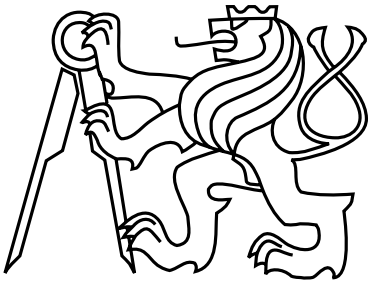
Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Computer Science and Engineering



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 18, 2009

Acknowledgements

I would like to thank Ing. Jan Vraný for his conscientious supervision of my work and valuable advice. Also, I really appreciate the great work atmosphere present CellStore team has created. Last but not least, I would like to thank my family and friends for neverending support during my studies.

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 18, 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 are also exhausting instructions for running CellStore database with remote access service and for compiling of 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 Interfaces of Existing Object & XML Database Engines 3
2.1.1 eXist 3
2.1.2 Sedna 5
2.1.3 NeoDatis ODB 6
2.1.4 GemStone/S 6
2.1.5 Oracle XML DB 7
2.2 CellStore Database Engine 8
2.2.1 XML:DB Interface 8
2.2.2 SELF Model Interface 11
2.3 Client-server Protocol for CellStore 12
2.3.1 Protocol Requirements 12
2.3.2 Implementation Approaches 12

3 RPC Programming in C and Smalltalk/X **15**
3.1 XDR Description 15
3.2 RPC Programming in C 16
3.2.1 Server 17
3.2.2 Client 17
3.2.3 Building and Running 17
3.3 RPC Programming in Smalltalk/X 19
3.3.1 Server 20
3.3.2 Client 21
3.3.3 Various Smalltalk/X RPC Enhancements 22
3.4 RPC in Other Programming Languages 23

4 Protocol **25**
4.1 Mapping Object World to ONC RPC 25
4.1.1 Handling the Objects Remotely 26
4.1.2 Exceptions 26
4.2 Message & Control Flow 26
4.2.1 Single-Call Operations 27
4.2.2 Multi-Call Operations 27
4.3 Large File Transfer 28
4.3.1 Data Transfer 29
4.4 Naming Conventions and Organization 31

4.5 Implemented Functionality . . . 32
4.5.1 XML:DB . . . 33
4.5.2 DOM . . . 33
4.5.3 OODB . . . 33

5 Realization **35**
5.1 Server Side - Smalltalk/X . . . 35
5.1.1 Multi-threaded RPC Server - RPCMTServer Class . . . 35
5.1.2 Server Core - RemoteServer Class . . . 36
5.1.3 Session Object Holder - SessionStorage Class . . . 38
5.1.4 Large Files Transfer - SocketJob Class . . . 39
5.1.5 Special Read Stream - NetReadStream Class . . . 42
5.1.6 CellStore Service - RPCService Class . . . 43
5.1.7 Remote Server Launcher - RemoteServerWizard Class . . . 43
5.1.8 Example - XML Import . . . 44
5.2 Client Side - C Library . . . 44
5.2.1 API Guidelines . . . 44
5.2.2 Library Structure . . . 45
5.3 Smalltalk/X Client example - RemoteClient Class . . . 46

6 Testing **49**
6.1 Unit Testing Tools Used . . . 49
6.1.1 Smalltalk/X - SUnit . . . 49
6.1.2 C Language - Check Library . . . 49
6.2 Test Coverage . . . 50
6.2.1 Server Side - Smalltalk . . . 50
6.2.2 Client Side - C . . . 51
6.3 Performance Measurements . . . 52

7 Conclusion and Future Work **55**

Bibliography **57**

A List of Used Abbreviations **59**

B Installation Instructions **61**
B.1 CellStore Installing . . . 61
B.2 Client Library Compiling . . . 62
B.3 Demo Applications . . . 63

C Programming with the Client Library **65**

D CD Content **69**

List of Figures

2.1 CellStore architecture [1] 9
2.2 XML database hierarchy in CellStore database console 10

3.1 Sample C server procedure 18
3.2 Sample C client code 18

4.1 Sample reply definition 27
4.2 Client-server protocol, capitalized labels represent RPC messages 28
4.3 File upload protocol from client side 31

5.1 Brief capture of client-server implementation architecture 36
5.2 Server - basic structure 37
5.3 Code of RPC operation with standard reply behavior 38
5.4 Code of RPC operation after enhancement 38
5.5 SocketJob hierarchy 40
5.6 Structure of the sample Smalltalk/X client 47

6.1 New TestRunner for Smalltalk/X SUnit tool 50

B.1 Demo applications: sample output 64

C.1 Whole tutorial source code without comments 68

List of Tables

6.1 Performance measurements - running times 53
6.2 Performance measurements - client-server protocol overhead 53

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 for real application development and deployment. 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 should be able, beyond the former XML database access, to work with any arbitrary objects.

This thesis describes the protocol design and implementation of both client and server components. Client part has to be implemented as a dynamic library written in C language. As a consequence, it can be also linked to any programming language that supports C calls to external libraries. Server side will be implemented as a module (service) of CellStore database engine in Smalltalk language.

Second chapter of this thesis gives a brief information about existing object or XML database engines and their client-server protocols. Then, it describes CellStore features and discusses its protocol requirements. Third chapter brings introduction to RPC programming on both platforms being used. Fourth chapter, Protocol, describes all aspects of protocol design, including guidelines for its extension. In fifth chapter, Realization, there are discussed main issues that had to be solved during the implementation of both client library and server module. In Testing chapter, test coverage of both components is described and a simple performance measurement is presented. Last chapter brings conclusion and aspects of future work and protocol expansion. In appendix, among others, there are installation instructions for both server part and client library, and the developer's guide for client applications.

Chapter 2

Analysis

At first, this chapter 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 and client API. Next section summarizes CellStore database engine capabilities and interfaces which can be used and which should be accessed remotely. In the last section, there are several variants for data encoding confronted with protocol requirements.

2.1 Interfaces of Existing Object & XML Database Engines

2.1.1 eXist

The project called eXist [2] was started by Wolfgang Meier in 2000. It's written in Java. It supports XQuery 1.0 and XPath 2.0. According to [2], 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 or remote access. It use Jetty as a web server providing those interfaces.

When running as a servlet within the web application, XQuery can be used for generating web output by similar way as JSP or PHP do. In this case, all HTTP requests to with URI leading to a *.xql file are caught and processed by XQuery using HTTP extensions, producing (X)HTML output. This way, simple web applications can be written.

The latter case is the most important for us as a comparison for this thesis. All three approaches will be discussed more in detail in following text.

2.1.1.1 REST API

REST is a term for common principles which define resource addressing and usage via HTTP protocol without using additional messaging layer [3]. Interfaces defined this way are also called "RESTful". In case of eXist and many other similar systems, REST API maps the URI in HTTP request to the resource provided by the XML database. It uses four HTTP methods for manipulating the data.

1. **GET**.
   Get method retrieves data from database. Request URI is mapped to the XML database structure and appropriate output is returned. If URI resolves to a XML resource, this will be returned. In case of collection, a list of child collections and resources (in XML format) is returned. XQuery or XSL transformation can be also applied. There request must be encoded to the URL together with the URI of the resource they have to be applied to. Also, it's possible to execute stored XQuery by sending request to resource with *.xql suffix. In this case, XQuery is interpreted in the same way as in Servlet deploying case.

2. **PUT**.
   Put request stores or updates given resource. Data is encoded in request.

3. **DELETE**.
   Delete simply erases collection or resource from database.

4. **POST**.
   Post method is used to upload XML fragments to server, such as XUpdate queries or large XQuery scripts which cannot be encoded to Get method.

2.1.1.2 WebDAV

WebDAV part of client-server protocol map each collection to directory path. Using that, user is able to upload, delete, view and edit resources as if they were files. There are many clients supporting WebDAV for all mainstream operating systems.

2.1.1.3 XML-RPC API

XML-RPC is the most important API for us since it can bring us an inspiration for designing our own protocol. eXist provides driver for Java XML:DB API introduced on [4]. It add some more functionality as User management and multiple database instances.

XML-RPC API provides all these features to programmers of client applications. But it doesn't map the RPC calls 1:1 to the XML:DB API. Many functions were made simpler to save network bandwidth.

The most significant change is in collections and resources addressing. These are not referenced by name and their parent node but using full path URL instead. All operations which affect some resource or collection (creating, moving, deleting) use full path as an identifier. For example `getDocument()` method takes full path including parent collections to return resource content. This approach minimizes amount of service

data transfered (XML-RPC is quite bandwidth ineffective since it sends all messages encoded in XML-like message).

Executing of queries is available via `executeQuery()` method. Executing previously stored query is possible as well. Results can be accessed selectively, by using `retrieve()` method. Using `query()` both previously noted operations are merged into one call.

To handle large data, eXist introduces some improvements to save server memory. Documents (resources) can be retrieved by chunks using `getDocumentData()` initial method and `getNextChunk()` for other parts, respectively. Importing can be done directly by calling `parse()` method or via temporary uploaded file using `upload()` and `parseLocal()` methods.

Rest of features such as XUpdate queries and user management are beyond the XML:DB specification.

2.1.1.4 SOAP

Nowadays, Simple Object Access Protocol (SOAP) [5] is more common than XML-RPC [6]. It supports Web Services Description Language (WSDL [7]) based code generation. eXist API provides two SOAP services, Query and Admin. The former one allows read-only access and querying while the latter one provide all available operations. The API is nearly identical to its XML-RPC counterpart.

2.1.2 Sedna

Sedna [8] is a free native XML database which provides a variety of database services - ACID transactions, security, indices, hot backup. XML processing facilities include W3C XQuery implementation, integration of XQuery with full-text search facilities and a node-level update language.

Sedna provides client-server protocol which is well documented and already implemented for various programming platforms. It's based on message sending.

Protocol support transactions, query executions and result retrieving. It resolves only three data formates: integer (4 bytes in network order), byte and string. Each message is formatted in following manner:

- The first four-bytes integer is an instruction code.
- The next four-bytes integer is length of the following body in bytes.
- The rest of message contains the message specific data of length specified by the previous field.

For large data handling, auxiliary messages are present. After `se_Execute` message is sent, the query is performed and the `se_QuerySucceeded` is sent back. After that, first item of result is passed from server using several `se_ItemPart` messages with data chunks and `se_ItemEnd` signaling no data in current item is available. If client wants another result item, it sends `se_GetNextItem` message and the retrieving process is repeated.

