iMagine

by Teemu Kinnunen, Jukka Lankinen and Sanaz Ahmadi

Introduction

Pictures are so much valuable to its owners since they all meant to be life time and supposed to remain the memory of great moments. These days, technology has advanced and people are more technology oriented than the last decades. Picturing different moments of life has become an interesting task. People own smart phones and other devices for capturing pictures. Therefore, managing, and organizing huge amounts of images has the first priority. Particularly, people from every corner of this world are trying to organize and manage their images and find as efficient storage as possible in order to adequately store their images. Therefore, with iMagine, we wanted to help and offer people the effective and efficient way of organizing their image collections. Images are first synced across multiple devices using a server. Furthermore, the server also categorizes images based on visual information.

Purpose and Idea

The purpose of iMagine is to offer a service to people in order to manage, organize, and categorized their images regardless of the number of images and the size. iMagine team has noticed and realized that, people own large numbers of images in their devices (Mobile phones, PCs, Laptops, External hard drives, and etc). In most cases, people are willing to easily transfer their images from their laptops to their mobile phones and vice versa. Besides, they are willing to have as safe storage as possible. Therefore, image synchronization between different devices is illustrated in Fig. 1. The idea is that people are able to access to their image collection from any device that has iMagine. Therefore, iMagine, with its large capacity of storage, high demand, and quality of services, is offering this service to people. People are then able to organize, manage, and categorized their images by iMagine.

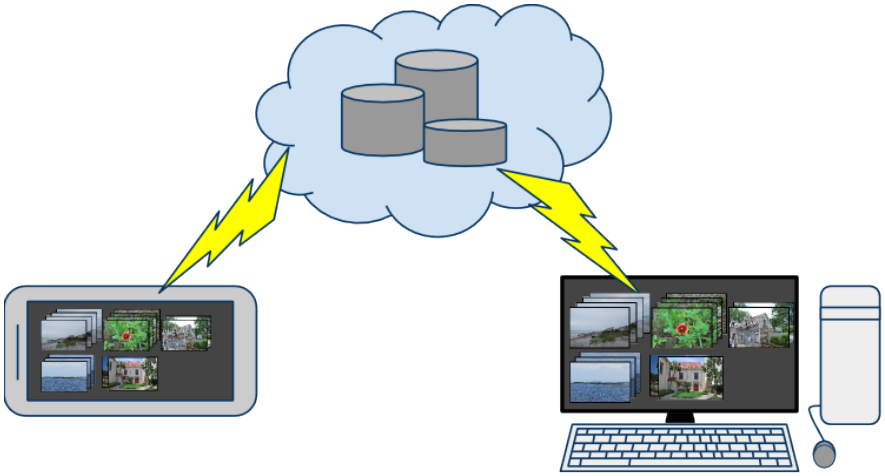

Figure 1: Image synchronization across different devices

Implementation

Implementation had two parts: the communication part between the server and the client and another which presented the data provided by the server in visual manner. A functional application requires Internet connection to sync between the server and clients (such as phones and laptops). The server does all the computing needed for finding similar images for easier browsing. To group visually similar images together we used the Self-organizing map and bag-of-features model which was presented in a paper by Kinnunen et al. [1].

Client

The client resizes (downsamples) the images, sends them to the server and shows groups of similar images. In the client, we had three different views: Login screen (SettingsView), Main view (StackView) and image view (ImageView). Instead of using Qt Creator to design these views, we implemented user interfaces by programming them. We chose this approach since the user interface is very simple, but dynamic. Thus, it was more convenient to implement UI by programming. In Fig. 5, we show the class diagram of iMagine client application. In Qt, events are replaced with signals and slots. Thus every user interaction and network event causes a signal that can be used to call functions.

In the Log In view (*SettingsView*), we used grid layout to divide screen into 4 x 2 grid. In the Log in view, we ask for a username, password and directory where images are stored at. When user click 'Log in' button, all the given information is stored in the user data widget which stores all the information on the disk (*ImageView::saveSettings()*). Therefore, the information is always available after it is given once. It is very convenient for the user since he/she does not need to give information more than once. After user information is stored, the application emits a new *submitted()* signal to the *MainWindow* which tells that user information is given and user would like to log in to the server. Screenshot of the Log In view is shown in Fig. 2.

