RetroShare: Writing Plugins

RetroShare Team

January 17, 2014

1 Introduction

So you have been using RetroShare for a while, and as a computer programmer, you want to add your favourite plugin to share with others.

Where to start?

Well, that is the purpose of this guide. It will provide a guide for developing it and getting it working with RetroShare with the least amount of effort possible.

This was written as a plugin was being developed, so the screen shots displayed are for that plugin.

2 Concepts

There are 3 types of plugins:

1. services based on friend communication eg VoIP
2. services based on turtle router eg ZeroReserve
3. services based on cache exchange eg LinksCloud

There are 3 basic parts to each plugin:

1. Serialiser - messages that you want to send over the network. You just need in the constructor of your class to declare each member and what type it needs to be serialized with.
2. libretroshare service - send/recv msgs and prepare data for GUI.
3. rs-gui window - get info from service, and display it!

All plugins working in the 0.5.* version and releases, will continue to work with 0.6 which changes to a new system using GXS.

3 Important Things

3.1 Language

RetroShare is written in c++ with some parts in c. You need to have a good understanding of c++. It will be possible to add other features from other languages that can be called from c/c++ and return values into the calling program eg Haskell.

3.2 Tools and Libraries

3.2.1 QtCreator

QtCreator is a free tool that can be used to develop the project, as well as editing, and compiling to create a library as either a *.so for Gnu/Linux or *.dll for Windows or *.so for Mac.

3.2.2 Editor

Many programmers use vim (from `http://www.vim.org/`) with some addons to make editing easier and faster, especially on older, slower computers. You can get a complete collection of plugins and settings for vim from `https://github.com/morpheusbeing/vim`.

Emacs is another editor that can be used.

3.2.3 Libraries

Unless your plugin is very complicated, all the required libraries should be available on your computer if you can compile a working version of RetroShare.

3.2.4 Version Control

RetroShare uses svn for its version control. If you are compiling your own, you will understand the use of the *svn up* command on Gnu/Linux.

For a project like this, git and `https://github.com` is a good option to manage a local and remote repository.

You will need to create an account if you don't have one, and create a repository to store your code in. This allows you to keep an offsite copy of your work, as well as allowing for collaboration. There are many useful tutorials on using git, but essentially it is as follows:

1. Login or create an account at `https://github.com`.
2. Create a new repository (best to use the same name with same character case as your plugin name).
3. cd into the plugin folder you are working with
4. create a README.md file and edit the contents of it.
5. Initialise your local repository:

   ```
   git init
   ```

6. Add the file you jsut created to your repository:

   `git add README.md`

7. Commit the changes made into your repository:

   ```
   git commit -m first commit
   ```

8. Map your repository to the remote repository using the command:

   ```
   git remote add origin https://github.com/[your user name]/[your plugin].git
   ```

9. Now add any other files you have created. To add a folder, just use the folder name, and will track all the files.

10. Now copy your repository to the remote :

    ```
    git push -u origin master
    ```

11. Now check the status of local git:

    `git status`

For more information about using git, check the internet for tutorials and information.

3.3 Structure

It is possible to put all the files in one folder of the plugin name so that it is ~/retroshare/plugins[your plugin] as shown in 1

As a suggestion, establish some folders in you plugin to store various parts of your project into (refer to 2)

You will notice a folder .git - this is for versioning control - refer to 3.2.4.

3.3.1 gui folder

3.3.2 interface folder

3.3.3 services folder

3.3.4 lang folder

This folder contains all the files required for translating the strings used in the plugin to the appropriate user language.

