

B.Comp. Dissertation

**Panex: Repository for the Management and
Exploration of Patient Data**

By

Mohit Singh Kanwal

Department of Computer Science

School of Computing

National University of Singapore

2012/2013

B.Comp. Dissertation

Panex: Repository for the Management and Exploration of Patient Data

By

Mohit Singh Kanwal

Department of Computer Science

School of Computing

National University of Singapore

2012/2013

Project No: H0401070

Advisor: Assoc. Prof. Leow Wee Kheng

Deliverables:

Report: 1 Volume

Program: 1 Volume

Database: 1 Volume

Abstract

Ever pervasive use of computational algorithms has changed the face of medical data analysis. Certain surgeries such as craniomaxillofacial (CMF) surgery require sophisticated planning. Hence computer-aided surgery planning through research algorithms becomes crucial to its success. Most analysis algorithms require themselves to be installed in a computer before they can be used. Some of the analysis algorithms also require technically intensive setups on local workstations, which medical clinicians don't have much experience with. This significantly prevents mass distribution and adoption of new algorithms in the medical research industry.

To advance the state-of-the-art in computer-aided surgery planning and medical data analysis, we need to collect and analyze a large amount of patient data. The goal of this project is to develop a repository and processing center for the remote management and processing of patients' medical data. The system will be a server-based system so that it can be extended to a multi-server system. It will have to manage text data, medical images and 3D models, but also the remote installation and batch execution of software algorithms to process the data, and reporting of processing results to the users. It should also support the access, retrieval and display of the data by authorized users. This ultimately will allow users to execute sophisticated algorithms on patient data without requiring them to go through extensive technical setups.

Subject Descriptors:

- C.2.1 Network Architecture and Design
- C.2.4 Distributed Systems
- D.2 Software Engineering
- D.4.1 Process Management
- H.3 Information Storage and Retrieval

Keywords:

Web Services, client/server systems, distributed systems, information retrieval, process management

Implementation Software and Hardware:

Ubuntu Linux 12.04 Precise Pangolin, Ruby 1.9.3, Rails 3.2.11, Qt 4.8, CMake 2.8.10

Acknowledgements

Few products are strokes of single-handed individual brilliance. Most of us need assistance to realize our full potential. This thesis is no exception.

I wish to thank my supervisor, Associate Professor Leow Wee Kheng, for his guidance and continuous support throughout the entire project. Prof. Leow helped me challenge myself and push towards higher goals. He taught me the right approach of looking at problems, formulating a good design and grounding myself in the core tenets of self-belief, consistency and coherence. Without his encouragement and insight, this thesis would have been impossible to complete.

Special thanks to Cheng Yuan, PhD candidate at the School Of Computing, for providing me with sample research programs and helping me debug them during the integration process.

A mention of thanks goes out to the numerous course instructors in the School of Computing who have helped me develop a passion for software and system development. Without these courses, I think I would not have even thought through this project.

I also would like to sincerely thank numerous unknown open source contributors without whom the technical directions of this project would be a decade's work for me.

I am grateful to all my friends and colleagues: Amulya, Rajul, Arvin, Navneeth, Vishnu, Cutee, Rohan, Gaurav, Ivan, Nayan, Nikhil, Vasu, my University Scholars Programme buddies: Haikel, Danielle, Naomi, Kenneth, Anyi, the entire senior's level and countless others who made my life as an undergraduate in Singapore memorable and provided words of hope and wisdom during the last days (nights) of thesis compilation.

Lastly, I thank my ever-awesome family for supporting me through the entire period and bearing with me for periods of minimal communication. Their endless love, unconditional support and encouragement have given me the strength to accomplish this journey.

List of Figures

Figure 1 Panex Interaction Ecosystem with different users, with a future platform application such as a Social Health App	2
Figure 2 A popular DICOM Image Browser OsiriX running on the Mac OSX platform	6
Figure 3 Entity-Relationship Diagram for Panex entities	10
Figure 4 Panex Architecture Diagram	13
Figure 5 A typical OAuth workflow (Credits: salesforce.com).....	15
Figure 6 Main Screen of Panex Client App showing patients (left) and their data (right)	20
Figure 7 Service Upload Dialog Box	20
Figure 8 Service Run Status Dialog displaying the server load	21
Figure 9 Upload Patient Data Dialog.....	21
Figure 10 Server side application architecture.....	25
Figure 11 Server Side Request Flow	26
Figure 12 A simplified Job Execution setup	29
Figure 13 A simplified setup of MVC used in the client side	32
Figure 14 screenshot of the Github Issues for the server code.....	34
Figure 15 Trello board with the task list divided into three categories	35

Table of Contents

Acknowledgements.....	iv
List of Figures	v
Table of Contents	vi
1 Introduction	1
1.1 Motivation	1
1.2 Project Objectives	3
1.3 Report Organization	3
2 Background.....	4
2.1 Medical Image Digitization	4
2.2 The DICOM standard.....	5
2.3 Existing Systems	5
2.4 Recent Developments	6
3 Requirements Analysis	8
3.1 Actors.....	8
3.2 Functional Requirements	8
3.3 Non-Functional Requirements	9
3.4 Security Requirements.....	9
3.5 Performance Requirements.....	9
3.6 Conceptual Data Graph.....	9
4 System Design	11
4.1 Overview	11
4.2 Proposed Solution.....	11
4.3 Architecture	12
4.4 Featured Architectural Components.....	14
4.4.1 User Functionality Components	14
4.4.2 Request Validation.....	14
4.4.3 Patient Data Management.....	16
4.4.4 Service Design	16
4.4.5 Service Run.....	17
4.4.6 Service Result Linkage	18
4.4.7 Application Download	19

4.5	Interface Design.....	19
4.5.1	Single unified user interface	19
5	Implementation and Testing	22
5.1	Overview	22
5.2	Technology Stack.....	22
5.2.1	Desktop Application Framework.....	22
5.2.2	Server Application Framework.....	22
5.2.3	Database	23
5.2.4	PACS Server	23
5.2.5	Web Server.....	23
5.2.6	Hardware.....	24
5.3	Server Side Components.....	24
5.3.1	Rails Model View Controller Architecture.....	24
5.3.2	Authentication and Authorization System	27
5.3.3	Job Queuing and Service Management.....	28
5.3.4	Process Management	30
5.4	Client Side Components.....	30
5.4.1	Qt Model View Architecture	31
5.4.2	Data Serialization and Uploading	32
5.4.3	Offline Data Management.....	32
5.4.4	Multi-threaded Upload.....	33
5.5	Testing Strategy.....	33
5.5.1	Unit Testing	33
5.5.2	Functional Testing	33
5.5.3	Integration Testing	34
5.6	Continuous Integration and Development Methodology	34
5.7	Miscellaneous.....	35
5.7.1	Documentation.....	35
5.8	Future Work	36
5.8.1	Administrator Service	36
5.8.2	Testing and Troubleshooting Service	36
5.8.3	Integration of diverse Data Sources	36
5.8.4	Integration of diverse Service Types	36
5.8.5	Integration of service execution time.....	36
5.8.6	Foundation for a telemedicine system	36
5.9	Implementation Evaluation	37

6	Conclusion.....	38
7	References	39
8	Appendix A – External Libraries	41
9	Appendix B – Service Developer’s Guide	42

1 Introduction

1.1 Motivation

The problem of standardizing medical image communication was a galvanizing field of research in the 1980s. It concluded with the establishment of a protocol known as **DICOM** (Digital Imaging Communication in Medicine). It is the de-facto standard for medical image communication between vendors, devices, hospitals etc. and widely supported in medical applications worldwide.

The research around the medical image digitization has revolved around retrieval of images (Onyebuchi, 2011). This has to some extent been resolved by the installation of a PACS (Picture Archiving and Communication Systems) server, which serves medical images in response to queries. In the medical data analysis research field, algorithms are often run on patient data. As such the more data available the better it is for these algorithms. However, local machines cannot be trusted with storing huge amounts of patient data and making it available to a number of different programs. We thus need a central repository for data storage, which is solved by the usage of PACS. To make the programs available to multiple users, some sort of program repository is needed. Further, those programs would be required to handle PACS integration by themselves. Moreover, multiple variations of the algorithms would be required to run for research using the data stored as a test bed, which further complicates integrating them with PACS. Software services are not supported on existing PACS systems (Wong & Huang, 1996). *However for the purpose of research and analysis it is desirable if patient data and analysis services were housed in a single centralized workstation and accessed through the network by multiple users.*

Moreover, even if services were built to integrate within PACS it would be difficult for them to interface with other types of possible input mediums which do not understand the DICOM protocol (Shiroma, 2006). An example would be analysis of textual data (output) from an algorithm working on patient images. Extensibility of the services to support various different types of inputs is therefore required as well. PACS and DICOM were designed to allow communication of image data to be

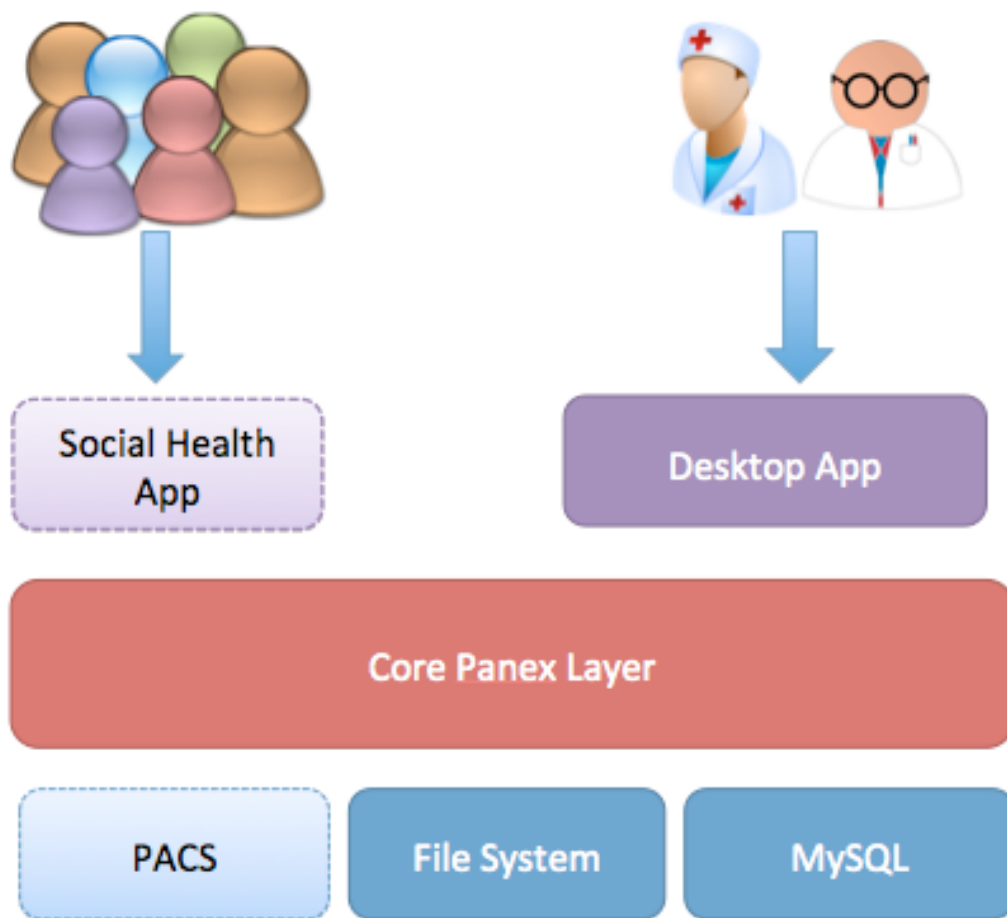


Figure 1 Panex Interaction Ecosystem with different users, with a future platform application such as a Social Health App

standardized. The proposed system through this project would attempt at unifying a multitude of distributed services, image databases and other documents behind a single user-interface, which a user, clinician etc. can use from the comfort of their desktops.

