Självständigt arbete på avancerad nivå

Independent degree project – second cycle

M.Sc. Thesis
within Computer Engineering C, course, 30 points

Distributed Electronic Health Record System based on Middleware
Xin Zhang
Distributed Electronic Health Record System based on Middleware

Xin Zhang

Mid Sweden University
The Department of Information Technology and Media (ITM)
Author: Xin Zhang
E-mail address: superzhangxin2009@yahoo.com.cn
Study programme: Computer Engineering MA, Final Project, 30
Examiner: Professor Tingting Zhang, Tingting.Zhang@miun.se
Tutors: Professor Tingting Zhang, Tingting.Zhang@miun.se
Scope: 12109 words inclusive of appendices
Date: 2013-05-19

M.Sc. Thesis
within Computer Engineering C, course, 30 points

Distributed Electronic Health Record System
based on Middleware

Xin Zhang
Abstract

With the rapid development of information technology, traditional health care is evolving to a more digital and electronic stage. The Electronic Health Record (EHR) is a patient’s basic information and health care related information conforming to the standard. It is not only able to provide useful information to medical workers, but also exchange resources with other information systems. However, with the growing complexity of electronic health record data sources, it becomes a significant challenge to set up a structure which allows different types of data sharing and exchanging in multi-platform applications. It is even more important to discover a method to support a large number of users from different application platforms in relation to sharing and exchanging data at the same time.

In this paper, we have proposed a distributed electronic health record system based on middleware to address the problem. Both permanent and real-time data should pass through the middleware provided by the system, and will be transformed into standard format for storage. Multi-thread and distributed server group design will allow the system to be more flexible and scalable, and will be able to provide services to users concurrently.

The system creates a standard data format for data transferring and storage. All raw data collected from different kinds of sensor systems will be formatted by means of an application programming interface (API) or software development kit (SDK) system provided before uploading to the system. Encryption methods are also implemented to ensure data security and privacy protection.

Keywords: electronic health record, distributed system, middleware, privacy protection.
Acknowledgements

First of all, I would like to thank Professor Tingting Zhang. As my supervisor in Mid Sweden University, she has given me the opportunity to take part in this project. She also patiently gave me suggestions about my project work and thesis writing.

Secondly, I want to express my sincere thanks to Professor Jiajin Le. As my supervisor in Donghua University, he has always taken care of me in both research and daily life.

Finally I would like to thank everyone who also participated in this project. The progress of this electronic health record system is the result of the great efforts of everyone involved and I want to express my sincere gratitude to all those who have supported me and helped me.
# Table of Contents

Abstract .......................................................................................................................... iii

Acknowledgements ......................................................................................................... iv

Table of Contents ........................................................................................................... v

Terminology ..................................................................................................................... vii

Abbreviations .................................................................................................................. vii

1 Introduction .................................................................................................................... 8
  1.1 Background and problem motivation ................................................................. 9
  1.2 Overall aim ............................................................................................................. 9
  1.3 Scope ..................................................................................................................... 10
  1.4 Ethical issues ......................................................................................................... 10
  1.5 Concrete and verifiable goals ............................................................................. 11
  1.6 Outline ................................................................................................................ 11

2 Related work ................................................................................................................. 13

3 Methodology ................................................................................................................ 16
  3.1 Modularization ...................................................................................................... 16
  3.2 Client Simulation Method ................................................................................. 17
  3.3 Middleware API and SDK solution ................................................................. 18
  3.4 OOP .................................................................................................................... 19
  3.5 Distributed Server Structure ........................................................................... 20
  3.6 Multi-Thread Server ......................................................................................... 21
  3.7 AES encryption .................................................................................................. 21

4 Design and Implementation .......................................................................................... 23
  4.1 Distributed Server Group .................................................................................. 23
  4.1.1 Main Functions ................................................................................................. 23
  4.1.2 General Structure ............................................................................................ 25
  4.1.3 Interface Server ............................................................................................... 26
  4.1.4 Dispatch Server ............................................................................................... 28
  4.1.5 Application Server .......................................................................................... 29
  4.1.6 Storage Server ................................................................................................ 32
  4.1.7 Load balance algorithm and implementation .............................................. 34
  4.1.8 Crash detection mechanism ......................................................................... 36
  4.2 Middleware API and SDK .................................................................................. 38
  4.2.1 Architecture .................................................................................................... 38
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.2.2</td>
<td>Main Function</td>
<td>40</td>
</tr>
<tr>
<td>4.2.3</td>
<td>Data Serialization and Transmission</td>
<td>42</td>
</tr>
<tr>
<td>4.2.4</td>
<td>Data Encryption and Decryption</td>
<td>42</td>
</tr>
<tr>
<td>4.3</td>
<td>Demo Client Application</td>
<td>43</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Android Client Application</td>
<td>43</td>
</tr>
<tr>
<td>4.3.2</td>
<td>PC Client Application</td>
<td>44</td>
</tr>
<tr>
<td>5</td>
<td>Evaluation</td>
<td>46</td>
</tr>
<tr>
<td>5.1</td>
<td>Test environment</td>
<td>46</td>
</tr>
<tr>
<td>5.2</td>
<td>Performance and capacity</td>
<td>47</td>
</tr>
<tr>
<td>5.3</td>
<td>Fault tolerance</td>
<td>53</td>
</tr>
<tr>
<td>5.4</td>
<td>Efficiency</td>
<td>54</td>
</tr>
<tr>
<td>5.5</td>
<td>Openness</td>
<td>55</td>
</tr>
<tr>
<td>5.6</td>
<td>Security and Privacy protection</td>
<td>55</td>
</tr>
<tr>
<td>5.7</td>
<td>Scalability</td>
<td>56</td>
</tr>
<tr>
<td>6</td>
<td>Conclusions</td>
<td>58</td>
</tr>
<tr>
<td>7</td>
<td>References</td>
<td>60</td>
</tr>
</tbody>
</table>
Terminology

Abbreviations

EHR  Electronic Health Record
AES  Advanced Encryption Standard
ECG  Electrocardiography
1 Introduction

With the rapid development of information technology, traditional health care is evolving into a more digital and electronic stage. The Electronic Health Record (EHR) is a patient’s basic information and health care related information conforming to the standard. It is not only able to provide useful information to medical workers, but also exchange resources with other information systems [1].

The idea of EHR has been created and developed with the transformation of medical and health modes. Original disease-centred medical and health modes have brought great pressure both in relation to nations and companies due to the rapidly increasing costs. EHR proposes a health-centred medical service pattern. The aim is to achieve the goal of disease prevention and health promotion by adapting to full population coverage and life coverage. The aim is to encourage people into acquiring a healthy way of life, in order to achieve the goal of active health, self management and personal health management by introducing an online personal health data inquiry. EHR can also support diagnoses by providing a patient’s full health history. Finally, EHR will establish information sharing between various health and medical systems and, as a result, medical costs should gradually be reduced and wastage of medical resources should also be prevented.

Currently, the development of the EHR system is facing many challenges. Considering the situation and difficulties that EHR systems are facing both home and abroad, the major factor appears to be in relation to the technical obstacles which are slowing down the development. This is despite the fact that it is a governmental policy issue. From the technical point of view, it is a huge project to integrate mass information from various health and medical systems and different databases, and to store them online for real time inquiry and resource sharing. Codes and standards do not exist at a national level as the health departments have not included electronic health system development into their future plans. In addition, the currently running EHR systems are built by various software companies, which leads to standard problems and makes it difficult for these system to connect and share resources.
1.1 Background and problem motivation

Our research group has been working on an E-health project for several years. The project’s goal is to set up an electronic health record system for people from all ages. Users can upload various health related data collected from sensor systems to the electronic health record system, and obtain both data analysis and health advice from the system. The system is also open to research groups who are able to carry out data mining activities with the data collected by the system.

