work is to describe whole process leading to design and implementation of client-server protocol for CellStore database engine. It contains discussion about selection of communication mechanism and data representation. Is also describes features and common guidelines of API which was being created. All source code of both client and server part, including API documentation, are appended on CD. In appendix, there is also exhausting instruction for running CellStore database with remote access ability and compiling client library and client programs.

Abstrakt

Cílem této práce je popsat celý proces návrhu a implementace klient-server protokolu pro databázový stroj CellStore. Obsahuje diskusi výběru komunikačního mechanismu a reprezentace dat. Dále uvádí a popisuje vlastnosti a společná pravidla vytvářeného API celé klientské knihovny. Na přiloženém CD jsou mimo jiné umístěny veškeré zdrojové kódy jak klientské tak serverové strany včetně dokumentace API. V příloze jsou také uvedeny informace pro zprovoznění databáze CellStore, včetně služby pro vzdálený přístup, a informace pro kompilaci klientské knihovny a na ní založených programů.

Contents

1 Introduction **1**

2 Analysis **3**

2.1 CellStore description . . . 3
2.1.1 XMLDB interface . . . 3
2.1.2 DOM interface . . . 3
2.1.3 SELF interface . . . 4
2.2 Existing object & XML database engines and their interfaces . . . 4
2.2.1 eXist . . . 4
2.2.2 GemStone/S - GemBuilder for C . . . 4
2.3 Choosing the network data representation . . . 4
2.3.1 Text-based protocol and data representation . . . 5
2.3.2 Binary protocol . . . 5
2.3.3 Remote Procedure Call . . . 5
2.3.4 Conclusion . . . 6

3 RPC programming in C and Smalltalk/X **9**

3.1 XDR description . . . 9
3.2 RPC in C - rpcgen . . . 10
3.2.1 Server . . . 11
3.2.2 Client . . . 11
3.2.3 Building and running . . . 11
3.3 RPC in Smalltalk/X . . . 13
3.3.1 Server . . . 14
3.3.2 Client . . . 15
3.3.3 Various Smalltalk/X RPC patches . . . 16
3.4 Other languages . . . 17

4 Protocol **19**

4.1 Mapping object world to non-objected language . . . 19
4.1.1 Handling the objects remotely . . . 19
4.1.2 Exceptions . . . 20
4.2 Message & control flow . . . 20
4.2.1 Single-call operations . . . 22
4.2.2 Multi-call operations . . . 22
4.3 Large file transfer . . . 22
4.3.1 Data transfer . . . 23

5 Realization **27**
5.1 Server side - Smalltalk/X . . . 27
5.1.1 Multi-threaded RPC server - RPCMTServer class . . . 27
5.1.2 Server core - RemoteServer class . . . 28
5.1.3 Session object holder - SessionStorage class . . . 31
5.1.4 EnhancedSessionStorage class . . . 31
5.1.5 Large files transfer - SocketJob class . . . 32
5.1.6 Special read stream - NetReadStream class . . . 34
5.1.7 Example - XML import . . . 35
5.1.8 CellStore service - RPCService class . . . 35
5.1.9 Remote server launcher - RemoteServerWizard class . . . 36
5.2 Client side - C library . . . 36
5.2.1 API guidelines . . . 37
5.2.2 Library structure . . . 37
5.3 Sample Smalltalk/X client . . . 38

6 Testing **41**
6.1 Unit testing tools . . . 41
6.1.1 Smalltalk/X - SUnit . . . 41
6.1.2 C Language - Check library . . . 41
6.2 Test coverage . . . 42
6.2.1 Server side - Smalltalk/X . . . 42
6.2.2 Client side - Check . . . 42
6.3 Performance measures . . . 42

7 Conclusion **43**

Bibliography **45**

A List of used abbreviations **47**

B Installation instructions **49**
B.1 CellStore installing . . . 49
B.2 Client library compiling . . . 50
B.3 Demo applications . . . 51

C How to implement protocol **53**

D CD content **55**

List of Figures

2.1 CellStore architecture [1] . . . 7

3.1 Sample C server procedure . . . 11
3.2 Sample C client code . . . 12

4.1 Sample reply definition . . . 20
4.2 Remote access protocol, capitalized labels represent RPC messages . . . 21
4.3 File upload protocol in detail . . . 25

5.1 Brief capture of client-server implementation architecture . . . 28
5.2 Server - basic structure . . . 29
5.3 Code of RPC operation with standard reply behavior . . . 30
5.4 Code of RPC operation after enhancement . . . 30
5.5 SocketJob hierarchy . . . 33
5.6 Structure of the sample Smalltalk/X client . . . 39

6.1 Smalltalk/X SUnit tool . . . 42

B.1 Demo applications: sample output . . . 52

Chapter 1

Introduction

CellStore [1] is a native XML database engine which was born at Department of Computer Science & Engineering at Czech Technical University in Prague, Faculty of Electrical Engineering. It's being developed both for educational and research purposes. It's entirely written in Smalltalk on Smalltalk/X platform running either on Windows or Linux systems.

Currently, it works as an embedded database so it can be run as local library only. But this is not enough or real application deployment or development. For these purposes we want to develop client-server based protocol to enable remote access to various CellStore database interfaces.

CellStore is formerly XML database but since new object interface was developed, database can handle any object data. Protocol should reflect that and be able, beyond the former XML database access, work with arbitrary objects (in some kind as Gemstone/S does).

(TODO: some information about target library - C language, portability etc. etc.)

Chapter 2

Analysis

At first, this chapter summarizes CellStore database engine abilities and interfaces which can be used and which should be accessed remotely. Latter section provides information about existing XML and object-oriented databases and focuses on their remote access possibilities. This information will be used for discussion about features we want the CellStore client-server protocol should support. Also this can bring us the inspiration how to design whole protocol or client API. In the last section, there are confronted several variants for data encoding with protocol requirements.

2.1 CellStore description

CellStore has layered architecture which allows access the database from many levels. Figure 2.1 shows CellStore architecture in its current form. You can see many access points symbolized by black dots at the top of the figure.

One of important layers is, in the middle, a new SELF engine. SELF [2] is a prototype-based object-oriented language which is even simpler than Smalltalk. SELF data model allows to store any data as objects without concerning about low level representation. On the top of SELF layer, there are several interfaces which provide access in many ways, including XMLDB API [3] and OODB (object-oriented database) API. In fact, OODB API only allows to perform SELF message passing. All important APIs are described in following sections.

At first, we will describe CellStore XMLDB interface which is the most significant for XML databases. In latter sections, some other interfaces, including SELF, are mentioned.

2.1.1 XMLDB interface

XMLDB interface was developed to fit needs for accessing XML-based data in database engines. (FIXME)

2.1.2 DOM interface

(FIXME)

2.1.3 SELF interface

(FIXME)

2.2 Existing object & XML database engines and their interfaces

2.2.1 eXist

The project called eXist [4] has started by Wolfgang Meier in 2000. It's written in Java. I supports XQuery 1.0 and XPath 2.0. According to [4], there are three ways how to run the database engine.

- In a Servlet Context. The database is deployed as part of a web application in servlet. It's the default setting.
- Embedded in an Application. In embedded mode, the database is basically used as a Java library, controlled by the client application. It runs in the same Java virtual machine as the client, thus no network connection is needed and the client has full access to the database.
- Stand-alone Server Process. eXist runs in its own Java virtual machine. It provides either XML-RPC, WebDAV or REST-style HTTP API for remote access. It use Jetty as a web server providing those interfaces.

The latter case is the most important for us as a comparison for this thesis. XML-RPC API offers all expected XMLDB features such as document retrieving and storing, manipulation with collections, querying and result retrieving. (FIXME: focus on xml-rpc implementation and xmldb api mapping.)

2.2.2 GemStone/S - GemBuilder for C

(FIXME)

2.3 Choosing the network data representation

CellStore API has some significant properties which should be considered during the protocol design. Concrete requirements are summarized in following enumeration.

1. Work with SELF API consist of many simple operations. Our protocol should be bandwidth effective in the light of this fact.
2. CellStore is still in development stage so protocol must be versatile and expandable to cover all requirements appearing in future.

3. Protocol should be implementable easily with various programming environments. Although C library will be the main implementation, making the protocol portable will be a plus.

4. It must support various data transfer, including long binary streams for resource upload.

There were several possibilities of data representation to decide between. Every of them is facing the requirements its own way.

2.3.1 Text-based protocol and data representation

Text-based representation is used in various widely spread network protocols, including FTP, SMTP of HTTP. It can be debugged simply because it's human readable. Also, implementation is simple. Problems come with binary data transfer. To keep data represented by readable characters, various coding methods are used. For example, MIME coding is used to encode binary data or non-ASCII characters in SMTP protocol (SMTP was designed for 7 bit ASCII). This brings some data overhead. Although CellStore does not support binary resources at present, protocol should be designed with respect to this possibility.

2.3.2 Binary protocol

Designing our own binary protocol would probably bring us the most data-effective result. All message codes, operation statuses and other control values would have their byte length as low as possible. But on the other hand, bad design can make further protocol extension impossible. For example, coding operation number into one byte would become a problem in future because it would limit the number of different operations available. Also, some complex data structures that weren't under consideration before can bring problems later.

2.3.3 Remote Procedure Call

Remote Procedure Call (RPC) is a common name for communication mechanisms allowing programmers to call program code remotely from another machine the same way as local functions. Several variations were developed like ONC RPC[1], XML-RPC, Corba etc. Their common property is data representation in platform independent format. This format can be either binary or text-based (mostly XML-like text). XML format brings visible overhead, thus it's not effective for sending of many short messages. Binary representation seems as a good option.

