QRODS User Manual

v0.0.1

MoDCS Research Group

[http://www.modcs.org]

CIn - Centro de Informatica UFPE

Cidade Universitaria - 50740-540 - Recife - Brazil

- Tel +55 81 2126.8430 -

August 11, 2015

Contents

1 Overview

The evolution of the data center and data has been dramatic in the last few years with the advent of cloud computing and the massive increase of data due to the Internet of Everything. The Integrated Rule-Oriented Data System (iRODS) helps in this changing world with virtualizing data storage resources regardless the location where the data is stored.

This user guide explains and demonstrates a library that extends the Qt abstract model interface to provide access to the iRODS data system from within the Qt framework. Qt is widely used for developing graphical user interface software applications that are display platform agnostic. This library intends to benefit Qt developers by enabling a transparent iRODS access. Moreover, it will allow developers to implement applications that access an iRODS data system to populate a single model that can be displayed as a standard Qt tree like structure.

This manual is organized as follows. Section 2 presents the QRODS library and shows its functionalities. Section 3 describes an example implemented, named RODEX, that uses QRODS library.

2 QRODS library

The following subsections present the QRODS library. First, we present preliminaries information followed by explaining how to include the QRODS library into a Qt application. Next, an overview of the library features is shown. Finally, the architecture and the class diagram are presented.

2.1 Preliminaries

This section presents important concepts for a better understanding of QRODS library. First, a brief overview related to iRODS is presented. Next, some concepts regarding QAbstractItemModel class and Jargon API are discussed.

2.1.1 iRODS

iRODS has become a powerful, widely deployed system for managing significant amount of data that requires extendable metadata. Typical file systems provide only limited functionality for organizing data and a few (or none) for adding to the metadata associated with the files retained. Additionally, file systems are unable to relate or structure what limited metadata is available and provide only a platform from which to serve unstructured file data. Within several fields, scientific research evolving instrumentation capabilities have vastly expanded the amount and density of unstrucuted file data, in which standard file systems can be a limiting factor in the overall use of data. The adopted iRODS versions to test the QRODS API were 4.0.3 and 4.1.3.

iRODS can be classified as a data grid middleware for data discovery, workflow automation, secure collaboration and data virtualization. As illustrated in Figure 1, the middleware provides a uniform interface to

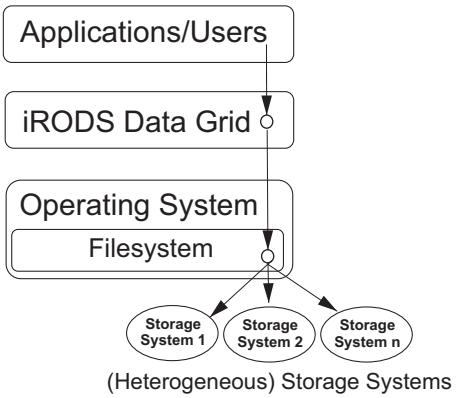

Figure 1: iRODS Overview [3].

heterogeneous storage systems (POSIX and non-POSIX). iRODS lets system administrators roll out an extensible data grid without changing their infrastructure and accessing through familiar APIs. The reader should refer to [2] for more details about iRODS environment.

2.1.2 QAbstractItemModel class

Qt is a comprehensive C++ application development framework for creating crossplatform GUI applications using a "write once, compile everywhere" approach. Qt lets programmers use a single source tree for applications that run on Windows, Linux, and Mac OS X as well as mobile devices. Qt libraries and tools are also part of Qt/Embedded Linux alike product that provides its own window system on top of embedded Linux.

The Qt framework provides a model/view controller approach (Qt MVC). QAbstractItemModel class provides an abstract interface for item model classes. Thus, programmers can populate one single model; such model enables the use of different ways for displaying a group of contents (files and directories). A QTreeView, for example, implements a tree representation of items from a model, whereas a QTableView implements a standard table representation.

2.1.3 Jargon API

Jargon is an API that implements the communication with iRODS protocol. The API allows development of iRODS-enabled Java applications. It is useful for developing mid-tier applications and services, as well as desktop-clients. These libraries also provide a foundation for a new set of interfaces that come with iRODS. Besides iRODS protocol implementation, Jargon is also able to acess iRODS data.

Jargon is implemented in Java, providing support for Java applications and not for other programming languages. Therefore, the iRODS Rest API based on Jargon has been developed to overcome such issue. The REST API provides support for developers to implement different client use cases for iRODS. Next session presents the QRODS, a library that adopts Jargon REST API to conduct the access from Qt applications to iRODS data system.

2.2 Features

QRODS is a library that enables software engineers to build Qt graphical user interfaces (GUI) which can access the iRODS storage platform. The current version of QRODS implements essential manipulation functionalities of files and directories (collections) including metadata object information. Therefore, the QRODS may perform the following proceedings:

- Download, upload and delete files;
- Create and delete collections;
- Add and delete metadata from an object;
- List content (files, metadata, collections) and
- Lazy loading (collection and files - no metadata).

In order to be able to perform any of the above functionalities, QRODS users must instantiate a QRODS object. After that, it is possible to call each specific method to conduct those operations.

2.2.1 Download, upload and delete files

- A call to the *getFile(QString &remotePath, QString &localPath)* method is conducted to download a file from the iRODS storage system, where *remotePath* is a string that represents the path on the iRODS system and *localPath* is a string that represents the local path.
- A call to the *uploadFile (QString &remotePath, QString &localPath)* method is performed to upload a file.
- A call to the *removeDataObject(QString &path, bool force)* method is performed to delete the file specified in the parameter path.

2.2.2 Create and delete collections

- To create a collection, a call to *addCollection(QString &path)* is performed, *path* is a string parameter that represents the path where the collection will be created.
- A call to the *removeCollection(QString &path)* method is performed to delete a collection, where *path* is a string that represents the path of the collection to be removed.

2.2.3 Add, delete and get a metadata

- A call to the *addCollectionMetadata(QString &path, MetadataEntry &map)* method is performed to add a metadata to a collection, where *path* is a string that represents the path of the collection in which the metadata will be added and *map* is a *MetadataEntry*.

- A call to the *removeDataObject(QString &path, bool force)* method is performed to delete a metadata from an object, where *path* is a string that represents the path of the metadata object to be removed.
- A call to the *getDataObjectMetadata(QString &path)* method is performed to get data object metadata, where *path* is a string that represents the path of the metadata to be received.

2.2.4 List content (files, metadata, collections)

- A call to the *getDataModel()* method is performed to get the data model to list files or collections. After that, it is necessary to call the *addView(QAbstractItemView *view)()* method that is responsible for adding the desired view (e.g., Tree, List or Table view).
- A call to the *setModel(QAbstractItemModel *model)* method is conducted to get the metadata information to be listed.

2.2.5 Lazy loading

Lazy loading functionalities have been implemented for collections and files.

- A call to the *getCollectionDataLazy(FileListingEntry entry)* method is performed to get the collection data using the lazy loading functionality, where *entry* is a FileListingEntry. An example of this method call is *d->collectionClient->getCollectionDataLazy(entry)*; where *d* is an instance of *QRODS* class, *collectionClient* is a *CollectionClient* atribute defined in *QRODS* and *entry* is a *FileListingEntry*.

