Laboratório VISGRAF
Instituto de Matemática Pura e Aplicada

Architectures for Distributed Mobile Applications

Gabriel Fernandes, Djalma Lucio, Bruno Silva, Luiz Velho

Technical Report TR-15-01 Relatório Técnico

January - 2015 - Janeiro

Architectures for Distributed Mobile Applications

Gabriel Fernandes, Djalma Lucio, Bruno Silva, Luiz Velho
VISGRAF Laboratory - IMPA

1 - Overview

This technical report discusses architectures for distributed mobile applications. One of the goals of the present study is to provide an understanding of the cloud and mobile phenomenon in modern day computing and collaborative data capture and analysis. For this purpose we will present a model application involving mobile photography. The structure of this application will be described and three different versions using alternative architectures will be developed and compared.

1.1 Application

The proposed mobile photo capture/share system helps understand the many challenges in cross platform app development / deployment and collaborative data upload/download. The app, originally called RPic, captures a photo on any device (mobile and desktop) and uploads it with any metadata available to a picture pool on a web server. Once the picture is received and stored, all the apps can view the photo and, depending on privileges, can edit part of the metadata. The main features are: take pictures, set/edit title, view and delete pictures. After a pre determined quantity of pictures have been stored, the user will be able to view and analyze them in a collaborative environment.

The platforms used for development and testing are Mac (OSX), Windows, Linux (Ubuntu/ Kubuntu), iOS (7 and 8) and Android. The server was based on Rest principles, as will be explained further on, each version/platform has a unique way of communicating with the server, although the end result is the same.

1.2 Data Model

A simple data model is proposed so it can be easily reproduced and manipulated on multiple platforms. The are two elements: Picture and User. There is only one relationship between them: each picture is owned by one user. The user element contains a unique id, username, password and pictures. The user *pictures* property relates to all pictures taken by user who uploaded them. For a user to upload content he must be registered and type his username and password once during the applications first run.

The picture element also contains a unique id, this id is defined from the total number of pictures uploaded by all users and can be used to locate a specific picture in picture pool. Other properties stored in a picture element are: user, title, date and coordinates (latitude and longitude). The title is optional and inserted by the user after the picture is taken, it can also be edit in the view mode after the picture is uploaded. The coordinates are sent if available. Availability depends on localization features activated and authorized by user on the current device, some devices might not even have localization options (such as laptops and old mobiles). If no coordinates are available the system stores the default coordinates (the IMPA location).

1.3 Architecture

The systems is straightforward, on the client: a registered user takes a picture, gives it a title and uploads to the server. The upload starts once the user confirms the title, and if, at title type menu the discard button is pressed, the app deletes the picture and returns to camera mode. On the server the picture is received, stored and the appropriate entries are made on a SQL database. Afterwards the user can view a list of past pictures and edit titles or delete them. For easy data mapping the database can be queried by REST (Representational State Transfer) abstraction principles. The *RESTful* query system was built over the Django framework, which will be explained further.

Each client queries web addresses composed of the data storage paths and receives *json* formatted objects, which after received by the client are mapped into platform specific objects and then displayed. The client also maps its own objects to a *json* object which can be sent back to user readable web addresses on the server, resulting in actions such as create, replace, edit or delete data. The expected end result of this system is a user built photo database which can later be viewed and analyzed in collaborative fashion by all the users. Bellow are examples of the address structure and mapping.


```
[
    {
        "url": "http://phong.visgraf.impa.br:8080/rpic/pictures/54/picture/",
        "id": 54,
        "user": "visgraf",
        "uuid": "F0ECEB2D-3C7E-449C-8230-0A82B4018C0E",
        "title": "Untitledppp",
        "created": "2014-11-26T18:54:58.411Z",
        "last_modified": "2014-11-26T18:54:58.632Z",
        "latitude": "22.9660072327",
        "latitude_ref": "S",
        "longitude": "43.2375679016",
        "longitude_ref": "W"
    },
    {
        "url": "http://phong.visgraf.impa.br:8080/rpic/pictures/53/picture/",
        "id": 53,
        "user": "visgraf",
        "uuid": "{8a6d2219-002c-4ebf-8f64-6a9bccd65a26}",
        "title": "Cool Face",
        "created": "2014-11-26T15:58:05.650Z",
        "last_modified": "2014-11-26T15:58:05.768Z",
        "latitude": "22.9660716667",
```