[1] Open Network Computing Remote Procedure Call

2.3.4 Conclusion

(TODO: cleanup, discuss pros and cons)

First two options discussed below have significant disadvantages. Text-based protocol misses efficiency du to the binary data coding. Furthermore if XML formatting was used, efficiency in case of many short calls would fall down.

The main disadvantage of binary protocol can be problem with extension. CellStore is still in development and nobody knows what features will be wanted to provide in future. Our protocol must be versatile in this point of view so custom built binary protocol is not a good idea in this stage of project.

We chose RPC solution because it is versatile and allows easy protocol extension in case of further changes and new features addition. From various kinds of RPC we finally chose ONC RPC [5] (also known as Sun RPC). It uses binary data representation (will be described later) so it seems to be bandwidth effective. Also, great advantage is that both SunRPC client and server are already implemented in Smalltalk/X distribution. Chapter 3 provides information about RPC programming in C and Smalltalk/X in detail.

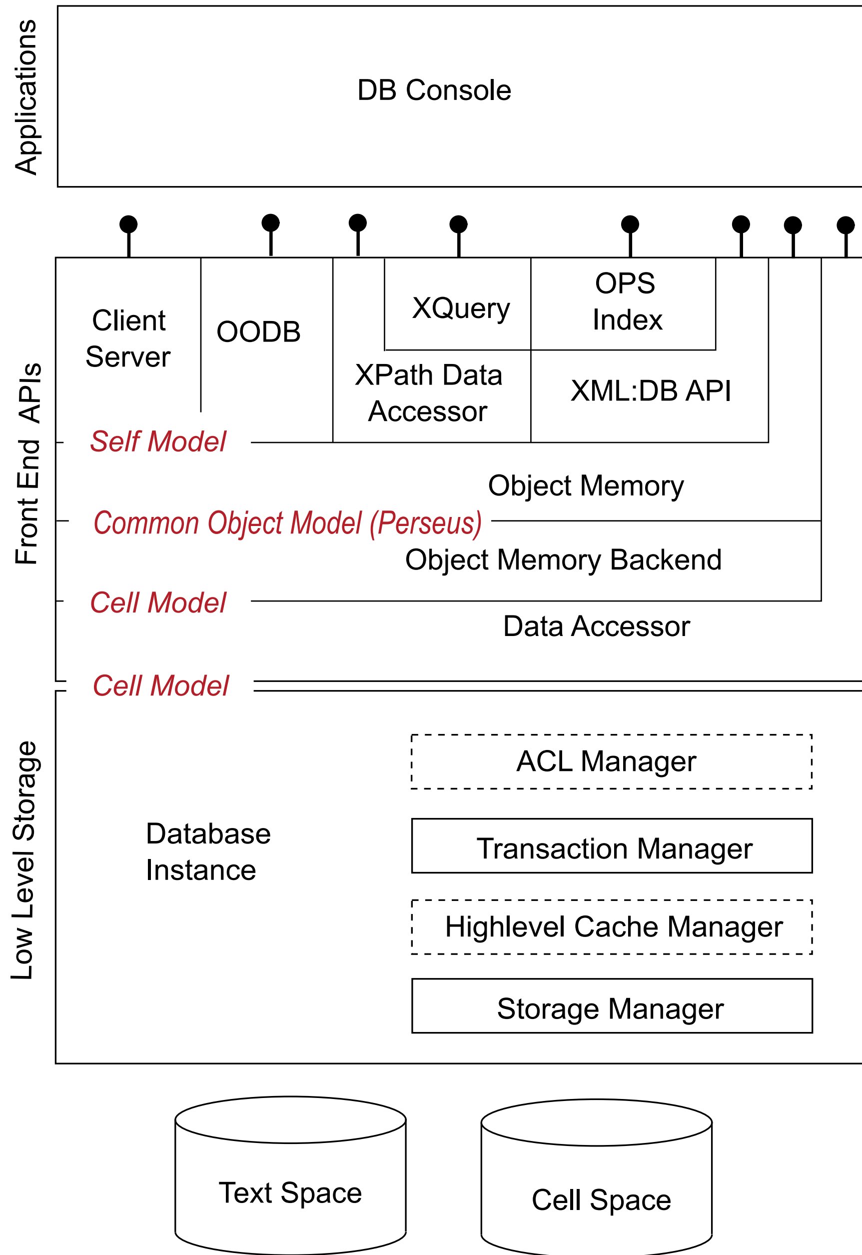


Figure 2.1: CellStore architecture [1]

Chapter 3

RPC programming in C and Smalltalk/X

(FIXME: cleanup)

From options discussed in previous chapter, we finally chose ONC RPC [5] (also called SunRPC). ONC RPC is a protocol formerly developed by Sun Microsystems [6] for their NFS protocol. It uses XDR [7] as an interface and data representation definition language. It identifies application by integer number which must by unique on server system. Remote procedures are identified by procedure number and program version number. Various transfer protocols, including TCP and UDP, can be used for connection. Remote port number of service can be obtained via portmap service (which, in fact, is also RPC application) or input directly during connection initialization.

3.1 XDR description

XDR [7] means for eXternal Data Representation, OSI presentation layer implementation, used with ONC RPC [5]. Using that, it's possible to transfer various data information between two interconnected systems running on different platforms.

XDR description uses own syntax similar to C language. It contains signatures of all procedures which can be run remotely, followed with protocol version and unique (should be) protocol identificator. Following example is taken from [8]. I had to make some changes because Smalltalk/X implementation lacks some syntax elements.

```
/* msg.x: Remote msg printing protocol */
typedef string stringArg<>;
program MESSAGEPROG {
   version PRINTMESSAGEVERS {
     void null(void) = 0;
     int PRINTMESSAGE(stringArg message) = 1;
   } = 1;
} = 200001;
```

In this example, one program `MESSAGEPROG` with number 200001 and one version, 1, is described. It has two procedures. The first one, `null`, numbered with 0, is intended for connection testing purposes and should be present. The second, `PRINTMESSAGE` takes exactly one argument - a string - and returns integer value.

3.2 RPC in C - rpcgen

For C language, there is a powerful tool called `rpcgen` that reads out the XDR definition mentioned above and generates both client and server stubs, conversion routines and application templates.

Useful commands are:

```
#generate XDR routines and common headers
rpcgen msg.x -N

#also generates sample client code, redirect it to file
rpcgen msg.x -Sc -N > msg_client.c

#generates sample server code
rpcgen msg.x -Ss -N > msg_server.c

#generates makefile template
#(must be edited - must contain list of files to be compiled)
rpcgen msg.x -Sm -N > Makefile
```

These commands are recommended to run in given order since Makefile generation automatically adds common headers and XDR routines.

The most important switches of `rpcgen` command are listed bellow.

- `-Sc` switch forces `rpcgen` to generate sample client code to standard output
- `-Ss` generates sample server code.
- `-Sm` generates quite versatile Makefile.
- `-N` option allows "new" style of programming. It means multiple arguments and easier RPC routines call so arguments are not needed to be passed as structures anymore. Default mode (without -N option) is for backward compatibility.
- `-M` generates multi-thread safe code. It's not used in this project yet.

```
int * printmessage_1_svc(stringArg message,  struct svc_req *rqstp) {
   static int  result;

   printf("Message: %s\n", message);
   result = 1;

   return &result;
}
```

Figure 3.1: Sample C server procedure

3.2.1 Server

Sample server code in `msg_server.c` file contains code stubs for each procedure (and version) declared in XDR definition. These procedures will be executed when particular RPC call is received. In our example we can simply implement the procedure for `PRINTMESSAGE` call as shown on figure 3.2.1. You can see that the procedure name is assembled from RCP procedure name and program version number. This code will print the received message to standard output. Integer value of 1 will be returned back to client.

Similar to this, we need to implement all exported procedures in all versions declared in XDR definition. Note that we do not have to add no code to `null` procedure stub since it really does nothing. It's aimed for connectivity tests and thus it should be declared in every RPC protocol and implemented.

3.2.2 Client

Generated sample client code in `msg_client.c` file shows the way of calling the remote procedures. I modified it a bit to make it simpler and the result is shown on figure 3.2.2

All we need to make it working is to implement shared procedures on both server and client side. We can use sample codes as a good start point. When using generated Makefile, we need add our own filenames to it.

3.2.3 Building and running

`rpcgen` tool can generate nice Makefile using the arguments discussed at the beginning of this section. The only thing we have to do is add all source file names to the variable definitions at the beginning of the Makefile. For our example, these variables should look like following.

```
CLIENT = msg_client
SERVER = msg_server

SOURCES_CLNT.c = msg_client.c
SOURCES_CLNT.h =
```

```
#include "msg.h"

int main (int argc, char *argv[]) {
   if (argc < 3) {
      printf ("usage: %s server_host message\n", argv[0]);
      exit (1);
   }
   CLIENT *clnt;
   int  *result;

   //creating connection handler, TCP transport is selected
   //MESSAGEPROG and PRINTMESSAGEVERS are defined in msg.h file
   clnt = clnt_create (argv[1], MESSAGEPROG, PRINTMESSAGEVERS, "tcp");
   if (clnt == NULL) {
      clnt_pcreateerror (argv[1]);
      exit (1);
   }

   //calling the procedure, name also contains program version
   result = printmessage_1(argv[2], clnt);
   if (result == (int *) NULL) {
      clnt_perror (clnt, "call failed");
   }
   else printf("Reply: %d\n", *result);

   clnt_destroy (clnt);
}
```

Figure 3.2: Sample C client code

```
SOURCES_SVC.c = msg_server.c
SOURCES_SVC.h =
SOURCES.x = msg.x

TARGETS_SVC.c = msg_svc.c   msg_xdr.c
TARGETS_CLNT.c = msg_clnt.c   msg_xdr.c
TARGETS = msg.h msg_xdr.c msg_clnt.c msg_svc.c
```

These variables are used for identifying all dependencies and link proper modules together. Also we must modify `RPCGENFLAGS` variable to make rpcgen using new style of coding everytime it is called.

```
RPCGENFLAGS = -N
```

To compile both client and server, use `make` command. To get client only, run `make msg_client` or `make msg_server` for server, respectively.