2.2.6 SSL communication

SSL enables security during data exchanging between QRODS and iRODS server. All data, including user and password credentials are encrypted and sent through HTTPS commands. This kind of functionality makes possible the use of our library in business environments as it avoids data sniffing between QRODS and iRODS.

- A parameter in the configuration file sets if the SSL will be used or not.
- The library modules detects the parameter and create a connection using the proper protocol (HTTP or HTTPS).
- For using security through SSL connections, it is necessary to configure the web server to accept this kind of connections. You may check the appendix 3.2.4 for configuring an Apache Tomcat web server to run apps with HTTPS and SSL.

2.3 How to include the QRODS library into a Qt application

This section illustrates the applicability of the proposed QRODS library through a simple Qt application. The main goal is to present the main QRODS features available that allows Qt developers to implement a Qt software that store data in an iRODS server.

2.3.1 Creating a dialog

The first step is to create an empty dialog in order to allow us to show the QRODS functionalities. Figure 2 presents the steps to configure a dialog that is created inside the QRODS library folder. The second step is to add a List View and a Push Button in the UI dialog created.

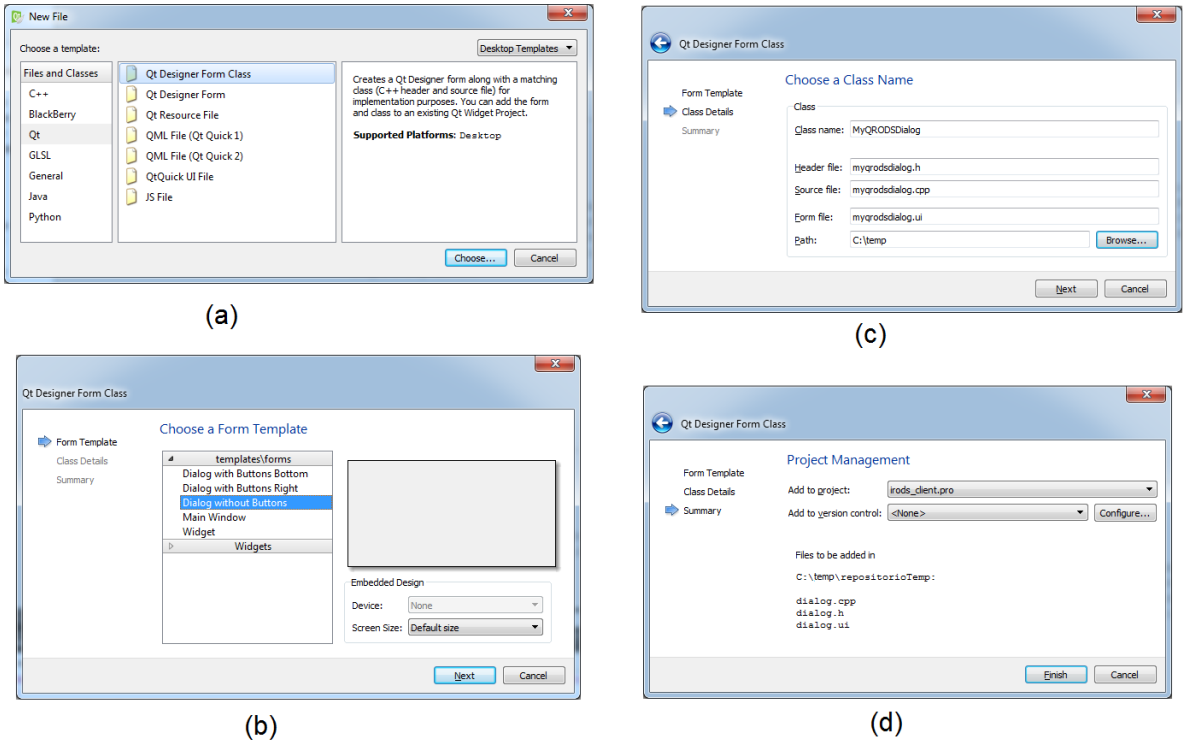

Figure 2: Creating a dialog example.

2.3.2 Uploading a File example

To upload a file, we changed the button name as well as the associated text to Upload File. After that, we create a clicked slot which is associated to the button. The event code is depicted in Listing 1.

Listing 1: Code for uploading a file.

```
void MyQRODSDialog::on_uploadButton_clicked()
{
    QString remotePath("/tempZone/home/rods/test.txt");
    QString localPath("C:/temp/test.txt");

    qrods->uploadFile(remotePath, localPath);
}
```

2.3.3 Configuring the constructor method and main.cpp

Before running this Qt application, it is necessary to configure the constructor method of our dialog. Listing 2 presents the constructor code. In this constructor, the host, user and password to connect to an iRODS server must be set. In addition, the port and the path must also be configured. After setting the constructor parameters, it is necessary to conduct a setup in the dialog.h created. For instance, we need to include our library in the Qt project as #include <qrods.h>. In addition, a pointer to QRODS must be defined in the private section of the dialog.h such as QRODS *qrods;. The following step is to configure the main.cpp as the code depicted in Listing 3.

Listing 2: Code for the constructor dialog.

```
MyQRODSDialog::MyQRODSDialog(QWidget *parent) :
    QDialog(parent),
    ui(new Ui::MyQRODSDialog)
{
    ui->setupUi(this);

    QString host("host");
    QString user("user");
    QString pass("password");

    qrods = new QRODS(this, host, port, user, pass);
    ui->listView->setModel(qrods);
    qrods->connectToServer("path", false);
}
```

Listing 3: Main.cpp example.

```
#include "myqrodsdialog.h"
#include <QApplication>
#include <QDebug>
#include "qrods.h"
#include <QString>
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    MyQRODSDialog w;
    w.setWindowTitle("Example Dialog");
```

```
    w.show();
    return a.exec();
}
```

Figure 3 shows the created Qt application in which we have the picture before and after running the upload process.

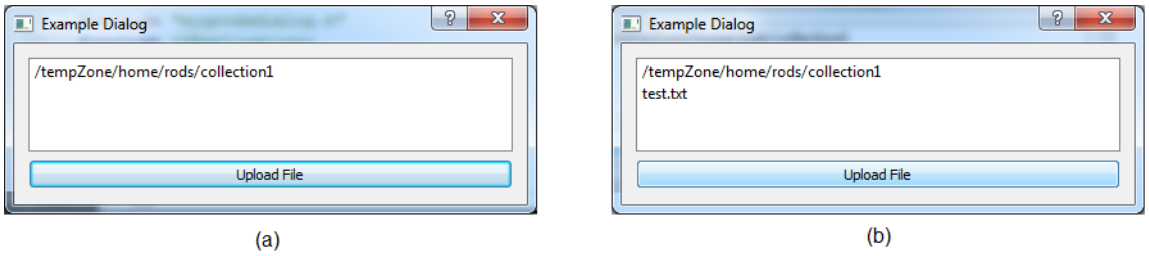

Figure 3: Uploading a file: (a)before the upload; (b)after uploading the file.

2.3.4 Downloading a File example

To download a file, we added a button and named it as uploadButton and associated to it the Download File text. After that, we created a clicked slot to the button. The following code represents the event that downloads a file from the iRODS system. Therefore, our simple Qt project example looks like the Figure 4.