Giving consideration to potential future demands, electronic health record may extend their function and scale to support an increased number of different kinds of data source and different kinds of applications from multiple platforms. The system’s structure should be sufficiently flexible to deal with various types of upcoming data. All parts of the system should be modulated to make deployment and upgrade easier.

Additionally, in order to give consideration to scalability, an extendable system structure should be created to replace the original structure. The staff deployed by the system can decide how many servers in each level are required based on the potential number of users of the system. The system structure should be distributed and easy to deploy and maintain. Different system modules should be interconnected with standard middleware, so partial updates or modifications are easily able to be carried out.

Special attention is required in relation to data security and privacy protection; users should be allowed to decide their own privacy policy.

The system provides an application programming interface (API) and software development kit (SDK) for 3rd party data sources and applications to gain access to the system resource to ensure the openness of the system. Different levels of user access rights should be implemented in the middleware.

1.2 Overall aim

This research work is mainly aimed at implementing and establishing a model of distributed electronic health record system based on middleware. The system should achieve the following goals:

(1) Be able to adjust the system scale according to potential number of users.
(2) Be able to provide an application programming interface (API) and software development kit (SDK) to 3rd party users and developers. The API and SDK should be presented with the concept of middleware. Middleware should make both the internal data structure and the transfer process transparent to users and developers.

(3) Be able to adapt a partial upgrade without interfering with other parts of the system.

(1) Be able to support both static health data and streaming health data in transfer and storage.

1.3 Scope
This research work concentrates on the implementation of the distributed electronic health record system and middleware design. The system will support several kinds of data as a demonstration, and leave a standard interface for upcoming data. A small scale test server group will be set up to test the overall performance and system workload.

Demonstration clients for multiple platforms will be developed to test the performance of the middleware. For testing and evaluation purposes, only the basic functions will be implemented instead of all functions.

The final outcome of the research will be a functioning laboratory prototype of the system, including all the necessary modules with basic functions. Middleware will also support basic authentication, upload and download functions.

1.4 Ethical issues
The research we conducted on electronic health record system will enable more informed decision making and enhanced quality of health care. It will save lives through remote consultation and create more efficient, convenient health care. With the above mentioned benefits, our research will surely make contributions to the health aspects of the general population.

To prevent the misuse of the system we have created and to firmly protect the privacy of users, we set up strict user right rules and anonyemities for the user data to ensure the identity of users will not enter the public domain. We also encrypted crucial users by means of an AES encryption method, which will also ensure the security of the user health related data.
1.5 Concrete and verifiable goals

(1) Do research on middleware, application programming interface (API) and software development kit (SDK). Summarize the concept and implementation of the technology. Learn the result of former researches. Analyze the advantage and disadvantage for all possible solutions. Decide what have been done and what is needed to be done.

(2) Do research on distributed server group structure, multi-thread server programming and encryption methods. Learning existing secured distributed system design, consider possible modification and improvement for this project.

(3) Design server group structure; decide ways of transferring and storage. Figure out function for each layer of the server group and data transfer protocol between them.

(4) Implement the server group and set up a testing server group for evaluation.

(5) Design and implement the middleware for 3rd party users and developers.

(6) Test and evaluate the middleware with demo clients from multiple platforms.

1.6 Outline

The structure of this thesis is described below:

Chapter 1 introduced the background and motivation of this research work, and both overall goals and concrete goals.

Chapter 2 introduced several related electronic health record system products and technologies.

Chapter 3 introduced the chosen methodology and methods to accomplish the research work. The described methodology and methods include modularization, middleware API and SDK solution, distributed server group structure, multi-thread server and AES encryption method.

Chapter 4 first briefly introduced the general structure and function of distributed electronic health record system. Then it introduced the
design and implementation of system in detail. Finally, it introduced the structure of middleware API and SDK, and its usage in demo clients.

Chapter 5 gave a detailed evaluation of the system in performance, easy-to-deploy and easy-to-upgrade, flexibility, scalability and security issues.

Chapter 6 summarized the whole research work and pointed out the future directions.
2 Related work

Jun Tang et al. [16] conducted a survey on wireless sensor networks for a home healthcare monitoring application. The paper mainly focuses on two prototypes relating to home healthcare monitoring application: daily activities monitoring application and medical status monitoring application. The paper presented the requirement analysis which starts from the causes of chronic diseases. The paper also discussed the challenges for current home healthcare monitoring applications. This paper provided guidance for the design and implementation of a home-based health data gathering system for an electronic health record system.

Xuchen Lu et al. [17] proposed a heterogeneous data source middleware. The middleware uses an abstract class to shield the differences among heterogeneous data sources and creates a data source service wrapper for each. Both permanent data info and real-time data flows should pass through the middleware during communication of data sources and application. In addition, by implementing a corresponding data source service wrapper, it is very convenient to add a new data source. At the same time, the middleware uses XML to accomplish data mapping and transmission, so as to solve the incompatibility of data sources schema and ensure platform independence. Further development has been conducted based on the prototype she proposed to form the functional middleware discussed in this paper.

Corey M. Angst and Ritu Agarwal have proposed an electronic health record system with the adoption of an elaboration likelihood model and individual persuasion. They have conducted research on users’ concerns in relation to privacy protection and analyse the appearance of the system to satisfy these basic concerns. They have conducted a survey to evaluate the possible acceptance rate of their proposed system model, the result indicates that individual’s CFIP interacts with issue involvement and this has an effect on their attitude towards EHR. Although the research did not provide an overall structure of the system, it did offer a very clear image with regards to how the privacy control of the system should appear. [12]

Bernd Blobel from the Faculty of Medicine, Institute of Biometrics and Medical informatics, Otto-von-Guericke University, Germany proposed an authorisation and access control method for the electronic health
record system. This method is based on EHR architecture and security standards CEN ENV 13606. The basic idea is the separation of structural roles and enabling specific acts using UML and XML. Integration of organisational, functional, informational and technological components follows specific rules. He also presented the analysis, design, implementation and maintenance of the system. However this research has not given a great deal of consideration with regards to scalability and flexibility issues. [13]

Researchers from the Departments of Biomedical Informatics and Pediatrics, Vanderbilt University, Nashville, U.S. have proposed a method of importing direct entries of clinical data into electronic health record systems. They defined a systematic collection of health care phrases that support clinicians’ entry of patient-related information into the system. According to their evaluation results, their means of modelling and generalizing the health data proved to be very effective. However, this research has not considered the problem of the upcoming variety of types of data and connection with 3rd party applications and data sources. [14]

Researchers from Oregon Health & Science University, Portland, U.S. provided a discussion with regards to the definitions, benefits and strategies of the electronic health record system. They also considered how an EHR system might function to benefit individuals, caregivers and health care providers. Although this research did not present the possible structure of the whole system, it served as a theory foundation for future EHR development. [16]

Researchers from Partners HealthCare System, Information Systems, U.S. presented a time consumption model for an electronic health record system. According to their result, EHR system’s response time must be under a given threshold in order to meet the original working efficiency of physicians, otherwise the adoption of the system will meet with strong barriers. This research offered a time evaluation model for evaluating the performance of EHR system. [17]

D. Kalra from University College London, United Kingdom provided an overview of the initiatives that are proceeding internationally to develop standards for the data exchange between EHR systems. The research analyzed the key factors for data exchange standardization and claimed that there is a strong prospect that a generic EHR standard can be
agreed at a European level. However this research did not provide any specific details regarding data format and structure. [21]

D.A. Ludwick from Department of Mechanical Engineering, University of Alberta, Canada presented a systematic overview of current EHR systems being used around the world based on the available literature. The research shows that the system’s graphical user interface design quality, feature functionality, project management do affect the implementation outcomes. Implementation also relied on factors such as privacy, patient safety, patient relations, staff anxiety, efficiency and liability. The article made clear comparisons between different EHR systems and offered suggestions in relation to building the most suitable EHR systems for particular occasions. [22]
3 Methodology

The first step and the key for project development is using proper methodologies and choosing appropriate corresponding methods. This chapter will introduce the methodology and concrete methods used to accomplish the research work.