2.1.3 NeoDatis ODB

Neodatis [9] is a simple representative of object oriented databases. It currently runs on the Java, .Net, Google Android, Groovy and Scala. It support object storing and fetching. Queries can be done using Native queries (it's a query implemented in client's native language), Criteria query (hierarchy of boolean expressions and other predicates) and by OID (object ID). It supports limited range of native data types.

It can be run either as embedded database or via client-server protocol. The client-server protocol is quite simple. It consist of set of messages and replies to them. All messages are defined as classes, for example, message invoking storing of an object is described in *StoreMessage* and the appropriate reply in *StoreMessageReply* class, respectively. Messages contain at least command code, an integer constant value defined in *Command* class. Messages are sent via standard TCP socket in serialized format since they implement *Serializable* interface.

Objects to be stored are sent in message body. NeoDatis database use three layers of data representation: client language native, meta representation and physical storage representation. Objects are transmitted in form of meta representation data which also implements *Serializable* interface to make the stream encoding possible. Errors can be present in reply message and they are represented as strings.

2.1.4 GemStone/S

Gemstone Smalltalk Object Server (GemStone/S) [10] is an object application server using Smalltalk as application language and providing physical storage for object-based application data. Server part consist of two kinds of nodes which can be distributed in network, *Gem*, the application server itself, and *Stone* object repository manager.

Gemstone has several client-server interfaces.

- **GemBuilder for Smalltalk**. It provides interface to access to the Gemstone from Smalltalk client applications
- **GemBuilder for C**. It allows connecting the C application with Gemstone server.
- **Topaz**, scriptable command line interface to Gemstone/S.
- **UserActions**. They are similar to Smalltalk primitives. They can be written using Gembuilder for C.

Unfortunately, the client-server protocol is not public, thus it's not described here. We will focus on GemBuilder for C in following section.

2.1.4.1 GemBuilder for C

GemBuilder for C is a library providing access to the GemStone application and the repository. It can be run either as linked code or via RPC connection.

It supports two ways of usage:

- Downloading objects from repository, using so called structural access, and performing actions with their data on client side.

- Sending messages to objects on GemStone/S server. This option is described in following text.

GemBuilder use unique 32-bit object-oriented pointer (OOP) to identify remote objects. In C program, it's represented by variable of *OopType* type. Also, identifiers of all GemStone/S kernel classes and special objects such as `nil`, `true`, `false` and errors are defined.

Some of GemStone objects are stored as immediate values within the OOPs so GemBuilder have macros to retype their appropriate representations in C. To send message to an object, `GciPerform` procedure is used. It takes three arguments: receiver OOP, array with arguments and its length. By calling the `GciNewSymbol` function, a symbol (which can represent, for example, a message name or dictionary key) is created. GemBuilder also provides several functions for complex Smalltalk code executing. `GciExecuteStr` takes C string with Smalltalk code and executes it on server. `GciExecuteStrFromContext` treats the string as the message send to some object.

2.1.5 Oracle XML DB

Oracle database is not a typical member of XML database group since its XML DB works as an extension of object-relational database system [11]. It brings new object type, *XMLType*, which represents XML structured data.

In Oracle XML:DB extension is implemented via PL/SQL packages. PL/SQL package DBMS_XDB is an API for managing resources and security through access control lists (ACLs). It also provides a basic XML DB configuration. Two new views were introduced to obtain information about stored XML resources. These are RESOURCE_VIEW and PATH_VIEW. Note that this API is not compatible with the one specified by XML:DB Initiative [4] and it sometimes specifies new terms (folders instead of collections and so on).

Oracle XML DB also provides an interface to manipulate data stored in XML resources in several ways. Its PL/SQL packages, JAVA or C APIs contain functions for manipulating XML data using DOM approach, calling XML parser, or produce output using XSL transformation. All DOM APIs are compliant with the W3C DOM Level 1.0 and Level 2.0 Core Recommendation.

All these features can be used by standard client protocols provided by Oracle. This includes various clients and interfaces such as Java Database Connectivity (JDBC) driver.

To make the access to the XML documents as easy as possible, Oracle supports several standard internet protocols. Documents hierarchy is mapped to directory structure and XML data can be uploaded or downloaded as files. Network protocols, such as HTTP(S) and WebDAV, or FTP are used for data transfer.

2.2 CellStore Database Engine

CellStore database provides wide range of interfaces – from low level ones to high level ones such as SELF model interface or XML:DB interface. Figure 2.1 shows architecture of CellStore as well as its interfaces.

One of the most important layers is so called SELF engine. One of important layers is, in the middle, a new SELF engine. This layer allows to store any data structured to objects provides generic and easy to use interface for higher-level interfaces to CellStore database.

On the top of SELF layer, there are several interfaces which provide various access methods, including XML:DB API [4] and OODB (object-oriented database) API. All important interfaces which will be provided by our new protocol are described later in this section.

At first, we will describe CellStore XML:DB interface which is the most important for us. In latter sections, some other interfaces, including SELF, are mentioned.

2.2.1 XML:DB Interface

XML:DB interface was developed to fit needs for accessing XML-based data in database engines. It was designed by XML:DB Initiative [4]. Last changes in API are dated to year 2001.

XML databases come with XML documents as a basic structure for storing data (it's often called as document-centric approach). These documents are stored in resources. XML resources can be accessed atomically by DOM[1] or SAX[2] interfaces. Both of them allow to inspect every element of XML document separately. Using SAX interface (defines set of events used by XML parsers), it's possible export the XML resource as set of SAX (parser) events for further processing in another tool. Some databases also allow to store binary resources containing arbitrary data.

Resources are organized hierarchically in the database. The nodes in the tree structure are called collections. XML database contains root collection by default. Sample structure is shown on CellStore database console on figure 2.2.

XML:DB API allows common operations with collections and resources like child nodes listing, adding, removing and information obtaining. Also, it provides few services to extend its functionality (transaction service, XPath service etc.). The specification can be found in JavaDoc on [4].

CellStore XML:DB API classes are located in *XMLDB* namespace and they reflect the XML:DB specification as much as possible. CellStore implementation adds several functionalities. XPath service was replaced by XQuery service which is not present in former XML:DB specification. XQuery [12] is a complex query language used with XML-based data. XPath is a subset of XQuery providing document elements identifying so there is no missing functionality.

Because CellStore is written in Smalltalk, XML:DB API lacks some features which are specific for other programming languages. For example, iterators are often used in C++ or Java code. But in Smalltalk, there is not usual to use iterators. Using of `#do:` method is a common habit. The most important XML:DB API classes include:

[1] Document Object Model
[2] Simple API for XML

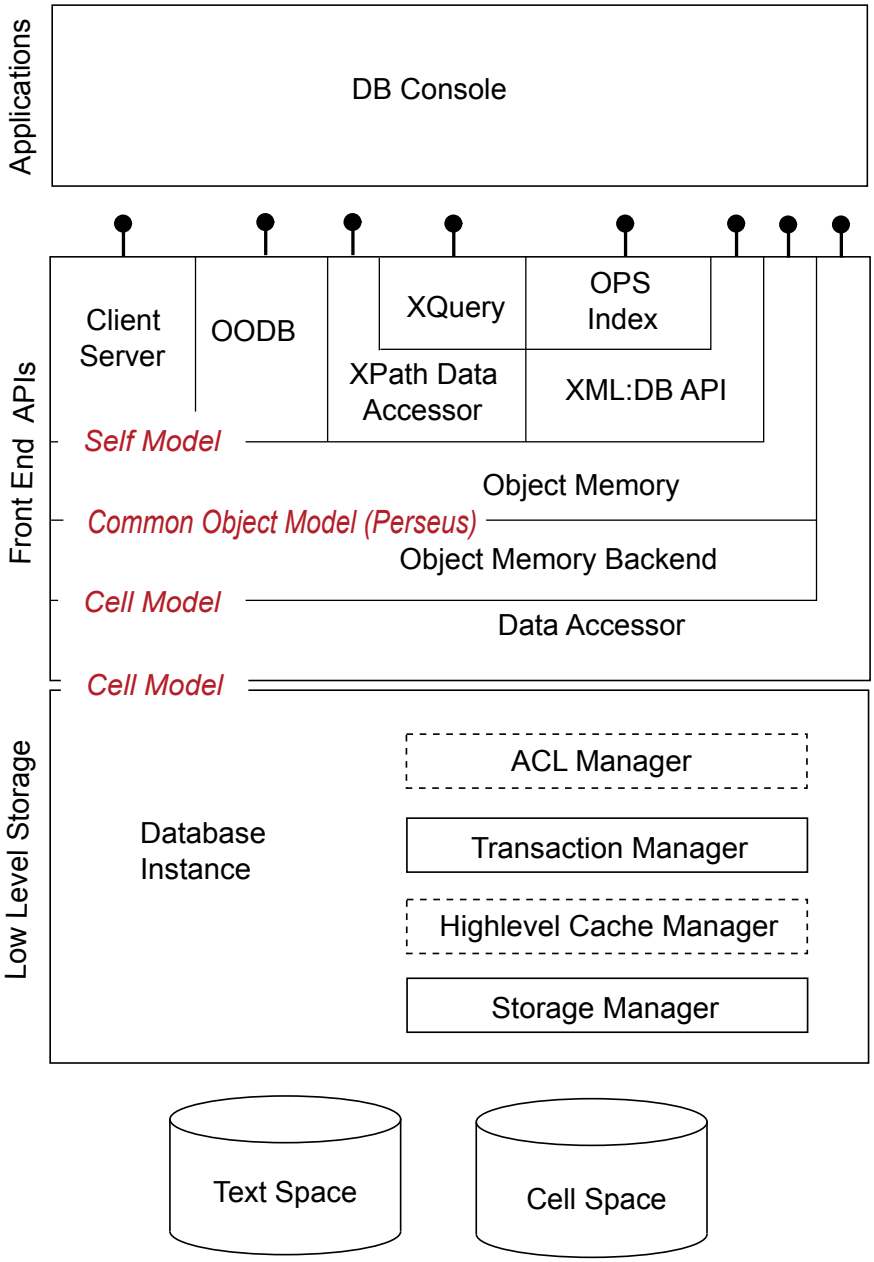


Figure 2.1: CellStore architecture [1]