Figure 2: Log In view (SettingsView)

MainWindow catches the *submitted*()-signal and processes the user information and tries to log user in. After user is successfully logged in the system. The application synchronizes images from the server and to the server. The image synchronization process and communication protocol is illustrated in Fig. 6. While the client and server are synchronizing images, we show a progress bar, since it can take a while to download and upload many images. After the images, image lists and image stack lists are synchronized, the application loads *MainView* widget for the Main View. In the Main View, we use a *QScrollArea* widget that covers the whole screen. Inside the *QScrollArea* widget, we have a grid layout widget. Inside

each cell, we have an image stack widget that we made. Image stack information is acquired from the server. The server gives a stack file where each stack is on a separate row. So it is easy to build an image stacks for the main view and image from the stack file. Image stack widget is inherited from *QWidget* and it uses *QPixmaps* to show a list of stacked images. First we paint the bottom image and then we paint an image that is on top of the first image and then the last image on the top of two first images. We show three or less images, since otherwise it becomes very confusing we are aiming at simple UI. When user clicks one of the image stack widgets, the application will show the images of the stack in the Image view. In practice, a mouse click from the user emits a *stackClicked*(*stackid*) signal which tells that one of the stacks have been clicked. It also gives id for the specific stack so the system can display the images of the correct stack in the Image View. The *MainView* captures *stackClicked*(*stackid*) signal and changes widget to the *ImageView* widget. Screenshot of the Main View is shown in Fig. 3.

Figure 3: A screenshot of MainView showing image stacks.

In the Image View, we show the first image of the stack at first and then the user can browse the rest of the images in the stack using left and right buttons in the view. Images are ordered in the stack. The top most image in the stack is the most representative image (the image with less error). In the Image view, we use grid layout to divide the view into 2 x 3. On the left, we have a *QPushButton* to display images on the top of the stack and on the right we have *QPushButton* to show images on the bottom of the stack. In practice, these two buttons emit signals which call the functions to decrease (if possible) or increase (if possible) the value of the current image stack index. Two columns in the middle of the grid are reserved for the current image to use most of the screen area to display the image. To display the current image, we used *QLabel* and *QPixmap*. In the bottom left corner, we have a button that emits a exited() signal to the *MainWindow* which changes current view widget back to the *MainView* widget. A screenshot of *ImageView* is shown in Fig. 4.

Figure 4: A screenshot of ImageView.

In Fig. 5, we show the central classes of our iMagine implementation. It also shows some of the most importan Qt-classes that we have been using.