Target Users

Panex, as described currently according to the requirements serves 3 kinds of users:

- **Clinicians** – This category of users are usually concerned with retrieval of images, uploading of images, running/stopping services etc.

- **Developers/Researchers** – This category of users house the system developers who add additional services and application to the system.
- **Casual Users** – Sample category of users who would use the platform for getting information about their own diseases, a precursor to telemedicine or e-Health form of information distribution.

1.2 Project Objectives

The core aim of the project is to come up with a design and implementation of a prototype system to allow storage and retrieval of images/documents relating to CMF surgery. Based on a client-server model and a data repository model, the system aims to be extendable and support a wide range of operations.

The project aims to fulfill the following objectives:

- Come up with the design and implementation of a prototype that allows images to be retrieved from a Server adhering to a Digital Imaging and Communication in Medicine (DICOM) format.
- Allows studies/reports relating to a patient to be retrieved from the server
- Allow background system services to be installed and run in the server and their results be retrieved and delivered to the user
- Serve as a storehouse of stand-alone medical applications, also allowing their retrieval from the server onto a local client (e.g. a Desktop).

1.3 Report Organization

The first chapter of the report deals with the introduction of the problem and the general survey of literature and existing tools in the arena. The second chapter of the report deals with system designs, architecture issues and the decisions that were made to support different kinds of data and usage scenarios. The third chapter deals with the technical details of how the system has been implemented including details about the platform, special optimizations, algorithms etc. In the end, there's Appendix, A which deals with the external and third party libraries used in the development of *Panex*. Appendix B contains information, which Service Developers must adhere to for their service-related binaries to work on the server.

2 Background

2.1 Medical Image Digitization

Digitization of medical images commenced in the field of radiology in the late 70s and early 80s. Several different implementations of digital medical images were carried out during these years (Eichelberg et al., 2004). However, the implementations lacked technical maturity and as such standardization of medical imaging technology was needed which started with the first Conference on Picture Archiving and Communication Systems (PACS) in 1982. PACS conferences were instrumental in introducing the wider medical community to the concepts of digital image communication and also paved the way for the acceptance of the DICOM communication standard (Onyebuchi, 2011).

PACS integrates many components related to medical imaging for clinical practice typically consisting of image and data acquisition, storage and display subsystems integrated by various digital networks (Zhang, Sun, & Stahl, 2003). PACS installation tend to be customized and can range from simple to pretty complex ones consisting of collaboration of a number of components (both hardware and software) within a hospital.

From a networking standpoint, a PACS consists of a central server that contains a database of associated medical images (McGeary, n.d.). The server is then able to communicate back to a local workstation portal where the physicians/clinicians can take a look at the images via a Local Area Network (LAN) or a Wide Area Network (WAN) such as the Internet.

To communicate the data within the images (similar to meta-data) the Digital Imaging Communication in Medicine (DICOM) standard was introduced (Huang, 2003). With the introduction of DICOM, the issue of transmitting, archiving and retrieval of medical images was largely considered as successfully solved and is being widely used in the world today or at least being considered from a digital standpoint (Suh, Warach, & Cheung, 2002). The DICOM standard does not only provide a

mechanism to communicate medical images to a repository (mainly the PACS server) but also provides a way of storing meta-data within the image record itself.

2.2 The DICOM standard

In the 1980s with the introduction of PACS it was felt that different workstations and devices needed to communicate in order to understand patient records and transfer information. Moreover, the PACS hardware and medical imaging devices were vendor-specific and therefore made it hard to integrate. As such in 1993 the DICOM standard was adopted among the American College of Radiology and the National Electrical Manufacturers Association (Grauer, Cevitanes, & Proffitt, 2009).

Contrary to popular belief, DICOM is not an image format like JPEG, GIF etc. It is a full-fledged data transfer, storage and display protocol built and designed to cover all functional aspects of digital medical imaging (Association, 1993).

A DICOM record consists of a (1) DICOMDIR file, which includes patient information, specific information about the acquisition of the image and a list of images that correspond to axial slices forming the 3D image; and (2) a number of sequentially coded images that correspond to the axial slices (Grauer et al., 2009). These slices form a 3D image of the subject when assembled in the correct sequence.

2.3 Existing Systems

Most of the current systems can be classified as satisfying the retrieval aspect of the medical images or providing services to a general audience at large. However, research related requirements could be different from general audience requirements. A number of commercially available DICOM viewers allow clinicians to communicate with data repositories and retrieve/store images locally or on the central server. Examples of such software are OsiriX, dcm4chee, dicombrowser etc.

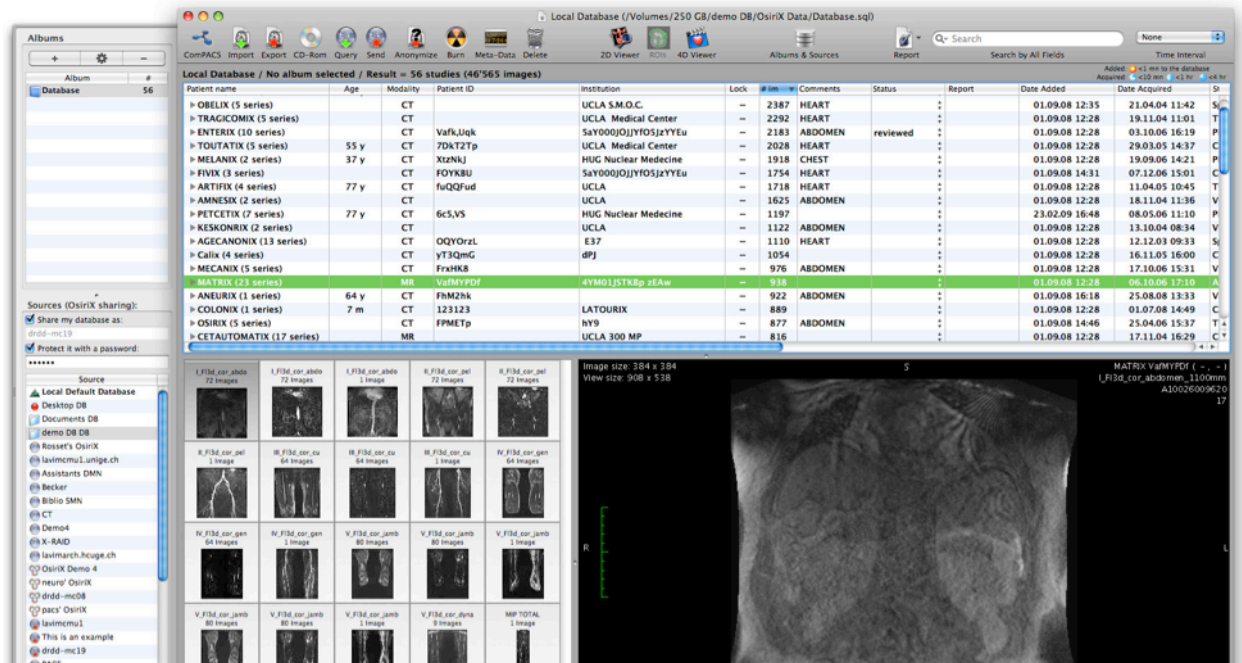


Figure 2 A popular DICOM Image Browser OsiriX running on the Mac OSX platform

Another category of software applications exists which provide services to the general audience at large. These are generally referred to as Electronic Patient Record Systems or telemedicine service providers. Examples of such systems are AmeriDoc, Consult A Doctor etc (Zhang et al., 2005). These systems centralize patient information databases and provide services such as consultations, diagnosis etc. They are targeted towards general public users and not specifically researchers.

2.4 Recent Developments

The question of how to store large number of digitized images in DBMS and retrieve such images has been an area of active research recently. *Now the focus of research is shifting to providing services relating to generation and acquisition of images to the post-processing and management of these images.* The motive is to realize the greatest possible benefit from the data that already exist (Wong & Huang, 1996). Various models of client-server based architecture have been proposed, some based on content-based retrieval, others based on meta-data-based retrieval. Some intelligent-systems based on Artificial Intelligence techniques are finding applications in the medical image retrieval industry as well (Onyebuchi, 2011).

With image retrieval being addressed through the means above, the next phase of medical imaging innovation resides in the ability to analyze and perform

applications on the collected repository of data. This has multiple ramifications and includes applications like telemedicine, surgery planning as possible by-products.

Today integration demands of medical image analysis require more than just analysis of images. Sometimes analysis of images along with medical data is required. Data (or Image) analysis can provide valuable clues to make crucial decisions. Therefore a comprehensive solution is required to make collection, archiving and analysis of medical images a reality.

3 Requirements Analysis

A requirement analysis phase was done in the first semester, which allowed identification of the requirements as described in the following sections

3.1 Actors

The actors in the system are identified to be:

1. **Doctors and the Medical Staff (Clinicians)**- The important and most frequent task of this group is obtaining information from the server and downloading DICOM images from the server to view them on their local machines
2. **IT Administrators** - Main function includes to be able to view logs, backup information, create new user accounts etc.
3. **Research Staff** - Main function is to be able to upload and analyze service outputs through the desktop client.
4. **Administrators** - Oversee the functionality of the system and resolve account related issues and approval of new signups.
5. **Patients** – Category of users who can themselves submit their own images using the system and view the output of the services

Of the described actors only 1 and 3 are critical to the functioning of the system, the administration and patient related requirement has been left out as a future work.

3.2 Functional Requirements

Functional requirements consist of the functionality that the system is supposed to deliver upon completion, this includes:

1. The application should support multiple users at the same time
2. Being able to retrieve patient images in DICOM format
3. Being able to handle uploads of DICOM images and store it in the server
4. Being able to query studies about a patient from the DICOM image meta-data
5. Allowing new users to create an account as clinicians and researchers, two of the most critical user groups
6. Providing functionality for software developers to upload their stand-alone applications to the server
7. The system should allow uploading of backend services

8. Allow running of backend services on patient data
9. It should provide functionality to allow users to download the stand-alone applications from the server
10. Allow download facility for the service results to be downloaded by the end users

3.3 Non-Functional Requirements

Non-functional requirements specify criteria that are required for the operation of the system. These include:

1. Uploading of patient data should be fast and efficient
2. Client application should not download the patient data multiple times, it should for efficiency reasons stores its own local cached copy
3. Being able to log each action performed by the user
4. Usage of open source technologies to prevent licensing constraints
5. Being able to deploy with minimum constraints

3.4 Security Requirements

1. Each web request should be validated and the request invoking user checked and verified
2. Only researchers should be allowed access to the backend services.
3. Passwords should not be stored in clear text
4. Command Line parameter injection within the service binaries should be prevented

3.5 Performance Requirements

1. The system should make sure that running backend services does not hog the CPU or slow down the server application
2. The web service API should be fast and able to handle multiple concurrent clients simultaneously

3.6 Conceptual Data Graph

A Conceptual data graph was drawn to better understand how the different data entities are related in the system. At the centre of the diagram is the patient and it is the single most important entity in the system. All the patient data uploaded through

the client will be linked to and searchable by the patient details. In healthcare, maintaining patient details is of utmost importance.