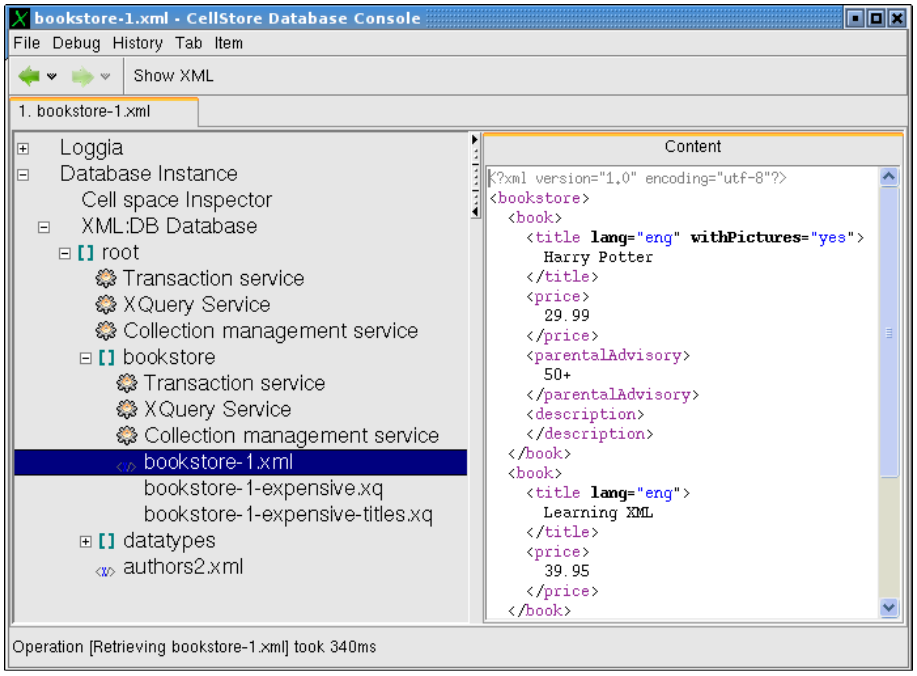


Figure 2.2: XML database hierarchy in CellStore database console

- *XMLDB::XMLDBDatabase*
  Main object representing the database instance on XML:DB level. It can be obtained using `#getXMLDBDatabase` method of *DatabaseInstance* object[3].

- *XMLDB::Collection*
  This class represents XML:DB collection and provides operation with it and its child collections and resources.

- *XMLDB::Resource*
  This is a superclass for all types of resources which can be stored in CellStore database. Currently, only *XMLResource* and *XQueryResource* are supported and working. They represent XML document and stored XQuery query, respectively. Both these resources can be exported using DOM or SAX interface. XQuery resource also have to be evaluated before it is exported. XML resource can be imported either from DOM or XML file. This file can be streamed so no additional memory is used.

- *XMLDB::XQueryService*
  This service provides XQuery interpreter which evaluates query on given collection. It can be obtained by sending `#getXQueryService` message to *XMLDB::Collection* object. XQuery is evaluated by passing `query:` message to the service. It returns *XQueryResult* object which is, in fact, a collection of *XQueryResultItem* objects. These items represent each part of XQuery result. Result of execution can be converted to DOM or exported as XML document using SAX interface and XML writer.

[3]This object represents whole CellStore database instance.

2.2.1.1 DOM API

The Document Object Model[13] is an interface which allows reading and modifying of various structured documents content, especially XML documents. Document model has hierarchical structure. Each node can have one of several types (element, attribute, etc.) and particular number of child nodes, according to its type. Document structure can be changed and stored back into database.

Huge disadvantage is fact that when using DOM interface with CellStore, all data are copied out from database to server memory. For example, `#getContentAsDOM` of *XMLDB::XMLResource* will copy whole structure from database to hierarchy of DOM nodes. These nodes can be modified, removed, or fresh nodes can be added to existing structure. Then, whole DOM hierarchy can be imported back to the database using `setContentAsDOM:` method. Large XML document generates huge DOM structure so the size is limited by available physical memory. Mapping the DOM interface directly to CellStore SELF data could be better solution since no additional memory in virtual machine would be necessary.

2.2.2 SELF Model Interface

Recent versions of CellStore database model all data structures in the database as a SELF-like objects. SELF [14] is a prototype-based object-oriented language which is even simpler than Smalltalk. In a reduced form, it serves as a universal data access provider for upper CellStore layers. SELF data model allows to store any data structured to objects without concerning about low level representation. The only operation in SELF is a message send.

Compared with Smalltalk, it doesn't differentiate between instance variables and methods. SELF has slots instead. The only way how to access a slot content is via message send. When a message is sent to an object, appropriate action depends on content of the slot. If the slot contains an object reference (this slot is called *data slot* in SELF language), this will be returned without further processing. If the slot contains method, its code will be executed and result value is returned. Every data-slot has implicit getter and setter. In CellStore, concrete SELF objects are represented by cell-pointers, pointers to CellStore physical storage.

There are several ways how to execute a SELF code:

- A message is sent to SELF object represented by cell pointer together with its arguments. These arguments must be another SELF objects.
- SELF code is executed in context of implicit receiver (SELF object). If no implicit receiver is specified, object called *lobby* is used[4]. This options allow performing message cascade. For example, evaluating of `"xmldb root"` with implicit receiver causes sending `xmldb` message to *lobby* and `root` to the result of previous call. As a result object representing XML:DB root collection is returned.

[4]Implicit receiver is a term of SELF language. It's called "implicit" even if it is specified explicitly.

For both ways of usage, SELF interpreter is required. It can be obtained by sending the `#newInterpreter` message to *ObjectMemory* object which represents the SELF memory space.

Objects of higher layer interfaces such as XML:DB (see figure 2.1) are mostly realized as object proxies. Object proxy allows to work with SELF object as if it was Smalltalk object. It contains cell-pointer to the SELF object it is associated with. Almost every of its methods consist of message sends to associated SELF object.

2.3 Client-server Protocol for CellStore

2.3.1 Protocol Requirements

Our protocol must respect few requirements:

1. **Bandwidth efficiency**. Work with SELF API consists of many simple operations calls. Our protocol should be bandwidth effective then.
2. **Extensibility**. CellStore is still in development phase and most likely the protocol or the API will change in near future. Therefore, protocol should be easily extensible by means of programmer's effort required to implement such changes.
3. **Portability**. If some additional toolkit is used, native implementation in other programming languages should be available to make the protocol implementing possible. Furthermore, our protocol implementation will be in form of C library. That makes it linkable to various programming languages which support C callouts.
4. **Various data transfer**. The protocol must be able to transfer any kind of data, including large binary objects.

2.3.2 Implementation Approaches

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

2.3.2.1 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 easily because it's human readable. It's extensible as far as extensible protocol parser is used. Problems come with binary data transfer. To keep data represented by readable characters, various coding methods are used. For example, MIME [15] format together with encoding techniques such as *quoted-printable* or *base64* is used for sending of binary data or non-ASCII characters using 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 option.

2.3.2.2 Binary Protocol

Designing our own binary protocol would probably bring us the most bandwidth-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 costly. For example, coding operation number into one byte would become a problem in future because it would limit the number of different operations available (which is 256 in case of 1 byte used).

2.3.2.3 Remote Procedure Call

Remote Procedure Call (RPC) is a common term for communication mechanisms allowing programmers to call program code remotely from another machine the same way as local functions. Several implementations were developed such as Open Network Computing Remote Procedure Call (ONC RPC, [16]), XML-RPC [6] and Corba [17]. 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 bandwidth-effective for sending of many short messages. Binary representation usually gives us better results by means of bandwidth-effectiveness.

2.3.2.4 Conclusion

First two options discussed above have significant disadvantages. Text-based protocol misses efficiency because of 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 is it's extensibility. CellStore is still in development phase and nobody knows what features will appear in future. Our protocol must be versatile in this point of view so custom binary protocol is not a good idea - simply because it's very costly to extend.

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 implementations we finally chose ONC RPC [16] (also known as SunRPC). It uses binary data representation (will be described later) so it seems to be bandwidth effective. Moreover, both SunRPC client and server are already implemented in Smalltalk/X and ONC RPC client libraries are available for all commonly used operating systems. Chapter 3 describes RPC programming in both C and Smalltalk/X.

Chapter 3

RPC Programming in C and Smalltalk/X

From options discussed in previous chapter, we finally chose ONC RPC [16] (also called SunRPC). ONC RPC is a protocol formerly developed by Sun Microsystems [18] for their NFS protocol and other network services. It uses XDR [19] as an interface and data 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 [19] means for eXternal Data Representation, OSI presentation layer implementation, used with ONC RPC [16]. Using XDR, 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 program number. Program number is used to identify the network application on portmapper. Program number space is partially regulated by IANA. See [16] for more information.

Following example is taken from [20]. I had to make some changes because Smalltalk implementation of XDR parser 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, program `MESSAGEPROG` with number 200001 and 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.

Among the basic types, XDR supports complex types like structures, unions, arrays and strings (both variable and fixed size). See [19] for more information.

3.2 RPC Programming in C

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. Following text expects usage of `rpcgen` version available in GNU/Linux distributions[1].

Useful commands are:

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

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

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

#generate makefile template
#(it 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 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.

[1] Unfortunately, several implementations of `rpcgen` are available for various platforms. Sometimes, they differ in their API or behavior and set of features than can be used within XDR definition.

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-based protocol and server should respond to it.

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

As you can see, `rpcgen` generates functions to all exported RPC procedures. We can start using them right after connecting. `clnt_create()` provides the simplest way to create connection handler.

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 to add all source file names to the variable definitions at the beginning of the Makefile. For our example, it should be like following:

```
CLIENT = msg_client
SERVER = msg_server

SOURCES_CLNT.c = msg_client.c
SOURCES_CLNT.h =
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 determining all build targets and dependencies. Also we must modify the `RPCGENFLAGS` variable to make rpcgen using new style of coding every time it is called.

```
RPCGENFLAGS = -N
```

```
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

```
#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

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 RPC request and can be terminated by keyboard interrupt (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, version 1, using TCP
rpcinfo -t localhost 200001 1

#call procedure 0 in program with number 200001, version 1, using UDP
rpcinfo -u localhost 200001 1
```

3.3 RPC Programming in Smalltalk/X

Smalltalk/X provides several classes for SunRPC located in *Net-Communication-RPC* category. They include RPC server and client, XDR parser and decoder/encoder, together with several implemented RPC applications, such as NFS server and Portmap server. Following introduction is inspired by the tutorial on eXept site [21]. It's adapted to the sample in section 3.2.