Picture pool address: http://serveraddress/rpic/pictures
This will return the following a *json* list of pictures and metadata.


```
{
    "url": "http://phong.visgraf.impa.br:8080/rpic/pictures/54/picture/",
    "id": 54,
    "user": "visgraf",
    "picture_file": "user/1/visgraf@impa.br_F0ECEB2D-3C7E-449C-8230-0A82B4018C0E.jpg"
}
```


Picture 54 address: http://serveraddress/rpic/pictures/54/picture/
This returns a json object with the address of picture file for download.

There was debate over wether to use or not the picture metadata header (EXIF and others) to store data. Although practical, the standard for writing and reading headers differs significantly

between platforms and presents a challenge all by itself. Although there are C libraries (such as *libexif* http://libexif.sourceforge.net/) which can be used, adapting the library to each platform lies outside the scope of this work. The final solution was to leave the metadata outside the image as database entries.

2 - Technologies

Key technologies for distributed mobile application development are related to: distributed standards for networked client / server systems; mobile platforms and programming languages. These technologies have experienced great improvement and innovation in recent years. Furthermore, the interplay between many of their aspects at all levels is non-trivial and have a significant influence in the design of new architectures.

2.1 Distributed Standards

REST (Representational State Transfer) is way of building abstract paths for accessing data. The paths are usually human readable and if implemented fully should allow users to navigate and manipulate the database almost in a intuitive way. Although there are platforms that help develop RESTfull servers, most off the work still lays on the developers for the process of customizing the address system, linking it to the databases and managing requests based on user privileges. This access layer is a interface for accessing databases, although not limited to, SQL.

CloudKit is a new developer framework introduced by Apple in iOS 8 to leverage the full power of iCloud resources and facilitate the development of client-server applications. With CloudKit, developers can focus on the client-side app, letting the framework taking care of the server-side logic in iCloud. Thus, it eliminates the need to write the server application. The framework provides an infrastructure for easily an securely store and efficiently retrieve the app structured itens in a database or general assets in iCloud. CloudKit provides authentication, private and public databases as well as asset storage services. It also enable users to anonymously sign in to the app with their iCloud Apple IDs, without sharing their personal information.

2.2 Mobile Platforms

iOS is Apples mobile operating system and is used on almost any Apple device which is not OSX. It resembles in development and design of the desktop platform, the main difference is touchscreen based system. Apple has made a significant effort in closing the gap between this mobile system and its desktop counterpart. Applications made for iOS must pass through XCode, which is Apples Software Development IDE.

Android is the the open source OS of Open Handset Alliance. Its based on Linux and is extremely customizable. Similar to iOS its centered on touchscreen systems, but can be used with mouse/keyboard if available. Although based on Linux it does not bring direct resemblance to any particular linux distribution. Applications for Android can be built in any system and using languages such as Java, C, C++. There are many different pipelines for developing apps in a series of different IDEs. The most popular IDE, in the past, being Eclipse and currently being replaced with Android Studio.

QT is not a OS, but a C++ cross platform library which can be used to make native apps for all platforms. It carries a overhead of the QT library with every application, which although large has a very small memory imprint during runtime. For each deployment platform it has a unique build kit which compiles the C++ project with optimal configuration. A application built with QT using its popular IDE Qt Creator, can be safely executed on most desktop systems and with less fidelity

in most mobile systems. It has a set of licenses (GPL, LPGL), which make it very popular in open and closed source projects or of commercial/non commercial nature.

2.3 Programming Languages

C++ is one of the most popular programming languages available, it is also considered to have good performance when used natively on mobile platforms.

Objective-C is Apple's main development language, and can work with C++. It adds many custom tailored features/classes especially made to work with Apple devices bringing optimal performance and fast development. Although very similar to C++ it is not directly portable to other platforms outside the Apple universe. When developing using Apple Xcode, there is also a interface creation tool called *storyboard* which helps design layouts and code the app together.