To break apart the complicated problem of distributed electronic health record system, modularization is required to be applied to the whole system. To support multi-platform applications and data sources, middleware API and SDK solution are proposed for 3rd party application integration to the system. As modularization is defined as the key part of object oriented programming, the programming for the whole system should also adopt object oriented thinking and utilize OOP. To increase the scalability and flexibility of the system, a distributed server structure is used. To ensure high performance for each server, multi-thread programming and mechanism are implemented. Since data security and privacy protection form an important part of the system implementation, an AES encryption method is used for data encryption.

3.1 Modularization

Modularization is technique in software design and development. The main concept of modularization is to divide software system into various parts based on data links, related functions and implementation considerations [2].

Programming with modularization think is also called “top-down design” and “stepwise refinement”. Its goal is to divide a software system into independent, interchangeable modules. In this way, each module is independent in logic and function, and only contains everything necessary to undertake one part of the desired functionality [2].

The result of modularization is simplification of complex problem. The big problem is divided into small and easy-to-solve problems. The big problem can eventually be solved after every small problem is solved. Modularization has a great advantage in group developing work. Several programmers can work on the same project but different parts at the same time. The modules of the program are easier to modify and update [2].
To realize the concept of modularization, our distributed electronic health record system has been divided into different parts and layers, and different modules in each layer. Also, the middleware API and SDK solution is based on modularization consideration. Further details of the system structure will be mentioned in next chapter.

### 3.2 Client Simulation Method

Client Simulation method is used to test the overall performance and capacity of the distributed server group. Client Simulator will generate random number platform-unrelated clients with multiple threads and generate concurrent requests. Parameters can be changed to simulate different situations such as free hour, busy hour and rush hour.

![Client Simulator project structure](image)

**Figure 3.1: Structure of Client Simulator project**

As shown in Figure 3.1, Client Simulator included Middleware API as a support class. Simulate data generators are also created to generate different types of random health data. The server response time for each request is recorded to measure the average response time in each situation.

To test and evaluate the overall scalability and performance of distributed electronic health record system, key features such as average system response time, transaction per second, CPU usage, memory usage, fault tolerance and failure rate.
We set up 3 scenarios for testing and evaluation. We assumed that system will provide health record storage and analysis service for 40000 residents in Sundsvall city. 20% of the population is over age of 65 [14]. Users gather and upload health related data twice a day. Users over age of 65 double the gathering and uploading rate to 4 times a day to provide intensive care. Every data gathering will generate 4 common data entries which represent blood pressure, body temperature, pulse rate and respiration rate. Every data gathering will also generate one stream data which represents 10 seconds of Electrocardiography (ECG) data.

As shown in Fig.3.2, user requests subjects to double peaks normal distribution in our test case. We assumed users will gather data around 10 am and 8 pm to ensure data reflexes accurate health status. 80% of daily transactions will happen from 8 am to 12 am and 6 pm to 10 pm. After calculation, we defined idle hours scenario of system has 1 to 3 concurrent users, busy hours scenario of system has 4 to 7 concurrent users and rush hours scenario has 8 to 10 concurrent users.

3.3 Middleware API and SDK solution
Middleware is a kind of computer program that connects different software systems together. Middleware is the key enabling technology for component-based software development and system modularization. Middleware is also responsible for realizing the interoperability between different components in the system [3].
Middleware is widely used in different types of projects. It provides data services, supports application functions and integrates components. The advantage of using middleware is that a relatively stable high-level application environment is created. The application interface remains the same regardless of how the server side and hardware change. Middleware can assist a software development group to significantly reduce investment in maintenance and development [3].

Application programming interface (API) is a protocol for the connection of different parts of the application. Since the scale of software has increased significantly during recent years, API is often used between system parts to make system maintenance easier. In addition, when a system is open to 3rd party application for integration, API is used to set the standard for data exchange protocol. The implementation of functions in API is transparent to users; users can use the function without having any knowledge of its mechanism. API has a similar concept to that of middleware, but API is more specific in its function and is platform based [4].

A software development kit (SDK) is a set of software development tools that allow 3rd party developers to create an application under the guidance of the system. SDK is a package of software, framework, demonstration codes and a technical document. API is often part of the SDK [5].

In our research project, we used middleware API between different parts of the system to increase the efficiency of the development. With middleware, we managed to set up an elegant framework which is loosely coupled and which has a high cohesion. We also provide both API and SDK for 3rd party users and developers in order to increase the openness of the system. A 3rd party can share data with the whole system easily and safely. Details of the implementation of the middleware API and SDK solution will be mentioned in the next chapter.

3.4 OOP

Object Oriented Programming is kind of programming framework. The basic concept of OOP is that a program is made up with several objects. Each object can carry out one kind of task. OOP is able to achieve 3 fundamental objectives of software engineering, which are reusability, flexibility and extensibility. In order to be put up to a whole system, each object must be able to process and exchange data with other objects [6].
In OOP concept, object is the foundation of modularization. An object is the basic unit which is made up with data and functions together. Data’s interaction with functions is also included in the object. The reason why OOP is distinguished from traditional process-oriented programming is that it includes features such as inheritance, data abstraction and encapsulation [6].

OOP makes programming closer to the way how human look at things naturally. OOP gives every object unique data members and functions, which makes programming more humane. To adopt object oriented programming framework, Java is used as programming language for the distributed electronic health record system. Java has platform independence property and support module concept as packages. Java classes can be exported to jar files and can be used by applications from other platforms [6].

### 3.5 Distributed Server Structure

Distributed server structure is a server structure which data and program is not located on single physical server, but spread to multiple servers. Distributed server structure has advantage in allocating and optimizing tasks in the whole system. Distributed server structure has overcome the bottleneck of traditional centralized system. It solved the problem of strained resource of central server and limited responses. It’s a great progress in geographical information system technology [7].

Distributed server structure is location transparent to users. Users feel the same when using centralized single server structure. But distributed server structure has much better scalability and flexibility than the traditional centralized server structure. The amount of server needed can be calculated by system load formula specific for the system.

Distributed server structure can work more stable than traditional structure. The system can automatically detect the failure part and redirect service to other server. Once the failure part is fixed, this part can be back online under the control of the system. Also distributed server structure can adopt hot-update which means upgrade the whole system without shut down the service [7].

In our distributed electronic health record system, server structure has 4 layers according to the main concept of distributed server structure and practical need of the system. Details will be mentioned in the next chapter.
3.6 Multi-Thread Server

Every program running in the operation system is a process. Each process consists of several threads. Process can be the dynamic execution of entire program or part of the program. Thread is a collection of commands. It can be executed independently. It is responsible for carry out multiple tasks in a single program [8].

Thread is a single controlling sequence in the program. Multi-thread is the mechanism of carrying out multiple tasks in single program using several threads [8].

An application implemented multi-thread mechanism can use system resource more efficiently. Its major advantage is making the best use of CPU free time slice. As the result, user’s request can be answered in minimum amount of time. The overall performance of the process can be gradually increased. Multi-thread mechanism can also increase the flexibility of application. All thread in the same process is sharing the same part of the physical memory. There is another advantage because no special data exchange mechanism is needed because threads are sharing data storage. So the coordination, exchange and distribution of resource among threads are easier to solve [8].

Multi-thread server is server application using multi-thread mechanism. It has better overall performance than traditional single thread server. It can make the best use of system resource and support multiple user requests nearly simultaneously.

In our distributed electronic health record system, every server application has implemented the multi-thread mechanism to gradually increase the overall performance and capacity of the system. Details will be mentioned in the next chapter.

3.7 AES encryption

Advanced Encryption Standard (AES) is also called Rijndael encryption method; it was published by the U.S. National Institute of Standards and Technology (NIST) in 2001. AES has being widely used as method for electronic data encryption [9].