Running the server is easy. Just type

```
./msg_server
```

Server will serve any request and can be terminated by keyboard interupt (e.g. Ctrl+C). Client has two command line arguments, as seen in example code. Run

```
./msg_client localhost "Your message"
```

Message is printed on server side and integer of value 1 is returned and printed by client.

We can also test connectivity using the `rpcinfo` tool:

```
#call procedure 0 in program with number 200001 using TCP
rpcinfo -t localhost 200001

#call procedure 0 in program with number 200001 using UDP
rpcinfo -u localhost 200001
```

Whole source code used as an example in this section can be found on appended CD or in SVN repository of the project.

3.3 RPC in Smalltalk/X

All classes usable for RPC programming are defined within SunRPC namespace. Nice tutorial can be also found on [9]. This introduction is partialy inspired by that.

Smalltalk/X's RPC implementation is very simple to use. All things that are needed to do is:

1. Subclass both *SunRPC::RPCClient* and *SunRPC::RPCServer* classes.
2. Write XDR description. Assign as a return value of `#xdr` class method for both server and client class.
3. Assign TCP/UDP ports to be used on server side. Implement instance method `portNumbers` that returns a collection containing all port that can be used.
4. Implement instance methods according to XDR definition on server side.
5. Implement instance methods for our own client API, using calls to procedures defined by XDR. This is optional since we can call remote procedures using default `#operation:arguments:` method (see sample provided later).

3.3.1 Server

3.3.1.1 XDR description

SunRPC::XDRParser omits some syntax features from C rpcgen so XDR file from [8] had to be modified slightly. Since I patched XDRParser a bit, it is now allowed to use names for procedure arguments (non-patched parser fails having argument names given). See later section for details.

XDR is assigned as a class method to server (and also to client class).

```
xdr
^'
/* msg.x: Remote msg printing protocol */
typedef string stringArg<>;
program MESSAGEPROG {
   version PRINTMESSAGEVERS {
     void null(void) = 0;
     int PRINTMESSAGE(stringArg message) = 1;
   } = 1;
} = 200001;
'
```

3.3.1.2 Assigning ports

Next step is to implement instance method `portNumbers` (server side only) returning collection containing all ports that can be used. These ports are tried one by one. When socket is successfully opened, current port is registered to Portmapper. Note that Smalltalk/X has its own Portmapper implementation which can be run automaticaly so it is not needed to have system Portmapper installed.

Simplest way:

```
portNumbers
   ^ (11000 to: 11100)
```

3.3.1.3 Implementing methods

We need to implement every method described in XDR file (as our server instance method). Methods have one argument - collection of all RPC procedure parameters, as they are described in XDR definition. These methods are automatically called when received by server.

```
PRINTMESSAGE:args
  Transcript showCR: (args at:1).
  ^ 1.
```

The `null` procedure is already implemented in superclass.

3.3.1.4 Controlling the server

Following code shows the way how to control our new server (expecting our server class is named *EchoServer*).

```
"start using TCP (default)"
EchoServer start

"start using UDP"
EchoServer startUDP

"stop, the most polite way I found"
EchoServer serversRunning first release

"program definition is stored in class object"
"after altering xdr method we need to force parsing it again"
EchoServer initDefinitions
EchoClient initDefinitions
```

Startup will fail when the program number is already registered in portmapper. To remove previous registration, use following command (as root) which will remove registration for program with number 200001 and version 1.

```
rpcinfo -d 200001 1
```

After sucessful server startup it's posible to use client from C section. Note that we used TCP protocol for our C client so server has to be inicialized using TCP. Smalltalk implementation does not run using both UDP a TCP at the same time.

Our implementation of remote protocol uses new *SunRPC::RPCMTServer* which is able to process more connections at once using separate Smalltalk processes. It is very simple enhancement of standard *RPCServer* class and also has the same API. It's discussed in detail in section 5.1.1.

3.3.2 Client

Client is also very easy to implement. XDR assignment is done the same way as in case of server. Tricky solution can be refering to server specification:

```
xdr
    ^ EchoServer xdr.
```

Default client also has universal methods to invoke remote procedures but in general, it's better to implement our own API. Following method does the message sending.

```
printMessage: string
   "prints message on remote screen"

   ^ self operation: #PRINTMESSAGE arguments: (Array with: string).
```

3.3.2.1 Controlling the client

There exist several ways how to connect. *RPCClient* has methods to connect without asking the portmap service. Following code will connect using portmapper query and call `PRINTMESSAGE` procedure which will print given message on screen (in case of C server) or Transcript (in case of Smalltalk server). Afterwards the return value is printed by client.

```
| client reply |
client := EchoClient toHost: 'localhost'
reply := client printMessage: 'hello world'
client close.
Transcript showCR: reply.
```

3.3.3 Various Smalltalk/X RPC patches

During the development process, I found that Smalltalk/X RPC implementation lacks some important features which were used in our protocol implementation.

- XDR parser is not compatible with `rpcgen`. It does not allow names of arguments in procedure definitions. But `rpcgen` requires them to generate C code successfully. It's annoying to convert the XDR definition file to Smalltalk RPC compatible form each time it is modified. Due to this, I modified the XDR parser in *SunRPC::XDRParser* class. The `#procedureDef` method represents procedure definition token in recursive descent implementation of top-down parsing model. I modified this method to make argument names optional (identifier token is read when found). Now parser can read unmodified content of `service_rpc.x` file.
- XDR coder in *SunRPC::XDRCoder* class had not have array encoding and decoding implemented. Several RPC procedures in our protocol use arrays as a return values. XDR parser in Smalltalk/X recognizes array definitions but XDR coder had not been able to use it. Array binary representation is described in RFC 4506 [7]. Encoding and decoding are performed in `#encodeArray:type:with:` and `#decodeArrayWithType:` method, respectively. Coder now supports both variable and fixed length array.
  - For fixed size arrays, always the same number of values are expected on stream.
  - In case of variable size, the array is prepended by 4 Byte value containing the number of values in oncoming array.
- Smalltalk/X SunRPC implementation is able to process at most one TCP connection or UDP datagram at once. Since we want to use TCP connection to be active all the time client is operating, our server has to be able to process more than one connection at once. This issue has been solved by former SunRPC server modification which brings multi-threaded processing. It's described in detail in chapter 5.1.1.

3.4 Other languages

(TODO: discuss this section placement)

ONC RPC is wide spread standard so several RPC implementations can be found. In consequence, our new protocol can be ported to various platforms. Several implementations for different languages exist. During a short search, I found following:

- Remote Tea, pure Java implementation of ONC RPC.
  `http://remotetea.sourceforge.net/`.
- rpcc - Python ONC RPC Compiler, together with demo RPC implementation seem usable.
  `http://www.cs.umd.edu/~gaburici/rpc/` and
  `http://svn.python.org/view/python/trunk/Demo/rpc/`
- (FIXME: add some implementations for windows C)

I have not tested them. However, RPC and XDR are quite simple to implement so porting might not be a big problem.

Chapter 4

Protocol

The aim of this chapter is to describe problems with remote protocol specification. First section comes with discussion about problems with mapping of object-oriented world to non-object environments, including the XDR interface. Next sections bring information about protocol message and control flow, including special behavior during large files upload.

4.1 Mapping object world to non-objected language

Object-oriented programming languages have some advantages but all of them cannot be used over the internet connection directly. There are two main issues when discussing object techniques mapping to procedural language:

- Object references. Remote handling must protect object from deletion caused by garbage collector on server side.
- Exceptions. These cannot be simply raised over network connection since client side might not have particular characteristic to handle them. For example, C language itself has no flexible exception mechanism.

4.1.1 Handling the objects remotely

All database operations are done on server side so we need a mechanism for referencing the objects we process. I used unique unsigned 4 Byte integer values which are transferred in RPC calls or replies, respectively, to identify remote objects.

There is no type control in protocol. All object references have the same type. The reason is simple. Target platform (e.g. C language) might not know inheritance and categorizing the references may be tough and not flexible because in some cases, remote operations can return reference to various objects. Due to this, type control is performed on server. Alternatively, user can ask for object class name, but this operation return string because class types cannot be identified by enumerated value (it's difficult to determine all used classes), so this method is useful for debugging only.

Due to the fact that garbage collector is present in Smalltalk/X, these references cannot be simply equal to object memory addresses. When garbage collecting process is run, objects addresses will probably change every time, but our references, stored on client, won't.

Also, we need to protect the objects from erasing. References from client don't affect garbage collecting process so these objects would be deleted if there weren't another reference (and there aren't for almost all objects). Both these problems are solved using SessionStorage class. It is described in chapter 5.1.3.

4.1.2 Exceptions

In object-oriented languages, errors are represented by exceptions, tiny objects which hold information about error that has occurred. In our protocol, we need to transfer the information they are holding towards the client side. For this purpose, exceptions are represented by enumerated values. Every exception raised on server is caught and converted to its proper code number. These codes are specified in `cs_status` enumerated value in `service_rpc.x` file. Finally, this code, together with the error message contained in exception, is packed into the reply and sent to client.

To acquire this way of error handling, every RPC procedure reply (except of `null` procedure) is a union value. First item, `status` contains error code or `CS_STATUS_OK` value (equal to 0) signaling no error. If operation finishes properly (status is equal to 0), next item, `value` will be present and will contain operation result. In case of failure, string item named `description` contains error message. Sample reply definition is shown on figure 4.1. Note that items in switch cannot have the same identificator because `rpcgen` tool generates code which cannot be compiled then.

```
union cs_reply_int switch (cs_status status) {
  case CS_STATUS_OK:
    unsigned int value;
  default:
    cs_arg_string description;
};
```

Figure 4.1: Sample reply definition

4.2 Message & control flow

Whole protocol is quite simple and straightforward. It can be described in following sequence (also, see figure 4.2).