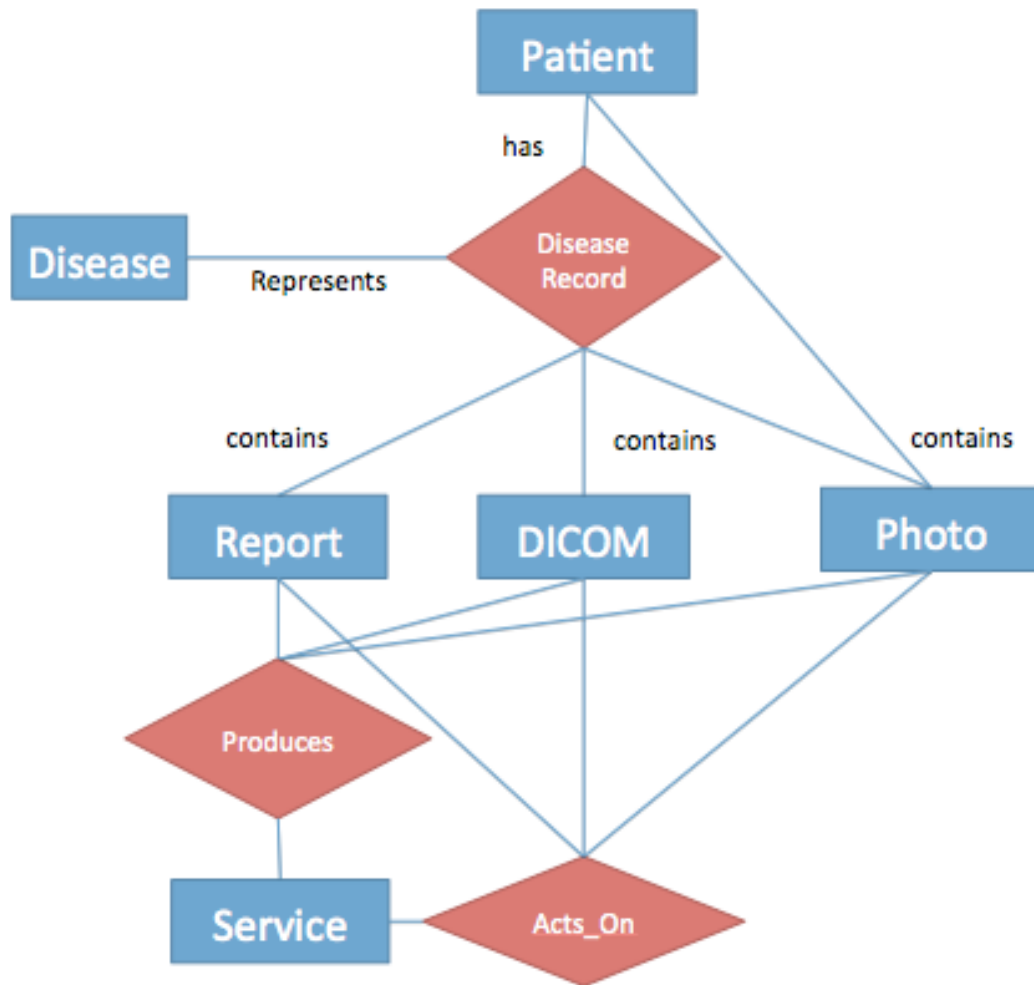


Figure 3 Entity-Relationship Diagram for Panex entities

In Figure 3 Entity-Relationship Diagram for Panex entities, the major entities within Panex can be seen. Every Patient can have multiple Disease Records, which represent a Condition and serve as a point of aggregation for the various different types of data supported within the system. The other major entity within Panex is the **Service**, which represents the developer research programs, or batch commands. Ideally, these will function on the various different types of data uploaded to the server and in turn will also produce a variety of different results. The results would be linked to the patient via the disease records. A patient can have multiple data belonging to any of the data categories. These will be handled in the appropriate manner.

4 System Design

4.1 Overview

Panex is a management tool designed for patient data as well as applications running on those data. There are three parties present in the ecosystem: the service clients, the server and the patient data and application repository. The service clients can be applications that consume the services provided by the server. For this project, only a desktop client has been implemented, perhaps in the future a mobile client can be implemented. The server manages retrieval and access of all patient data through a variety of data sources: file system, database management systems or PACS server.

In principle, this allows separation of concerns of the display of data from the storage. In addition, the server abstracts a lot of other components, for example: the job executors which run the services requested by the end-users as well as other pre-processing and post-processing of patient data. The client applications therefore only need to adhere to the Application Programming Interface provided by the server.

4.2 Proposed Solution

The proposed solution relies on a **three-tiered, client-server** architecture for the application. Since it is required that the application be available to multiple concurrent users, therefore a network based client-server application is proposed. It incorporates a relational database, a secure web service, middleware software, network storage and other technologies. The three-tiered layered architecture design is a time-tested paradigm that allows users access to distributed resources through interaction with a single interaction point (Wong & Huang, 1996).

Clients would interact with the application through the use of a desktop application. The language of implementation of the desktop application will be Qt since it is easy to learn and most importantly cross-platform. The presence of DCMTK, a DICOM Imaging Toolkit, essentially a wrapper written in C++ to facilitate DICOM related operations also helps to make the case for Qt. Moreover, Qt's plugin based object oriented design system makes it easy to make minor changes to the code base to extend functionality (Blanchette & Summerfield, 2006).

Furthermore, the server would house **services**, essentially a collection of binaries or runtime modules written for the Linux Platform that would provide a mechanism to analyze patient data in the background. This will be done invisibly from the user's notice. The server runtime environment would provide the standard libraries etc. for the services to run in a stand-alone fashion.

Another type of application service provided by the project would run **stand-alone** in a user's desktop computer. These would be delivered via the means of software updates or manually if a user wants to download and install by themselves.

4.3 Architecture

The architecture of Panex derives from the well-known three-tier architecture for network applications. There is a server data layer, a server business logic layer and a client-facing layer. Figure 4 shows a detailed architectural view of the system. The server handles client requests and contains components for handling the different requests dealing with different categories of operation. The client side follows the Model View Controller paradigm for the display of data and interaction with the user. Model View Controller separates the notion of data and information from its display. As such, the source of data can be changed from server to a local repository without affecting the display of data. This helps in the scenario where there might be the need to access data in offline mode or when the server is unreachable. The server side has two loosely coupled types of components: the job executor and the request handlers. The job executor is a self-contained module housing the necessary components required to make services run on patient data in the background. The requests handlers are housed within the server web-container handling client requests and providing the necessary responses. The job executor is not housed in the server web-container and runs as a separate module, allowing it to be scaled across machines seamlessly. All of them though, are tied to the database, which stores the patient data in addition to the results of the various services.

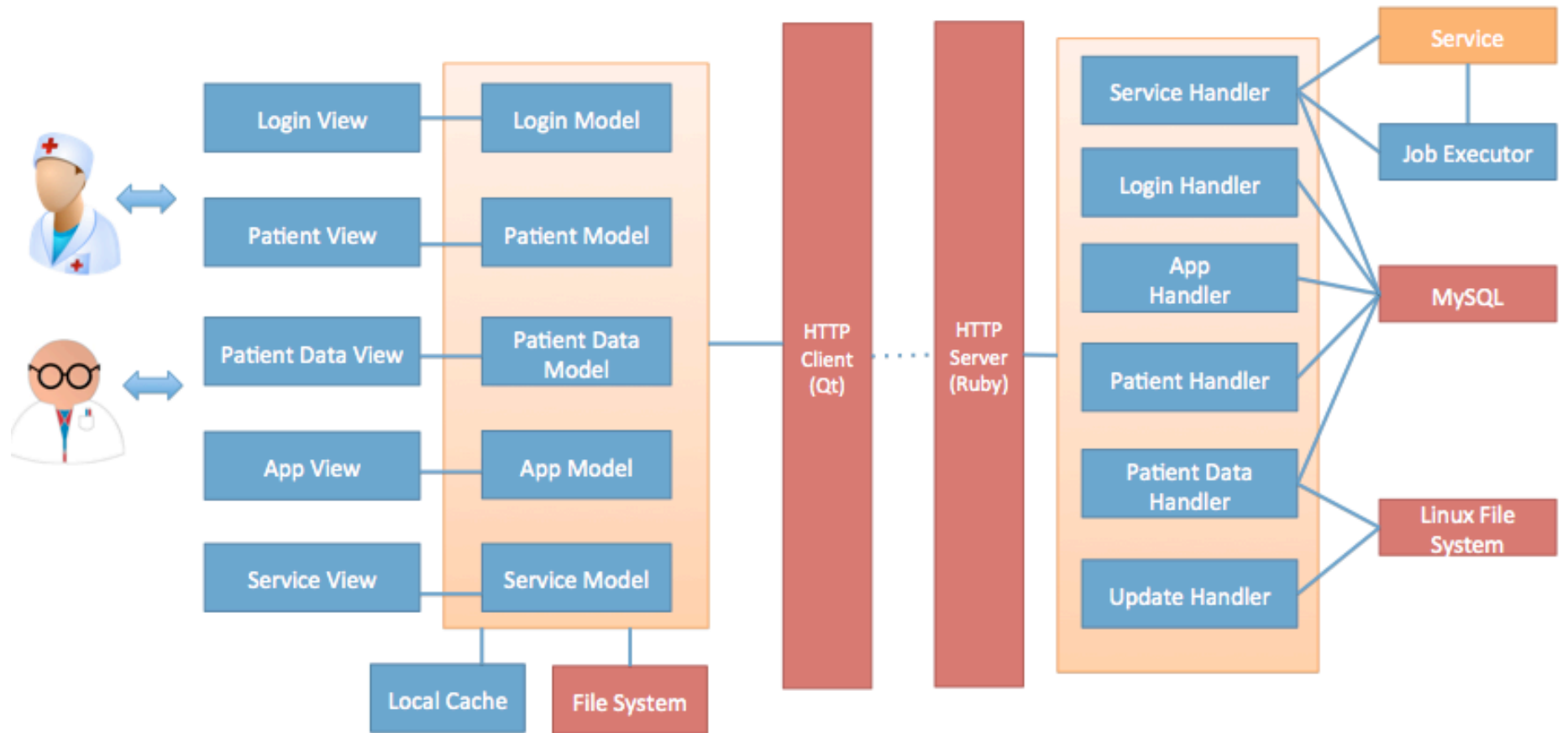


Figure 4 Panex Architecture Diagram

The server provides an Application Programming Interface (API) for client application programmers to take advantage of the different functions provided by the server backend.

4.4 Featured Architectural Components

In this section, we present the major categories of components in system architecture and give a brief overview.

4.4.1 User Functionality Components

The functional requirements of the software state that new users can be allowed to create their accounts as well as login to the application remotely and start enjoying its features. Therefore, the system design takes in consideration of the different types of users and their details. It allows a login mechanism through an authentication module for web requests.

The design also handles tracking of the user's action through various access controls. Each user has one or more roles defined. An Access Control List (ACL) stores the allowable user roles that can access a particular resource on the server. The patient handler and the Service handlers check for the ACL listing before executing any data related queries. This satisfies some of the security requirements as described in Chapter 2.