AES replaced the Data Encryption Standard (DES), which was published in 1977. AES is a symmetric-key encryption algorithm; same key is used during encryption and decryption process. AES has high performance in
both software and hardware because it is based on substitution-permutation network design principle [9].

AES encryption process is operated on a 4×4 bytes matrix. 4 steps of process circulate during the encryption processes which are AddRound-Key, SubBytes, ShiftRows and MixColumns. The number of cycles is depend on the length of the key. For example: 10 cycles is needed for 128 bit keys, 12 cycles is needed for 192 bit keys and 14 cycles is needed for 256 bit keys [9].

AES encryption method is used to encrypt user account information and health record data in our distributed electronic health record system. With the high security grade of AES, user’s data will be safely stored without concern of be cracked. Details will be mentioned in the next chapter.
4 Design and Implementation

This chapter will give a detailed look into the distributed server group and its server components. The structure and communication protocol of server components will be presented. Then the design and implementation of middleware API and SDK will be described in detail. At last, a series of demo clients using middleware API and SDK will be presented for testing purposes.

4.1 Distributed Server Group

The core of distributed electronic health record system is the distributed server group. Distributed server group is responsible for all the data communication and authentication. The design of distributed server group mainly focused on high performance, high stability, high flexibility, high scalability and low deployment cost. All applications from different platforms get services from distributed server group through middle API and SDK, and whole communication process is transparent to users.

4.1.1 Main Functions

As shown in Figure 4.1, the distributed server group has 4 parts of main functions. This section will introduce each function of distributed server group.
The user setting function is related with user information and privacy settings. Users need register and login process to get access to the system resource. System creates a unique profile for each user and assigns specific user right to ensure user only can get access to specific data. User can decide which part of the data should upload to the system for health analysis through privacy setting. The user profile and privacy setting is stored in the distributed server group, so user will always adopt same privacy setting no matter which device he is using.

The data service function is responsible for receive health record data from users and store data. Data service will also provide data to user when requested. Distributed server group can currently handle 2 types of data. The first type is common string health record data. Such as blood pressure, heart rate and body temperature. This type of data has limited length and can be recorded with type name string and value string. The second type is stream health data. Such as Electrocardiography (ECG) wave data and Electroencephalography (EEG) wave data. This type of data has various lengths and consists of huge amount of atom data entries. The storage of this type of data needs special consideration in performance and size. Distributed server group will also provide health data analysis and gives out health advices when user got irregular data which indicates illness and unhealthy body condition.

System control function is mainly responsible for system running status adjustment. System always dynamically running a service provider list, in which stores the information of fully functioning servers. Distributed server group can automatically detect any crashed server and remove them from service provider list for reboot and maintenance. Server group can also automatically detect any new coming server and add them into service provider list. To ensure the overall performance and balance in the system, a load balance mechanism is implemented in the distributed server group. System will always try to average the workload among the servers and tried to avoid the situation of unbalanced workload distribution.

User management function is designed for system administrators. System administrators can adjust users’ role in the system by giving them different user rights. Administrators can also manage sensors’ ownership to connect data in storage with users. Administrator can update users’ contact information as well.
4.1.2 General Structure

As shown in Figure 4.2, the distributed server group consists 4 layers. The number of servers in each layer can be adjusted according to the potential user amount. The formula of calculating server amount in each layer will be presented in the next chapter.

![Figure 4.2: General structure of distributed server group](image)

The 4 layers in the distributed server group are interface layer, dispatch layer, application layer and storage layer. Interface layer is the entrance portal for user to get access to system resource. Its main function is to provide dispatch server’s connection for users. Interface server also will check dispatch server’s status according to schedule, and detect errors or failures in the dispatch layer. Interface server will remove any crashed dispatch server from service provider list to ensure the functionality of the dispatch layer. Details about the implementation of interface layer will be mentioned in 4.1.3.

The dispatch layer is responsible for set up connection between clients and application server. Dispatch layer check status of application layer according to schedule and automatically balance the user amount between different application servers according to current linked user amount of each application server. Details about the implementation of dispatch layer will be mentioned in 4.1.4.
Application layer is responsible for carrying out business logic transactions for client requests. Application layer requests data from storage layer according to the different needs of client, and then format the data before send back to clients. Application server reports its status and current workload to dispatch layer according to schedule. Details about the implementation of application layer will be mentioned in 4.1.5.

Storage layer is responsible for the storage of both user profiles and user health record. Storage layer is transparent to client and only answer to the request of application layer. The key value of all the data in the storage layer is encrypted using AES encryption method. The connection between storage layer and application layer is achieved by middleware. This inter-layer middleware defines the standard data format and data exchange methods. Details about the implementation of storage layer will be mentioned in 4.1.6.

### 4.1.3 Interface Server

Interface server serves as the entrance of client for using system resource. Interface server has static public IP address and it is the only part in the distributed server group which is visible to the clients. Interface server implemented the multi-thread mechanism thus can handle multiple requests at the same time. The application of interface server is coded with Java.

![Figure 4.3: Structure of Interface server application](image-url)

As shown in Figure 4.3, the application for interface server consists 3 layers. The first layer is the event generating layer which is responsible for accept the request and deliver them into the 2\(^\text{nd}\) layer. The second
layer is the event handler layer which is responsible for carry out the business logic transaction according to the event generated in the first layer. The 3rd layer is the storage layer for data and system log.

When a new client socket connection is established, multi-thread socket connection pool will create an independent thread for this socket connection. This mechanism will ensure each socket connection doesn’t interfere with any other connections and tasks can be carried out simultaneously. After the connection is closed, system resources will be collected and get ready for the next connection.

Scheduled check thread sends check request to every dispatch server recorded in the interface server database. The frequency of checking can be adjusted according to the actual need.

When a client connects to the interface server and request for dispatch server IP address, the Redirect Dispatch IP service will first scan the memory if this client has delivered the same request a certain amount of time ago. If it has, the same dispatch server IP address will be send to the client if the dispatch server is currently working. Otherwise, the Redirect Dispatch IP service will chose the dispatch server with least workload, send the IP address to client and record this transaction to the memory. We call this record mechanism “soft-state server”.

Soft-state server keeps temporary data of clients to improve performance. Distributed servers don’t exchange temporary data with each other. Soft-state server combines the advantage of state server and stateless server, and also gets rid of their weakness.

When a dispatch server from dispatch layer sends a status report to interface server, the Dispatch Report Handler will deal with the status report. The dispatch server’s information in the database will be updated, and if this dispatch server is new to the interface server, a new record will be created. The current workload of the dispatch server will also be included in the status report. So interface server will record the workload of each dispatch server and make auto-balance according to them.

Every incoming request and transaction will be recorded by System Logger. System logs are vital for error detection and correction.

Check Result Handler is responsible for deal with the result of Scheduled Check Thread. It’s necessary to have both dispatch report and scheduled
check. Because the consideration of scalability and flexibility, the new interface server should be able to inform all dispatch servers through scheduled check mechanism. The new dispatch server will also be able to inform all the interface servers through scheduled report mechanism.

4.1.4 Dispatch Server

Dispatch server serves as the coordinator of client for using system resource. Dispatch server decides which application server will carry out user’s request. Dispatch server is invisible to the clients. Dispatch server implemented the multi-thread mechanism and it can handle multiple requests at the same time. The application of interface server is coded with Java.

![Diagram of Dispatch server application](image)

**Figure 4.4: Structure of Dispatch server application**

As shown in Figure 4.4, the application for dispatch server consists 3 layers. The first layer is the event generating layer. The second layer is the event handler layer where business logic transactions are carried out. The third layer is the storage layer for data and system log.

After client get the dispatch server IP address from interface server and establish connection to dispatch server, multi-thread socket connection pool will create an independent thread for this socket connection. After the connection is closed, system resources will be recollected and get ready for the next connection.