Listing 4: Code for downloading a file.

```
01 void MyQRODSDialog::on_downloadButton_clicked()
02 {
03     QString remotePath("/tempZone/home/rods/test.txt");
04     QString localPath("C:/temp/test.txt");
05     qrods->getFile(remotePath, localPath);
06 }
```

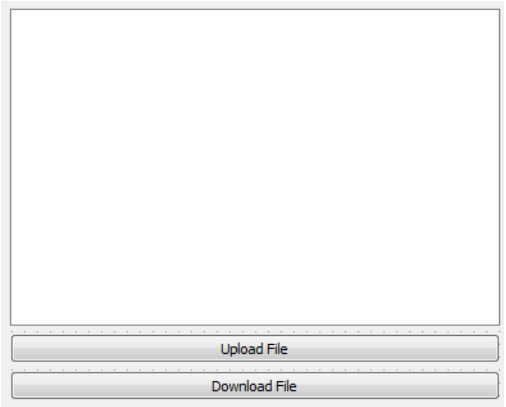

Figure 4: Downloading a file.

2.3.5 Deleting a File example

To delete a file, we added a button and named it as deleteButton and associated to it the Delete File text. After that, we created a clicked slot to the button. The following code is the event that deletes a file from the iRODS system. Therefore, our simple Qt project example is shown through Figure 5.

Listing 5: Code for deleting a file.

```
01 void MyQRODSDialog::on_deleteButton_clicked()
02 {
03     QString remotePath("/tempZone/home/rods/test.txt");
04     qrods->removeDataObject(remotePath);
05 }
```

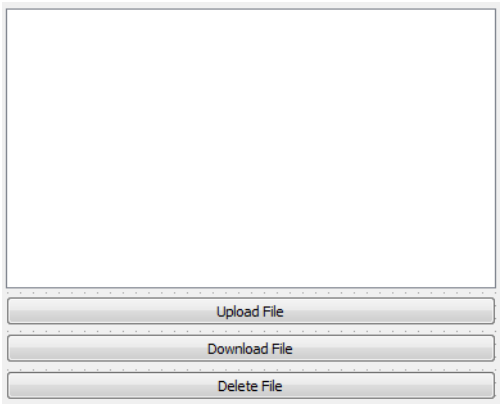

Figure 5: Delete a file.

2.3.6 Creating a Collection example

To create a collection example, we added a button, named it as createCollectionButton and associated to it the Create Collection text. Afterwords, we created a clicked slot to that button. The following code is associated to that event that creates a collection on the iRODS system. Therefore, our simple Qt project example looks like the Figure 6.

Listing 6: Code for creating a collection.

```
01 void MyQRODSDialog::on_createCollectionButton_clicked()
02 {
03     QString remotePath("/tempZone/home/rods/newcollection");
04     qrods->addCollection(remotePath);
05 }
```

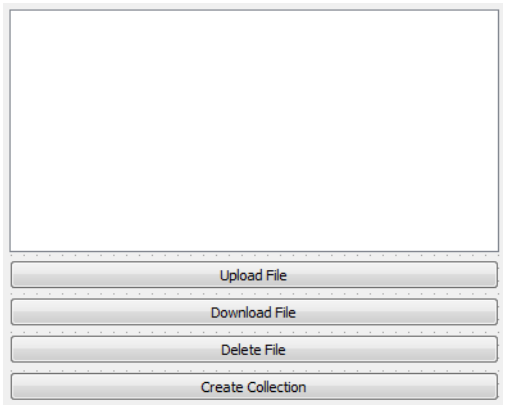

Figure 6: Creating a collection.

2.3.7 Removing a Collection example

To remove a collection, we added a button, named it as removeCollectionButton and associated to it the Remove Collection text. After that, we created a clicked slot to the button. The following code is associated to that event that removes a collection on the iRODS system. Therefore, our simple Qt project example looks like the Figure 7.

Listing 7: Code for removing a collection.

```
01 void MyQRODSDialog::on_removeCollectionButton_clicked()
02 {
03     QString remotePath("/tempZone/home/rods/newcollection");
04     qrods->removeCollection(remotePath);
05 }
```

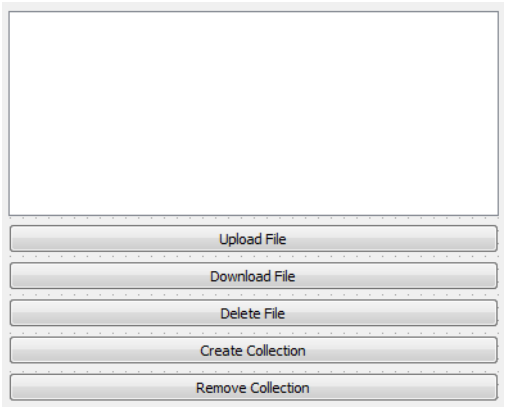

Figure 7: Removing a collection.

2.3.8 Creating a Metadata example

To create a metadata example, we added a button, named it as addMetadataButton and associated the Create Metadata text to it. After that, we created a clicked slot to the button. The following code is associated to the event that creates a metadata on the iRODS system. Therefore, our simple Qt project example looks like the

Figure 8.

Listing 8: Code for creating a metadata.

```
01 void MyQRODSDialog::on_addMetadataButton_clicked()
02 {
03     QString attribute ("attribute");
04     QString value ("value");
05     QString unit ("unit");
06     MetadataEntry entry(attribute, value, unit);
07     QString remotePath("/tempZone/home/rods/test.txt");
08     qrods->addCollectionMetadata(remotePath, entry);
09 }
```

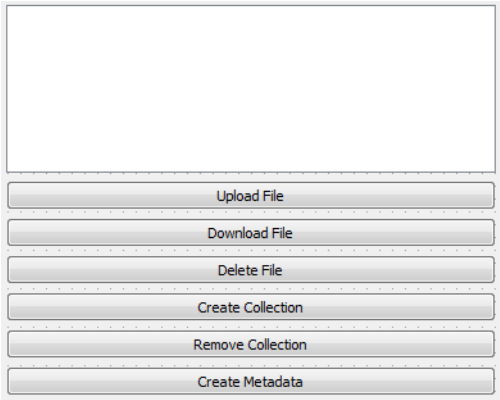

Figure 8: Creating a metadata.

2.3.9 Removing a Metadata example

To remove a metadata example, we added a button, named it as removeMetadataButton and associated to it the Remove Metadata text. After that, we created a clicked slot to the button. The following code is associated to the event that removes a metadata on the iRODS system. Therefore, our simple Qt project example looks like the Figure 9.

Listing 9: Code for removing a metadata.

```
01 void MyQRODSDialog::on_removeMetadataButton_clicked()
02 {
03     QString attribute ("attribute");
04     QString value ("value");
05     QString unit ("unit");
06     MetadataEntry entry(attribute, value, unit);
07     QString remotePath("/tempZone/home/rods/test.txt");
```

```
08      qrods->deleteCollectionMetadata(remotePath, entry);
09  }
```

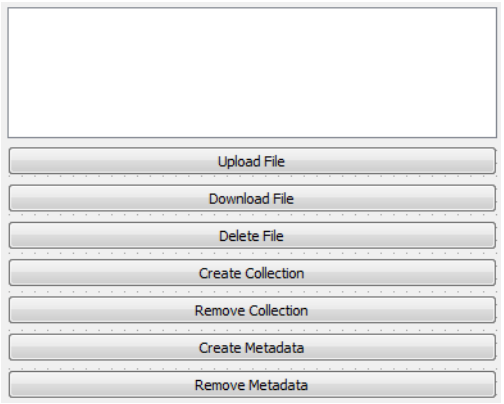

Figure 9: Removing a metadata.