Another requirement is to allow services to be executed on the server. In order to achieve this, a web service is designed which can handle HTTP requests regarding service invocation and execute the necessary background queries and tasks.

4.4.2 Request Validation

With a lot of requests being sent to server using the network, it is important that the service requests are validated in all the scenarios. One of the possible ways of securing such requests would be to use a token mechanism, which can allow verifying of the origin of these requests. OAuth is one of the techniques that allows for authentication and then authorization of HTTP requests. Figure 5 shows a typical OAuth workflow.

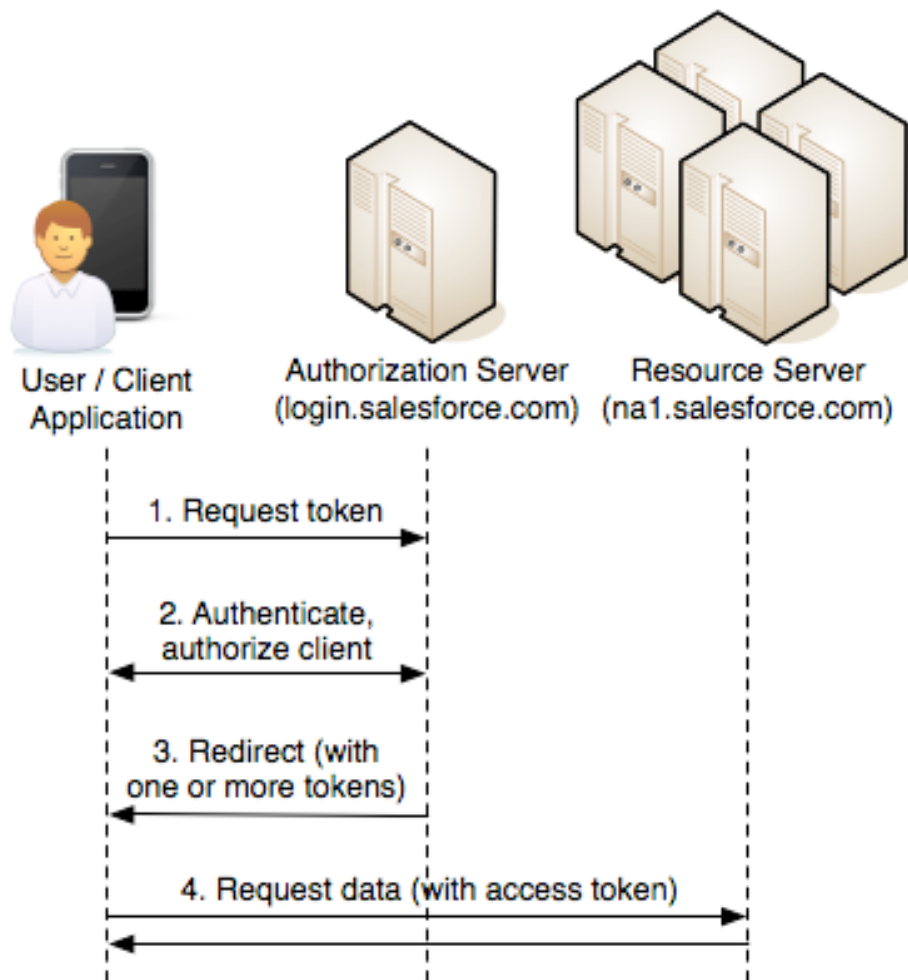


Figure 5 A typical OAuth workflow (Credits: salesforce.com)

It provides several advantages as compared to various different approaches:

- The tokens which are meant to legitimize requests with the server, are tied to a particular user and client application, so even if an attacker sniffs them in plaintext, they would need access to the client application for authorization purposes.
- Also, OAuth specifies that the tokens be expired every few days or hours; this prevents the client application to store the token forever and prevent it from sending requests on behalf of the user. This implies, a client would need to login once tokens are expired.

Panex uses OAuth to authorize client application requests. The authentication module is separated from the main application. As such, all requests to the Handlers must pass through the authentication module first before being acted upon.

4.4.3 Patient Data Management

Maintaining data integrity and safety of patient data is of utmost importance. The requirement is to support different types of data. Patient data usually comprises of DICOM files, text and pdf reports and other type of image data. The services will produce the same type of files. The system should be able to handle the diversity of data involved but at the same time provide a uniform interface to access the patient data and allow its manipulation.

As such, the system design makes use of the Linux file system to store the patient data since storing such information in the database in binary form would cause the database to grow very fast ultimately putting strain on it. Another advantage of using the File System is that it allows for efficient copying of the data when services are invoked. Instead of querying the database, the services are given a copy of the patient data to work with. A MySQL database allows storage of the meta-data related to the data elements for faster retrieval and searching. Furthermore, the data storage can be scaled across different hard disk drives or even different machines and data can be retrieved using FTP (File Transfer Protocol).

Users can therefore upload different types of data related to a patient through the client application. The upload process for the different types of data is the same.

4.4.4 Service Design

The requirements states that users can invoke batch services on the server. A service is an external program, which can be invoked through a command line with some additional parameters. Currently, the researchers run these services on their personal machines and analyze the results produced. Panex takes a different approach. The web service approach and the splitting of client and server allow the services to be invoked via network. The external programs are given the location of the patient data files in which they are interested through a parameter in the command line. They are also given an output directory parameter where they can write the results.

This design is language agnostic. As such, services can be written in various different languages such as Ruby, Python, Perl, Java, and C++ etc. and be linked to the libraries provided by the system. This design allows the services to be separate from the core application logic. This is important, as services should not be allowed to interfere with the normal working of the web service.

4.4.5 Service Run

This section gives an overview of the different design decisions related to the running of the services within the server environment.

4.4.5.1 Sandbox

Some of the performance requirements state that external service invocation should not cause interruptions in the normal execution of the application server. Running external programs can cause lots of issues. Some programs might want to try malicious activities, such as accessing illegal data in other directories within the file system. As such, the Panex design proposes services need be run within a sandboxed environment complete with the right access permissions. Some of the other problems associated with running external programs are that some programs might run for a long period of time. The server must be able to pre-empt the running of such programs and time them out so that other queued services might be able to run successfully.

In order to cater to the security setup and make the server have complete control over the running of such services, using a separate user account is advised. Thus, within the server environment there is a separate user account that has access to only the necessary input and output folders and nothing else on the file system. The user service does not have any special privileges and therefore is sandboxed within the server environment working at a privilege level lower than that of the server application.

4.4.5.2 Job Queues

Another design issue related to the running external services is issue of running them synchronously or asynchronously. Running the services synchronously would cause a user to wait for any output files. However one cannot estimate the runtime of the external programs therefore an asynchronous model makes much sense. A separate component called the **Job Executor** takes the responsibility of executing the services. It reads the services that are to be run from a queue called the **Job Queue**. The external programs are placed in a queue and then executed whenever the Job executor is idle.

By making a Job Queue, we can asynchronously execute the tasks without having to worry about when the tasks get finished. Separating the job executor allows us to have multiple instances of the executor running which can then read the jobs

from different queues or from the same queue. Moreover, each queue can have a different priority, which is then factored by the executor when executing services from different queues.

4.4.5.3 Notification Hooks

In addition to the above-described service run setup, we also have different stages of a service run and notifications hooks are fired at different stages of the service run. The different stages are as follows:

- **Enqueue Hook:** this is the hook fired when a job is enqueued to the job queue. This means that the job is in a ready state
- **Before Hook:** this is the hook fired when a job is about to be run, that is this is immediate stage before the actual running of the job
- **Success Hook:** this hook is fired when a job has finished execution and exited with a zero-status i.e. the program exits normally with a return value of 0
- **Error Hook:** this hook is fired when a job exits with a non-zero exit value or if an exception would be raised during the job run.
- **After Hook:** this is the hook fired when a job has been completed, the after hook is always fired. It is possible that a success/error hook might be fired as well, but in the end of it, the after hook gets invoked.

The design of the notification hook system is based on the observer design pattern. Other modules can subscribe to these hooks and then execute specific functions like cleaning up the temporary directory after a service, copying files or sending notifications regarding a successful service run to the users.

4.4.6 Service Result Linkage

Another important requirement of the service run design is to be able to link back the results to the patient so that they can be searchable and be queried at a later stage. This would require access to the patient database as well as to the file system; as such the services themselves cannot be trusted to do the linking. This would be required to be carried out by the main application server. The design of notification hooks is very useful in this particular scenario since the necessary functions can subscribe to the success notification hook and execute the necessary linking operations.

Not only that, other notifications hooks can be subscribed to as well and the appropriate notifications be sent to the users.

4.4.7 Application Download

Another feature offered by the system is to act as a repository for some of the services that cannot be run within the server over the patient data. These are the stand-alone applications that require graphical interactions from the user in being able to carry out the necessary operations. In this case, the server provides a mechanism for end-users to be able to download such applications onto their workstations and use it locally like any other locally installed application. There is an important distinction to note. Unlike the services, these applications are completely stand-alone in nature. They do not have access to any patient data. Their mode of operation is completely dictated by the end-user. The client application simply helps the users to download these apps to their computers.

The developers (researchers) do the uploading of such applications. They can make applications in any language they wish and for any platform. These will be presented as downloads to the end user who would be responsible for taking care of installing it on the required platform.

4.5 Interface Design

4.5.1 Single unified user interface

The client application presents the same interface to the different categories of users involved. The main user interface consists of a toolbar, a menu bar, a main window area and a status bar.

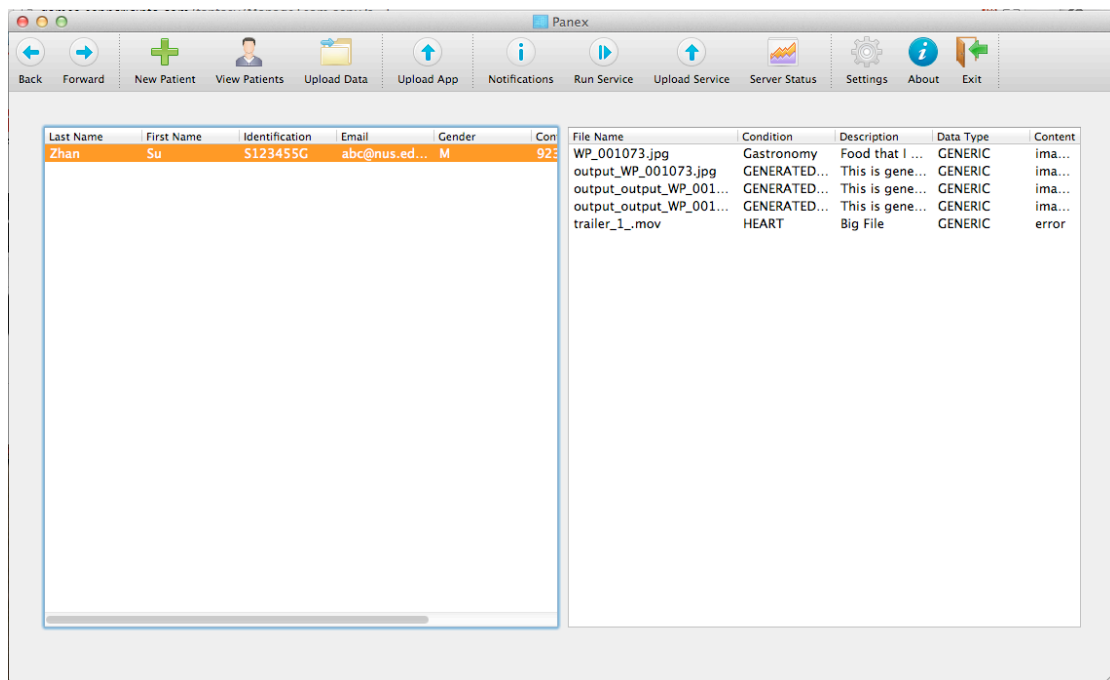


Figure 6 Main Screen of Panex Client App showing patients (left) and their data (right)