Scheduled check thread sends check request to every application server recorded in the database. The frequency of checking can be adjusted according to the actual need.
Scheduled report thread sends status report to every interface server recorded in the database. Status report contains dispatch server’s ID, current IP address and current workload. The frequency of reporting can be adjusted according to the actual need.

When a client connects to the dispatch server and request for application server IP address, the Redirect Application IP service will first scan the memory if this client has delivered the same request a certain amount of time ago. If it has, the same application server IP address will be send to the client if the application server is currently working. Otherwise, the Redirect Application IP service will chose the application server with least workload, send the IP address to client and record this transaction to the memory.

When an application server from application layer sends a status report to interface server, the Application Report Handler will deal with the status report. The application server’s information in the database will be updated, and if this application server is new to the dispatch server, a new record will be created. The current workload of the application server will also be included in the status report. So dispatch server will record the workload of each application server and make auto-balance according to them. The algorithm of auto-balance is always assigning the application server with lowest workload to client.

Every incoming request and transaction will be recorded by System Logger. System logs are vital for error detection and correction.

Check Result Handler is responsible for deal with the result of Scheduled Check Thread. It’s necessary to have both application report and scheduled check. Because the consideration of scalability and flexibility, the new dispatch server should be able to inform all application servers through scheduled check mechanism. The new application server will also be able to inform all the dispatch servers through scheduled report mechanism.

4.1.5 Application Server

Application server serves as the coordinator of transactions requested by clients. Application server’s IP address is invisible to clients although it is send to the clients for setting up connection. Application server will fulfill client’s request and interact with storage layer for data. Application server implemented the multi-thread mechanism and it can handle
multiple requests at the same time. Same to other servers, application server is also coded with Java.

![Diagram of Application Server Structure](image)

**Figure 4.5: Structure of Application server**

As shown in Figure 4.5, the application server consists 3 layers. The first layer is the connection layer. It handles all the connection with clients and storage layer. It provides connections for business logic transaction in the second layer. The second layer is the event handler where business logic transactions are carried out. The third layer is the storage layer for data and system log.

After client get application server IP address from dispatch server and establish connection to application server, multi-thread socket connection pool will create an independent thread for this socket connection. After connection is closed, system resources will be recollected and get ready for the next connection.

Scheduled report thread sends status report to every dispatch server recorded in the database. Status report contains application server’s ID, current IP address and current workload. The frequency of reporting can be adjusted according to the actual need.
As shown in Figure 4.6, the entire client request handling process is carried out by User Request Handler and Storage Result Handler. After Multi-Thread Socket Connection Pool received and established the connection with client, User Request Handler will exchange and confirm the protocol code with client. Each protocol code represents a kind of transaction. This protocol code mechanism has the advantage of easy update. It's able to support different versions of protocols at the same time. After client sends the protocol parameters to User Request Handler, it passes the data request to Storage Result Handler. Storage Result Handler will get connection to storage layer with the help of Storage Connection Manager and then get data from storage layer as requested. Storage Result Handler will pass the data to User Request Handler and then passes to the client.

The Storage Connection Manager is an instance of middleware SDK used between servers. Each kind of storage system will have specific middleware SDK to use. Details will be mentioned in chapter 4.2.

Every incoming request and transaction will be recorded by System Logger. System logs are vital for error detection and correction.
4.1.6 Storage Server

Storage server is responsible for storage work in the system. Storage server only answers to the request of application servers and it is completely transparent to clients. Storage server implemented the multi-thread mechanism and it can handle multiple requests at the same time. MySQL database is used for storage because it’s flexibility and small size. The storage server is also coded with Java.

As shown in Figure 4.7, the storage server consists 3 layers. The first layer is the event receiving layer. The second layer is the event handler layer where business logic transactions are carried out. The third layer is the core storage layer for data and system log.

After application server precedes the request of clients and passes the data request to storage server, multi-thread socket connection pool will create an independent thread for this socket connection. After the connection is closed, system resources will be recollected and get ready for the next connection.

Request Handler will carry out a series of update or select actions to the database according to the request. After the interaction with database is finished, Request Handler delivers the result to application server through the original connection.

<table>
<thead>
<tr>
<th>Table: t_user</th>
</tr>
</thead>
<tbody>
<tr>
<td>u_id</td>
</tr>
<tr>
<td>-------</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
As shown in Figure 4.8, table t_user is used to store the profile of users. We only put user name into it for demo purposes and column expansion is possible.

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>u_name</td>
<td>varchar</td>
</tr>
<tr>
<td>u_password</td>
<td>varchar</td>
</tr>
<tr>
<td>u_lastlogin</td>
<td>datetime</td>
</tr>
</tbody>
</table>

**Figure 4.8: Structure of table: t_user in database**

As shown in Figure 4.9, table t_sensor is used to store the link between users and sensors. Each user can have multiple sensors of various functions. It is possible to change the ownership of a single sensor to other user.

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>s_id</td>
<td>int</td>
</tr>
<tr>
<td>s_uid</td>
<td>int</td>
</tr>
<tr>
<td>s_sid</td>
<td>int</td>
</tr>
</tbody>
</table>

**Figure 4.9: Structure of table: t_sensor in database**

As shown in Figure 4.10, table t_sensordata is used to store the common string form health record. Application can decide the type string of the health record and read the entries according previous decided string.

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>sd_id</td>
<td>int</td>
</tr>
<tr>
<td>sd_sid</td>
<td>int</td>
</tr>
<tr>
<td>sd_type</td>
<td>varchar</td>
</tr>
<tr>
<td>sd_value</td>
<td>varchar</td>
</tr>
<tr>
<td>sd_time</td>
<td>datetime</td>
</tr>
<tr>
<td>sd_uploadtime</td>
<td>datetime</td>
</tr>
</tbody>
</table>
The time of original creation and upload are both recorded in the database.

<table>
<thead>
<tr>
<th>Table: t_sensorstream</th>
</tr>
</thead>
<tbody>
<tr>
<td>ss_id</td>
</tr>
<tr>
<td>ss_sid</td>
</tr>
<tr>
<td>ss_type</td>
</tr>
<tr>
<td>sd_time</td>
</tr>
<tr>
<td>sd_uploadtime</td>
</tr>
</tbody>
</table>

Figure 4.11: Structure of table: t_sensorstream in database

As shown in Figure 4.11, table t_sensorstream is used to store the stream form health record like ECG or EGG. Application can decide the type string of the health record and read the entries according previous decided string. The time of original creation and upload are both recorded in the database. This table only contain the label data such as type and time. The core data is stored in another table. Because the stream data is usually big, and select operation is very slow in tables with big data. So the data id will be selected first in this table and the core data will be extracted according to data id.

<table>
<thead>
<tr>
<th>Table: t_sensorstreamdata</th>
</tr>
</thead>
<tbody>
<tr>
<td>ss_id</td>
</tr>
<tr>
<td>ss_value</td>
</tr>
</tbody>
</table>

Figure 4.12: Structure of table: t_sensorstreamdata in database

As shown in Figure 4.12, table t_sensorstreamdata is used to store the core data of stream form health record like ECG or EGG. The data in this table is indexed with only id.

Every incoming request and transaction will be recorded by System Logger. System logs are vital for error detection and correction.

4.1.7 Load balance algorithm and implementation

Load is a measurement for resource usage of all running processes on a single server. Load Index (LI) is an evaluation criterion for quantitative
load analysis. Different definitions of load index will lead to different load figure, so it’s crucial to set up an appropriate load index [23].