1. Ask Portmapper for application port. This option is recommended since server may use different port every time it is run. Application is identified by program number.

2. Connect. Client have to open new TCP connection to port retrieved from portmap service.

3. Send HELLO request. This is not mandatory, but recommended. When maximum connection is reached, server responds with `CS_ERROR_TOO_MANY_CONNECTIONS` error code to the first procedure being called. For this reason, it's recommended for client libraries to use this first call to get informed and disconnect before any operation attempt is done. In future, HELLO procedure may serve for more purposes.

4. Call operations as they are requested.

5. Disconnect.

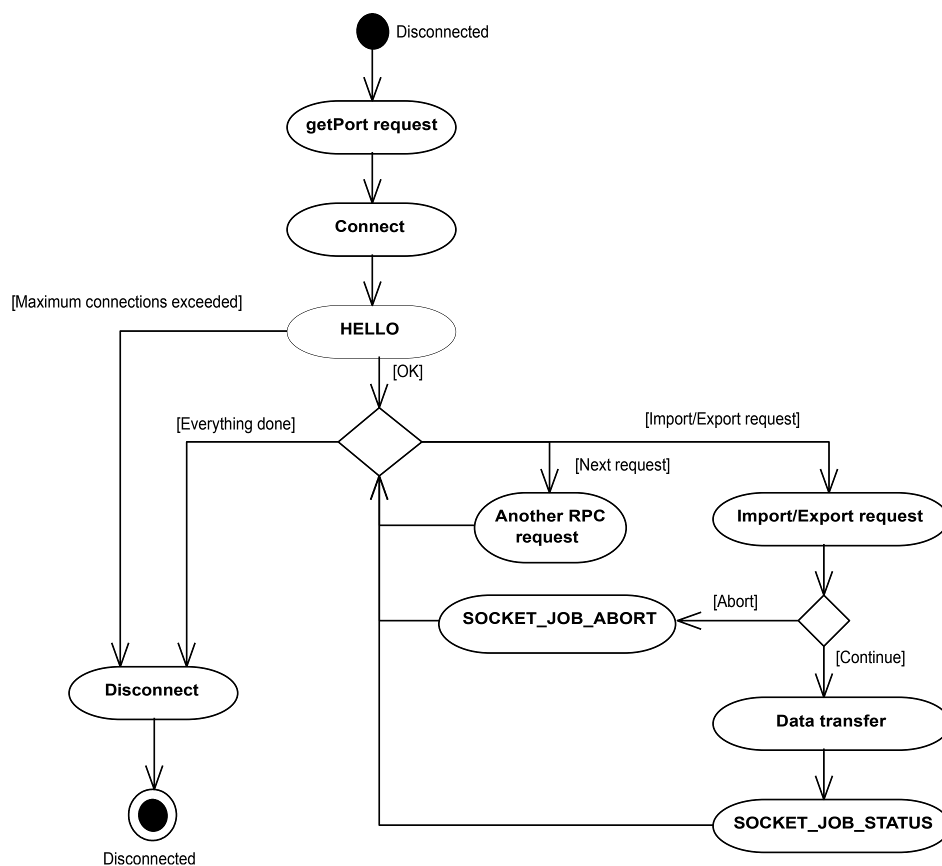


Figure 4.2: Remote access protocol, capitalized labels represent RPC messages

As seen on figure 4.2, there are two possibilities how to perform an operation in connected state:

- singe-call operations,
- multi-call operations.

4.2.1 Single-call operations

Almost all operations related to current CellStore features are single-call. They're performed as the message is delivered and decoded. In final API implementation, these operations are mapped 1:1 to the protocol definition and they're atomic.

4.2.2 Multi-call operations

Multi-call operations consist of more than one messages and other action. To reach operation final state successfully, client has to perform several operations in given order.

At present, the only multi-call operations are large files import and export actions. They're called "Socket jobs" since they use another connection (represented by network socket). To achieve proper behavior, several auxiliary operations are needed. Large files transfer is described closely in chapter 4.3, concrete implementation is explained in chapter 5.1.5.

4.3 Large file transfer

When not mentioned, all issues in this section will be explained on case of file imports. Download operations can be thought similarly.

Transfer of large files can be problem for RPC based protocol. It's neither possible nor acceptable to store whole file in a memory to encode it into XDR stream and receive it at the opposite side. Even if we divide the file into chunks and send them separately in many RPC requests, some temporary memory (RAM or hard drive) is needed to store these chunks together before processing.

Since XML readers or writers work on streams, it's memory efficient to parse the file or generate output, respectively, directly on the network stream without storing to memory or temporary file. Both XML reader or writer is invoked by calling the only method having a stream as an argument. As a consequence, the whole processing operation is atomic. During the RPC call, client is in blocking state and waits for reply so he is unable to send or receive. As a result, import and export operations cannot be implemented in one RPC call.

As explained above, data transfer cannot be atomic from the protocol point of view. As seen on figure 4.2, multi-call operation have 3 phases:

1. initialization,
2. data transfer and processing,
3. checking status.

First and last phase are implemented as RPC calls. Initialization does all required actions to prepare server side for data transfer. For example `XMLDB_UPLOAD_RESOURCE` procedure call announces XML database resource upload. Checking status is done with `SOCKET_JOB_STATUS` message.

Note that only one Socket job is allowed at the same time. Once new initiation procedure is called, previous job is aborted.

The middle action (data transfer) has several possibilities how to solve it:

- Receive whole data in small chunks via RPC calls and save it to the temporary file or memory, then process it on server side. Memory is not good solution since the file may be large and virtual machine has limited memory. File seems as a suitable emergency option but better solutions follow.
- Receive data in small chunks via RPC requests and push them into server-local pipe which is directed to the parser running in another process. This require additional data processing at server side.
- Use RPC connection stream to serve data to the importer. This solution can be difficult to implement because all possible error states have to be under consideration to ensure that RPC connection won't be broken during the unexpected error. Also, it must be save to return the stream to the RPC processing mode (to receiver another request). This option may be unacceptable on some platforms and client implementations since we need to get to the RPC connection socket descriptor.
- Use separate TCP stream to serve data. This option is better in relation to RPC server and error handling is easier.

The last option was selected because it's easy to implement on both client and server side. It versatile enough to provide platform for various features which can be implemented in future. To allow process to be aborted in any time by another RPC call easily, importer should run in separate process. See realization notes in chapter 5.1.5 for details.

4.3.1 Data transfer

Protocol of data transfer depends on the direction of the transfer. Download jobs are simpler so they will be described at first.

4.3.1.1 Data download

When download job is initiated (for example, by calling `XMLDB_DOWNLOAD_RESOURCE` RPC procedure), all the client has to do is:

1. Connect to server address and given port. Port number is retrieved in reply of initiation procedure call.
2. Read out all data until the EoF[1] flag is detected (remote side closes the connection). Using C sockets, EoF is detected when blocking read return no data. If export operation fails on server, remote connection will be closed immediately.
3. Close the opened socket.

After data transfer is finished, client must ask server for operation status. This is done with `SOCKET_JOB_STATUS` call. Procedure will contain `CS_STATUS_OK` status value if everything is done. Otherwise, code value representing exception which caused the error is returned. Also, reply contains message extracted from the exception.

[1] End of File

4.3.1.2 Data upload

For uploading data from client to server, several modifications had to be applied. It is expected that XML parser reads data until the end of file (or stream, respectively) is reached. Converted to network socket, it means that connection have to be closed to reach the EoF signalization.

But that behavior is not acceptable since client needs to wait for ACK[2] message. It's important for client to ensure that import is finished before upload job status is checked to avoid non-consistent states. The best way to make client waiting is to use blocking read operation. So connection can be closed just after ACK message is received.

To achieve requested behavior, we must emulate EoF signaling other way. Data being sent to server are divided into blocks. Before every block is written to socket, client has to send header first. This header is 4 Byte integer value in network format announcing the length of the oncoming block. Header with zero value indicates reaching the end of input file.

Block size can vary during the transfer, server is able to process every length. But too short blocks are not recommended because server operations can sometimes want to read long data. In this case, server must merge the data from more chunks divided with headers. This brings small performance fall. Also data transfer efficiency will fall when short blocks are used because more header data are sent. In general, block of sizes in hundreds of Bytes are long enough.

After sending the zero header, client waits for ACK message, 4 Byte integer, which has currently value of 7777 (but this is not important since client does not any value check). After receiving, the connection is closed. Then, client does `SOCKET_JOB_STATUS` procedure call to check the upload status.

Structure of data communication is shown on figure 4.3.