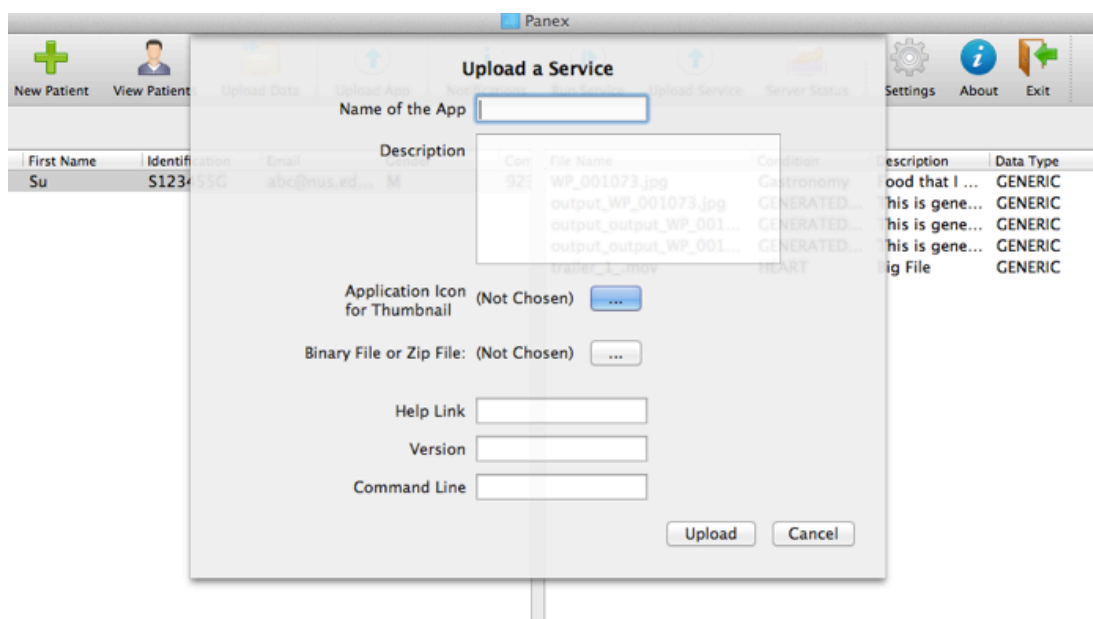


Figure 7 Service Upload Dialog Box

A separate web interface for the monitoring of running and queued jobs is also available. It can be also reached from the client application toolbar.

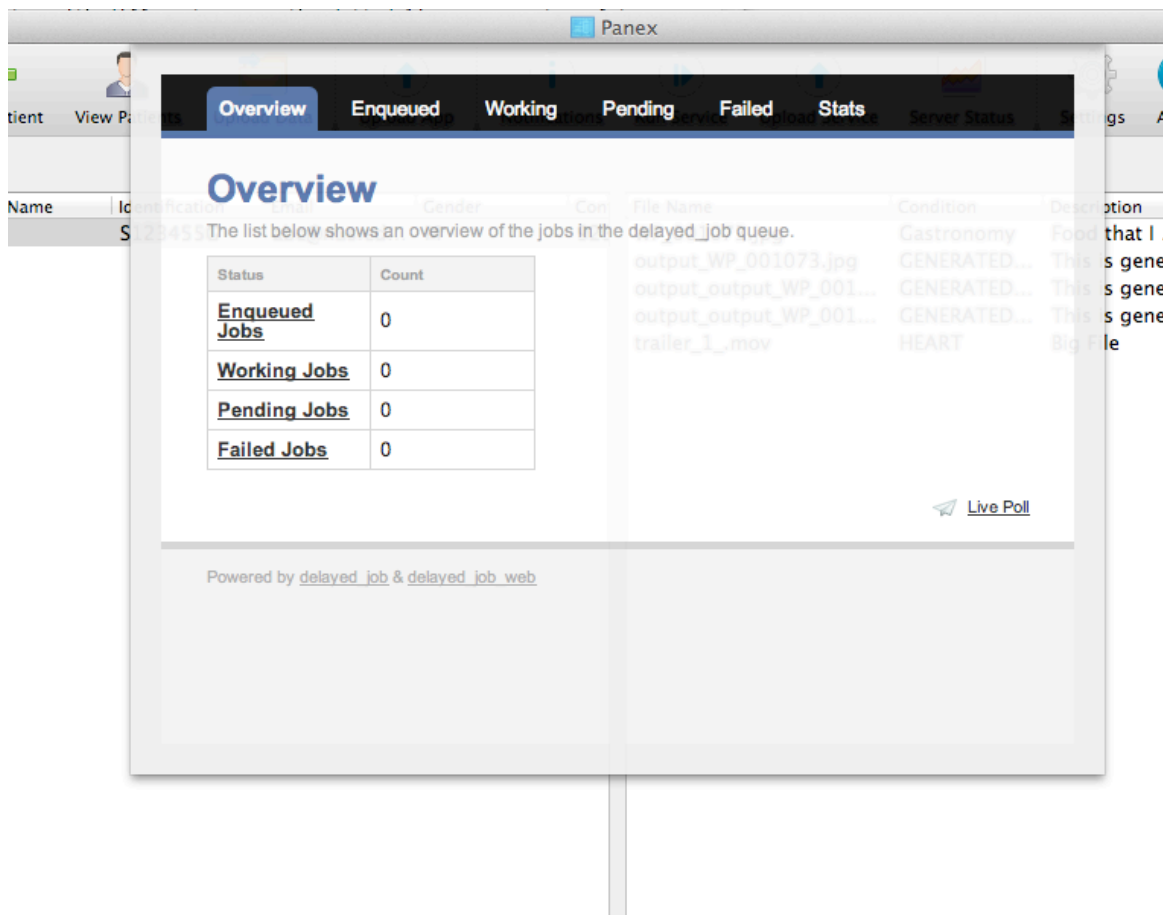


Figure 8 Service Run Status Dialog displaying the server load

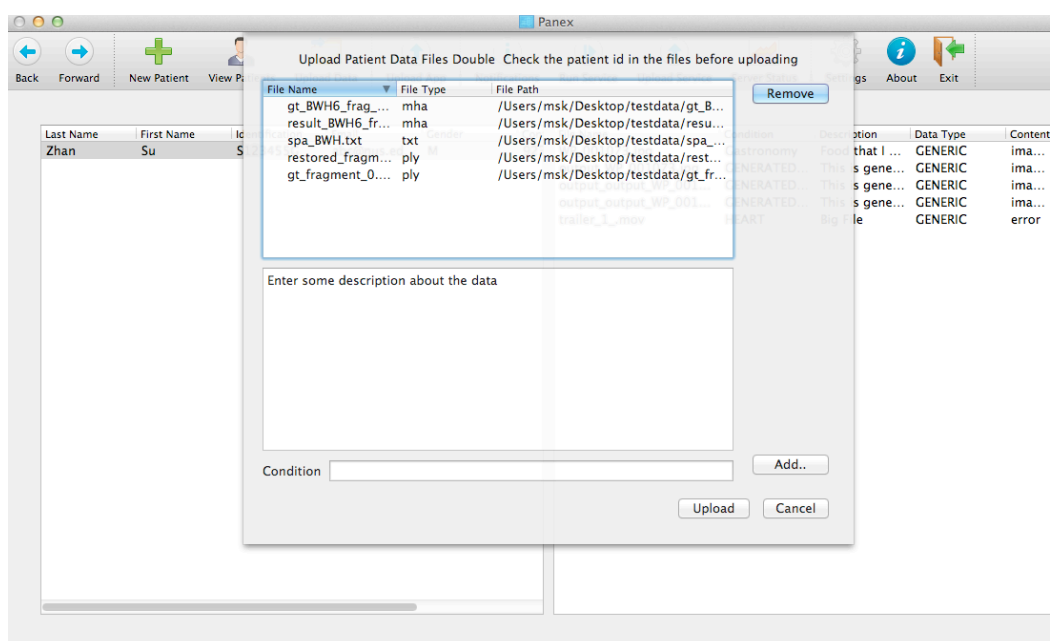


Figure 9 Upload Patient Data Dialog

5 Implementation and Testing

5.1 Overview

This section describes the technical implementation and the testing strategies used in the development phase of the software system. The server component has been developed in Ruby on Rails framework, which is open source and provides huge advantages in terms of available plugins, community support etc. The desktop client application is developed in Qt, a C++ based cross-platform application development framework¹. It is available in two types of licenses. The open source version of Qt can be used with a Lesser GNU Public License (LGPL) applied onto the project. A commercial version is available for commercial applications, which has to be bought from **Digia Inc**, the current owner of the Qt framework. The application therefore is developed in the LGPL license to make use of the open source version of Qt.

5.2 Technology Stack

5.2.1 Desktop Application Framework

Qt appeared to be the choice for reasons as described above. The requirement for the application framework was to be able to interact with DICOM files and be cross platform. Several alternatives were considered such as a web-browser based application, a HTML5 desktop application etc. However, the alternatives lacked support for DICOM files. Qt can make use of robust DCMTK libraries. Another alternative was to use Java, however, with regards to high performance and runtime-efficiency Qt seems to be more suited for medical applications where rendering 3D images might be needed (Gui & Dalheimer, n.d.).

5.2.2 Server Application Framework

Several frameworks were considered, Rails, Java, PHP etc. However, Ruby on Rails seemed to be the most simple and lightweight without requiring heavy configuration, which is the case with Java-based web application frameworks. Initially, PHP was considered and indeed a first prototype was developed in PHP. However, on encountering various low-level thread and process-based manipulation,

¹ Qt was originally produced by TrollTech, which was acquired by Nokia. In 2012, Digia purchased Qt from Nokia. The project uses Qt 4.8 which was the last open source version released by Nokia

Ruby was found to be more effective as compared to PHP because of the immense amounts of gems (plugins) available which extend Ruby's functionality.

5.2.3 Database

Some alternatives considered were MongoDB, Redis and other key-value based NoSQL databases since they scale well over large scale data entries. However MySQL is a mature technology and moreover, medical data tends to be pretty structured and the structure hardly changes over time. Therefore using MySQL Cluster Database made more sense in this case. Other relational databases considered included PostgreSQL, Oracle DB etc. However, one of the core non-functional requirements is to only make use of open source libraries. As such Oracle DB and other non-open source software cannot be used. Between MySQL and PostgreSQL, the former is easier to manage through configuration and a web based interface (phpMyAdmin). For development purposes, SQLite was used, a lightweight database. Also, for the storage of client preferences, a special type of Key, Value based database called QSettings² (provided by Qt) was used. It stores user data and offline file locations.