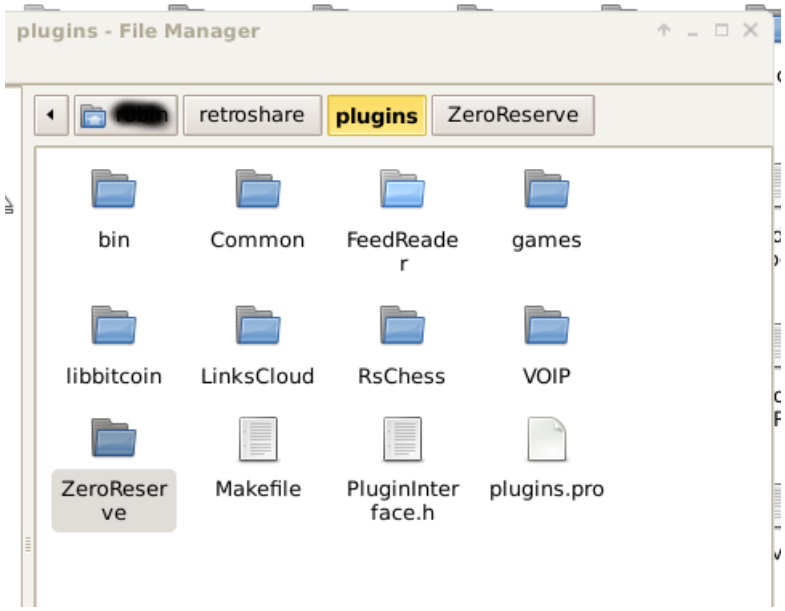

Figure 1: Folder Structure For Plugin

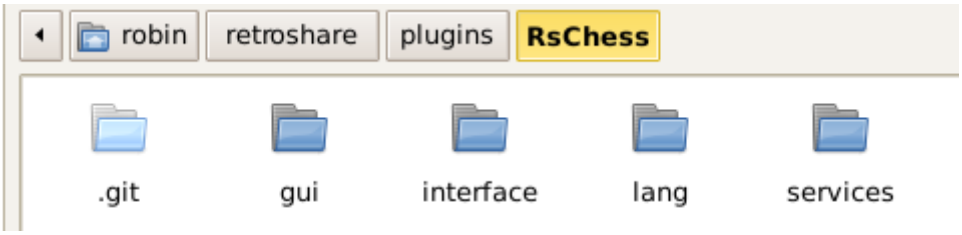

Figure 2: Sub-folders for the plugin (suggested)

3.4 Coding Style

4 Putting It Together

4.1 Serialiser

5 Example

This is a simple example that just displays a popup message.

6 Documentation Processes

RetroShare uses doxygen to comment all of the code.

There are several different styles of doxygen coding that can be used.

The @ and \ are interchangeable.

6.1 Blocks

Blocks are the basic units of documentation for Doxygen. At first it will feel like overkill to use blocks, but realize that Doxygen was initially designed to operate on header and source files, and then the blocks of documentation would be before the definition or declaration of the methods, functions, etcetera. Doxygen is used to operating on blocks, and that's why we need to reproduce them in our dox files.

Blocks should adhere to the following standards:

- All blocks open with

  /*!

- and close with

  */

- The documentation is placed in between these markers.
- All the contents in between the markers is indented by tabs.
- The tab length should be four.
- Between blocks, there should be two empty lines.
- The maximum width of the contents between blocks is 80 columns. Try not to cross this limit, because it will severely limit readability.

Example

```
/*!
        \brief Append an item to the list.
    \detail More detailed description.
    \param item The item to add.
    \retval true The item was appended.
    \retval false Item was not appended, since resizing the list failed.
\sa AddItem(void *item, int32 index)
*/
```

6.2 File Header

```
/*!
  \file thing.ext
  Description of the file
```

```
  \author Jake Smith
  \version 1.0
  \date 1999
  \remarks starts a remarks paragraph, leave empty line after.
  \bug A known bug

*/
```

6.3 Function

```
/*!
   \brief Brief (one-line) description of function

   Extended description (may extend over several lines).

   @code
   NSString *example = @"example string!";
   @endcode

   We can also use lists.

     - item 1
     - item 2

   @param first Description of first param
   @param second Description of second param
   @return Description of returned value
   @retval describes the return value of the function
   @exception e Description of e
   @throw Same as exception
   @attention Starts an attention paragraph

   @warning Bad things can happen

   @todo Something to be done

   @note Using on your birthday will provide a bonus

   @see [name-list]
*/
```

6.4 Class

```
/*!
  \class

*/
```

Index