[2] Acknowledgment

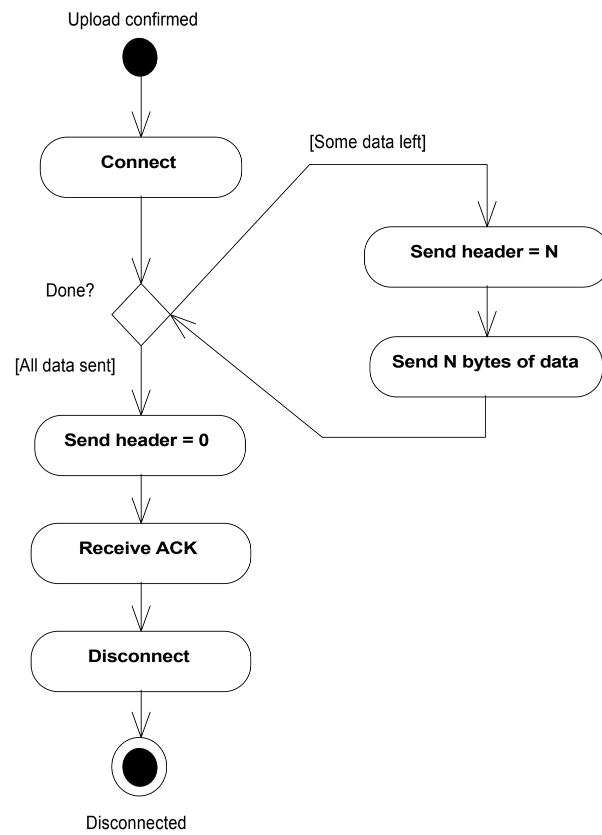


Figure 4.3: File upload protocol in detail

Chapter 5

Realization

This chapter describes the most important issues which had to be solved during the design and implementation of remote protocol. There also discussed few patches of original Smalltalk/X SunRPC implementation code to make some features fit our needs.

Figure 5.1 provides brief look at whole architecture of client-server protocol. Server side is represented by Smalltalk code. Client side is primarily done in C language, as explained in former chapters.

5.1 Server side - Smalltalk/X

As seen on architecture figure 5.1, RPC based protocol depends (not ultimately) on portmap service which registers running service and provides information about port the service is running on. Portmapper daemon often runs on Unix systems or can be installed. Also, Smalltalk/X installation contains its own portmapper so it's used when the system one is not found.

Whole server layout is described on figure 5.2. Server consist of many important components represented by concrete classes. Most of them are described in this section.

(FIXME: strange square in figure)

5.1.1 Multi-threaded RPC server - RPCMTServer class

Smalltalk/X comes with SunRPC server implementation which is able to process at most one TCP connection at once (or sequential UDP requests). For our purpose, where long term connections from clients are expected, we need to handle more connections simultaneously. On that account, I enhanced the basic *SunRPC::RPCServer* with features described below.

I created *SunRPC::RPCMTServer* as a child class of former *RPCServer* class. It inherits most of its functionality and adds ability to process more connections at once. Note that this feature, by principle, works with TCP connection only.

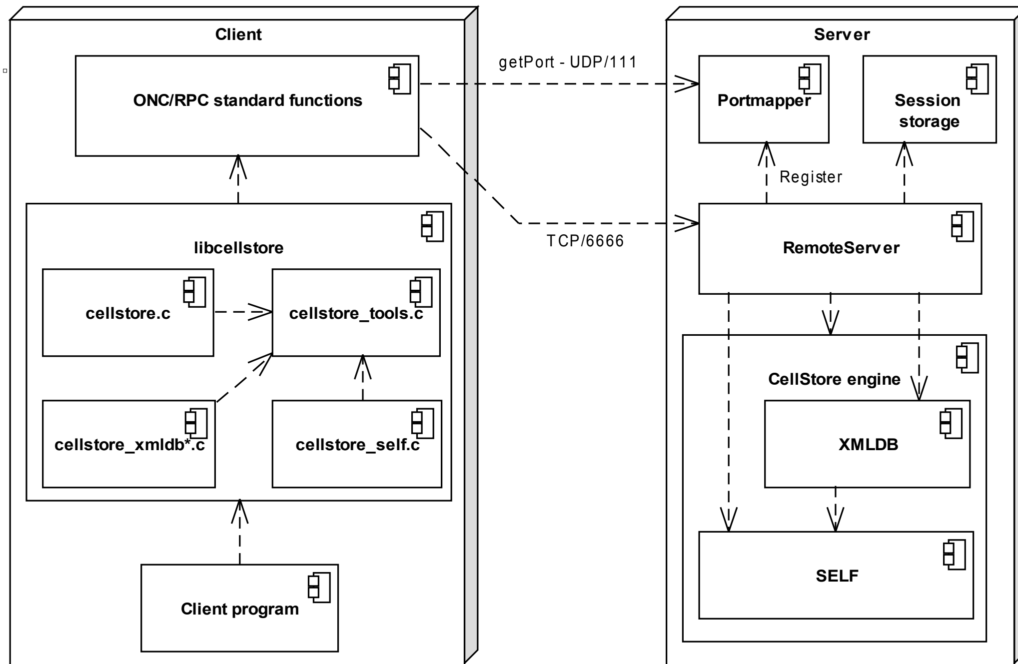


Figure 5.1: Brief capture of client-server implementation architecture

Once connection is accepted by main server process[1], the new process is created and the whole connection is handled in this new process. During the fork, also new server object is created and assigned to current connection. Child server objects are connected with their parent via instance variable. This link is used to share some resources, e.g. reference to CellStore database instance.

Once forked, process calls former RPCServer code. In fact, changes are minimal. To make basic protection against overloading, number of simultaneous connections is limited. When maximum amount of child processes is reached, all new connections will be closed by the main server process immediately.

Few further changes have been made in concrete implementation in *RemoteServer* class which is discussed in another chapter. These changes are specific for current client-server protocol and aren't related to general multi-threaded modification.

5.1.2 Server core - RemoteServer class

This object exist for each current remote connection and represents instance of RPC session. In fact, it inherits *SunRPC::RPCMTServer* class and implements all exported operations.

Main object purposes are:

[1]Note that term "process" stands for Smalltalk/X process and means something different than Unix process. Smalltalk virtual machines have their own scheduler and memory management. Smalltalk processes are rather similar to threads because they have access to whole virtual machine memory (as opposite to Unix processes which have their own virtual memory space)

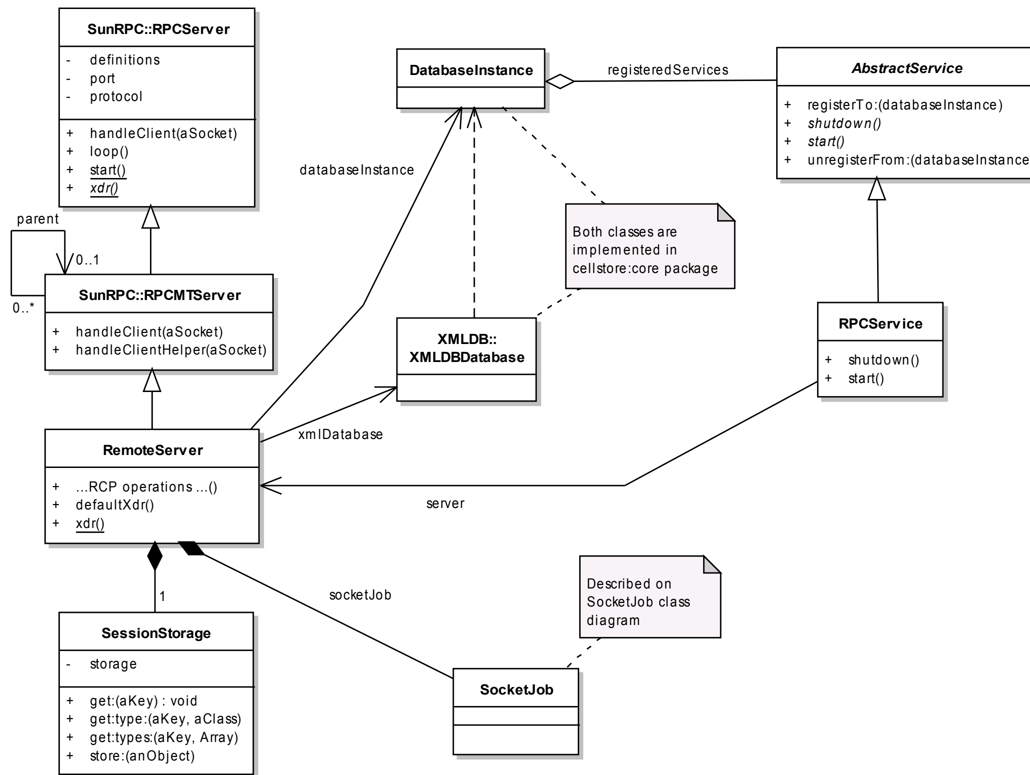


Figure 5.2: Server - basic structure

- *RemoteServer* overloads some of *RPCServer* and *RPCMTServer* code to get better behavior and allow simpler implementation of new functions.
- Defines XDR description of protocol. `#xdr` method was improved to load current XDR definition from file located in SVN[2] working copy or local file. This feature was used during development and ensures that the most recent code is used.
- Implements all exported RPC operations.

5.1.2.1 Operations

As shown in RPC introduction chapter, all RPC calls are simply mapped to the methods with the name equal to procedure name defined in XDR definition file. As noted in chapter 4, all result values of RPC calls consist of at least two values. The first one is always the status code. The second one depends on status. When operation end properly, it contains reply value.

In Smalltalk/X implementation of RPC, struct-like (struct, union) reply values are expected as dictionaries. For example, reply to `DOM_LIST_COUNT` procedure call might look like code on figure 5.3.

[2]Subversion. Currently, the best way to get whole server code is to load it directly from Subversion repository

```
DOM_LIST_COUNT: args
  |reply|
  reply := (self sessionStorage getNodeList:(args at:1)) length.

  ^ Dictionary new
      at: 'status' put: #CS_STATUS_OK;
      at: 'value' put: reply.
```

Figure 5.3: Code of RPC operation with standard reply behavior

```
DOM_LIST_COUNT: args
  |reply|
  reply := (self sessionStorage getNodeList:(args at:1)) length.

  ^ reply.
```

Figure 5.4: Code of RPC operation after enhancement

But this example only shows solution in case that operation finished correctly. If operation raises an exception, this will not be caught and not stored into the reply. In addition, server process will be aborted due to the exception. So every exception must be caught and the reply value must be modified according to protocol specification.