5.2.4 PACS Server

dm4chee was chosen since it is one of the very few popular open source PACS servers available. Some other open source PACS servers available in the market are PacsOne, softneta etc. However, dm4chee is the only open source version that is in active development. Most of the other including pacsOne or softneta don't have an active open source community and the versions hardly update. They do however have commercial versions available but that would go against the requirements. It is written in Java and completely supports the DICOM standard. For the current implementation version the PACS server simply serves as a backup data source for the DICOM images being uploaded.

5.2.5 Web Server

Nginx is the web server used in the application. A number of other open source alternatives were considered. Apache being the most popular open source web server used in 68% websites around the world. Apache is a process-based server while nginx

² For more information on QSettings refer to <http://qt-project.org/doc/qt-4.8/qsettings.html>

is an event-based web server. However event-based servers are asynchronous in nature and this has a significant advantage. In process-based servers, each new connection requires a new thread, creating which has significant overhead³. As such under heavier loads, process-based servers like Apache can't scale well as compared to Nginx. Furthermore, Nginx is far more efficient at serving cached assets faster and with a lesser memory overhead. For these reasons, Nginx is the preferred choice of the web server.

5.2.6 Hardware

One server belonging to the project supervisor placed in the School of Computing has been used. It is running on Ubuntu Linux 12.04 LTS with VTK, DCMTK and other common runtime libraries installed as maybe required by the services. Linux as an operating system was chosen because it is open source and it has a well-established mechanism for controlling the OS specific operations using C++ and bash commands.

5.3 Server Side Components

Panex server application component is written in Ruby. It makes use of a widely popular rapid application development (RAD) framework called Rails. Among fostering a rapid web development approach, Rails has other benefits⁴:

- A lot less code
- A lot less configuration data
- Easy to learn
- Bringing up basic functionality quickly
- Building out new functionality incrementally
- Integrated Testing

5.3.1 Rails Model View Controller Architecture

In this section, we describe the main design of the backend system. We also consider some of the design patterns used and some design considerations.

³ More Apache versus Nginx benchmarks available at the Linux Journal online: <http://www.linuxjournal.com/article/10108>

⁴ For details about Rails principles and benefits refer to: <http://learnruby.com/about-rails.html>

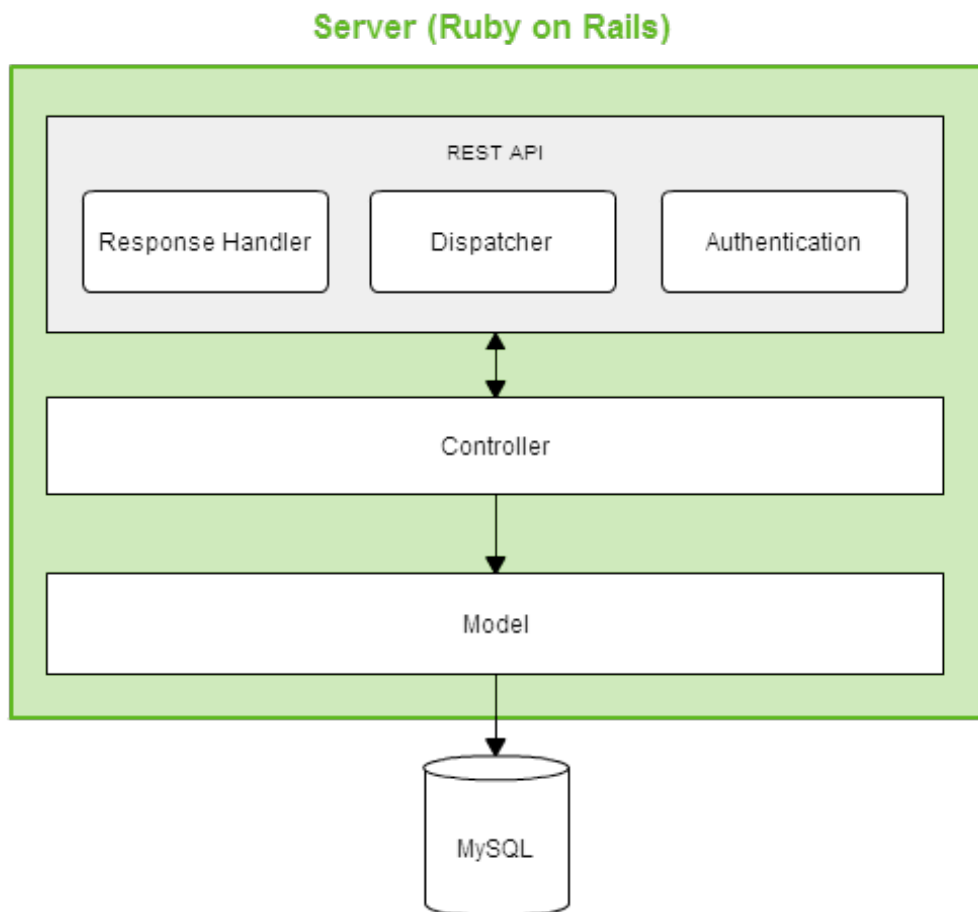


Figure 10 Server side application architecture

Model-View-Controller pattern is applied to our architecture diagram. By using this pattern, it provides a clean separation of concerns, which promote better code organization, extensibility, scalability and code re-use.

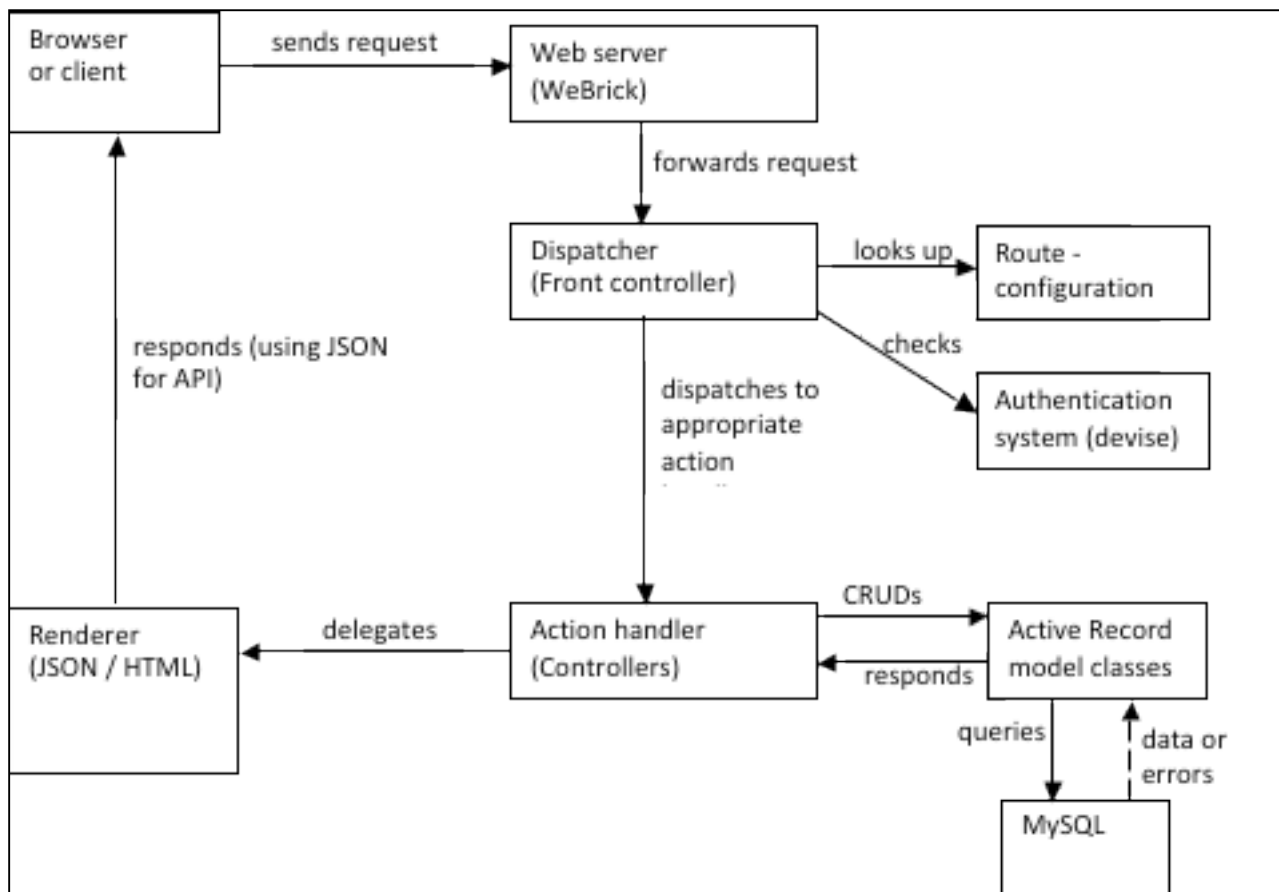


Figure 11 Server Side Request Flow

Figure 11 provides an overview of the flow of web requests within the system. The major components involved in this flow are described below:

Table 1 Major Software Components in the server architecture

Web server (WeBrick)	Ruby on Rails comes with a default web server: WeBrick. However, it can work with several other web servers as well. The role of the web server is to accept requests from client and forwards it to the Dispatcher (Front Controller)
Dispatcher	Dispatcher is the Front Controller of Ruby on Rails framework. It is the controller where all the requests get forwarded. It performs some common tasks to all requests (eg. checking authentication), gets parameters from the request, and looks up the route from configuration file to route the

	request dynamically to the correct action handler.
Controllers	Controller classes are the “C” of the MVC triad. Their role is to respond to handle the requests from clients. They will get data from data model, calling the functions from model to perform appropriate tasks with data (such as creating/reading/updating/deleting - CRUD some resources), and then pass the results to the renderers (the view) to render.
Model classes (Active record)	Model classes are the “M” of the MVC triad. In Ruby on Rails, they are all subclasses of <i>ActiveRecord::Base</i> and follows Active Record pattern: they provide an object mapping to the database, so that data can be easily accessed and manipulated. They automatically emit SQL query to connect and commit changes to the Database when needed.
Renderer	Renderers are the “V” of the MVC triad. Their role is to render the response (HTML page or other type of response) to return to the client. In our case, since we employ front-end rendering technique (client will retrieve data from server and render it using desktop related rendering techniques), so the renderer's role is to render the JSON response to the client.
Database	The role of database is to store all the data. The database used is: MySQL (relational database)

5.3.2 Authentication and Authorization System

For authentication system, we make use of the library Devise for Rails. This library provides several modules, each caters to different functionality for authentication, including: module for registration (checking password complexity, checking if password confirmation matches, checking for email validity), module for sign in/ sign out (creating and deleting the user session on server), and several other modules. It takes advantages of the Front-Controller pattern for RoR (Ruby on Rails), and every request can be check for authentication (whether user has been authenticated or not) before routing to the appropriate action handler. Different action

handlers can also be specified to handles different cases when users are logged-in or not.

For authorization and access control, we make use of the library **CanCan** for Rails. The way authorization is done is that: Our user model defines several different "Roles" for user, and each role will have specified rights to access or edit certain resources (resource refers to different system models).

When an user makes a requests to access some resources, before the resource is accessed, the system will find out who is logged in (using the authentication system), finding the roles of this user (could be more than one), and check if the user has the right to access or make change to the resource. This follows the authorization pattern, which checks for users' authorization for any access request before granting access to the resource.

5.3.3 Job Queuing and Service Management

For job queuing and scheduling we make extensive use of the library **delayed_job** for Rails⁵. The library provides a mechanism to allow different job queues to be defined and background processes to be controlled.

Figure 12 represents a simplified setup of the job execution setup in the server.