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 [20] had to be modified slightly. Smalltalk/X XDR parser implementation is unable to parse the definition with identifiers of procedure arguments. I made improvement to allow them. See section 3.3.3 for details.

XDR is assigned as a class method to server (and also to client) class. During the server initialization, XDR is parsed and particular information is stored to `Definitions` class variable.

```
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` 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. Smalltalk/X also has its own Portmapper implementation which can be run automatically so it is not needed to have system Portmapper installed. But in this case, Smalltalk/X virtual machine must have administrator privileges to open port 111 on Unix/Linux systems.

Simplest way of defining the ports:

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

3.3.1.3 Implementing methods

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

```
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*).

```
|server|

"start using TCP (default)"
server := EchoServer start

"or start using UDP"
server := EchoServer startUDP

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

Our implementation of client-server 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. To avoid actualization problems, client should take XDR definition from server.

```
xdr
    ^ EchoServer xdr.
```

Default client has universal methods to invoke remote procedures but in general, it's better to implement our own API. Following method calls the `PRINTMESSAGE` procedure:

```
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* provides methods to connect without asking the portmap service. Following code will connect using portmapper query and will call `PRINTMESSAGE` procedure which will print given message on screen (in case of C server) or Transcript (in case of Smalltalk/X 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 Enhancements

During the development process, I found that Smalltalk/X RPC implementation lacks some important features which are 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 [19]. 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 RPC in Other Programming Languages

ONC RPC is wide spread standard so several RPC implementations can be found. In consequence, our new protocol can be implemented on different platforms. Several implementations of ONC RPC exist for various languages. 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 seems usable.
  `http://www.cs.umd.edu/~gaburici/rpc/` and
  `http://svn.python.org/view/python/trunk/Demo/rpc/`
- ONC/RPC for Windows, free implementation.
  `http://oncrpc-windows.sourceforge.net/`

Chapter 4

Protocol

The aim of this chapter is to describe problems with client-server 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 binary objects transfer.

Complete XDR specification of CellStore protocol can be found in `service_rpc.x` file in client library sources or in `#defaultXdr` method definition in *RemoteServer* class in server part.

4.1 Mapping Object World to ONC RPC

Object oriented languages has several features which cannot be handled by ONC RPC directly. This section describes these features and their mapping to ONC RPC.

In Smalltalk, every operation is a message send to some object. Even `'1 + 2'` expression represents message send. "1" is an object (*SmallInteger* class) which receives message `#+` with one argument, another integer with value "2". The former object (with value "1") is called receiver.

This can be easily emulated by regular ONC RPC procedure calls. Message will be represented by a standard function and reference to the receiver will be passed as its first argument followed by message arguments. By this way, we can solve the problem in our protocol since we are able to send and receive references to server objects we work with. Except of this easy issue, there are two other problems which are more difficult to solve:

- **Object references**. Remote handling must protect objects from deletion caused by garbage collector on server side.
- **Exceptions**. These cannot be simply raised over network connection since ONC RPC wasn't designed to handle exceptions.

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 as object identifiers. These are transferred in RPC calls or replies, respectively.

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 might be not flexible since remote operations can return reference to various objects (Smalltalk itself is not type language). Basic type control is performed on server side on demand. Server operations can check the referenced object type where needed. This can prevent from unexpected errors caused by object mismatch. More information about server-side type control can be found in section 5.1.3.

Alternatively, user can ask for object class name, but this operation returns string. Class types cannot be identified by enumerated value because it's difficult to determine all used classes. Object class name should be used for debugging purposes only.

Due to the garbage collector in Smalltalk/X, these references cannot be simply equal to object memory addresses (actually, it's not easy to get the memory address of an object in Smalltalk). 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 will be deleted if there is no other 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` is present and contains 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.