This is done in overloaded `#performOperation:withArguments:` method. To distinct each exception type from others, every significant class that describes some important error has `#cellstoreIdentify` method defined. This method returns unique status code. These codes correspond with definition of `cs_status` enumerated value in XDR definition. Once caught, the identification method is called and the return value is set as a operation status. Default value, returned by base *Object* class, is `CS_ERROR_OTHER`.

This enhancement also allows programmers to return operation value only. The whole reply value is packed into the reply in `#performOperation:withArguments:` so the resulting code of each operation may look simpler as shown on figure 5.4.

Sometimes it's necessary to construct whole reply in method which implements the operation. For this purpose, reply value packing mechanism was modified to detect whether whole reply is received from concrete operation method already. To distinguish this situation, operation must return reply as *ReplyDictionary* object. This class is private in *RemoteServer*. If return value with this class identity is detected, the reply is passed unchanged as is.

To identify its own errors within the client-server protocol project, *RemoteServer* also contains another private class, *CustomError*. This is an exception class providing several status codes to identify various custom errors. It also implements `#cellstoreIdentify` method to get the proper code for RPC reply value.

The server core also contains some modifications for testing and debugging purposes. These modifications are discussed in chapter 6.2.1 (Testing).

5.1.3 Session object holder - SessionStorage class

To handle garbage collector issues mentioned in chapter 4.1.1, I created a structure which stores references to objects being used during the session and assigns them unique reference numbers. This structure is implemented in *SessionStorage* class and its enhanced version (discussed later) in *EnhancedSessionStorage*.

Basic method `#store:` adds given object to storage and return unique reference. Backwards, `#get:` method return object based on given reference or throws an exception if none with such reference exists.

Storage can also provide basic type control. Using the `#get:type:` or `#get:types:` method, server-side operations can check that they're obtaining object with proper class or subclass. Note that this is the only way how to ensure that proper object is processed since client side has no type control. Checking for type protects client application programmers from "method not understand" errors caused by their mistakes. For example, when `XMLDB_DOWNLOAD_XML` routine is called, it ensures that the object given is either XML-like resource or XQuery result. Many methods for concrete cases were created as wrappers for universal methods mentioned above. For instance, `#getResource:` calls

```
self get:aKey type:XMLDB::Resource
```

If `aKey` variable does not contain reference to object of *XMLDB::Resource* class or its subclass, an exception with `CS_ERROR_OBJECT_TYPE_MISMATCH` code is raised.

5.1.4 EnhancedSessionStorage class

Various CellStore interfaces return fresh objects every time they are called. This property is not a plus and can be reduced easily since almost objects handled by session storage are "object proxies". These objects point to an object in SELF memory space. When called three times, the same method always return new instance of object proxy pointing to the same object in SELF memory space. Standard behavior of *SessionStorage* is to always assign new unique ID when `#store:` method is called. As a consequence, client application cannot do simple identity compare based on remote object reference equality.

Using the property of the object proxies, *EnhancedSessionStorage* is able to detect that the object to be stored has the same content - cell pointer (object proxies with the same cell pointer refer to the same object in SELF memory). It uses cross-directed identity dictionary having objects as keys and their remote reference as values. This dictionary can be used to quickly search for already stored object. When detected, old reference value is returned. In this case, client is able doing quick compare by reference value compare only.

Unfortunately, this feature couldn't be used in current remote server configuration. Not all objects handled by remote server are object proxies. For example, DOM objects are stored outside SELF memory and their content does not say anything about their identity. For example, two elements with the same name and same attributes are not

identical. If we treat them as identical, we won't can change content of only one of them. Also DOM object cloning wouldn't work.

For this reason, *EnhancedSessionStorage* is not used until DOM operations are re-implemented to direct access operations on SELF storage. At this time, this seems as the only precondition for allowing enhanced storage usage.

To allow client to compare identities and values, two operations were created. `OBJECT_EQUAL` compares remote objects by their content, `OBJECT_IDENTICAL` by their identity, respectively.

5.1.5 Large files transfer - SocketJob class

Protocol for large files transfer is described in section 4.3. Implementation on server side can vary independently on protocol specification.

Running import or export operations in main process (in fact, process associated with concrete client connection) is dangerous since when launched, importer cannot be stopped any other way than closing the connection. But we cannot guarantee that all importers or exporters will react to this behavior properly. When separate process is used, whole job can be simply cancelled by killing this particular process from another RPC call.

SocketJob class is a root of a hierarchy responsible for large object transfer. Whole hierarchy is illustrated on figure 5.5. *SocketJob* implements methods common to both directions of data transfer. The most significant are:

- `#portNumbers` (class)
  Specifies the range of available ports that can be used for incoming connection.
- `#fork:name:`
  Runs given block in a new process. Kills previously initialized process if there is any.
- `#abort`
  Aborts already running job.
- `#status`
  Returns current job status. Retrieves error code if some exception has occurred or `CS_SOCKET_JOB_WORKING` if process is running yet. Otherwise, it returns `CS_STATUS_OK`.

On next level of hierarchy, there are generic object for uploading and downloading, *UploadJob* and *DownloadJob*, respectively. They implement whole code which is common to all jobs and prepare interface for implementing concrete upload or download job. Significant methods on this level are (explained on *UploadJob* class, *DownloadJob* has similar behavior).

- `#prepare`
  Creates new socket and makes it waiting for incoming connection. It returns a port number of the listening socket. This method is called during the file transfer initialization RPC call (e.g. `XMDLB_UPLOAD_RESOURCE`).

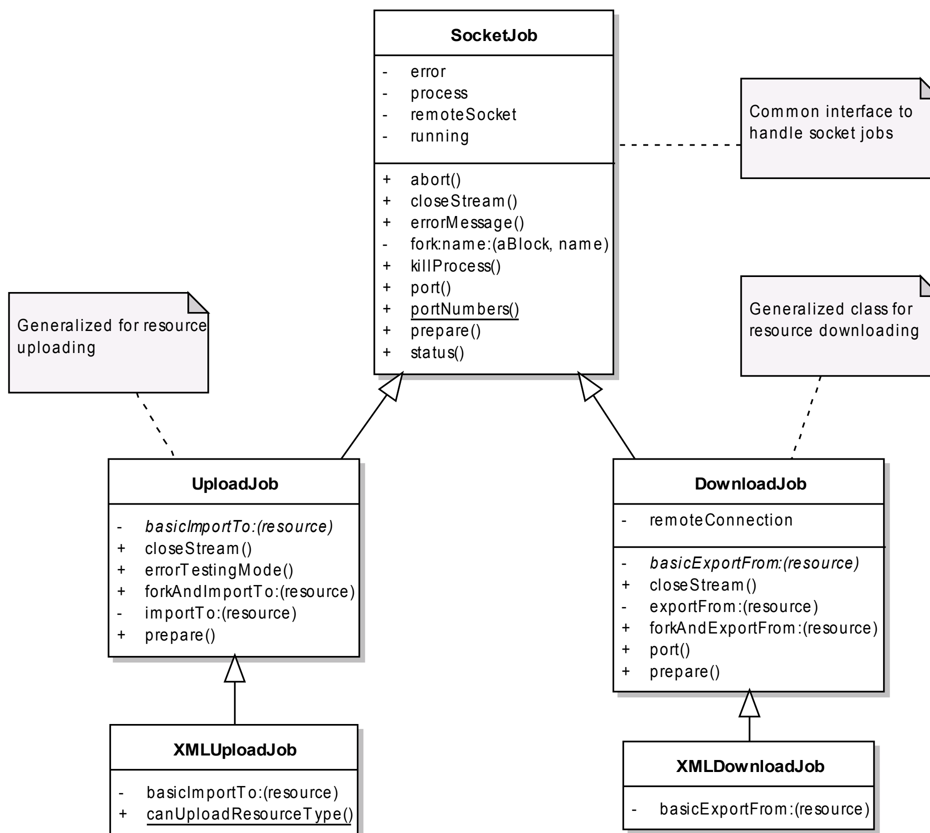


Figure 5.5: SocketJob hierarchy

- `#forkAndImportTo:`
  Creates new process and runs `#importTo:` in it.

- `#importTo:`
  Waits for incoming connection from client and runs importing process in `#basicImportTo:` method which has to be implemented in every particular class and does operations specific for concrete resource being uploaded. I note that these method shouldn't be called directly from main process.

When implementing new process, all programmer has to do is to inherit subclass from *UploadJob* class and implement `#importTo:` method which imports data from object local stream (or socket) to resource gives as an argument. Similarly, make subclass of *DownloadJob* and implement `#ExportFrom:` method.

In following sections, details about data transfer are discussed.

5.1.5.1 Downloads

Download process can be realized very easily. In Smalltalk/X, network socket (realized in Socket class) behaves as any other stream. So it can be passed to XML parser or

writer, respectively, instead of the file stream or any other stream. In case of upload, situation is slightly different and it's described below.

In this case, new socket which is used for data transport has always to be initialized. So at first, request message is sent. This operation prepares new socket and other required things. Then it replies with port number the socket is listening on. After that, client can connect to remote server address and given port and send or receive data, respectively.

When data is transfered or connection is closed unexpectedly, client sends `SOCKET_JOB_STATUS` message to check previous operation status. In case of failure of any kind at server side, reply will contain proper exception code and error message.

5.1.5.2 Uploads

Upload processes should use NetReadStream class instead of standard Socket to allow client wait until all server operations are finished. This prevent client from asking for operation status before it is finished.

5.1.6 Special read stream - NetReadStream class

(FIXME: needs heavy rewriting)

To allow import operations using the remote connection socket the same way as in case of download actions, an enhanced network stream (*NetReadStream* class) was created. It allows EoF signaling while connection is not closed. As discussed before, data being sent from client are divided into blocks. Each block is prepended with special header created by one integer value indicating the block length. When no data is available, zero value is sent to announce end of file.