⁵ For more information on `delayed_job`, refer to the documentation at: https://github.com/collectiveidea/delayed_job

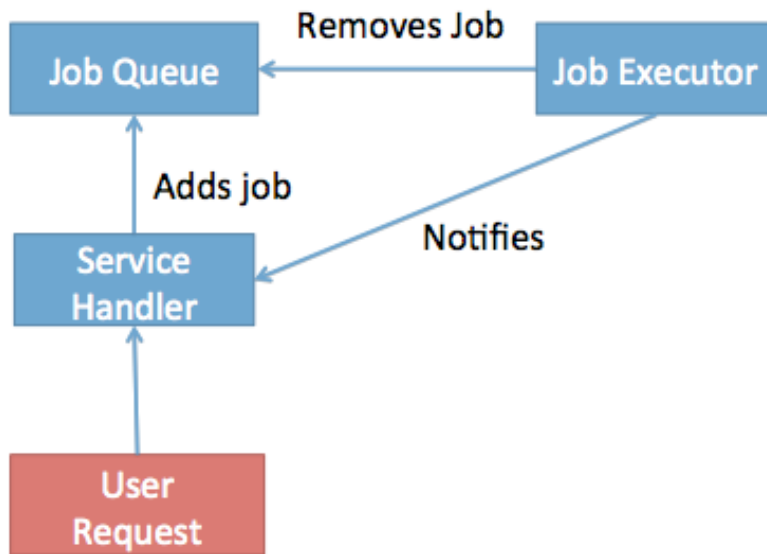


Figure 12 A simplified Job Execution setup

A **job** is a representation of a particular running instance of backend service acting on a particular set of patient data invoked by a particular end-user. Whenever a user requests invocation of a particular type of a service, the service request handler, creates an instance of a job and adds it to the job queue. The job executor is responsible for removing jobs out of the queue and running them on the server one job at a time. Whenever there are multiple jobs, the job executor checks to see if the priority of a particular job is higher than other jobs in the queue. This setup vaguely mimics a CPU scheduler. The Job Executor acts as a CPU scheduler, scheduling jobs and pre-emptively terminating them if required (in the case where they might take up too much running time or memory). The job executor also fires notifications regarding the different phases of the job. The service handler subscribes to particular job stages and executes different functionality required for different stages of the job.

Such a loosely coupled setup allows us to satisfy some interesting requirements. There can be multiple job queues and multiple job executors. They can remove jobs from a single job queue or multiple jobs queues. On multiprocessor core systems, jobs can be run in parallel by having different number of job executors.

When a user requests a service run invocation, the service handler checks to see which input data is required to complete the invocation. It may require patient data or other type of textual, image data. It then creates a temporary directory in the file system and copies the data there. It then copies the service binaries to the same directory. User permissions are setup so that the service cannot access any information outside the directory. If there are any compilation or library linking steps required prior to running the service, those are carried out. After a service has been successfully run, the service invoker is notified of it. On being notified, the service invoker associates the output data produced by the service to the patient so that it can be searchable at a later stage. At the end, the temporary directories are destroyed and cleaned up.

5.3.4 Process Management

The server requires process management modules in order to execute the services in the background. For this, extensive usage of the library `delayed_job` is done. `Delayed::Job` (or DJ) encapsulates the common pattern of asynchronously executing longer tasks in the background. The `delayed_job` library also provides functions to manipulate processes and kill them if needed. It provides an abstraction layer on top of the operating system process management layer. It has various extension points, which can be overridden to execute custom Ruby code. This allows for the necessary customizations to be carried out.

One of the extension points currently overridden is the mechanism to be able to control the location of the service command and its ability to access certain directories within the file system. This is done in order to make sure the services run in a sandbox environment and that the server backend has complete control over the running of these services so that they can be pre-emptively stopped if required.

5.4 Client Side Components

The client application companion for Panex runs on all desktop application platforms (Mac, Linux and Windows). This is due to the fact that it uses Qt, a very popular and fast cross-platform application development platform written in C++. Qt uses standard C++ but makes extensive use of a special code generator called the Meta Object Compiler, or moc together with several other macros to enrich the

language⁶. The user interface was designed in Qt Creator, which is an IDE (Integrated Development Environment) for Qt projects.

5.4.1 Qt Model View Architecture

Qt also uses a flavor of the Model View Controller architecture for the separation of concerns regarding data management and its display. In model/view architecture, the view and the controller objects are combined together. This still separates the concern of how to display the data to the user from the data management itself, but provides a simpler framework while keeping to the philosophy behind MVC in general. All the views within the interfaces are implemented as either Qt Widgets or Dialogs communicating to various APIs exposed by the application server.

The client data models in the application provide an abstraction to the actual data layer. The views make use of the data models to display the necessary information to the users. The data models either store a local copy of the data or get it through network communication with the server. Such separation allows for interesting scenarios. Since the views are shielded away from any network communication by the data models, they can maintain their own local cache of data and provide it to the views when they require it. The application will continue to work even when there is no network link to the server because of the above-described setup. One such offline data management strategy is described in Section 5.4.3.

⁶ For more information on Qt Development please refer to: <http://www.qt-project.org/>

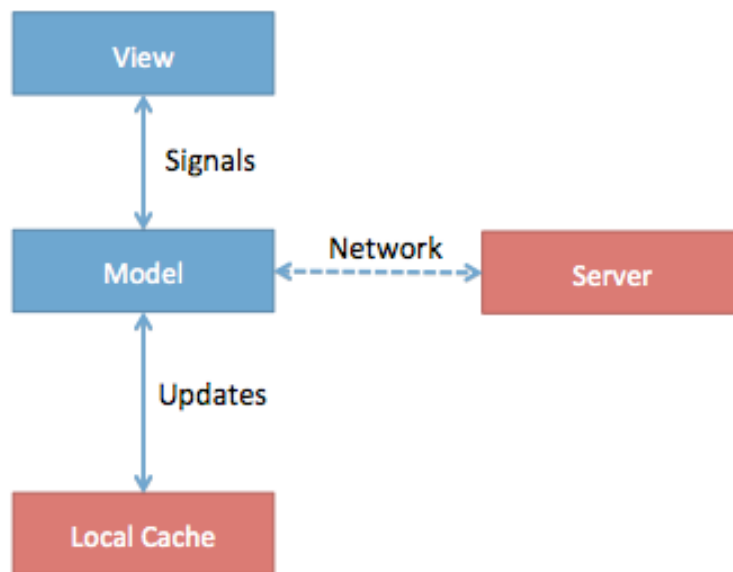


Figure 13 A simplified setup of MVC used in the client side

5.4.2 Data Serialization and Uploading

The client application makes use of the JSON (JavaScript Object Notation) format of data serialization to communicate with the server. A library called **QtJson** has been used to serialize/deserialize the request/response data.

In addition, a client library called **Form Post** was written to facilitate uploading of multiple files in conformance with RFC 1867⁷, which describes in detail the structural overview of a file upload HTTP request.

5.4.3 Offline Data Management

The client application uses an offline storage mechanism in the form of the client's file system as well as making use of **QSettings**. It stores the files downloaded from the server in a special directory locally. When a user wishes to view these files, the software will first query the local directory. Only when the file is not found locally, the software will issue a network request to download the file. This has two advantages:

- The server resources are not strained excessively for downloading of data

⁷ For details on the RFC refer to: <http://www.ietf.org/rfc/rfc1867.txt>

- Even when the server is not accessible or if the application is offline, the client software can still show the data to the user, at least the ones which are already downloaded.

In some aspects, offline data management strategy doubles up as a cache for the downloaded data. Moreover, since the location of the offline data directory can be completely decided by the user,

5.4.4 Multi-threaded Upload

A multi-threaded Qt upload library was written to facilitate faster uploading of DICOM files to the server. The current server can handle up to 100 concurrent HTTP connections, the client library takes advantage of this and splits into a number of threads, each making its own HTTP connection with the server and uploading the DICOM files. Since a DICOM series can consist of hundreds of files, each file being around 50-60 Kbytes, this approach thus is useful in making the upload process finish faster in the case of DICOM files. Since DICOM meta-data carries information about the Series it belongs to, thus each upload can be made independently from the other. The server looks up the Series information in the meta-data and adds it to the appropriate database.

5.5 Testing Strategy

Only Unit Testing, Functional Testing and Integration testing (manually) has been carried out, security testing, performance testing and compatibility testing has been left out as future work.

5.5.1 Unit Testing

The unit testing is used to test model classes, attributes and methods to make sure that the model works as intended. Ruby on Rails comes with a unit-testing library called RSpec, which helps to achieve functional as well as unit testing.

5.5.2 Functional Testing

The functional testing is used mostly to test the controller classes to see whether they could handle the requests, selects the corresponding models and calls the appropriate functions on models. For the Rails application, sample test cases were created and a separate testing database is setup.









<input type="checkbox"/>	 Result linking + Patient Integration	#10
	Opened by creativepsyco 12 days ago (updated 6 days ago)	
<input type="checkbox"/>	 Delayed Job Integration	#9
	Opened by creativepsyco 12 days ago (updated 6 days ago)	
<input type="checkbox"/>	 Service api Integration	#8
	Opened by creativepsyco 18 days ago (updated 18 days ago)	
<input type="checkbox"/>	 Adding helper to generate App Zip File	#7
	Opened by creativepsyco 19 days ago (updated 19 days ago)	
<input type="checkbox"/>	 Stable Uploads now safe to do a pull request	#6
	Opened by creativepsyco 19 days ago (updated 19 days ago)	
<input type="checkbox"/>	 Patient api	#5
	Opened by creativepsyco 25 days ago (updated 25 days ago)	
<input type="checkbox"/>	 Changing default value of role	#4
	Opened by creativepsyco a month ago (updated a month ago)	
<input type="checkbox"/>	 User Signup works now via JSON	#3
	Opened by creativepsyco a month ago (updated a month ago)	

Figure 14 screenshot of the Github Issues for the server code

5.5.3 Integration Testing

Integration testing for the system is not an easy task. This is because we employ both client rendering and back-end server rendering of data (instead of the traditional server-side web). Therefore integration-testing needs to make use of automation tool for testing at the client side as well, and for server-side, only unit tests and functional testing can be done. As such, the client-server integration mechanism was tested manually.

5.6 Continuous Integration and Development Methodology

Another aspect of the implementation is the heavy usage of agile software development methodology, which focuses on shorter development and deployment cycles. For making sure that features were acted upon and taken care of in shorter development cycles and for other Project Management requirements, an online tool called Trello was used. For more code related features, Github Issues was seen a good choice. Version controlling was done via the distributed version control system Git.