Swift is a multi-paradigm modern interpretation of the Objective-C language influenced by different concepts and languages. Its main focus is keeping the code safer from errors and smaller, it resembles distantly some scripting languages such as Javascript, Qml, Ruby, C# and others. Although Objective-C maintains many similarities with C++, the Swift language steps away making harder to reuse the code on other platforms.

Javascript is scripting language popularly used to create interactive and dynamic web pages. The current Javascript V8 engine used in Chrome and Qt/Qml compiles the script to native machine code before executing. Javascript plays an essential role in Qt/QML interface development and can be mixed with C++ calls for higher performance when needed.

Python is a general purpose high-level programming language. Python supports multiple programming paradigms, including object-oriented, imperative and functional programming or procedural styles. It features a dynamic type system and automatic memory management and has a large and comprehensive standard library.

3 - Client App

The client app revolves around a set features common in all versions, although sometimes expressed in different layouts due to variations in implementation. The features available to the user are:

- Input login and password, later verify if user is valid.
- View a list of pictures on the web server.
  - Access one picture, view it, and if necessary edit the title.
  - Delete a picture.
- Camera mode:
  - Viewfinder / Preview Picture
  - Take Picture
  - Set Title
  - Save/Discard.

Other features behind the scenes are:

- Send login data to server as a http request.
- Send http GET request for picture list.
- Send http PUT/POST request for picture metadata.
- Send http PUT/POST request for binary data (the picture itself).
- Send http PUT/POST request for updating title.
- Send http DELETE request for removing picture from server.
- Order received data by date.

All versions implement these features with or without the help of additional libraries or frameworks.

3.1 QPict

The QPict was developed using the Qt library and QtCreator (5.3.2). It has code written in C++, QML and Javascript. The application interface was written in QML, all of the graphical elements derive from the basic QML components, such as Rectangles, Text, Image and List. These elements where customized and arranged in a scene-graph which represents the app UI. Also the auto-adjust-layout behaviour is crafted inside the components through dynamic property bindings and anchor system (which resembles Apple's constraints).

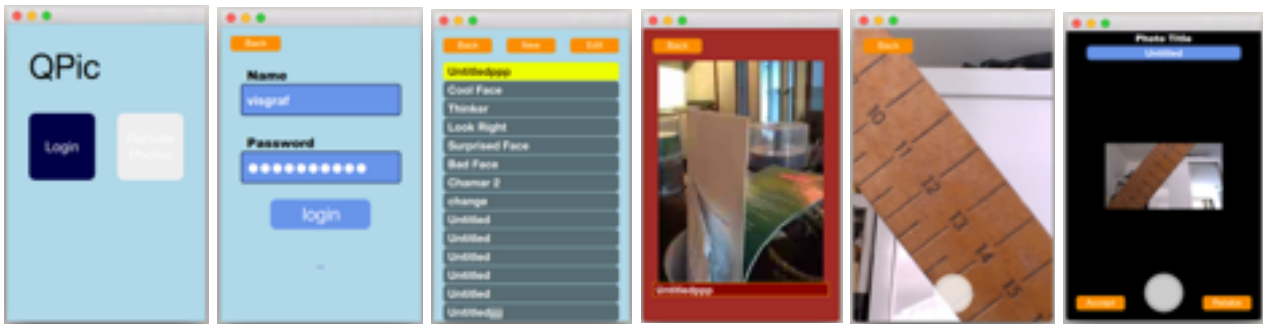
Since there was no need for high performance in any aspect off the app, most functionality was placed inside the QML code in the form of javascript. This javascript code is called in a event system triggered by user input, such as touching a button on the screen.