At the server side, *NetReadStream* reads out the header and remembers the number of bytes available for receiving. Then, it maps all read operations to socket methods and decrements the value according to read bytes. Seek operations, similarly to standard Socket class, are disabled. When available bytes are exhausted, next header is read. If more than announced data is required, reading is divided to more phases. Bytes already announced are read. Then new header is read and finally, remaining bytes are received (by doing a recursive call on itself). Both parts of data are merged into one collection and returned. After zero value in header is read, *NetReadStream* indicates end of file.

As noted before, client side waits for ACK message after all data is sent. When XML parser reaches the end of file, the job is finished successfully. For this purpose, `#ack` method is implemented. It sends ACK message to client. Then, the connection can be closed.

The most important methods are:

- `#initSocket`
  Creates listening socket where connection from client will be expected. Returns port number.

- `#accept`
  Waits for incoming connection.

- `#readBlockHeader`
  Reads out the header (4 Byte integer in network format). If zero header is received, it will set EoF flag.

- `#ensureHeaderRead`
  If no more bytes are left announced, reads next header. Otherwise, does nothing.

- `#ack`
  Sends ACK message to client and closes connection.

- `#close`
  This method does nothing. Parsers can close stream as they reach the end of file. But we do not want them to do this.

- (TODO: note about read functions and their behavior, chunking to read long data etc...)

5.1.7 Example - XML import

1. Client asks for upload into given resource by calling `XMLDB_UPLOAD_RESOURCE` operation. It passes resource reference as the only argument.

2. Server prepares new socket for upload, makes it listening on free port and forks process with connection accept and parsing code. All these actions are established by *XMLUploadJob* object.

3. Server sends back the port number of listening socket as a reply to previous request.

4. Client connects to remote port and writes whole file into socket by the way specified before. In the same time, *XMLUploadJob* accepts connection and starts parsing.

5. Client waits for acknowledge message (in blocking read). It is sent by the server when file is completely parsed.

6. After receiving, it closes the upload connection and asks for import result using the `SOCKET_JOB_STATUS` procedure.

5.1.8 CellStore service - RPCService class

RPCService class represents CellStore service. This service has interface unified with other services that can be available to CellStore. Service can be registered easily, by evaluating

```
CellStore::RPCService new registerTo:instance.
```

`instance` instance variable is expected to be CellStore *DatabaseInstance* object. Service is run when the `#startServices` method of *DatabaseInstance* is called. CellStore service adds another instance variable, `server` which points at main (listnening) *RemoteServer* instance. It implements two methods only:

- `#start` - code for service startup,
- `#shutdown` - code for service shutdown and cleanup.

5.1.9 Remote server launcher - RemoteServerWizard class

Launching the database instance together with services cannot be proceeded with only one command. To make running the server as easy as possible, RemoteServerWizard class was written.

Actually, it's only wrapper for several commands and provides few tasks with database instance. All routines are implemented as class methods.

- `#start`
  Create clean database with remote server.
- `#stop`
  Stop the remote server and remove database.
- `#toggle`
  Toggle between running and stopped state. This method is assigned to the button which is added to Launcher toolbar during the *service_ rpc* package load.
- `#console`
  Show database console.
- `#loadData`
  Load testing data from SVN repository into database.
- `#databaseInstance`
  Return database instance.
- `#xmlDatabase`
  Return the XML database main object.

The code included in this class is a very good reference to get familiar with database instance and remote server handling.

5.2 Client side - C library

(FIXME)

Client library in C is based on code generated by `rpcgen` tool. Whole library is organized into modules and most of them correspond to some subset of exported CellStore functionality.

5.2.1 API guidelines

Almost all operations in *libcellstore* follow similar coding habits. Every function which invokes remote procedure call returns operation status code. This status code represents relevant exception code or contains 0 (`CS_STATUS_OK`). All return values (object references, XML strings, string arrays, etc.) are passed via pointer arguments.

Few datatypes were defined in `cellstore.h` as mappings over XDR defined types to provide the same type naming conventions along the *libcellstore* library. These are

- `CellStoreStatus` - operation status code,
- `CellStoreSession` - pointer to RPC client connection,
- `CellStoreObject` - reference number to remote object,
- (TODO: make it complete)

(TODO:Example declaration and code use)

5.2.2 Library structure

(FIXME: finish modules description)

- `cellstore.c`
  Main module, its responsible for connection handling (including direct collection opening using XMLDB URI) and basic operations such as remote objects handling.
- `cellstore_xmldb.c`
  Main module for XMLDB interface. It provides basic XMLDB operations, including collection openning.
- `cellstore_xmldb_collection.c`
  This module provides interface for manipulating with XMLDB collection properties. In fact, all operations correspond with CellStore XMLDB interface.
- `cellstore_xmldb_resource.c`
  Actions with XMLDB resources.
- `cellstore_xmldb_download.c`
- `cellstore_xmldb_upload.c`
- `cellstore_xmldb_query.c`
- `cellstore_dom.c`
- `cellstore_oodb.c`
- `cellstore_tools.c`
  Provides private functions for whole *libcellstore*.

- `cellstore_error.c`
  Private functions for work with error statuses and global error message variable. As mentioned before, all functions returns operation status code. Also, they set up global error message string which can be accessed via this module.

- `cellstore_errmsg.c`
  This module is automatically generated by cascade of several scripts (`xdr_parse.sh`, `xdr_parse.awk` and `xdr_parse.sed`) from XDR description in `service_rpc.x` file. It contains the only function. This function writes out the textual description of given error code and appends the error message. Its made of one huge switch statement containing all status codes from XDR file. Textual description is taken from comment in *cs_ status* enumerated type declaration. Every value of enum is expected to be in form as following:

  ```
  CS_ERROR_OTHER = 1, /** Unsorted Error **/
  ```

  It will produce code similar to following:

  ```
  case CS_ERROR_OTHER: fprintf(stderr,"[Unsorted Error] %s\n", msg);
  break;
  ```

5.3 Sample Smalltalk/X client

To illustrate guidelines for creating of Smalltalk client, I have made simple example code of this client. Since Smalltalk client is not important for us, it's not complete at all. But it can show the way of implementing the interface in object-oriented languages.

Smalltalk client object layout of XMLDB interface should match with XMLDB API [3]. Figure 5.6 shows layout of sample client code illustrating guidelines for Smalltalk/X client creating. Other parts as SELF interface and DOM also should be like their server side complement.

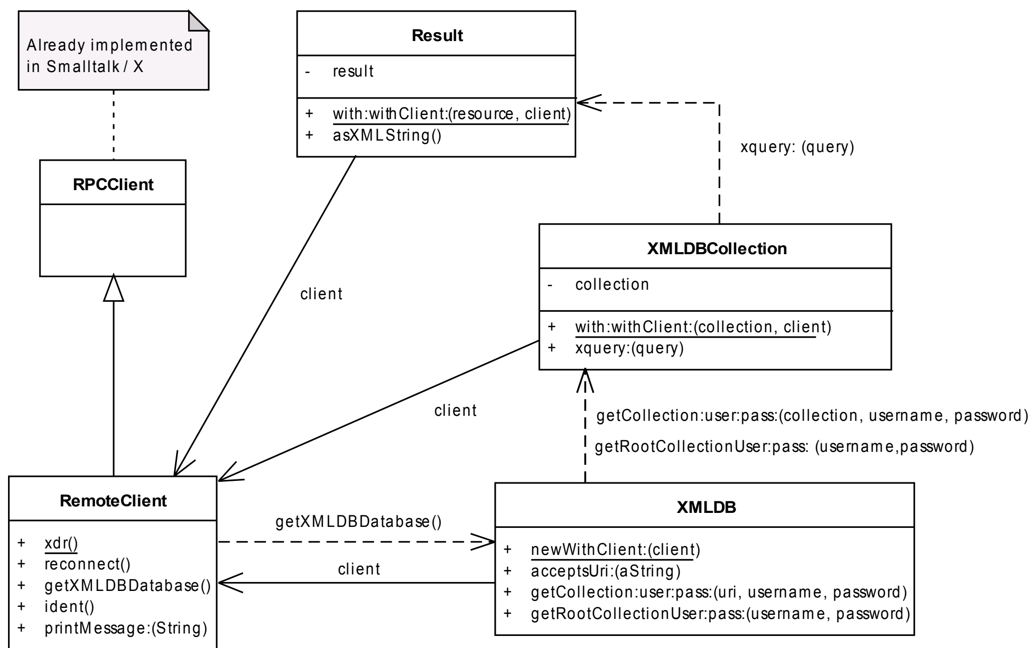


Figure 5.6: Structure of the sample Smalltalk/X client

Chapter 6

Testing

6.1 Unit testing tools

Unit testing i a crucial principle of so-called Extreme programming or Test-driven development techniques.

Former approach uses short development iterations and unit tests as a way of checking of the code correctness. Unit tests also allow programmers to do heavy refactoring.

Using the latter approach, the test covering all features which have to be implemented is written. Then, using tools like SUnit, tests are run and the program is being developed until the tests passes all cases.

6.1.1 Smalltalk/X - SUnit

Smalltalk/X have its own implementation of SUnit tool. It runs tests divided into categories or, in new version, into packages. When some tests fails, developer is allowed to debug appropriate test case in Smalltalk debugger and correct the code. Then he is able to rerun the test which have failed before.

6.1.2 C Language - Check library

Check [10] is one of existing unit testing frameworks for C language. It has a simple interface for defining unit tests. Tests are run in a separate address space, so Check can catch both assertion failures and code errors that cause segmentation faults or other signals.

In contrast to SUnit, Check needs whole API to be declared before running the tests because it has to compile them first. Without functions and data types being declared, there is no way to compile them. Small example of unit tests using Check framework is shown in later section.