After analyzed the concurrent data exchange environment of our project, we defined following compound type load index. To make traditional definition of load index more adaptive to network environment, we added network connection quality index Ni into the formula. LI(i) is calculated according to (1) to evaluate the load of server i.

\[
LI(i) = \begin{cases} 
K_1U_iL_iN_i & M_i \leq RAM(i) \\
\frac{M_i}{RAM(i)}U_iL_iN_i & 0.4M_i < RAM(i) < M_i \\
K_2 & 0.4M_i \geq RAM(i)
\end{cases}
\] (4.1)

K1 and K2 are adjustment factor based on experience, 0<K1≤1 and K1<K2<+∞. K2 should be large enough. Ui stands for CPU usage or server i. Li stands for the length of process queue. Mi stands for memory usage of all the processes in the queue. Ni stands for network connection quality between dispatch server and application server i, this index is calculated with average round trip time. RAM(i) stands for free memory of server i.

The load balance process will be triggered only when partial load unbalance occurred. We implemented two ways to monitor the partial load unbalance and start load balance process. Each server in application layer will monitor its own LI. When it reaches a threshold value, this server will report partial load unbalance and call for load balance process.

Servers in dispatch layer will also monitor the load layout in whole application layer. Degree of load balance (LBD) is calculated according to (2) to decide whether to start load balance process.

\[
LBD = \frac{\sum_{i=1}^{n} LI(i) / W_i}{n \max_{i=1}^{n} \{LI(i) / W_i\}}
\] (4.2)
Wi stands for computational capacity of server i, this should be adjusted according to different scenarios.

When system LBD reaches a pre-decided threshold value, dispatch server will initiate the load balance process. It will redirect client request to the application server with lowest LI(i)/Wi. When dispatch server received partial load unbalance report from an application server, the load balance process will also be triggered.

Load balance process will last for certain amount of time. Process is ended when system overall degree of load balance decreases to threshold or when application servers call to cancel load balance process due to decrease of server load. The termination threshold of load balance process equals 80% of activation threshold. This configuration guarantees the effectiveness of load balance process is not influenced by accidental fluctuation in load spread. The reason why we didn’t start load balance process at the beginning is that dispatch server takes extra time to process the load balance calculation. We try to improve the overall system performance by initiate load balance process only when partial load unbalance occurs.

Since there is no data exchange channel implemented between servers in dispatch server, the process of start load balance is asynchronous. But the asynchronies between dispatch servers will be corrected within several heartbeats.

Load balance algorithm we proposed and implemented in this project should be able to dynamically balance the load among servers and make the best use of system resource, thus improve the scalability and performance of the system. Detail evaluation will be presented in Result and Evaluation chapter.

4.1.8 Crash detection mechanism

In distributed server group, various kinds of failures can occur, such as power failure, disk fault and fake crash due to overload [11]. We implemented a crash detection mechanism to dynamically detect crashed servers and automatically accept new servers into service provider group.

Our crash detection mechanism is based on lease mechanism. Dispatch server issues lease with a period of validity to application servers. Lease contains a time length data which determined the time length of validity period. Application server which holds a valid lease can join the service
provider group, otherwise application server need to request for a new lease. Dispatch server maintains a list of issued leases and remove outdated lease of list every certain amount of time.

When there is a failure occurs in application server, it will run a pre-decided failure handle process. As shown in Fig.4.13, a diagnosis process will be carried out to decide whether this application server is capable to provide service. If the failure is serious that the diagnosis can’t be carried out, the system goes to shutdown state and waits for manual setup. If the diagnosis process discovered that this application server can’t provide service temporarily, the process will cancel the request for new lease, wait for a lease time and run diagnosis process again. If the diagnosis process confirmed that this application server has recovered from failure and is able to provide service, this application server will request for new lease from dispatch layer.
From dispatch layer’s point of view, any application server which does have a valid lease shouldn’t be assigned with client requests.

Time synchronization is not necessary between dispatch layer and application layer because the time length is used instead of time stamp. This mechanism will work if time interval is close among dispatch layer and application layer. Dispatch server will always give extra time on lease validity time period to compensate the time cost on transmitting lease.

When new servers arrived in the system to increase computational capacity, crash detection mechanism ensures no configurations need to be changed among other servers beside the new one. The new application server only needs a list of running dispatch servers and it will send lease request to these servers to register itself into service provider group.

Crash detection mechanism we proposed and implemented in this project should be able to increase the fault tolerance of the system and allow system administrators to expand system easily for larger capacity, thus improve the scalability and performance of the system. Detail evaluation will be presented in Evaluation chapter.

4.2 Middleware API and SDK

All clients from different platforms get access to system resource with Middleware API the system provides. The connection between internal parts of distributed server group is achieved by Middleware SDK. Middleware increased the openness of distributed electronic health record system and allow different applications from different platforms share their data with each other. The difference between Middleware API and Middleware SDK is that API strictly protect and hide the communication method from clients, only leaves interface functions visible to clients. But SDK is used for internal development or external system integration, so some of the data format and processes are exposed to users for better understanding of the system. The code of API and SDK are almost the same, so there will be no separate description in the following sections.

4.2.1 Architecture

Middleware API embedded all data processing and transmission process into the package and compiled to a JAR (Java Archive) file. JAR file is kind of platform-unrelated file type and can be executed on any plat-
form which supports JRE (Java Runtime Environment). Clients can get access to system resource without knowing the data processing mechanism but simply follow the data format rule.

Figure 4.14: Structure of Middleware API

As shown in Figure 4.1.4, Middleware API serves as the bridge between client and distributed server group. Connection Manager is used to establish and maintain the connection information between client and distributed server group. The connection information includes application server IP address and user id. Authentication module is used to verify user identity. Connection Manager will only accept and carry out client requests after Authentication module returns succeed message.
Protocol Switch will analyse the client request and select corresponding function to handle the request. Data Format module will transform the custom data format into standard data format. This is the preparation of data serialization. Data Serialization and Data De-serialization modules are responsible for the conversion between standard data object and serialize string. String Compression and String Uncompression modules will zip and unzip the string data for better performance. Data Encryption and Data Decryption modules are responsible for the encryption and decryption of serialized string using AES encryption method. After all the data processing finished, Data Transmission module will send data to application server using specific protocol.

4.2.2 Main Function

To handle both string form data and stream data, 3 data classes were created. As shown in Figure 4.14, eHealthCommonData class consists 4 data members. Clients can decide type and value when creating an instance of this class.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>int</td>
</tr>
<tr>
<td>type</td>
<td>String</td>
</tr>
<tr>
<td>value</td>
<td>String</td>
</tr>
<tr>
<td>datetime</td>
<td>Date</td>
</tr>
</tbody>
</table>

Figure 4.15: Data member of eHealthCommonData

Stream data is a series of atomic data entries in chronological order. Each data entry is an instance of eHealthStreamData. eHealthStreamSeries class consists specific number of eHealthStreamData and can be manipulated as a whole.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>datetime</td>
<td>Date</td>
</tr>
<tr>
<td>value</td>
<td>Float</td>
</tr>
</tbody>
</table>

Figure 4.16: Data member of eHealthStreamData
Distributed Electronic Health Record
System based on Middleware
Xin Zhang

Design and Implementation
2013-05-19

Figure 4.17: Data member of eHealthStreamSeries

<table>
<thead>
<tr>
<th>Name</th>
<th>Parameter</th>
<th>Return</th>
</tr>
</thead>
<tbody>
<tr>
<td>getConnection()</td>
<td>String username, String password</td>
<td>int</td>
</tr>
<tr>
<td>getData()</td>
<td>String type, int lastmsgid</td>
<td>eHealthCommonData Array</td>
</tr>
<tr>
<td>uploadData()</td>
<td>int sensorid, eHealthCommonData data</td>
<td>int</td>
</tr>
<tr>
<td>getStream()</td>
<td>String type, String lastmsgid</td>
<td>eHealthStreamSeries Array</td>
</tr>
<tr>
<td>uploadStream()</td>
<td>int sensorid, eHealthECGSeries data</td>
<td>int</td>
</tr>
</tbody>
</table>

Figure 4.18: Functions of Middleware API

As shown in Figure 4.17, Middleware API of distributed electronic health record system has 5 functions for demo purposes. Client needs to initiate the Middleware API object by getConnection function. If it returns 1 which means succeed in connecting the application server and authentication, then the object can carried out client request. Otherwise, re-connection is needed.

getData and uploadData are a pair of functions to support client request of uploading and downloading string form health record data. getStream and uploadStream are a pair of functions to support client request of uploading and downloading stream form health record data. All the communication is done with socket and all functions are synchronized.
4.2.3 Data Serialization and Transmission

Serialization technology allows programmer to convert an object into a binary stream and store it into database or file system. The current status and data of the object will be saved as if it’s frozen.