4.2 Message & Control Flow

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

```
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

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.

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 RPC request 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 actions. 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 file transfer is described in detail in chapter 4.3, concrete implementation is explained in chapter 5.1.4.

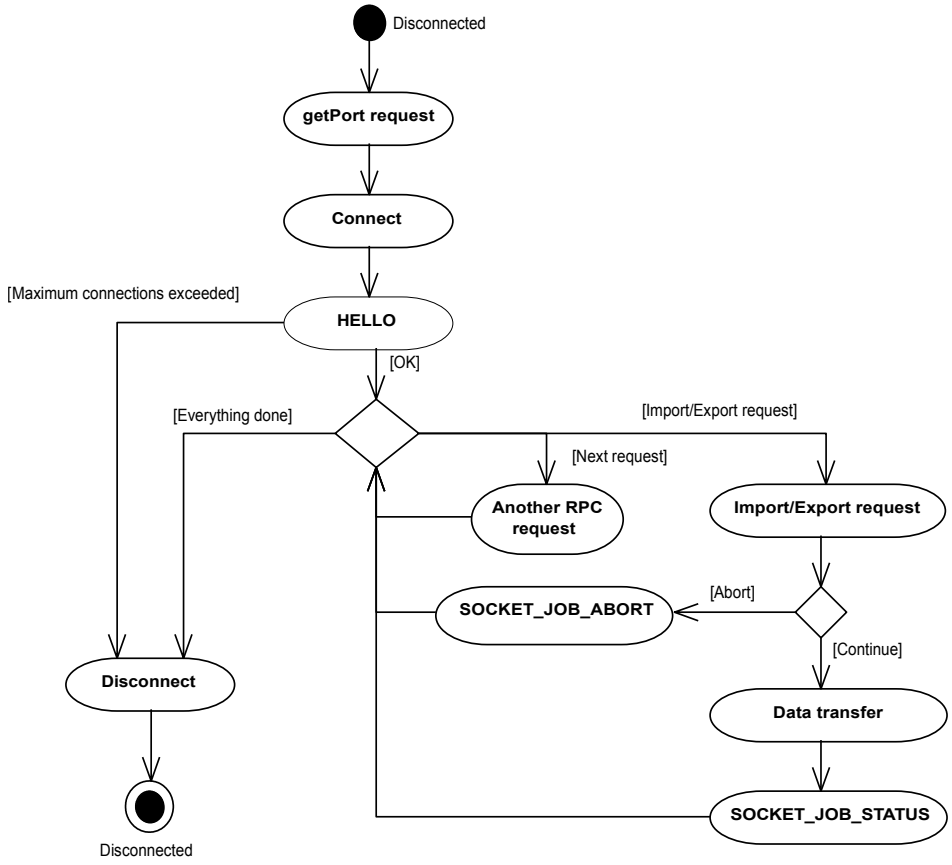


Figure 4.2: Client-server protocol, capitalized labels represent RPC messages

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 data. 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 receive another request). This option may be unacceptable on some platforms and client implementations since we need to get 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's versatile enough to provide platform for various features which can be implemented in future. Auxiliary connection should be initiated from client to pass possible firewall on client side. 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 5.1.4 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 returned in a reply of the 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 returns no data. If export operation fails on server, remote connection is 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.

4.3.1.2 Data Upload

For uploading data from client to server, several modifications has to be applied. It is expected that XML parser reads data until the end of file (or stream, respectively) is reached. If *Socket* is used, the connection has to be closed to indicate end of file.

But connection closing 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 blocks of different length. Too short blocks are not recommended. If operations want to read long data, server must merge the data from more blocks divided with headers. This brings small performance drop. Also, if short blocks are used, the bandwidth efficiency will decrease because more headers have to be sent. Practically, blocks of sizes in order of kilobytes are long enough.

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

Structure of data communication is shown on figure 4.3.

[1] End of File
[2] Acknowledgment

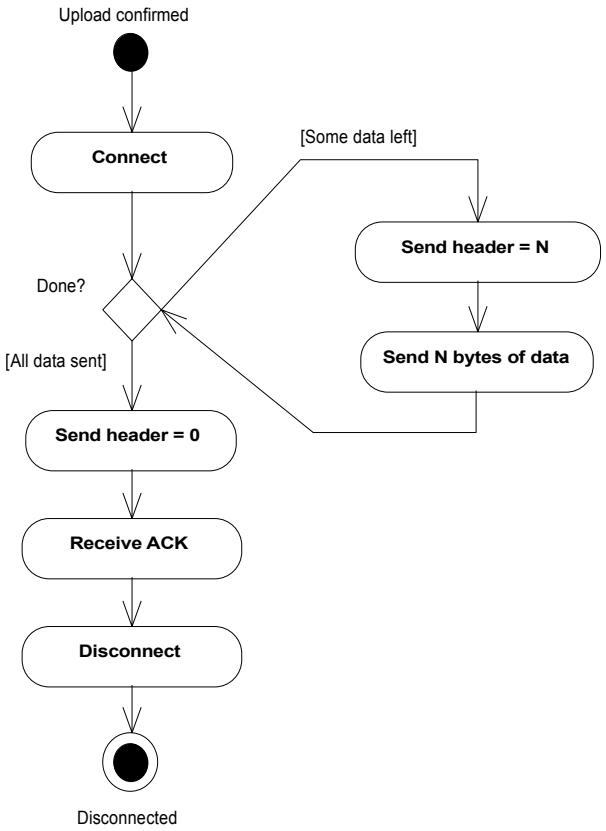


Figure 4.3: File upload protocol from client side

4.4 Naming Conventions and Organization

To keep the protocol messages organized, some system is brought to the remote procedures naming. Procedure names contain prefixes which map them to the hierarchy of packages and objects.

Some core and auxiliary procedures are left without prefixes but almost all are named in following manner. The leftmost prefix assigns the messages to the package. These prefixes are listed below. Operation code (procedure number) patterns for each category are placed in parentheses.

- (no prefix) - various auxiliary procedure calls (0000xx).
- `SOCKET_JOB_` - common functions for controlling socket jobs - large file transfers (0001xx).
- `OBJECT_` - functions for remote objects handling (2000xx).
- `XMLDB_` - XML:DB API functions (3xxxxx).
- `DOM_` - DOM API methods (4xxxxx).
- `OODB_` - SELF oriented operations (5xxxxx).

Some packages also have separation on next level. The most significant example is XML:DB API. It is differentiated by next prefix, according to the class which the procedures are mapped on.

- `XMLDB_` (no further prefix) - *XMLDB::XMLDBDatabase* functions (3000xx).
- `XMLDB_COLLECTION_` - *XMLDB::Collection* (3001xx).
- `XMLDB_COLLECTION_TRANSACTION_` - Transaction control on collections (30015x).
- `XMLDB_RESOURCE_` - *XMLDB::Resource* and subclasses (3002xx).
- `XMLDB_RESULT_` and `XMLDB_RESULTSET_` - operations with XQuery result (3003xx).
- `XMLDB_UPLOAD_` and `XMLDB_DOWNLOAD_` - large files transfer initiation (301xxx).

Error codes are assigned in the same manner using similar prefixes. For example XML:DB API error codes defined in `cs_status` enumeration have their names prefixed with `CS_ERROR_XMLDB_` and values in form of `3xxxxx`.

Custom arguments have their names prefixed with `cs_arg_` and return value types with `cs_reply_`. As explained before, every reply value type has to be a union. It is switched by status value (`CS_STATUS_OK` or appropriate error code) and contains return value itself or error message string. See code in figure 4.1 for example. The name of the new return value type is derived from the inner value type. For example if we want to return integer value (*int* type), the appropriate union type will be named as `cs_reply_int`.

4.5 Implemented Functionality

Following section aims to describe functionalities implemented in current version of protocol. It only defines the range of operations provided and adds info about some special calls. Formal description of all available operations together with comments are specified in `service_rpc.x` file.

Operation categories are mentioned in previous section. Expect of those described in their own subsections, there are some groups left.

- Auxiliary session procedures. This include `HELLO` message described before and `IDENT` which return some info about server.
- Socket job control messages. `SOCKET_JOB_STATUS` returns current job status or error if occurred. Using `SOCKET_JOB_ABORT` can abort that job. See chapter 4.3 for more information.
- Remote object handling procedures. The most important is `OBJECT_DROP`. This allow to drop any remote object when it's not needed by the client anymore. Programmers have to release unused objects to avoid wasting the memory during the session. Also, `OBJECT_IDENTICAL` and `OBJECT_EQUAL` are available for comparing two object by identity or content, respectively.

4.5.1 XML:DB

XML:DB API was implemented completely as it's available in CellStore. Every method is mapped to RPC procedure using naming conventions described in previous section. Operations which are available via XML:DB services (like transaction or XQuery services) are mapped directly so no previous service obtaining (using operations like `#getService:version:`) is necessary.

Among these operations, there are two special procedure calls which are not mapped to concrete API method. They prepare the large files import / export using auxiliary socket. These procedures are `XMLDB_UPLOAD_RESOURCE` and `XMLDB_DOWNLOAD_RESOURCE`. See chapter 4.3 for large files protocol specification and section 5.1.4 for server implementation.

4.5.2 DOM

As noted before, DOM interface has one serious disadvantage. Before any operation can be done, whole structure must be copied to new structure in memory. We expect DOM interface outage as soon as direct node access to XML resources and XQuery result will be available. Due to this, DOM operations provided by our protocol are not complete and probably it won't be extended at all. Only node-related procedure calls has been defined and employed, allowing read-only access to object model nodes, their child nodes, values and attributes. Also, its possible to export the object tree represented by its root node to the XML string.

It possible to check the node type by calling the `DOM_GET_TYPE` procedure. It returns value defined in `cs_reply_dom_type` enumeration type in XDR description.

4.5.3 OODB

OODB package provides methods for remote work within SELF layer of CellStore database. This API is similar to one provided by GemBuilder for C described in section 2.1.4. Available messages can be divided into several groups:

- **Obtaining the object pointer**. `OODB_GET_OOP` retrieves SELF memory pointer from known upper layer objects like XMLDB collection or resource.
- **Simple object creating and fetching**. Only strings and integers are available. For example, `OODB_CREATE_STRING` will create a string in server object memory. Backwards, using the `OODB_FETCH_STRING` procedure, string is read from object memory and sent back to client.
- **Code executing**. `OODB_EVALUATE` and `OODB_EVALUATE_WITH_IMPLICIT_RECEIVER` will execute given code in context of default (lobby) or explicitly given receiver. Alternatively, `OODB_SEND_MESSAGE` or `OODB_SEND_STORED_MESSAGE` will send given message to given receiver with arguments passed. Later option refers to message selector stored as interned string on server.

Chapter 5

Realization

This chapter describes the most important issues which had to be solved during the implementation of both server component and client library.

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 previous 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.

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 subclass of *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.

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

[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)

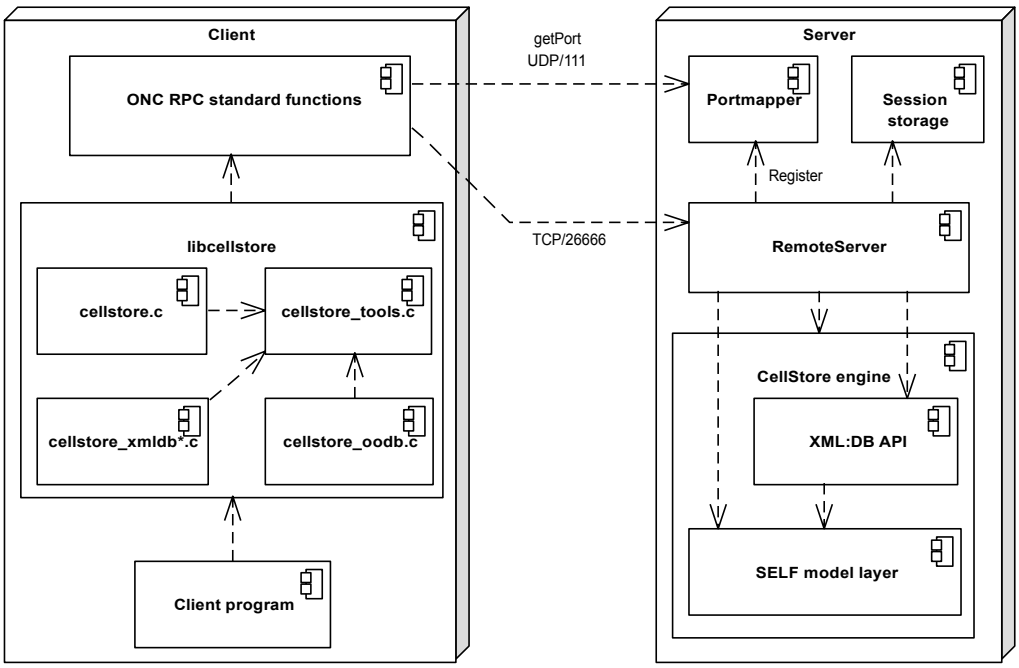


Figure 5.1: Brief capture of client-server implementation architecture

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, new process continues executing as if the server was single-threaded. 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 of server component which is discussed in section 5.1.2. 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 represents the RPC server for CellStore client-server protocol, listening on a TCP port. Also, another instance is created for each incoming connection to store session specific information like connection socket or session object storage (session storage will be explained in chapter 5.1.3). In fact, it inherits *SunRPC::RPCMTServer* class and implements all exported operations.

Main object purposes are:

- Define XDR description of protocol. `#xdr` method was improved to load current XDR definition from local file. This feature is used during development and allows to share the same protocol specification with client library.
- Implement all exported RPC operations.

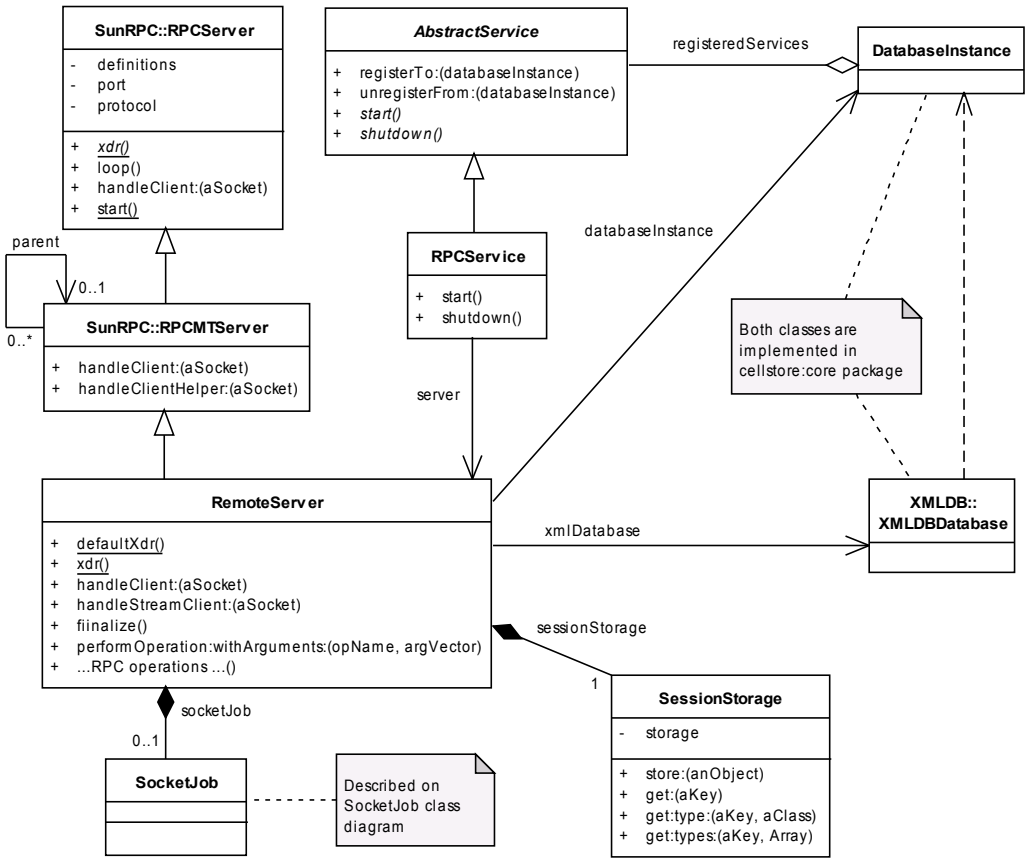


Figure 5.2: Server - basic structure

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 items. 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 expressed as dictionaries. For example, reply to `DOM_LIST_COUNT` procedure call might look like code on figure 5.3.

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 overridden `#performOperation:withArguments:` method. To distinct exception type from others, every exception class 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

```
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

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.

To identify errors generated by various classes of server component, *RemoteServer* also contains another private class, *CustomError*. This is an exception class providing several status codes to identify various custom errors.

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

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 returns unique reference. To get the referenced object back, `#get:` method is used. If the reference is not valid, an exception is thrown.

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 passing reference to wrong object. 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.