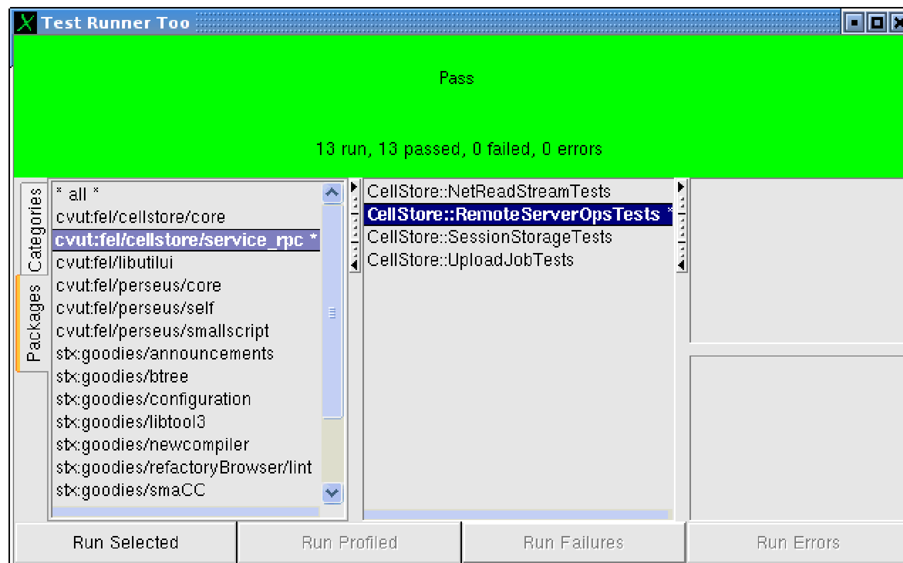


Figure 6.1: Smalltalk/X SUnit tool

6.2 Test coverage

6.2.1 Server side - Smalltalk/X

Server side was primarily tested for proper handling of exceptions and return values. Most of remote operation tests were inspired by client side tests. RPC server operations are called from point in which the RPC reply is completely assembled and all exceptions have been caught and processed in the reply. So the tests allow to check proper return values of error codes.

XDR coder can encode enum values from both numbers and symbols. Numbers are used when error code is calculated, for example in XMLDB::XMLDBException class. To make testing as easy as possible, all server operations convert numeric values into symbolic ones. To achieve that, the dictionaries representing enum values conversions are stored into RemoteServer class object during the initial XDR parsing. These dictionaries are represented by DOMTypes and StatusCodes class variables. During the evaluation, server operation use #symbolicValueFor: method to convert numeric value to symbolic one.

In addition, separate tests covering features of SessionStorage, NetReadStream and asynchronous upload/download jobs have been written.

6.2.2 Client side - Check

(FIXME)

6.3 Performance measures

(TODO: compare running time of directly called ops to remotely called ops)

Chapter 7

Conclusion

Blah, blah, blah. I really enjoyed this work.

Bibliography

[1] CellStore project web-site.
http://cellstore.felk.cvut.cz/.

[2] David Ungar and Randall B. Smith. Self: The power of simplicity. In *OOPSLA '87: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 227–242, New York, NY, USA, 1987. ACM.

[3] XML:DB Initiative for XML Databases.
http://xmldb-org.sourceforge.net/.

[4] eXist - Database Deployment.
http://exist-db.org/.

[5] RFC 1831 - RPC: Remote Procedure Call Protocol Specification Version 2.
http://tools.ietf.org/html/rfc1831.

[6] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems: Concepts and Design.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1994.

[7] RFC 4506 - XDR: External Data Representation Standard.
http://tools.ietf.org/html/rfc1831.

[8] Dave Marshall: Remote Procedure Calls (RPC).
http://www.cs.cf.ac.uk/Dave/C/node33.html.

[9] SunRPC Remote Procedure Call Implementation.
http://www.exept.de:8080/doc/online/english/programming/TOP.html

[10] Check: A unit testing framework for C.
http://check.sourceforge.net/.

Appendix A

List of used abbreviations

ACK Acknowledgment

API Application Programming Interface

DOM Document Object Model

EoF End of File

FTP File Transfer Protocol

HTTP Hypertext Transfer Protocol

MIME Multipurpose Internet Mail Extensions

NFS Network File System

OODB Object-Oriented DataBase

ONC RPC Open Network Computing Remote Procedure Call

RPC Remote Procedure Call

SMTP Simple Mail Transport Protocol

TCP Transmission Control Protocol

UDP User Datagram Protocol

XDR eXternal Data Representation

XML eXtensible Markup Language

XMLDB XML DataBase

Appendix B

Installation instructions

Following chapter gives detailed information about installing of CellStore with remote access ability and compiling client library.

B.1 CellStore installing

The way described bellow is the simplest one from many various options. Currently, it's recommended.

1. Download and install Smalltalk/X with Subversion support enabled. Use download page at `http://smalltalk.felk.cvut.cz/`. There is either Debian repository setting information or tarball with whole Smalltalk/X installation. Although CellStore and remote server should work on Windows systems, they've been primarily tested on GNU/Linux. Therefore using of Linux machine is recommended.

2. Ensure you have Subversion installed on your system. Simply type `svn` command and see whether it runs.

3. Run Smalltalk/X and assemble new image.

4. Load CellStore, RPC Service and others from Subversion repository. Evaluate following code in Workspace.

   ```
   Smalltalk loadPackage: 'cvut:fel/cellstore/service_rpc'
   ```

   Package location is mapped into subversion repository so *service_rpc* package and its dependencies (including CellStore core) will be downloaded into your image.

5. Create your database instance and register RPC service. Then startup services. You can either use *RemoteServerWizard* class (see chapter 5.1.9) or evaluate code similar to following.

   ```
   |instance|
   instance := DatabaseInstanceBuilder buildOnDirectory:'.'.
   CellStore::RPCService new registerTo:instance.
   instance startServices.
   ```

First command will create new database instance. Second one registers RPC service to created database instance. Third one invokes services startup.

6. Now you can access database with your *libcellstore*-based programs.

7. You can also access data from database console. Simply evaluate

```
(CellStore::DatabaseConsoleV2 new)
        open;
        fileMenuOpen:instance.
```

B.2 Client library compiling

Library is provided as source code. It has Makefile with several targets which can be used.

- client (default) - Compiles testing client. Currently it's a sandbox for implementing new features only and has no specific functionality.
- debug - Compiles testing client with debugging symbols and messages. Note this option does not make different targets so before compiling a debug target after non-debug version (and vice versa), use `make clean` command first.
- lib - Compiles library and places it into `dist/` subdirectory together with header files needed to compile library-based applications. You can use this directory content as distribution package.
- install - Installs compiled library to system. Copies shared library from `dist/` subdirectory into `/usr/local/lib` and required headers into `/usr/local/include` This action requires root privileges.
- uninstall - Remote library from system. Also needs root privileges.
- check - Run tests on local database, need *check* [10] library installed on your system. Note that tests are destructive, so do not have important data in database before running these tests.
- clean - Removes all non-source data (object files and other binaries).

Distribution also contains Doxyfile so you can generate library documentation, if not present, using Doxygen.

(FIXME: sync this information to current makefile, remove targets run, demo)

B.3 Demo applications

Few demo applications have been made to illustrate some of *libcellstore* features. You can find them in `demo/` source subdirectory. Current distribution contains three applications. Makefile provided with this demo applications compiles them with shared library in `dist/` subdirectory. To run these applications, you have to have *libcellstore.so* installed or `LD_LIBRARY_PATH` environment variable defined. Alternatively, you can use *run.sh* script which sets this environment variable automatically before run. Usage is simple. Simply add `./run.sh` prefix before command you want to run.

```
./run.sh xquery xmldb://localhost "doc('xmldb:authors2.xml')//surname"
```

Applications provided are (names correspond with Makefile targets):

- xquery - performs XQuery query on given collection. It recursively creates all collections in URI it they're not exist.
- xmlupload - uploads new XML resource into given collection.
- xmldblist - lists content of given collection as a tree.

Few examples of usage together with their output are shown on figure B.1.

```
$ ./xmlupload xmldb://localhost/test/ books.xml < books.xml
Moving to child collection: test
Resource uploaded successfully: books.xml

$ ./xquery xmldb://localhost/test/ "doc('xmldb:books.xml')//title"
<title lang="eng" withPictures="yes">
  Harry Potter
</title>
<title lang="eng">
  Learning XML
</title>
<title>
  1984
</title>

$ ./xmldblist xmldb://localhost/
Listing XMLDB collection [root] at xmldb://localhost/

root/
   test/
     test2/
       - books.xml [XML]
     - books.xml [XML]
   bookstore/
     - bookstore-1.xml [XML]
     - bookstore-1-expensive.xq [XQ]
     - bookstore-1-expensive-titles.xq [XQ]
   - authors2.xml [XML]
$
```

Figure B.1: Demo applications: sample output

Appendix C

How to implement protocol

(TODO: this section will be deleted soon)

There are two ways how to get CellStore client-server protocol working with your favorite platform.

1. Use given XDR specification and other protocol description to write your own library. Use information from chapter 4 as reference. Several implementations for different languages exist. During a short search, I found following:
   - Remote Tea, pure Java implementation of ONC RPC.
     `http://remotetea.sourceforge.net/`.
   - rpcc - Python ONC RPC Compiler, together with demo RPC implementation seem usable.
     `http://www.cs.umd.edu/~gaburici/rpc/` and
     `http://svn.python.org/view/python/trunk/Demo/rpc/`
   -
2. Link *libcellstore* library to your platform. Many platforms, including Python, PHP, some Smalltalk implementations, allow programmers to write their own interfaces to linkable libraries.

Appendix D

CD content

Nothing yet