Javascript has a mature and stable *json* parsing object which served well for mapping *json* objects to javascript objects. The mapped objects where further on placed inside QML components to be displayed as lists. Javascript also has a http request object called *xmlhttprequest* which worked well for requests without authentication (partly supported) and binary data (no support). Later on it was necessary to replace this request system with the *QHttpMultiPart* and *QNetworkRequest* in C++ (from the Qt library), so it would be possible to send binary data. The first attempt to mix javascript requests and Qt requests was complex and hard to debug, so in the last version all requests where made using the C++ objects. The *json* still where sent to and from C ++ as *strings and* parsed through javascript, as this was more convenient. Sending binary files was the most challenging process in Qt and bellow is the method and configuration used for this feature retrieved from (*rpicqtengine.cpp*):

```
void RPicQTEngine::sendBinaryFile(int pictureId, QString filepath, QString uuid)
{
    qDebug() << "C++ - sendBinaryFile received: id:" << pictureId << "path:"+filepath << "uuid:"+uuid;
    connect(jsonManager, SIGNAL(finished(QNetworkReply*)), this, SLOT(replyBinaryFinished(QNetworkReply*)));

    QHttpMultiPart *multiPart = new QHttpMultiPart(QHttpMultiPart::FormDataType);
    QHttpPart imagePart;

    QString outFilename = getUsername() + "_" + uuid + ".jpg";
    qDebug() << "C++ - sendBinaryFile outFilename: "+outFilename;

    imagePart.setHeader(QNetworkRequest::ContentTypeHeader, QVariant("image/jpeg"));
    imagePart.setHeader(QNetworkRequest::ContentDispositionHeader, QVariant("file; filename=\""+outFilename+"\"; name=\"picture_file\""));

    QFile *file = new QFile(filepath);
    qDebug() << "Exists: " << file->exists();

    if (file->open(QIODevice::ReadOnly)) {
        qDebug() << "C++ - sendBinaryFile OPEN OK";
        imagePart.setBodyDevice(file);
        file->setParent(multiPart); // we cannot delete the file now, so delete it with the multiPart
        multiPart->append(imagePart);

        QString sendPath = "http://phong.visgraf.impa.br:8888/rpic/pictures/"+QString::number(pictureId)+"/picture/";
        qDebug() << "C++ - sendBinaryFile sending to:" << sendPath;
        QUrl url(sendPath);
        QNetworkRequest request(url);
        QString concatenated = getUsername()+":"+getPassword();
        qDebug() << "C++ - login:password => "+concatenated;
        QByteArray data = concatenated.toLocal8Bit().toBase64();
        QString headerData = "Basic " + data;
        request.setRawHeader("Authorization", headerData.toLocal8Bit());

        qDebug() << "C++ - Put Image!";
        jsonManager->put(request,multiPart);
    } else {
        qDebug() << "C++ - sendBinaryFile OPEN ERROR";
    }
}
```

QML has many cross-platform components for accessing camera and location, so the metadata was easy to acquire on all platforms. The camera component is called *Camera* and the location is called *PositionSource.* They are easy to write, configure and customize and have extensive documentation and examples.

Although being easy to create, this version has the lowest level programming in comparison with the others. The overall program layout can be managed through a QML tree view (QtCreator), which shows all components organized, and allows component re-parenting by drag and drop interaction. It has good flexibility and cross-platform functionality, but must altered with caution as many options have complex configurations which are not obvious or intuitive. Bellow are screenshots of the QPict App:



Main Menu	Login/Admin	Album View	Photo Detail View	Camera	Save Picture

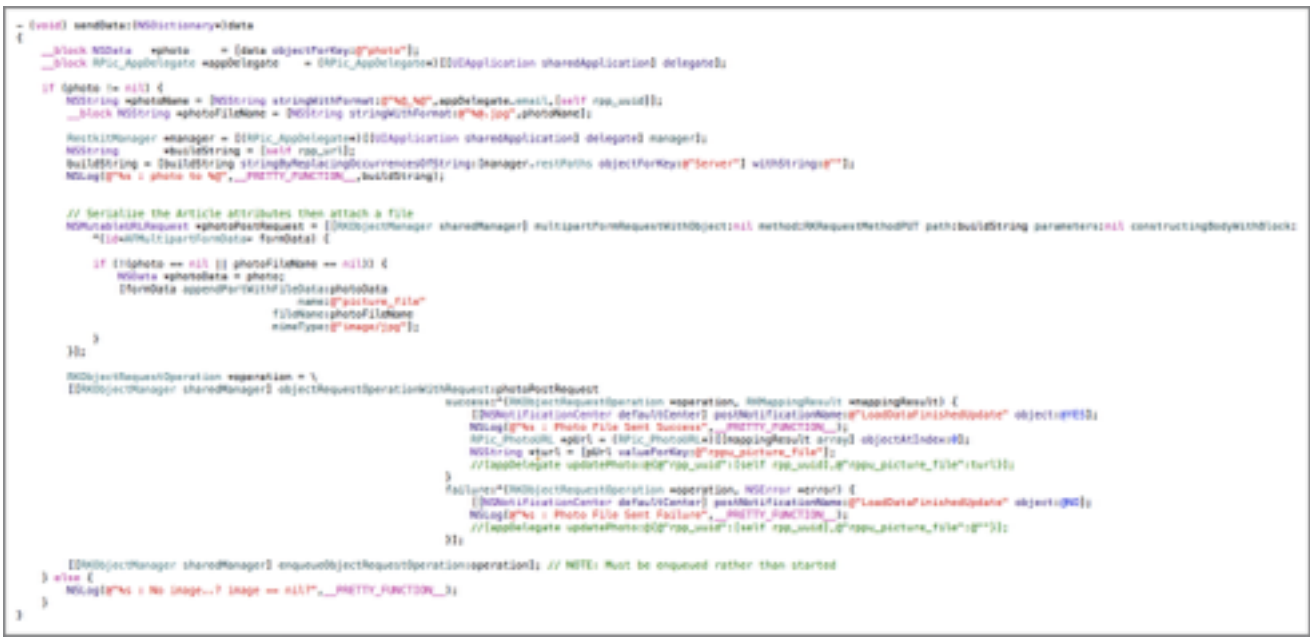
3.2 RPict

The Rpict is the Objective-C version of the client created in Xcode for the iOS platform. Its code is based on the Model-View-Controller paradigm. Apps developed in Xcode for Apple platforms can benefit from a series of tools to help organize and plan development. One of these tools which mark the beginning of the app design is the Storyboard. Storyboarding apps is a well

known concept, but Xcode has well integrated tool for designing storyboards and integrating them with code. The storyboard contains the Views which will be displayed and the relationship between them in graph. This planning shapes the user interface and certain behaviours before a single line of code is written. Also, all adapting layout (or constraints) can be configured through this layout without any direct coding. Below is a zoomed out view of the RPict Storyboard:

Each view has a controller class which contains the functionality of UI elements. On the top branch there is a path which leads to the login and authentication UI. The lower path leads to the server picture list and through buttons such as New or Edit, it is possible to navigate to the Camera View and Picture View.

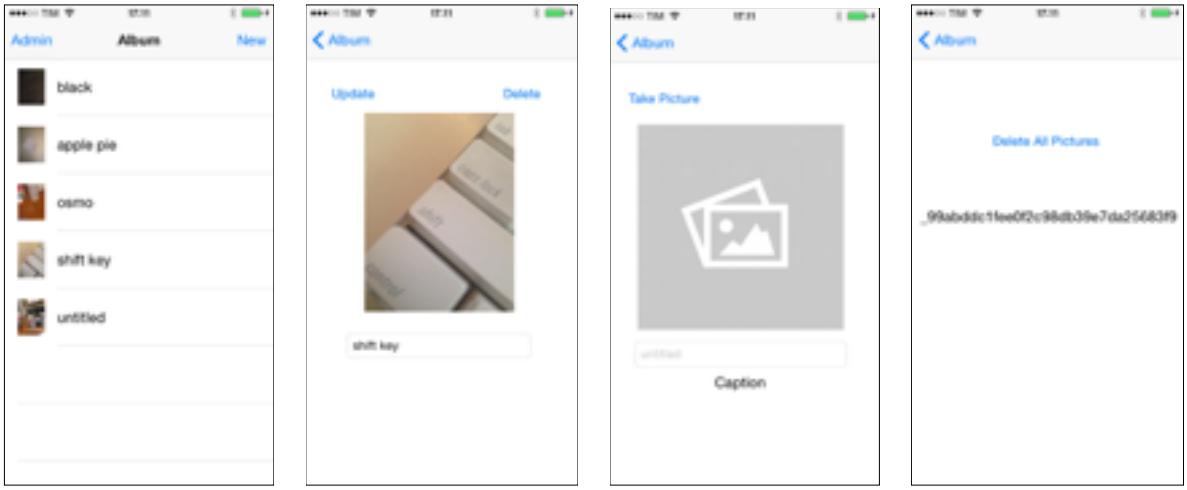
All the server communication is managed through a framework called RestKit. This framework sets up the path address system which makes sending requests easy and more Restfull. All possible requests are presented in group of methods which send requests and manage responses without the need for any low level coding. The objects are also sent and received in *json* format, but Restkit, after configuring mapping paths, maps the data directly to objects or a local database. Although a first version did keep a local database which worked, the current app only uses objects. Authentication is also easily configured and presented no challenge. Binary data was the most sophisticated request, but still was simpler than its Qt counterpart. The binary data POST request can be seen on the code bellow:



3.3 CPict

The CPict main goal was to experiment with CloudKit and Swift, recently introduced by Apple with iOS 8 in the Fall of 2014. Allegedly, the company's motivation to create these new resources was to make the development of client / server mobile applications easier and more intuitive. In that respect, CPict app is the simplest of the three versions of the client application.

CPict was also developed in Xcode as RPict, and took advantage of the storyboard tools for the design of the user interface. The app's UI is basically the same as RPict, it contains a main view presenting the photo album, a detail view for displaying individual pictures, a camera view for taking pictures and an administration view for managing the user's configuration. The figure below shows all these four views. Their layout was designed entirely using the Storyboard.



(a) Album View (b) Photo Detail View (c) Camera View (d) Admin View

The app's code was written in the Swift language. Differently than Objective-C, Swift does not have header files to declare the object's. The compiler handles automatically almost all global aspects of the project. Consequently, CPIct is composed of only seven source files: four of them implement each of the UI's views, two of them are dedicated to the CloudKit data objects and the last one is the main app object. The whole project has around 250 lines of Swift code. Below we describe the main objects and functions.

The data objects encapsulate CloudKit entities, they are proxies for Records in the database (CKRecord). The most important object is Pict, it represents a picture that is composed

of a name, a thumbnail image and a full image. The full image is a binary asset, while the name and thumbnail are record fields, respectively a string and binary data. The other object is User, which is automatically handled by the CloudKit framework.

In order to implement the interface of the data objects with the UI views, a picture manager object was created. The PictManager is responsible for the communication between the CloudKit database with the rest of the app. It contains the functions to load the pictures from the database container, as well as, to add, update and delete individual pictures. Just to illustrate how simple is to implement this functionality with the CloudKit framework, below is the code to update a picture record.

```
func updatePict(pict : Pict)
{
        beginRefresh()
        func recordSaved(record: CKRecord?, error: NSError?) {
            if (error != nil) {
                println(error)
            } else {
                endRefresh()
            }
        }
        database.saveRecord(pict.record, completionHandler: recordSaved)
}
```

4 - Server System

The server side of the photo-mobile app is responsible for persistent maintenance of the user's data that is manipulated by the mobile client side.

Basically, the server provides storage and access to the data. The database contains a collection of picture records for individual users. The system implements operations for user creation, authentication, operations to query and retrieve sets of pictures, as well as, operations to insert, delete and modify individual pictures. While the user and picture records are directly represented in the database, the raw image data is maintained separately as binary files with their associated handles kept in the database.

We have developed two versions of the server system. They both implement the basic functionality described above. One of them, employs a web server platform and a sqlite database. This server is used by the clients QPict and RPict. The other server takes advantage of the CloudKit framework and employs the iCloud infrastructure. This server is used by the CPict client.

4.1 Web

The web server is responsible for accepting requests and respond with data that was created using Django web project and a module named Django REST framework. The server is hosted in a machine running the Ubuntu operating system.

There are two ways of accessing the data in the server: through the web interface or by making REST calls directly by command line. Both ways allows for query only, if the user is not authenticated, and for query and modification otherwise.