5.1.3.1 EnhancedSessionStorage Class

SessionStorage assigns new reference value every time the `#store:` method is called even if the objects are identical. This forbids identity comparing based on reference value equality. During the object storing, it seems useful to find out whether the object is present already and return its reference. But CellStore interface methods return fresh objects every time they are called so identity check won't help.

Almost all objects handled by session storage are either object proxies or cell-pointers themselves. Although object proxies representing the same object are not identical, they can be compared by their content (using `#=` message) because they have identical content - cell pointer.

EnhancedSessionStorage uses cross-directed dictionary having objects as keys and their remote references 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 identity compare based on reference value.

Unfortunately, this feature cannot 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 won't work.

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

5.1.4 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 client handling process 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

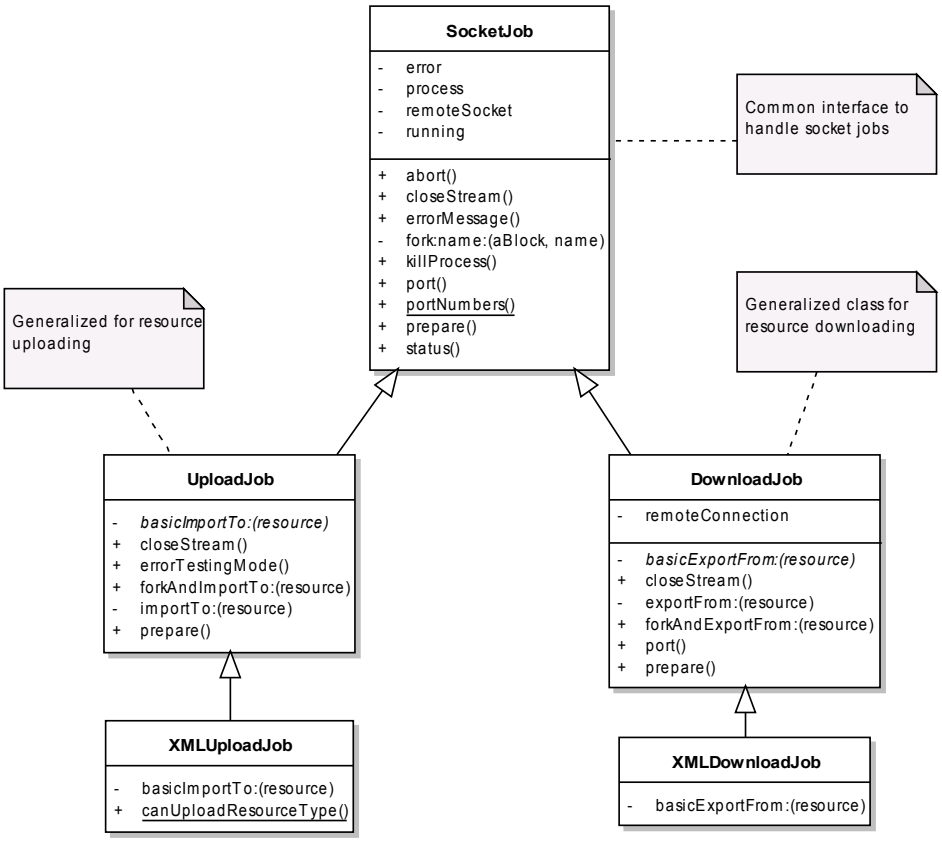


Figure 5.5: SocketJob hierarchy

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 important 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`
  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 classes for uploading and downloading, *UploadJob* and *DownloadJob*, respectively. They implement whole code common for

particular type of socket jobs and define interface for concrete upload or download jobs. Important 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`).
- `#forkAndImportTo:`
  Creates new process and runs `#importTo:` in it.
- `#importTo:`
  Waits for incoming connection 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 upload job (upload of new type of resource), all programmer has to do is to inherit subclass from *UploadJob* class and implement `#basicImportTo:` method which imports data from object local stream (or socket) to resource gives as an argument. Similarly, to create new download job, inheriting from *DownloadJob* and implementing of `#basicExportFrom:` method are required only.

In following sections, details about data transfer are discussed.

5.1.4.1 Downloads

Download process can be realized very easily. In Smalltalk/X, network socket (realized in Socket class) behaves like 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 later.

In this case, new socket which is used for data transport has always to be initialized. So at first, request message (e.g. `XMLDB_DOWNLOAD_RESOURCE`) is sent. This operation prepares new socket. 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.

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.

5.1.4.2 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.

5.1.5 Special Read Stream - NetReadStream Class

XML parsers read data from streams. *Socket* is also kind of stream. But due to reasons discussed in section 4.3, the data flow of import stream is different than in case of export. *NetReadStream* was created as a replacement of standard *Socket* having the same interface as generic *ReadStream*. It solves upload protocol issues mentioned in section 4.3. It can be passed to parser as if it was ordinary stream such as *FileStream*. In fact, it contains listening TCP socket receiving remote connections for data transfer.

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 in current block left, 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.
- Methods from *ReadStream* interface. They are modified to ensure that they can read from stream by calling the `#ensureHeaderRead` method.
- Seek operations, similarly to standard *Socket* class, were disabled. Parsers usually do not need them.

After listening socket is bound and connection from client is accepted, *NetReadStream* is passed to parser. While the parser is calling all the methods for data reading, *NetReadStream* maps them to socket operations. Beyond that, it holds the number of bytes available for receiving. Read operations decrement this value. 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 and stored to temporary variable. 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.

When XML parser reaches the end of file, the `#ack` method is called and ACK message is sent to client. Then, the auxiliary connection can be closed.

5.1.6 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 refers to main (listening) *RemoteServer* instance. It implements two methods only:

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

5.1.7 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.

5.1.8 Example - XML Import

This section shows simple example of large data upload and import. To successfully import the XML document, following actions have to be performed:

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. Then it creates new process which accepts remote connection and executes parsing. All these actions are performed by *XMLUploadJob* object methods.
3. Server sends back the port number of listening socket as a reply to `XMLDB_UPLOAD_RESOURCE` RPC call.
4. Client connects to remote port and writes whole file into socket using protocol specified in section 4.3. At 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.

5.2 Client Side - C Library

Client library in C, named *libcellstore*, is based on code generated by `rpcgen` tool. `rpcgen` and XDR library are available on Linux and other Unix-based systems by default. Although implementation for Windows exists (one was noted in chapter 3), *libcellstore* cannot be compiled on Windows natively without code changes because BSD network sockets API were used. Cygwin can be used instead.

Whole library is organized into modules and most of them correspond to some subset of exported CellStore functionality.

5.2.1 API Guidelines

All functions accessible from *libcellstore* follow the same naming and coding habits.

First of all, we will introduce data types defined in `cellstore.h` header file. They act as mappings over XDR defined types to provide the same type naming conventions along the *libcellstore* library and isolate them from the XDR description to avoid API changes in case of future protocol modifications. These are types are:

- `CellStoreStatus` - operation status code,
- `CellStoreSession` - pointer to RPC client connection,
- `CellStoreObject` - reference number to remote object,

- `CellStoreResourceType` - type of XML:DB resource,
- `CellStoreDOMType` - type of DOM node.

Functions are organized using prefixes in manner similar to protocol definition described in section 4.4. All functions names are prefixed with `cs_` ("cs" for CellStore), XML:DB API functions names begin with `cs_xmldb_` etc.

Every operation returns status code. This code is equal to `CS_STATUS_OK` if no error has occurred. Otherwise, it contains corresponding code. All return values are passed via pointers in function arguments. In case of error, no return variable will be modified. Function arguments have following order:

1. Return values. First of all, pointers for storing results are passed. In most cases, only one pointer is enough. To return variable array, two pointers are needed. The first one is a pointer to empty array pointer, for example `int**` for array of integers. The latter one is `int*` for passing the array length. Note that all functions returning variable-length data (arrays and also strings) allocates them dynamically so programmer is responsible for their releasing.
2. Reference to affected object, if any. At most cases, RPC call is mapped to method dispatching. So this argument refers to the receiver.
3. Other arguments.

5.2.2 Library Structure

- `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 transactions control.
- `cellstore_xmldb_collection.c`
  This module provides interface for manipulating with XMLDB collections, including child collections and resources creating.
- `cellstore_xmldb_resource.c`
  Actions with XMLDB resources.
- `cellstore_xmldb_download.c`
  Functions for resource download.
- `cellstore_xmldb_upload.c`
  Functions for XML resource upload.
- `cellstore_xmldb_query.c`
  XQuery and XQuery result related functions.

- `cellstore_dom.c`
  DOM API.

- `cellstore_oodb.c`
  Object oriented database API.

- `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 the global error message string which can be accessed via this module. The only function accessible from API is `cs_errorMessage()` which returns pointer to the error message string.

- `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 comments 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 Smalltalk/X Client example - RemoteClient Class

To illustrate guidelines for creating of native Smalltalk client, I made this very simple example which is able to perform XQuery query only. It can be used as an inspiration for developing client in other object-oriented languages.

Smalltalk client object layout of XML:DB interface should match with XML:DB API [4]. Figure 5.6 shows layout of sample client code illustrating guidelines for Smalltalk/X client creating. Other components as SELF interface and DOM also should follow the particular parts of CellStore API.