2.3.10 Getting the Metadata Information

To get the metadata information, we added a button, named it as getMetadataButton and associated to it the Get Metadata text. After that, we created a clicked slot to the button. The following code is associated to the event that gets the metadata information from the iRODS system. Therefore, our simple Qt project example looks like the Figure 10.

Listing 10: Code for getting the metadata information.

```
01  void MyQRODSDialog::on_getMetadataButton_clicked()
02  {
03      QString remotePath("/tempZone/home/rods/test.txt");
04      QList<MetadataEntry *> *list = qrods->getDataObjectMetadata(remotePath);
05      QMessageBox messageBox;
06      if(list->size() != 0)
07      {
08          messageBox.setText("attribute " + list->at(0)->attribute() + "\n value " +
                list->at(0)->value() + "\n unit " + list->at(0)->unit());
09          messageBox.exec();
10      }
11      else
12      {
13          messageBox.setText("The object has no metadata");
14          messageBox.exec();
15      }
```

16 }

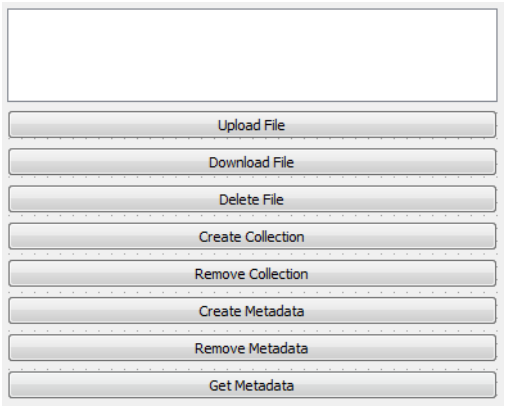

Figure 10: Getting a metadata information.

2.4 Library Architecture

Figure 11 depicts the QRDOS library architecture. As previously mentioned, the main goal of this library is to provide an interface that allow Qt developers to build applications with graphical user interfaces communicating to the iRODS data system. In order to accomplish that, Jargon REST API has been adopted to implement the iRODS protocol communication between our proposed library and iRODS. Jargon is an API that implements the communication with iRODS protocol. Although Jargon has been implemented in Java, it also provides a REST API for allowing tools implemented through different languages to communicate with iRODS. Therefore, QRODS library performs REST calls to the Jargon API that communicates with iRODS protocol through XML.

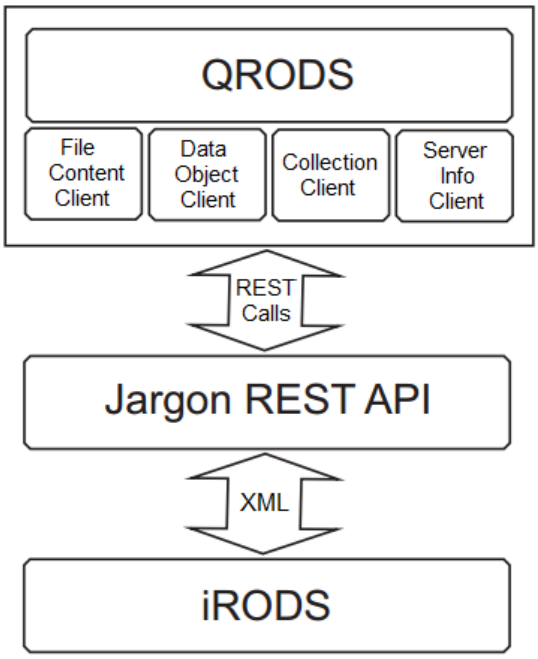

Figure 11: QRODS library architecture.

The QRODS library has been composed of four different types of clients: (i) file content client, which

is responsible to conduct all operations related to folders; (ii) data object client implements the metadata operations; (iii) collection client, which provides all functionalities for data collection; and (iv) abstract HTTP client, which provides secure communication functionalities. The following lines provide more details about the functionalities implemented for each type of client.

FileContentClient class

FileContentClient class manages iRODS files through two main functions: upload and download objects. As the name suggests, Qt applications upload files to iRODS using the *uploadFile()* function. The local and remote file names (including the entire file path) are parameters of that method. The equivalent iRODS command for this method is *iput*. Similar to uploading functionality, Qt applications download iRODS files by adopting the `downloadFile()` function. Local and remote paths represent the parameters used on that method. The equivalent iRODS command for this function is *iget*.

DataObjectClient class

DataObjectClient class manages metadata, which includes the add, remove and list functions. Qt applications associate a metadata to an iRODS object through the *DataObjectClient()* constructor. To remove an object metadata, the *removeDataObject()* method is adopted. A metadata can be listed by the *getDataObjectMetadata()* method or by its asynchronous version *getDataObjectMetadataAsync()*. The corresponding iRODS command for this method is *imeta*. In addition, a metadata collection can be added by the *addCollectionMetadata()* method.

CollectionClient class

CollectionClient class manages iRODS collections. This class contains methods to list collection contents, delete and create new collections. The equivalent iRODS commands to the CollectionClient class functionalities are *ils*, *irm* and *imkdir*. In order to delete a collection, the *removeCollection()* function must be called and the collection path is passed as parameter. Similarly, the *createCollection()* method receives the remote path as parameter to create new iRODS collections.

The listing functionality is implemented in two different ways:

- *Asynchronous Listing*: Using the *getCollectionDataAsync()* method, all the content of a collection is asynchronously listed. However, depending on the collection size, this function may take some time to finish.

- *Lazy Listing*: The asynchronous *getCollectionDataLazy()* method is adopted to perform collection lazy listing. By using this function, just a group of collection objects is retrieved per function call. This method is called several times to list all the collection objects. Therefore, this functionality is suitable for huge collections.

AbstractHTTPClient class

This class provides asynchronous functions associated with GET, POST, PUT AND DELETE HTTP calls. The `doGet()` method retrieves information identified by the requested URI. The `doPost()` method sends a post request with an enclosed entity to a given resource identified by the respective URI.The `doPut()` method requests the enclosed entity to be stored under the supplied requested URI. The `doDelete()` method requests that the resource identified by the requested URI to be deleted.