All data members in the serialized class must implement get and set function in order to be restored. So eHealthCommonData, eHealthStreamData and eHealthStreamSeries are all implemented “java.io.Serializable” in order to be serialized.

<table>
<thead>
<tr>
<th>Name</th>
<th>Parameter</th>
<th>Return</th>
</tr>
</thead>
<tbody>
<tr>
<td>serialize()</td>
<td>Object original</td>
<td>String</td>
</tr>
<tr>
<td>deserialize()</td>
<td>String serializedstr</td>
<td>Object</td>
</tr>
<tr>
<td>objectToByteArray()</td>
<td>Object original</td>
<td>byte[]</td>
</tr>
<tr>
<td>byteArrayToObject()</td>
<td>byte[] bytearry</td>
<td>Object</td>
</tr>
</tbody>
</table>

Figure 4.19: Functions of SerializeUtil class

As shown in Figure 4.18, 4 functions were implemented in SerializeUtil class. All functions are protected so client can’t directly use these functions but other classes in the Middleware API package are able to use these functions freely.

4.2.4 Data Encryption and Decryption

128 bit AES encryption key is used to encrypted all the user profile data and user health record data. The encryption and decryption processes are done in the Middleware API with AESUtil class.

<table>
<thead>
<tr>
<th>Name</th>
<th>Parameter</th>
<th>Return</th>
</tr>
</thead>
<tbody>
<tr>
<td>Encrypt()</td>
<td>String content, String password</td>
<td>String</td>
</tr>
<tr>
<td>Decrypt()</td>
<td>String content, String password</td>
<td>String</td>
</tr>
<tr>
<td>parseByte2HexStr()</td>
<td>byte buf[]</td>
<td>String</td>
</tr>
<tr>
<td>parseHexStr2Byte()</td>
<td>String hexStr</td>
<td>byte[]</td>
</tr>
</tbody>
</table>
All data will be encrypted before uploaded to distributed server group or stored locally. All functions are protected so client can’t directly use these functions but other classes in the Middleware API package are able to use these functions freely.

4.3 Demo Client Application

To test the functionality of Middleware API and overall performance of distributed server group, demo client application on Android and PC platform were created. These applications only implemented basic UI and data exchange functions because other members in the project group are working on more complicated client application. The purpose of these demo client applications is mainly demo and testing.

4.3.1 Android Client Application

Android is a Linux-based mobile operating system which is widely used in Smartphone and tablet computers. Android occupies 75% of Smartphone market by the end of 2012. Most of the potential users possess Android so develop an Android client application for the system is necessary.

As shown in Figure 4.20, the demo version of Android client application for patient implemented the basic function of viewing data. Patient can view his own health record data in both list form and line chart form.
Figure 4.22: Screenshot of Android client application for care-giver

As shown in Figure 4.21, the demo version of Android client application for care-giver implemented the basic function of user right control and viewing data. Care-giver can view health status data of all patients under his watch.

Both client applications use Middleware API to get access to system resource. User needs to log in with username and password before getting into main interface of the application. User right control in the distributed server group ensures that patients can’t use application for care-giver.

4.3.2 PC Client Application

PC client application is coded with Java and used SWT (Standard Widget Toolkit) as GUI (Graphical User Interface) structure.
As shown in Figure 4.22, PC client application implemented basic functions of viewing health status which allows community care-givers view the health status of all the patients in the community. No upload function was implemented since PC client application will be used only for monitoring purpose.

PC client application uses Middleware API to get access to system resource. User needs to log in with username and password before getting into main interface of the application. User right control in the distributed server group ensures that patients can’t use application for care-giver.
5 Evaluation

This chapter will analysis the distributed electronic health record system based on middleware from scalability, performance and capacity, openness, security and privacy protection.

5.1 Test environment

To test the performance and capacity of distributed server group, a LAN version of demo server group was established. Because there are not enough physical servers, Vmware is used to create virtual computers.

![Diagram of demo server group]

Figure 5.1: Structure of demo server group

As shown in Figure 5.26, 8 virtual servers are created on 2 physical computers. We set up 1 interface server in the interface layer, 2 dispatch servers in the dispatch layer, 4 application servers in the application layer and 1 storage server in the storage layer. Each virtual server runs on Window Server 2003 operating system and each of them has independent IP address. The connections between each layer are pre-configured.
The demo server group and client simulator (Mentioned in 4.3.3) compose the test environment of distributed electronic health record system based on Middleware.

5.2 Performance and capacity

To test and evaluate the performance and capacity of the distributed server group, we used client simulator to simulate concurrent client requests. There are 5 types of client request: login, getData, getStream, uploadData and uploadStream. We measured response time and transaction per second as 2 main feature representing system’s performance and capacity. We tested from 1 concurrent client request to 10 concurrent client requests. Each test measures 500 random generated client requests.

As shown in Fig. 5.2, the response time is the sum of client processing time, internet transmission time, server processing time and database processing time. In our case, response time equals to the sum of C1, C2, A1, A2, A3, N1, N2, N3 and N4.
Figure 5.3: Average response time of different functions

As shown in Fig. 5.3, average response time of all functions increased smoothly with the amount of concurrent users. Get data actions’ response time increased faster than upload data actions is because selection process in database cost more time. As a matter of fact, majority of user requests are upload data request while only a few are get data requests. Overall response time remains acceptable with growing amount of concurrent user request. The result indicates that distributed server group with multi-thread mechanism holds system response time to a satisfying degree.

![Graph showing average response time of different functions](image)

Figure 5.4: Transactions per second of different functions

As shown in Fig. 5.4, the maximum transactions per second (TPS) of functions except upload data are around 5 TPS. The reason why upload data action reaches 25 TPS is upload data action requires least calculation and database process. System’s transaction processing capacity meets the requirement of testing case, and the result indicates that the concurrent performance of system is satisfying.

![Graph showing transactions per second of different functions](image)
As shown in Fig. 5.5, after we increased the test concurrent user amount, the system response time increased smoothly before concurrent user amount reached 150, after that, the response time gradually increased.

As shown in Fig. 5.6, after we increased the test concurrent user amount, the system’s average throughput stayed stable from 20 concurrent users to 150 concurrent users. After concurrent user amount reached 150, system throughput drops due to gradually increased system response time.
According to commonly used 2-5-10 principle in client based system evaluation, the definition of fast response is system response time is under 2 seconds. The definition of ok response is system response time is between 2 to 5 seconds. Users will feel slow when system response time is between 5 to 10 seconds. Users will feel unbearable when system response time is over 10 seconds.

![Proportion of system response time](image)

**Figure 5.7: Proportion of system response time**

As shown in Fig.5.7, when the concurrent user amount is around 75, 61% of the users get OK response. When concurrent user amount reached 150, 78% of the users get slow response. Based on this, we define the optimum number of concurrent users of this system is 75, and the maximum number of concurrent users is 150.
As shown in Fig. 5.8, we divided the response time zone into 3 sub zones, light load zone, heavy load zone and buckle zone. The threshold between light load zone and heavy load zone is optimum number of concurrent users and the threshold between heavy load zone and buckle zone is maximum number of concurrent users.