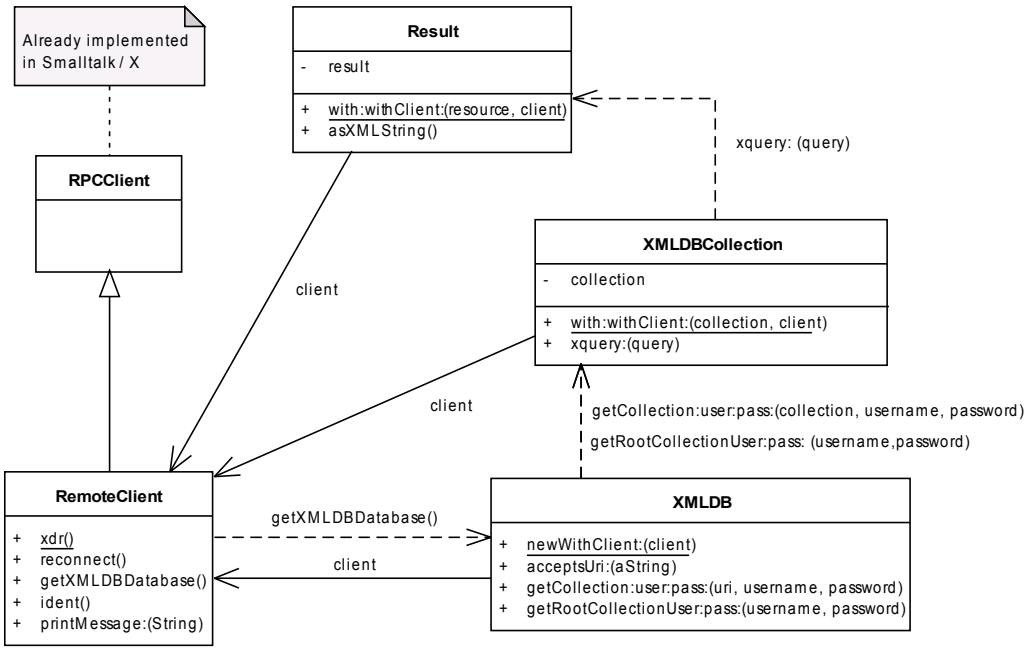


Figure 5.6: Structure of the sample Smalltalk/X client

Chapter 6

Testing

All parts of both client and server component are covered with unit test. Unit testing is a crucial principle of so-called Extreme programming and 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 pass all cases. Our protocol and its implementation are developed progressively so using of agile methods is suitable for this project.

At first, this chapter provides brief description of testing tools used for this project. Then, it gives information about test coverage on both server and client side. Finally, it shows results of simple performance measurement.

6.1 Unit Testing Tools Used

6.1.1 Smalltalk/X - SUnit

Smalltalk/X have its own implementation of SUnit tool. SUnit is one of eldest unit testing tools ever. It runs tests divided into categories or, in 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. Figure 6.1 shows new version of TestRunner for SUnit.

6.1.2 C Language - Check Library

Check [22] 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.

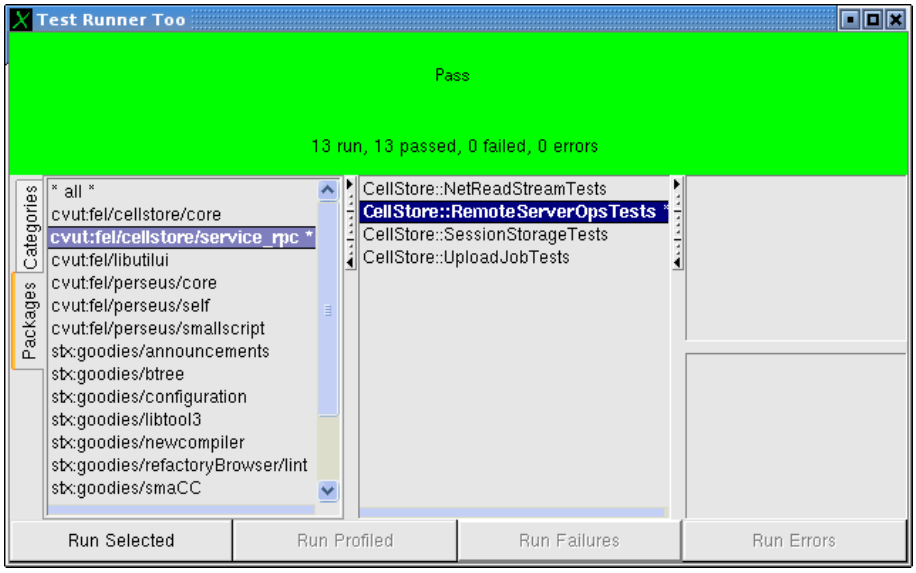


Figure 6.1: New TestRunner for Smalltalk/X SUnit tool

6.2 Test Coverage

6.2.1 Server Side - Smalltalk

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.

Also, some other components of server part were covered with tests. Test cases are realized in following classes in *CellStore-ClientServer-tests* category:

- *NetReadStreamTests*
  Tests of *NetReadStrem* class.
- *RemoteServerOpsTests*
  Testcase for checking behavior of all server operations.
- *RPCMTServerTests*
  Tests aimed at multi-thread processing in *SunRPC::RPCMTServer*.

- *SessionStorageTests*
  Tests of *SessionStorage* and *EnhancedSessionStorage* classes.

- *SocketJobsTests*
  Tests of socket job related classes (*SocketJob* and subclasses).

6.2.2 Client Side - C

On client side, all test are performed against the final API so whole operations or complex scenarios are tested. These tests are run against the current database and its content so first requirement is running database with remote server on local machine. First of all, all data from database are erased to prepare the environment. Due to this, running the tests against the database with important data is not recommended.

To partially check error handling, new protocol message was added. Calling the `ERROR_TESTING` procedure, current connection is switched to error testing mode and server functionality is changed. Then, modified server is able to partially simulate accidental disconnecting.

Using that, tests can check client library behavior in case that connection is lost. This is important for upload or download operations since they consist of more than one RPC call and other operations. If any of these operations crashes, an error must be always signaled properly.

Tests are located in `test/` subdirectory and are divided into several files by the part of API they are aimed at.

- `test/main.c`
  Main tests code. It contains procedures for tests setup.

- `test/test_core.c`
  Tests of common API, including tests for remote objects handling and maximum connection number exceeding.

- `test/test_dom.c`
  Tests covering DOM API.

- `test/test_errors.c`
  Tests checking various connection related error states. They're discussed above.

- `test/test_oodb.c`
  Tests of OODB API.

- `test/test_xmldb.c`
  Tests of XML:DB API, including upload and download operations.

These files have to be linked together after compilation. Easiest way to run tests is executing `make check` command from main library sources directory. Note that *Check* library has to be installed and CellStore database instance with RPC service must be run at the same machine. Again, I note that these tests are destructive.

6.3 Performance Measurements

To measure the overhead of client-server protocol, I made several test scenarios covering different parts of operations provided by protocol. These test cases are:

1. **Many Calls**.
   Two simple XML:DB operations are called (getting child collection and getting parent collection).
2. **OODB**.
   Several SELF operations are executed on XML resource (creating a string, message sends, value fetching).
3. **XQuery**.
   XQuery query is executed and the result is retrieved as XML string.
4. **Big Import**.
   Import of big XML file (1.2 MB) is performed.

These scenarios were implemented as local tests (*CellStore::PerformanceTest* and subclasses) using CellStore API directly, and also as *libcellstore* based client program (`test/performance.c` file).

These tests were executed on notebook with Intel Celeron M 430 (at 1.73GHz) processor and 1.5GB of RAM, running Smalltalk/X 5.4.4 on Debian GNU/Linux system. Client part of remote performance tests was run on another notebook with Intel Core2 Duo T7250 (at 2.00GHz) and 2GB ram, also running Debian GNU/Linux, connected via ethernet switch.

Executing times were compared and the result is shown in table 6.1. This table shows execution times of both remote and local API calls for each test case. To make the measurement more precise, these scenarios were called in loop.

- # **of loops** - number of loops being executed for each test case for both local and remote calls.
- **total time** - total time elapsed (all loops).
- **average time** - average time calculated to one loop. This is the most important column since it shows the execution time of each test case.

Table 6.2 shows the differences in execution times between remote calls and local API usage. Each test case consist of few remote calls so the total difference has to be divided by the number of calls. *Big import* test case consist of different operations than RPC calls so we won't calculate average time per call.

- **Calls per loop** - number of remote calls performed in each test case (in 1 loop).
- **Overhead per loop** - Difference between execution times for remote and local usage.

	Local API calls			Remote API calls		
Test case	# of loops [-]	total time [ms]	avg. time [ms]	# of loops [-]	total time [ms]	avg. time [ms]
Many calls	100 000	64335	0.643	1 000	80996	80.996
OODB	100 000	24044	0.240	1 000	162144	162.144
XQuery	1 000	73225	73.225	1 000	156264	156.264
Big import	1	65839	65839	1	68285	68285

Table 6.1: Performance measurements - running times

	Calls	Overhead	
Test case	per loop [-]	per loop [ms]	per call [ms]
Many calls	2	80.353	40.176
OODB	4	161.904	40.476
XQuery	2	83.039	41.520
Big import	-	2446	-

Table 6.2: Performance measurements - client-server protocol overhead

- **Overhead per call** - Previous time recalculated to one remote call. This value represents real overhead of remote call compared to local API call.

In all three RPC-based test cases, the delay of remote call is about 40 milliseconds. During the first two test cases, the CPU utilization didn't get over 15 per cent. So we can assume that this delay is caused by I/O waits. In case of XQuery test case, the CPU utilization was much higher. During the XML file import, the overhead ratio is about 3.7 per cent.

Chapter 7

Conclusion and Future Work

Client-server protocol for CellStore database engine was successfully designed. It covers all important facilities provided by CellStore, including full support of XML:DB API operations and direct access to SELF model interface.

CellStore database was improved with a native server implementing this protocol. For client application, C library was created to make new client application development as easy as possible. Its source code also contains few demo applications showing the API and its usage. As a side effect of server part implementation, several improvements of Smalltalk/X SunRPC implementation were made. They were sent to the author of Smalltalk/X and they will appear in future Smalltalk/X releases.

Of course, our protocol will be expanded in future as new functions will be implemented in CellStore database engine. The most important improvement are access control lists which will allow managing the security within the database. In this stage of CellStore project, it's not possible to include access lists to protocol since there is even no use case model specified. Also, creating of DOM model-based direct access to nodes of stored XML resources will rapidly improve XML:DB usability. It will also avoid unnecessary memory usage caused by DOM model objects building in server main memory.

The future of the client-server protocol is connected with the CellStore development. The quality of protocol design will be verified as new functionality requirements appear.

Bibliography

[1] CellStore project web-site.
`http://cellstore.felk.cvut.cz/`, 2009.

[2] eXist - Open Source Native XML Database.
`http://exist-db.org/`, 2009.

[3] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures, dissertation.* University of California, Irvine, 2000.
`http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm`.

[4] XML:DB Initiative for XML Databases.
`http://xmldb-org.sourceforge.net/`, 2003.

[5] W3C: SOAP Version 1.2 Part 1: Messaging Framework (Second Edition).
`http://www.w3.org/TR/soap12-part1/`, 2007.

[6] XML-RPC specification.
`http://www.xmlrpc.com/spec`, 2003.

[7] W3C: Web Service Definition Lanugage.
`http://www.w3.org/TR/wsdl`, 2009.

[8] Sedna XML database.
`http://modis.ispras.ru/sedna/`, 2009.