Qt Application Architecture using QRODS

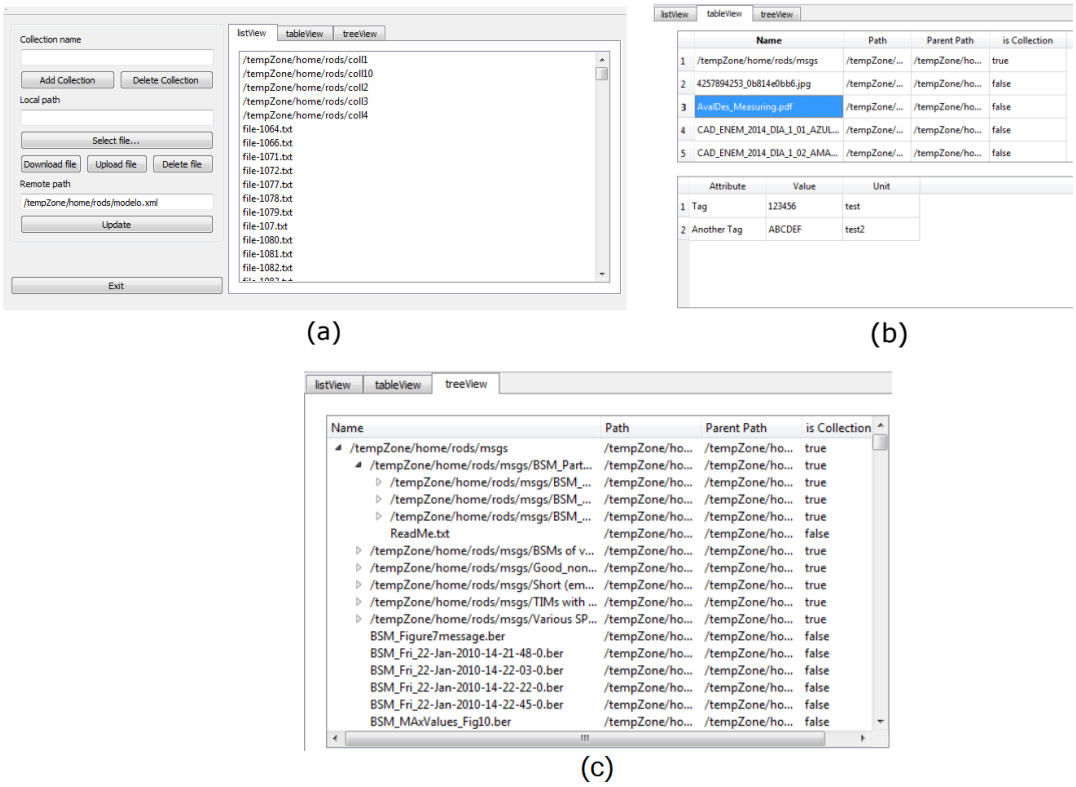

Figure 12: QRODS: List View (a), Table View (b) and Tree View (c).

Figure 11 presents the architecture of a Qt application that includes the QRODS library to directly communicate with iRODS data system. Besides that, it is important to stress that the list, tree and table views were implemented in our QRODS library. Therefore, Qt developers may show the iRODS data as a list, a tree or a table view as depicted in Figure 12.

QRODS class Diagram

Figure 13 presents the QRODS class diagram. QAbstractItemModel is a Qt model view class which provides the abstract interface for item model classes. This class defines functions that are required to support table, tree and list views. QAbstractItemModel class cannot be directly instantiated. Instead, a subclass must be implemented to create new models.

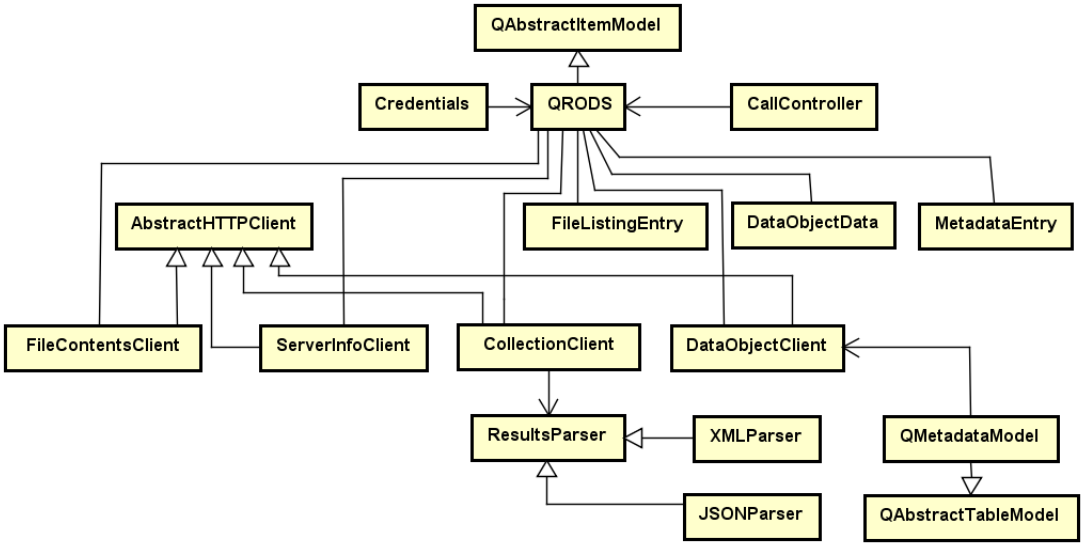

Figure 13: QRODS class diagram.

QRODS extends QAbstractItemModel overriding its main methods, such as: (i) *index()*, which returns the item model index specified by a given row, column and parent index; (ii) *parent()*, which returns the item model parent of a given index; and (iii) *headerData()* that returns the data for the given role and section in the header with the specified orientation.

QRODS is associated with one or more clients (e.g., FileContentsClient, ServerInfoClient, CollectionClient or DataObjectClient). The reader is redirect to Section 2.4 for more details about these clients.

FileListingEntry represents one node in the file system (file or collection) having pointers to the father and child objects. FileListingEntry is used by the aforementioned three types of views (see Section 2.4).

AbstractHTTPClient provides secure communication through HTTP commands using SSL protocol and encrypted channels. In addition, passwords have been stored using AES 128 standard. MetadataEntry class encapsulates metadata information for a specific QRODS object. More specifically, it aggregates the corresponding attribute name, value and unit.

2.5 QRODS Test Suite

Aiming to guarantee that all functionalities of QRODS satisfied the project specifications a number of functional tests were implemented. Functional testing is a type of black box testing that bases its test cases on the specifications of the software component under test. Functions are tested by feeding them input and examining the output, and internal program structure is rarely considered (different from white-box testing).

2.5.1 Tests coverage

The QRODS Test Suite covers the main functionalities of QRODS library. Therefore, tests were create intending to encompass the three types of objects:

- **Files**: Tests have asserted the proper operation of downloading and uploading files.

- **Collections**: Tests to add and remove collections.
- **Metadatas**: Tests to add and remove collections.

2.5.2 Source code details

The QRODS Test Suite is delivered separated from core library source code in a project called QRODS-TEST, as illustrated in Figure 14.

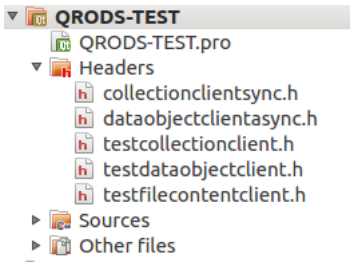

Figure 14: QRODS Test Suite folder illustration.

Following we list and describe shortly the functional tests that QRODS Test Suite embraces. The tests are associated with the three main client classes, they are: DataObjectClient, CollectionClient and FileContentClient.