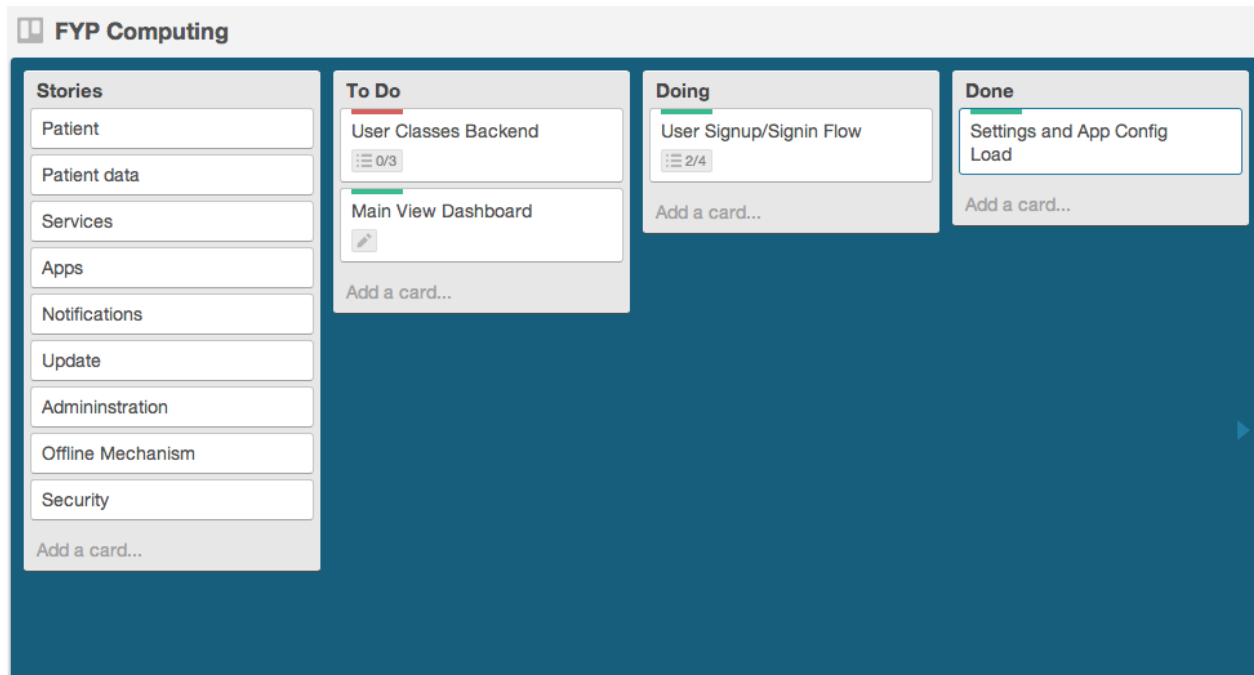


Figure 15 Trello board with the task list divided into three categories

The way the development methodology worked is as follows:

- New features in the application were written in separate branches
- These branches were periodically merged with the master branch once they were felt to be stable
- Once merged, automated tests were carried out via TravisCI to make sure no previously implemented functions were broken.

TravisCI⁸ is an open source hosted continuous integration server available for open source projects to use. It allows automated compilation and testing of software whenever code is pushed to the master branch. This made sure that the version being deployed is bug free.

5.7 Miscellaneous

5.7.1 Documentation

The source code is available via Github repositories online. Also a private wiki is available which explains the source code in detail.

⁸ Please refer to: <https://travis-ci.org/> for more details

5.8 Future Work

5.8.1 Administrator Service

Currently the users can register themselves through the client application. Ideally, users should be approved before they can be allowed to use the Panex Platform. As such, an administrator overseeing this operation would be required. This could be implemented in the future to allow the platform be more secure and accountable.

5.8.2 Testing and Troubleshooting Service

Currently the services uploaded by the users cannot be confirmed to be bug-free and work out of the box on the server. It is possible that some may encounter errors during their run. As such, a troubleshooting service would be required to inform the developers about such errors so that they can rectify the services.

5.8.3 Integration of diverse Data Sources

Only files are currently supported, but the design allows extension and support for different data types as long as the Patient Data interface is adhered to. In the future, perhaps more varying types of data could be supported.

5.8.4 Integration of diverse Service Types

Currently we support services that require themselves to be triggered manually. Also the input files are given as parameters to the service. Perhaps there might be services that may perform some post-processing on patient data as soon as it is uploaded or there might be services, which might work on all of the patient data. Future work could extend the types of services supported and come up with a more uniform mechanism of managing them.

5.8.5 Integration of service execution time

Currently, there is no way to inform the users how much time remains for a service to finish execution, perhaps in the future there may be an interface designed to allow running services to provide their estimated completion times. This is a hard problem since services will be written in different languages and linked to different command line libraries.

5.8.6 Foundation for a telemedicine system

The Panex platform could perhaps be transformed into a telemedicine like system where users can upload their images, reports etc. and let the diagnosis services

analyze them in the background. They can then be delivered notifications via various means such as mobile clients, desktop application client etc.

5.9 Implementation Evaluation

All major milestones scheduled at the beginning of the project have been completed and a beta version of Panex Client Application has been compiled together along with the continuously deployed web service. In the first semester, the project problems were tackled and detailed design issues related to architecture were sorted out. The deliverables at the end of semester 1 included:

- Detailed Design Documents regarding architecture
- Background into medical applications and terms
- Technology overview for desktop and server
- Sample flow mockups

In semester 2, the design documents were refined; implementation of the client and server components was carried out. The features accomplished during this phase are as follows:

- Implementation of desktop client written in Qt
- Web service implementation in Ruby on Rails
- Service Job queue implementation and background process management integration with the web service
- REST API for the client
- Offline Storage feature in the desktop client
- Persistent storage of patient data in database and File System
- Upload and Download handling of Apps and Services

As such, the implementation has satisfied all the requirements goals laid out at the beginning of the project.

6 Conclusion

Through this Final Year Project, a framework for the management and analysis of patient data has been developed. Within a period of 10 months, Panex has evolved from a mere idea to a working prototype. It is hoped that graduate and post-graduate medical application researchers can use the framework to run analysis services on patient data and reduce the turn around time for research work. The system also facilitates sharing, as all the uploaded services are available to the researchers. They can make use of the services uploaded by other fellow researchers and conduct detailed research analysis.

With the platform in relatively good shape, various client applications can be built in the future exploiting the advantages of a robust API interface for running services. A future extension can be the development of a telemedicine service over the current setup, which can be opened up to a large number of users.

In conclusion, this thesis contributes to:

- Analysis of software requirements of a medical research oriented analysis system
- Development of a framework that combines storage services for patient data and batch programs called services
- Development of a stable service execution pipeline, and enabling it to be accessed through a web service
- Development of a web service to expose the functionality offered by the framework
- Development of a prototype cross-platform client application to access the web service

7 References

- Association, N. E. M. (1993). Digital Imaging and Communications in Medicine (DICOM). doi:10.1007/978-3-540-74571-6
- Blanchette, J., & Summerfield, M. (2006). *C++ GUI programming with Qt 4*. Retrieved from <http://books.google.com/books?hl=en&lr=&id=DyGwIpiSoG0C&oi=fnd&pg=PR5&dq=C%2B%2B+GUI+Programming+with+Qt+4&ots=HrHEezzOp-&sig=n4uIJADUORVRsBJpUmZflLcFqjw>
- Eichelberg, M., Riesmeier, J., Wilkens, T., Hewett, A. J., Barth, A., & Jensch, P. (2004). 10 years of DICOM. (R. L. Galloway, Jr., M. J. Yaffe, A. A. Amini, J. M. Fitzpatrick, O. M. Ratib, D. P. Chakraborty, W. F. Walker, et al., Eds.), 5371, 57–68. doi:10.1117/12.534853
- Grauer, D., Cevdanes, L. S. H., & Proffit, W. R. (2009). Working with DICOM craniofacial images. *American journal of orthodontics and dentofacial orthopedics : official publication of the American Association of Orthodontists, its constituent societies, and the American Board of Orthodontics*, 136(3), 460–70. doi:10.1016/j.ajodo.2009.04.016
- Gui, I., & Dalheimer, M. K. (n.d.). Qt vs . Java A Comparison of Qt and Java for Large- Development, 1–12.
- Huang, H. K. (2003). Some historical remarks on picture archiving and communication systems. *Computerized medical imaging and graphics : the official journal of the Computerized Medical Imaging Society*, 27(2-3), 93–9. Retrieved from <http://www.ncbi.nlm.nih.gov/pubmed/12620299>
- McGeary, D. (n.d.). PACS--an overview. *Biomedical instrumentation & technology / Association for the Advancement of Medical Instrumentation*, 43(2), 127–30. doi:10.2345/0899-8205-43.2.127
- Onyebuchi, S. (2011). Medical image storage system for metadata-based retrieval.
- Shiroma, J. (2006). An introduction to DICOM. Retrieved from <http://veterinarymedicine.dvm360.com/vetmed/Medicine/ArticleStandard/Article/detail/509565>
- Suh, E., Warach, S., & Cheung, H. (2002). A Web-based Medical Image Archive System. *Medical Imaging 2002*, 4685, 31–41. Retrieved from http://dcb.cit.nih.gov/publications/download/SPIE_MI_Paper.pdf

- Wong, S. T. C., & Huang, H. K. (1996). ISSUES OF INTEGRATED DESIGN METHODS AND ARCHITECTURAL MEDICAL IMAGE DATA BASE SYSTEMS, *20*(4), 285–299.
- Zhang, J., Sun, J., & Stahl, J. N. (2003). PACS and Web-based image distribution and display. *Computerized medical imaging and graphics : the official journal of the Computerized Medical Imaging Society*, *27*(2-3), 197–206. Retrieved from <http://www.ncbi.nlm.nih.gov/pubmed/12620310>
- Zhang, J., Sun, J., Yang, Y., Chen, X., Meng, L., & Lian, P. (2005). Web-based electronic patient records for collaborative medical applications. *Computerized medical imaging and graphics : the official journal of the Computerized Medical Imaging Society*, *29*(2-3), 115–24. doi:10.1016/j.compmedimag.2004.09.005

8 Appendix A – External Libraries

The table below provides an overview of the external libraries used in the software. All the libraries are available with an open-source license either LGPL, MIT or an Apache GNU License.

Delayed Job	Provides an abstraction for the operating system's process management routines.
Devise	Ruby gem (Plugin) to allow authentication schemes to be implemented. Handles encryption of password and user details.
Can Can	Role based authenticator for Rails.
Ruby Dicom	Implementation of DICOM support in Ruby, reading and modification of meta data inside DICOM files
Ruby Zip	Handling of File Zip utilities implemented in Ruby, useful for transferring data within the file system and also providing as stand-alone downloads
Qt Json	Serialization and De-serialization of data in JSON format. Useful in talking to the web service in standards compliant way

9 Appendix B – Service Developer’s Guide

For developers who are interested in developing backend services for the online system, you must make sure of the following things:

- Panex will accept only services packaged as zip files, don't submit individual libraries etc.
- Panex comes pre-installed with Linux based versions of ITK, VTK and DCMTK. In order to allow your binaries to link to these libraries, you must supply the proper command line flags as the runtime environment will not supply these.
- Make sure your packaged out-of-the-box service runs as scheduled, from command line, otherwise you as a developer will be sent emails about the service failing to run. Keep to the server's configuration regarding the usage of the various libraries.
- If there are scripts that you wish to run before the actual performing of the command, you can do so in a file called '.setup' supplied in your zipped collection. It is a bash script and does your initial setup job. This can include compilation of the program, setting up a local database etc.
- Your program binary will be invoked as specified in the command line field during upload. e.g. a valid command line can be `java myprogram`, this of course assumes you run `javac myprogram.java` in your .setup file.
- In addition you will be provided two more parameters `input_dir` and `output_dir` in the command line, you must use these as the location of input and output files respectively. This is done since there might be some services, which will work on only some files or perhaps on all the files and produce multiple/single results. All these will be linked back to the patient and browsed as such.
- As such you must make sure to run through all the files within the directory yourself. The service is run in a sandbox so you will not have access to other directories other than the current one (in which your program will be placed)