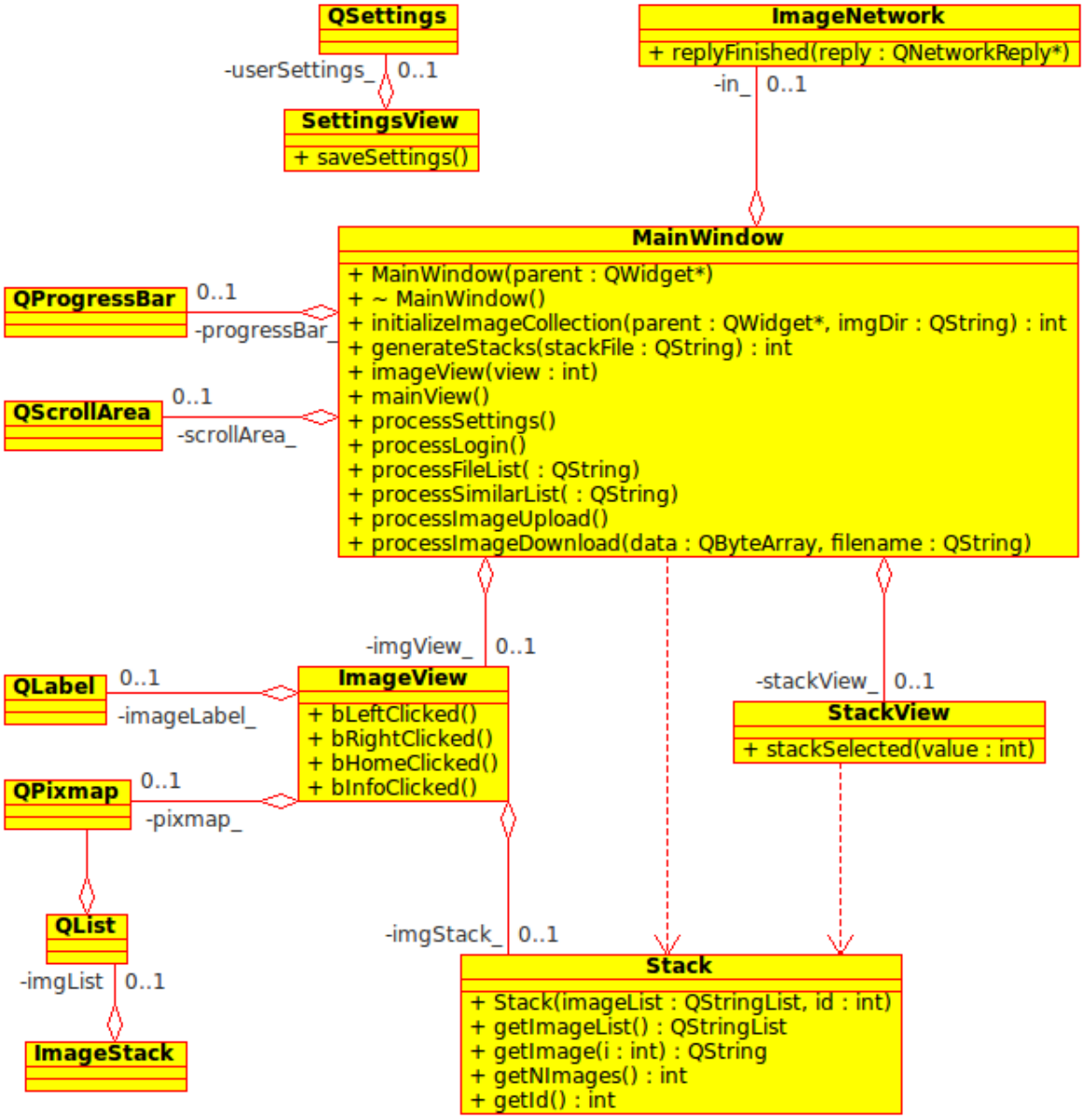

Figure 5: Class diagram for the client

Client-Server Protocol

Fig. 6 describes communication protocol between the client (a smart phone or a computer) and the server.

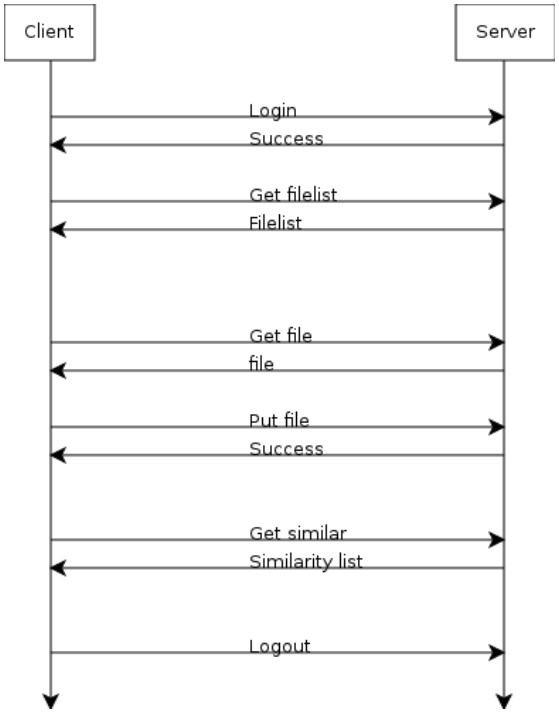

Figure 6: A simple example of how the protocol behaves.

An example of the protocol can be seen in Figure 6. Here a presumable client connects to Server with Login command with a username and a password hashed with MD5. All the commands sent to the server are done with HTTP POST messages. These messages are then replied with either data (such as lists or images) or a simple success/fail message.

To sync, Client sends a GETFILELIST message which is then replied a filelist. These files are then fetched from the server with GETFILE one by another. The server responds with either data or with an error message. In some cases user has added new images on the client side so these images are then sent to the server with PUTFILE command.

The galleries are then constructed by using an image list file provided by the server with GETSIMILARLIST command. This file has a N number of filenames on each line corresponding to a certain image stack in the Client.

References

[1] Teemu Kinnunen, Joni-Kristian Kämäräinen, Lasse Lensu, Heikki Kälviäinen, *International Conference on Pattern Recognition*, Unsupervised visual object categorisation via self-organisation, (2010).