- **TestFileContentClient**: This test includes to download and upload files:
  - **Download files**. The download test focuses on to fetch a file from iRODS.
    - *testGetFile()*: Test to fetch a file from the iRODS. The steps are the following:
      - Create a local file
      - Assert the local file exists
      - Upload the file
      - Checks (in a loop) whether the file was uploaded
      - Verifies the remote file existence
      - Delete the local file
  - **Upload files**. Related to this functionality a larger number of tests were implemented, they are:
    - *testUploadFilesWithLargeNames()*: Test the upload of files with different name sizes in terms of caracteres (8,65,128,200,251).
    - *testUploadOnlyOneSmallFile()*: Uploads a file of 1.3 KB and verifies whether the file was uploaded.
    - *testUpload_10000Files_10MB()*: Uploads 10000 files of 10 MB and verifies whether the files were uploaded.
    - *testUpload_1000Files_100MB()*: Uploads 1000 files of 100 MB and verifies whether the files were uploaded.

        * *testUpload_100Files_1GB()*: Uploads 100 files of 1 GB and verifies whether the files were uploaded.
        * *testUpload_10File_10GB()*: Uploads 10 files of 10 GB and verifies whether the files were uploaded.
        * *testUpload_1File_100GB()*: Uploads 1 file of 100 GB and verifies whether the files were uploaded.
        * *testUploadDifferentSizeFiles()*: Test the upload of files with different name sizes in terms of caracteres (8,65,128,200,251).

- **TestCollectionClient**: CollectionClient has the goal of managing collections. Thus, the funcionalities are add and remove collectoins. The respective method tests are:
  - **Add Collections**
    * *testAddCollection()*: Test to add one collection with one file inside and assert its existence on iRODS.
    * *testAddManyCollections()*: Test to add a number of collections (variable numberOfCollections) and assert its existence.
  - **Remove Collections**
    * *testRemoveCollection()*: Test to remove one collection from iRODS and assert that it does not exists anymore.

- **TestDataObjectClient**: DataObjectClient is responsible for dealing with metadatas. The covered functionalities are:
  - **Add metadata**
    * *testAddFileMetadata()*: Test to add metadata to a file on iRODS
    * *testMetadataCapacity()*: Test to insert a huge number of metadata to a file on iRODS
  - **Remove metadata**
    * *testRemoveDataObject()*: Test to remove a file from the iRODS

There are two auxiliary classes used by the tests, CollectionClientSync and DataObjectClientAsync. CollectionClientSync deals with the asynchronous requests and signals/slots, making it possible to assert the return of a collection only when it really has arrived to the client. Another important class is the main.cpp used to run all test suite at once. DataObjectClientAsync, similarly to CollectionClientSync, tries to fetch metadatas managing signals/slots, allowing the tests to assert the asynchronous request. There is a class used to run all tests at once (main.cpp) but it is described in the next subsection.

Finally, there are some macros, inherent to all tests, that are important to highlight here:

- **QSKIP**: You can use it to skip tests that wouldn't make sense in the current configuration. This macro is commented in all tests, but if user desires to run only a subset of test she just needs to uncomment.

- **QCOMPARE**: compares an actual value to an expected value using the equals operator.
- **QVERIFY**: checks whether the condition is true or not.

2.5.3 How to build and run the QRODS Test Suite on QTCreator

The only task to be done is to update the infomation in config.ini setting credentials to irods. Build and run the project using the QTCreator commands. Finally, press the play button illustrated in Figure 15 to build and run the tests.

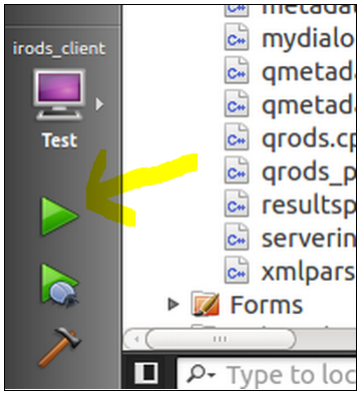

Figure 15: Running the tests.

2.5.4 How to build and run the QRODS Test Suite on Jenkins

Jenkins is an application that monitors executions of repeated jobs, such as building a software project. Considering that the user already have a Jenkins installed and a project configured, the user only have to insert a script to run the tests. Figure 16 shows the corresponding script.

Figure 16: Jenkins commands.

3 RODs EXplorer - RODEX

The main goal of this section is to illustrate the applicability of the proposed QRODS library in an implemented Qt application. Therefore, RODs EXplorer (RODEX) was developed to show the main functionalities of our library. RODEX application is able to upload and download files, create and delete files or collections. This application also allow one to add, delete and list file metadata, and to list content (files, metadata or collections) of iRODS data system. In order to add such functionalities to the Qt framework, Qt developers just need to include QRODS library into their project.

3.1 Graphical User Interface

Figure 12 (a) depicts the RODEX application, a common Qt framework application using QRODS library that allows the access to iRODS collections and files. Users may add/delete a collection by providing the collection path. Related to files, it is possible to download, upload and delete files. For download and upload it is necessary to inform local and remote paths. Aiming to delete files only remote path field is required. Finally, button “Next Page" may be used when loading and visualizing collections at left side window. Files are loaded in low portions, obeying a fixed offset number.

3.2 Description of functionalities

RODEX has been developed in Java language, which provides platform independence. The graphical interface allows one to:

- Create and delete files or collections;
- Download and upload files;
- Add and delete metadata from an object; and
- List content (files, metadata, collections).

This tool is composed of three different views: Table, List and Tree views. Thus, users can adopt the most appropriate view according to their needs. In the following sections, we are going to present methods implemented in the MainWindow.cpp class that comunicates with iRODS data system. The main goal of these examples is to show that the adoption of QRODS library is quite similar to the codes that Qt developers are used to implement.

3.2.1 Create and delete files or collections

Users may add a collection by providing the collection path and clicking the Add Collection button as highlighted in Figure 17. Listing 11 shows the correspondent code adopted for performing such operation. Line 3 of the on_addCollectionBtn_clicked() method is responsible for holding the string that contains the name of the

collection to be created. Line 4 calls the addCollection() method (provided by the QRODS API) which conducts the whole process operation. It is important to stress that the *model* variable was previously defined as QRODS *model.

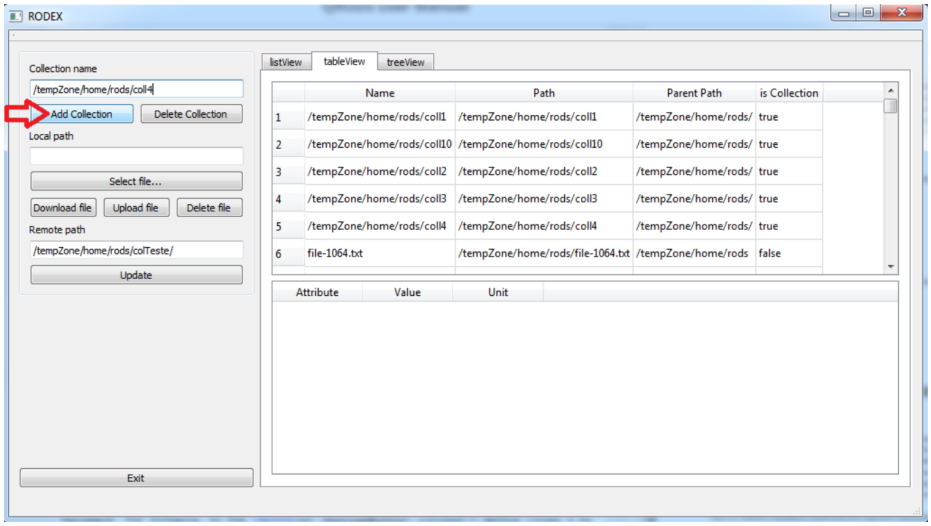

Figure 17: RODEX - adding a collection.