[9] NeoDatis ODB, open source object database for Java, .Net & Mono.
`http://www.neodatis.org/`, 2009.

[10] GemStone Smalltalk Object Server.
`http://www.gemstone.com/products/smalltalk/`, 2009.

[11] Oracle XML DB Developer's Guide 11g Release 1.
`http://download.oracle.com/docs/cd/B28359_01/appdev.111/b28369/toc.htm`, 2008.

[12] W3C: XQuery 1.0: An XML Query Language.
`http://www.w3.org/TR/xquery/`, 2007.

[13] W3C: Document Object Model (DOM) Level 3 Core Specification.
`http://www.w3.org/TR/DOM-Level-3-Core/`, 2004.

[14] 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.

[15] Multipurpose Internet Mail Extensions (MIME), Part One: Format of Internet Message Bodies.
`http://tools.ietf.org/html/rfc2045`, 1996.

[16] RFC 5531 - RPC: Remote Procedure Call Protocol Specification Version 2.
`http://tools.ietf.org/html/rfc5531`, 2009.

[17] CORBA Component Model, v4.0.
`http://www.omg.org/technology/documents/formal/components.htm`, 2008.

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

[19] RFC 4506 - XDR: External Data Representation Standard.
`http://tools.ietf.org/html/rfc4506`, 2006.

[20] Dave Marshall. Remote Procedure Calls (RPC).
`http://www.cs.cf.ac.uk/Dave/C/node33.html`, 1999.

[21] SunRPC Remote Procedure Call Implementation.
`http://www.exept.de:8080/doc/online/english/programming/goody_sunrpc.html`, 2009.

[22] Check: A unit testing framework for C.
`http://check.sourceforge.net/`, 2009.

Appendix A

List of Used Abbreviations

ACID Atomicity, Consistency, Isolation, Durability

ACK Acknowledgment

ACL Access Control List

API Application Programming Interface

Corba Common Object Requesting Broker Architecture

DOM Document Object Model

EoF End of File

FTP File Transfer Protocol

HTTP Hypertext Transfer Protocol

HTML HyperText Markup Language

IANA Internet Assigned Numbers Authority

JDBC Java Database Connectivity

JSP Java Server Pages

MIME Multipurpose Internet Mail Extensions

NFS Network File System

OID Object Identifier

OODB Object-Oriented Database

OOP Object-Oriented Pointer or Object-Oriented Programming

ONC RPC Open Network Computing Remote Procedure Call

OSI Open Systems Interconnection

PHP Hypertext Preprocessor (former meaning was Personal Home Page)

PL/SQL Procedural Language/Structured Query Language

REST Representational State Transfer

RPC Remote Procedure Call

SAX Simple API for XML

SMTP Simple Mail Transport Protocol

SOAP Simple Object Access Protocol

TCP Transmission Control Protocol

UDP User Datagram Protocol

URI Uniform Resource Identifier

XDR eXternal Data Representation

XSL Extensible Stylesheet Language

W3C World Wide Web Consortium

WebDAV Web-based Distributed Authoring and Versioning

WSDL Web Services Description Language

XML eXtensible Markup Language

XML:DB 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. Note that client library cannot be compiled on Windows.

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. Debian version is installed in `/opt/stx/` so run

   ```
   user@box:~$ /opt/stx/bin/stx
   ```

4. Load CellStore, RPC Service and others from Subversion repository. Evaluate following code in Workspace (type it in, then select and chose `Do it` from right-click menu).

   ```
   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. This requires Subversion to be installed on system.

5. Create your database instance and register RPC service. Then startup services. You can either use *RemoteServerWizard* class (see chapter 5.1.7) or evaluate code similar to following.

```
|instance|
instance := CellStore::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.

- `lib` or `library` (default) - 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 a 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 previously installed library from system. Also needs root privileges.
- `check` - Run tests on local database, need *Check* [22] library installed on your system. Note that tests are destructive, so do not have important data in database before running these tests.
- `demo` - Builds demo applications. See section B.3.
- `client` - 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.
- `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.

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 and one sample code used in developers' tutorial in chapter C. 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.
- `tutorial` - see tutorial in section C.

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

Programming with the Client Library

Following tutorial gives brief information about usage of *libcellstore* API. It expects that *libcellstore* is successfully installed on system.

All *libcellstore* API functions and types can be declared by including the `cellstore.h` file located on include search path. In this tutorial we will open some file, so `stdio.h` is needed, too.

```
#include <cellstore.h>
#include <stdio.h>
```

Every operation returns status code so we need a variable for it.

```
CellStoreStatus status;
```

Connected session is identified by *CellStoreSession* variable.

```
CellStoreSession session;
```

Results of operations are returned via pointers passed before other function arguments. Error codes have to be handled. Following code will initiate the session.

```
status = cs_session_init_simple ( &session, "localhost" );
if (status != 0) {
   printf("Connection error.\n");
   return 1;
}
```

Every remote object from database is identified by *CellStoreObject* variable

```
CellStoreObject collection;
```

We will open the root collection of XML database. Username and password are ignored since CellStore has not ACLs implemented yet.

```
status = cs_xmldb_get_rootcollection ( &collection, "username",
                                       "password", session );
HANDLE ( status );
```

HANDLE() is a macro that is not defined in `cellstore.h`. It will help us to work with status codes. It simplifies processing of operation statuses and prints error message. Real application requirements on error handling will be probably different.

`cs_printError()` takes two arguments, status code and a message text. If 0 is passed instead of message, last error message is taken. Macro have to be defined at the beginning of the source as follows:

```
#define HANDLE(status)  if ( status != CS_STATUS_OK ) {\
   cs_printError ( status, 0 );\
   cs_session_destroy ( session );\
   return 1;\
}
```

Now, we will open a file with XML document.

```
FILE * file = fopen ( "books.xml", "r" );
if ( file == 0 ) {
   perror ( "File open error" );
   cs_session_destroy ( session );
   return 1;
}
```

We will use function that creates new XML resource within the root collection and uploads data from previously opened file. Resource ID (set as "books.xml") is an identifier in XML database.

```
CellStoreObject resource;
status = cs_xmldb_quick_xml_upload ( &resource, file, "books.xml",
                                     collection, session );
fclose ( file );
HANDLE ( status );
```

Previous function gave us reference to *XMLResource* object on server. We don't need this object since data are safely stored in database. It's a good practise to drop objects we do not need. Following call drops the object reference and object will be freed during the next upcoming garbage collecting process on server.

```
status = cs_object_drop( resource, session );
HANDLE ( status );
```

Reference variable is now invalid. Following call tries to retrieve parent collection of the resource. If error occurs, status is nonzero and global last error message variable is set. We won't use our macro because we don't want to exit the program.

```
status = cs_xmldb_resource_get_parent(&collection, resource, session);
if ( status != 0 ) { cs_printError(status, 0); }
```

Something like following will be printed:

```
[No such object] Object does not exist
```

The text in the brackets is a textual identification of status code. The text after brackets is an error message.

Following call performs XQuery and returns reference to *ResultSet*

```
CellStoreObject result;
status = cs_xmldb_xquery ( &result, "doc('xmldb:books.xml')//title",
                           collection, session );
HANDLE ( status );
```

There exist several way how to retrieve resource or XQuery result from server.

For large data, `cs_xmldb_download()` is recommended. It will retrieve the data directly into the opened file. `cs_xmldb_download_string()` downloads the data to new string. These functions are very polite to server memory because the data is being generated directly to network stream.

Function `cs_xmldb_result_as_xml()` is better for small data because it doesn't initialize auxiliary connection for transfer. On the other hand, it requries the result XML to be allocated as a temporary string on server.

In this example, we will use the last option.

```
char * str;
status = cs_xmldb_result_as_xml( &str, result, session );
HANDLE ( status );
```

Print it. We must free the string because it was allocated dynamically.

```
printf("%s\n", str);
free (str);
```

That's all so we will close the session

```
cs_session_destroy ( session );
```

Whole source code is printed without comments on figure C.1.

To compile the program, you must link it with the *libcellstore* library (shared).

```
gcc -o tutorial tutorial.c -lcellstore
```

```
#include <cellstore.h>
#include <stdio.h>
#define HANDLE(status)  if ( status != CS_STATUS_OK ) {\
      cs_printError ( status, 0 );\
      cs_session_destroy ( session );\
      return 1;\
}

int main ( int argc, char * argv[] ) {
   CellStoreStatus status;
   CellStoreSession session;
   status = cs_session_init_simple ( &session, "localhost" );
   HANDLE ( status );
   CellStoreObject collection;
   status = cs_xmldb_get_rootcollection ( &collection, "username",
                                          "password", session );
   HANDLE ( status );
   FILE * file = fopen ( "books.xml", "r" );
   if ( file == 0 ) {
      perror ( "File open error" );
      cs_session_destroy ( session );
      return 1;
   }
   CellStoreObject resource;
   status = cs_xmldb_quick_xml_upload ( &resource, file, "books.xml",
                                        collection, session );
   fclose ( file );
   HANDLE ( status );
   status = cs_object_drop( resource, session );
   HANDLE ( status );
   status = cs_xmldb_resource_get_parent(&collection, resource, session);
   if ( status != 0 ) { cs_printError(status, 0); }
   CellStoreObject result;
   status = cs_xmldb_xquery ( &result, "doc('xmldb:books.xml')//title",
                              collection, session );
   HANDLE ( status );
   char * str;
   status = cs_xmldb_result_as_xml( &str, result, session );
   HANDLE ( status );
   printf("%s\n", str);
   free (str);
   cs_session_destroy ( session );
}
```

Figure C.1: Whole tutorial source code without comments

Appendix D

CD Content

```
|-- bin/ - Binaries of client library.
|     |
|     |-- linux-i686/ - 32 bit version (Intel).
|     |
|     |-- linux-x86_64/ - 64 bit version (AMD).
|
|-- doc/ - Documentation.
|     |
|     |-- client/ - Client library (Doxygen).
|     |
|     |-- server/ - Server part (Smalltalk/X documentation HTML export).
|
|-- src/ - Code sources.
|     |
|     |-- svn/ - Copy of SVN repository with all packages needed to load CellStore.
|     |
|     |-- client/ - Client library sources.
|     |
|     |-- server/ - Server part (without dependencies).
|
|-- thesis/ - Thesis text.
      |
      |-- src/ - LaTeX sources of thesis.
      |
      |-- plickm1_thesis.pdf - Thesis compiled to PDF.
```