4.1.1 Available request calls

The server is developed to answer requests for pictures and users data, and return them in json format.

4.1.1.1 Picture request calls • Metadata list

Returns a list of all the pictures stored in the database.

Example:

```
curl -H 'Accept: application/json; indent=4' http://example.com/rpic/pictures/
[
        {
        "url": "http://example.com/rpic/pictures/4/picture/",
        "id": 4,
        "user": "bob",
        "uuid": "D954AA48-57AF-451E-AE94-61CCF7F10E71",
        "title": "Mac Keyboard Madness",
        "created": "2014-05-26T17:58:32.844Z",
        "last_modified": "2014-11-05T18:14:16.326Z",
        "latitude": "22.9659866667",
        "latitude_ref": "S",
        "longitude": "43.237545",
        "longitude_ref": "W"
        }, {
        "url": "http://example.com/rpic/pictures/1/picture/", "id": 1,
        "user": "alice",
        "uuid": “0bcb9d35-0e6f-4ca0-973f-df9d73f7a40a",
        "title": "Nice Photo",
        "created": “2014-05-23T19:14:11.728Z",
        "last_modified": "2014-06-06T21:38:44.571Z",
        "latitude": "22.9657523459",
        "latitude_ref": "N",
        "longitude": "43.2363758653",
        "longitude_ref": "E"
        }
]
```

- **Metadata detail**

  Returns the metadata detail for a specific picture.

Example:

```
curl -H 'Accept: application/json; indent=4' http://example.com/rpic/pictures/1/
{
        "url": "http://example.com/rpic/pictures/1/picture/", "id": 1,
        "user": "alice",
        "uuid": "0bcb9d35-0e6f-4ca0-973f-df9d73f7a40a",
        "title": "Nice Photo",
        "created": "2014-05-23T19:14:11.728Z",
        "last_modified": "2014-06-06T21:38:44.571Z",
        "latitude": "22.9657523459",
        "latitude_ref": "N",
        "longitude": "43.2363758653",
        "longitude_ref": "E"
}
```

- **Picture detail**

Returns the picture's file path with id and username association.
Example:

```
curl -H 'Accept: application/json; indent=4' http://example.com/rpic/pictures/4/picture/
{
        "url": "http://example.com/rpic/pictures/4/picture/",
        "id": 4,
        "user": "bob",
        "picture_file": "user/2/D954AA48-57AF-451E-AE94-61CCF7F10E71.jpg"
}
```

4.1.1.2 User request calls

- **User list**

Returns a list of all the registered users .
Example:

```
curl -H 'Accept: application/json; indent=4' http://example.com/rpic/users/
[
        {
        "url": "http://example.com/rpic/users/1/", "id": 2,
        "username": "bob",
        "email": "bob@example.com",
                        "pictures": [
                                        "http://example.com/rpic/pictures/4/picture/",
                        ]
        },
        {
        "url": "http://example.com/rpic/users/1/",
        "id": 1,
        "username": "alice",
        "email": “alice@example.com",
        "pictures": [
                "http://example.com/rpic/pictures/1/picture/",
        ]
        }
]
```

- **User detail**

Returns the details for a specific user.
Example:

```
curl -H 'Accept: application/json; indent=4' http://example.com/rpic/users/1/
{
        "url": "http://example.com/rpic/users/1/", "id": 1,
        "username": "alice",
        "email": "alice@example.com",
        "pictures": [
                "http://example.com/rpic/pictures/1/picture/", ]
}
```

4.2 iCloud

The iCloud-based server uses the CloudKit Framework [2].

5 - Conclusion

The concepts underlying new technologies and architectures for mobile distributed applications and the comparative examples of a client / server photo-mobile app presented in this technical report aim to establish insights and guidelines for software development.

References

[1] Swift Programming Language - https://developer.apple.com/swift/

[2] iCloud Framework for Developers - https://developer.apple.com/icloud/index.html

[3] Webservice API framework for Django - http://tastypieapi.org

[4] Django REST Framework - http://www.django-rest-framework.org/

[5] Django Project - https://www.djangoproject.com/

[6] Qt Project - http://qt-project.org