Listing 11: Adding a collection.

```
...
01 void MainWindow::on_addCollectionBtn_clicked()
02 {
03     QString str = ui->lineEdit->text();
04     model->addCollection(str);
05}
...
```

Users may remove a collection by selecting the collection to be removed from any of the three views available (list, table or tree views) or by providing the collection path name to be removed and clicking the Delete Collection button. Listing 12 presents the correspondent code for the delete collection functionality. Line 3 holds the collection name to be removed. Lines 4 to 7 removes the collection considering that a user has specified the collection name to be removed. Otherwise, users must select from any of the three views a collection to be removed. Lines 11 to 25 select the specific collection to be removed according to the view under use. Lines 26 to 30 conduct a check in order to know if the selected entry is a collection or not. Assuming that a collection was selected, a call to the removeCollection() method (which is provided by QRODS API) is performed to remove this collection.

Listing 12: Removing a collection.

```
...
01 void MainWindow::on_deleteCollectionBtn_clicked()
```

```
02 {
03     QString collectionName = ui->lineEdit->text();
04     if(!collectionName.isEmpty())
05     {
06         model->removeCollection(collectionName);
07     }
08     else
09     {
10         FileListingEntry* entry;
11         if(ui->tabWidget->currentIndex() == 0)
12         {
13             entry = static_cast<FileListingEntry *>
14                 (ui->listView->selectionModel()->selectedIndexes().at(0).internalPointer());
15         }
16         else if(ui->tabWidget->currentIndex() == 1)
17         {
18             entry = static_cast<FileListingEntry *>
19                 (ui->tableView->selectionModel()->selectedIndexes().at(0).internalPointer());
20         }
21         else
22         {
23             entry = static_cast<FileListingEntry *>
24                 (ui->treeView->selectionModel()->selectedIndexes().at(0).internalPointer());
25         }
26         if(entry->isCollection()){
27             QString remotePath;
28             remotePath.append(entry->pathOrName());
29             model->removeCollection(remotePath);
30         }
31     }
33 }
...
```

To delete a file, users may select the file from a view and click on the Delete File button. Listing 13 shows the correspondent code associated to the Delete File button. Lines 3 to 18 select the entry file to be deleted according to the correspondent view under use. Lines 19 to 26 are responsible for adjusting the path after removing the

desired file. Line 25 removes the selected file by removeDataObject() method which is provided by QRODS API.

Listing 13: Deleting a file.

```
...
01 void MainWindow::on_deleteBtn_clicked()
02 {
03     FileListingEntry* entry;
04     if(ui->tabWidget->currentIndex() == 0)
05     {
06         entry = static_cast<FileListingEntry *>
07           (ui->listView->selectionModel()->selectedIndexes().at(0).internalPointer());
08     }
09     else if(ui->tabWidget->currentIndex() == 1)
10     {
11         entry = static_cast<FileListingEntry *>
12           (ui->tableView->selectionModel()->selectedIndexes().at(0).internalPointer());
13     }
14     else
15     {
16         entry = static_cast<FileListingEntry *>
17           (ui->treeView->selectionModel()->selectedIndexes().at(0).internalPointer());
18     }
19     if(!entry->isCollection()){
20         QString remotePath;
21         remotePath.append(entry->parentPath())
22           .append("/")
23           .append(entry->pathOrName());
24         QString adjustedPath = remotePath.mid(1, remotePath.length() -1);
25         model->removeDataObject(adjustedPath,false);
26     }
27 }
...
```

3.2.2 Download and upload files

Listing 14 shows the code that is associated to the Download and Upload File buttons. For downloading a file, users need to select the file to be downloaded from the list, table or tree view as shown in Figure 12 (a). Besides

that, the user must select the local path to download the file from the iRODS data system. The local path is achieved in line 3, and the remote path is represented through line 22. Lines 5 to 19 defines from which view the user has selected the file to be downloaded. Afterwords, a call to the *getFile()* method is performed (line 25). Finally, a message box opens showing that the specified file has been downloaded in the defined path.

Lines 11 to 17 show the *on_uploadButton_clicked()* method that is responsible for uploading files to iRODS data system. For instance, to upload a file, a call to the *uploadFile()* method is performed (line 16). This method receives as parameters the local and remote paths. The local path is reached from the line 13 and the remote path from the line 14.

Listing 14: Download and Upload examples.

```
...
01 void MainWindow::on_downloadButton_clicked()
02 {
03     QString localPath = ui->localPathEdt->text();
04     FileListingEntry* entry;
05     if(ui->tabWidget->currentIndex() == 0)
06     {
07         entry = static_cast<FileListingEntry *>
08           (ui->listView->selectionModel()->selectedIndexes().at(0).internalPointer());
09     }
10     else if(ui->tabWidget->currentIndex() == 1)
11     {
12         entry = static_cast<FileListingEntry *>
13           (ui->tableView->selectionModel()->selectedIndexes().at(0).internalPointer());
14     }
15     else
16     {
17         entry = static_cast<FileListingEntry *>
18           (ui->treeView->selectionModel()->selectedIndexes().at(0).internalPointer());
19     }
20     if(!entry->isCollection()){
21         QString remotePath;
22         remotePath.append(entry->parentPath())
23                 .append("/")
24                 .append(entry->pathOrName());
25         model->getFile(remotePath, localPath);
```

```
26
27          QMessageBox messageBox;
28          messageBox.setText("File"+entry->pathOrName()+"\n downloaded to"+localPath);
29          messageBox.exec();
30      }
31}
...
32 void MainWindow::on_uploadButton_clicked()
33{
34      QString localPath = ui->localPathEdt->text();
35      QString remotePath = ui->remotePathEdt->text();
36      model->uploadFile(remotePath, localPath);
37}
...
```

3.2.3 Add and delete metadata from an object

To add or delete a metadata, users need to double click on a file or collection on the list view. The correspondent metadata dialog for that file or collection opens. This metadata window is depicted in Figure 18. Users are able to add a metadata by informing the metadata parameters (Attribute, Value and Unit) and pressing the Add metadata button shown in Figure 18. To remover a metadata, users select the metadata to be removed and press the Remove metadata button. The correspondent code for adding metadata is shown in Listing 15. The *model* variable was previously defined as QRODS* qrsm.

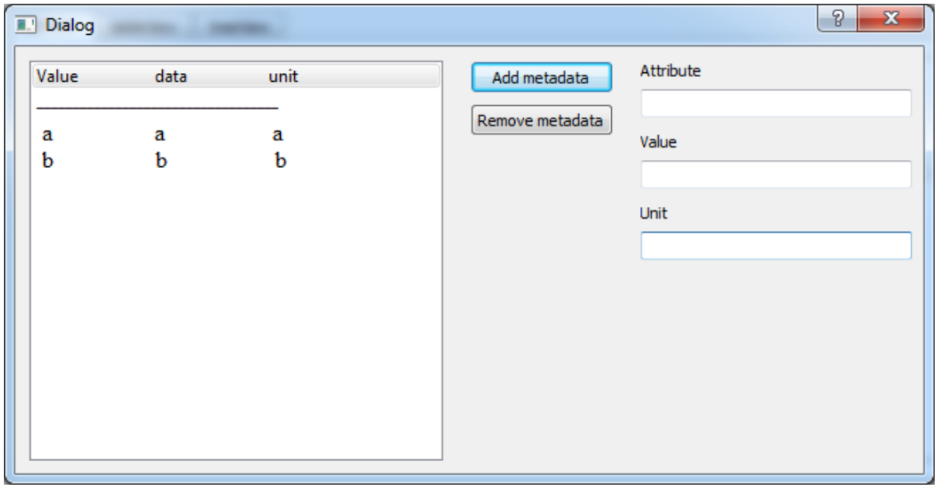

Figure 18: RODEX - metadata window.

Listing 15: Add metadata example.

```
...
```

```
01 void MetadataDialog::on_addMetadataBtn_clicked()
02 {
03     QString attribute = ui->attributeEdt->text();
04     QString value = ui->valueEdt->text();
05     QString unit = ui->unitEdt->text();
06     MetadataEntry entry(attribute, value, unit);
07     qrsm->addCollectionMetadata(path, entry);
08 }
...
```

3.2.4 List content (files, metadata, collections)

Once RODEX application is executed, a window shows iRDOS data through three different views. Listing 16 code is responsible for performing such operations that presents iRODS data as list, tree and table views. Lines 5 to 19 set the parameters needed (e.g., user, host, port and password). After that, Lines 20 to 22 associate the QRODS model to each view. Line 24 updates the QRODS model by associating iRODS data according to the path passed as parameter. Therefore, according to the view that is selected, the iRODS data can be shown.

Listing 16: Main Window - listing content.

```
01 MainWindow::MainWindow(QWidget *parent) :
02     QMainWindow(parent),
03     ui(new Ui::MainWindow)
04 {
05     QString user(getUser());
06     model = new QRODS(this);
07     model->setUser(user);
08     QString * password = new QString("rods");
09     if (password == NULL){
10         qDebug() << "Candidate password does not match!";
11         delete ui;
12     }
13     else{
14         qDebug() << "Password object is OK!"  ;
15         ui->setupUi(this);
16         QString host(getHost());
17         model->setHost(host);
18         model->setPort(getPort().toInt());
```

```
19          model->setPassword(*password);
20          ui->tableView->setModel(model);
21          ui->listView->setModel(model);
22          ui->treeView->setModel(model);
23          QString path("/tempZone/home/rods");
24          model->connectToServer(path, secureHttp());
25          ui->metaDataView->setModel(model->getDataModel());
26          model->addView(ui->treeView);
27          connect(model, SIGNAL(layoutChanged()), this, SLOT(layoutChanged()));
28      }
29  }
```

A Configuring Apache Tomcat for HTTPS

By default, irods-rest users perform clear HTTP connections to exchange data with the Jargon-REST server. That might expose username, password and other private fields. Mainly because it will send data by non-protected HTTP connections. In this tutorial we will create a server certificate, add it to a keystore, configure the tomcat server, and change the irods-rest app for HTTPS.

A.1 Pre-requisites

Before starting, make sure a JAVA Virtual Machine (JVM) is installed as well as the Apache Tomcat.

```
$ java -version
```

If JAVA is not installed, you can download and install it from Oracle:

```
$echo "deb http://ppa.launchpad.net/webupd8team/java/ubuntu precise main" | tee -a /etc/apt/sources.list
$ echo "deb-src http://ppa.launchpad.net/webupd8team/java/ubuntu precise main" | tee -a /etc/apt/-
sources.list
$ apt-key adv –keyserver keyserver.ubuntu.com –recv-keys EEA14886
$ apt-get update
$ apt-get install oracle-java8-installer
```

After that, download Apache Tomcat binaries [1]. **Apache Tomcat version 8.0.15** or above is recommended. For an easy installation, it is recommend the core version provided by a .zip file.

A.2 Creating the keystore

Go to the bin folder inside your JAVA home. If there is not an environment variable called JAVA_HOME on your machine, you need to find manually or check at the Oracle Java site. On a Linux installation, this would be located at: **/usr/lib/jvm/java-8-oracle/**

```
$ cd $JAVA_HOME/bin
```

or `$ cd /usr/lib/jvm/java-8-oracle/bin`

Inside this folder you may see a file name `keytool`. It is exactly a tool to generate keystores. Let's generate a keystore. By default, it will be saved on your home folder (/.keystore).

Pay attention! When asked about your first and last name you should provide the machine address (e.g. localhost, 210.75.14.146, mydomain.com). Moreover, don't forget the provided password!

```
$ keytool -genkey -alias tomcat -keyalg RSA
```

A.3 Configuring Apache Tomcat for HTTPS

Go to the conf folder inside Apache Tomcat installation. Following, edit the server configuration file server.xml:

Find the following tag (may be a little different):

```
<Connector port="8443" protocol="HTTP/1.1" SSLEnabled="true" maxThreads="150"
scheme="https" secure="true" clientAuth="false" sslProtocol="TLS" />
```

Change it to:

```
<Connector port="8443" SSLEnabled="true" acceptCount="100"
clientAuth="false" disableUploadTimeout="true" enableLookups="false" maxThreads="25"
keystoreFile="/home/yourUserName/.keystore" keystorePass="myPassword"
protocol="org.apache.coyote.http11.Http11NioProtocol" scheme="https"
secure="true" sslProtocol="TLS" />
```

Pay attention to keystoreFile and keystorePass tags!

A.4 Configuring the irods-rest web app

Now you will need the .war file or an exploded directory containing the irods-rest app files.

The .war file must be renamed to "irods-rest.war". If you have an exploded folder, you must rename it to "irods-rest". Both of them must be put inside the *webapps* tomcat folder.

Following, find and open the **web.xml** file inside the WEB-INF directory (extract the .war file if you have the .war version):

Add the following text right before the *</web-app>* tag :

```
<security-constraint>
<web-resource-collection>
<web-resource-name>securedapp</web-resource-name>
<url-pattern>/*</url-pattern>
</web-resource-collection>
<user-data-constraint>
<transport-guarantee>CONFIDENTIAL</transport-guarantee>
</user-data-constraint>
</security-constraint>
```

Now you are able to run the Apache Tomcat and test the irods-rest app.

If the browser requests your credentials, the irods-rest app is properly set for HTTPS. You may notice that the server certificate must be accepted manually. Generally, the browser asks you about the certificate.

References

[1] Apache tomcat web page. `http://tomcat.apache.org`, 2015.

[2] iRODS. The integrated rule-oriented data system (irods). http://irods.org/, 2014. Acessed: 04/25/2015.

[3] iRODS. irods executive overview. http://irods.org/wp-content/uploads/2014/09/iRODS-Executive-Overview-August-2014.pdf, 2014. Acessed: 04/25/2015.