As shown in Fig. 5.9, partial load unbalance happens when there are around 10 concurrent users. This leads to sudden increase of system resource usage. But after we enabled the load balance mechanism, the level of system resource decreased compared to original situation.
Fig. 5.10: CPU usage of application servers in 20 seconds

Fig. 5.10 shows a piece of 20 seconds CPU usage record of application servers. When partial load unbalance occurs on Application server 3 and its load index rose above threshold, load balance mechanism was triggered. The CPU usage of application servers gradually averaged after load balance mechanism started. Then system was back to normal after load balance process ended.

The result shows load balance algorithm and mechanism we implemented can work effectively on system resource control.

To summarize the performance test above, distributed server group successfully prevented the dramatic increase of response time when there are more concurrent client requests. The average max transactions per second are around 250tps after calculated the individual TPS of each operation and their possibility.

Two formulas and one relationship can be concluded as the result of test. The relationship is that each dispatch server can support maximum 16 application servers, and each interface server can support maximum 16 dispatch servers. The least amount of dispatch servers and application servers are always 2 to ensure there is a backup server.

\[
A = \frac{m(1 + p)}{3 \times 10^5} \quad (5.1)
\]
In formula 5.1, A stands for amount of application servers in the system, m stands for potential user amount and p stands for the percentage of users which are over 65 years old.

\[ A = \frac{t}{3 \times 10^6} \]  \hspace{1cm} (5.2)

In formula 5.2, A stands for amount of application servers in the system and t stands for the total transaction amount per day.

With the 2 formulas above, the amount of application servers can be calculated after the user amount or system workload is known.

**5.3 Fault tolerance**

We simulated common failures such as network connection lost, service thread shutdown and system overload, to test the fault tolerance ability of the system. We compared average system response time and crash detection time in scenarios with and without crash detection mechanism we implemented.

We set the lease validity time in crash detection mechanism to 5 seconds, and we scheduled failure at the beginning of the test. Since we got 4 application servers, we simulated one and two servers crash to represent 25% and 50% of server loss.

![Graph showing system response time with and without crash detection mechanism](image)
As shown in Fig.5.11, average system response time remained stable when there is crash server situation with the help of crash detection mechanism and load balance mechanism. Average system response time increased more than 45% when server group lost 25% of its servers, but with the help of crash detection, measures have been taken to make increase of average system response time less than 19%. Since the lease validity time is 5 seconds, the test result shows that all crash situations were detected less than 5.5 seconds. This result proved that our implemented lease mechanism work well. This result also shows that system scalability is greatly improved by crash detection and management mechanism.

5.4 Efficiency

To evaluate the efficiency of middleware when processing and transmitting health related data, we measured total processing time include data formatting, data encryption and data transmission of upload data action and download action. We compared the performance and functionality of AES encryption and string compression in middleware to check whether these modules are scalable.

As shown in Fig.5.12, average processing time of single transaction increased slightly after data is encrypted with AES encryption method, this is due to extra time cost from encryption and transmission. Data encrypted with AES encryption method will gain around 10% of its size.
But after data compression, average processing time drops dramatically. This is due to time saving in transmission after compression.

The result shows AES encryption and string compression modules we implemented will actually improve the system performance. When dealing with large amount of health related data, middleware will significantly lower the overall time cost in data processing and transmitting. The power efficiency on mobile platform is closely related to time efficiency. Middleware is proved to be power efficiency based on the fact of its time efficiency.

The test proved that middleware works stable and scalable when processing large amount of health related data, thus improve overall system scalability.

5.5 Openness

Openness means the ability to exchange data with heterogeneous applications of a system. The Middleware API implemented in the distributed electronic health record system enables 3rd party applications and systems to share and exchange data with it freely.

During the test, Middleware API showed full functionality in Android client application, PC client application and client simulator. Clients can use same Middleware API even there are updates in distributed server group.

5.6 Security and Privacy protection

All the user profile and user health record are encrypted with AES encryption method. All the encryption and decryption processes are done in client side.

All data communications are done through Middleware API, and the data communications are embedded in the Middleware API. Clients don’t have the access to take a look at the detail communication protocol.

User rights are strictly distinguished in the system. Users of certain level will be only allowed to get access to specific data, such as patients can only view their own data but care-givers can view health record data of all patients under their watch.
Users also have the right to decide which part of the health record can be uploaded for analysis. If the user decides not to upload any data to the system, they can still view their own health record data on their own device.

The only part of the distributed server group visible to the outside is the interface layer. This layer does not contain any user-related information, thus preventing the system from being attacked.

5.7 Scalability

Scalability is the ability of a system or an application to handle a growing amount of workload and can be adjusted according to the amount of workload to achieve better performance.

Distributed server group and multi-thread server design handle concurrent user requests stably and promptly. Load balance algorithm and mechanism we implemented are proved to be able to dynamically adjust system resources in the system for better performance. Crash detection mechanism we implemented is also proved to be able to detect crash situations in time and make necessary adjustments. Data processing modules we implemented in middleware can provide convenient access to system resources while maintaining power efficiency.

The test result shows our test server group’s optimum user number is around 75 and means the server group can support 75 concurrent users with average response time lower than 2000 ms in rush hour. With this standard, the required amount of application servers can be calculated in real deployment scenarios. Total user amount and peak concurrent requests amount are main features need to be predicted in real deployment scenarios. Each dispatch server can support 4 to 16 application servers. So the required amount of dispatch servers can be calculated based on the amount of application servers used.

We also made comparison between existing health record systems and the distributed health record system we designed and implemented. As discussed in health record system related surveys and analysis, despite workflow designing and staff training issues, the most critical problems for existing health record systems are performance, stability, and scalability. Most of the existing health record systems can’t meet the average response time standard mentioned in [18], as the result, some of the existing health record systems actually decrease the work efficiency in care-giver facility [15]. The distributed health record system we designed and implemented meets the system performance and scalability stan-
dard described in [18]. The openness and stability of our system also surpass the existing health record systems mentioned in related papers.

After concluded the test result on system’s concurrent performance, resource usage, fault tolerance and efficiency, it is proved that our effort in achieving scalability in distributed health record system is effective. System showed satisfying scalability and performance in designed testing scenarios.
6 Conclusions

Electronic Health Record deals with a patient’s basic information and health care related information as it must conform to the standards. However, with the growing complexity of the electronic health record data source, it becomes a significant challenge to set up a structure which supports a massive data exchange concurrently. We proposed scalability oriented distributed electronic health record system to address the problem.

We designed and implemented a distributed server group as the core of the electronic health record system, and developed middleware in order to provide applications from multiple platforms access to the system resource.

We proposed and implemented a load balance algorithm and a crash detection mechanism in the distributed server group to ensure the system scalability and performance. The system creates and defines a series of standards in transferring and storing regular health data and stream health data. All raw data collected from different kinds of health related sensor systems will be formatted by means of the middleware system provided before being uploaded to the system. An encryption method and privacy control mechanism is also implemented to ensure data security and privacy. Power efficiency is granted by means of a string compression module which has been implemented.

To evaluate the overall performance and capacity of the system, a test server group and client emulator was implemented. The evaluated results proved that the designed system structure is scalable and flexible. Our efforts in achieving scalability in a distributed health record system are confirmed by theoretical analysis, simulation and comparison with existing health record systems. The designed and implemented distributed health record system meets the generally accepted standard and has better scalability in comparison to the existing health record systems. The implemented load balance algorithm and mechanism has proved to be able to dynamically adjust the system resource in the system in order to provide a better performance. The implemented crash detection mechanism has also proved to be able to detect a crash situation in time and to make the necessary adjustments. Data processing modules im-
implemented in the middleware can provide convenient access to the system resource while maintaining power efficiency.

Our future work will continue with exploring the key features of scalability, as well as other important issues concerning the development of an electronic health record system such as wireless sensor networks, application frameworks and security. Data security and privacy control are particularly important issues. While users’ health related data must be analyzed, privacy protection is crucial in relation to the confidence of the users in the system. A hierarchical user right system must be set up to manage user access to the system resources. We will also continue our research in related fields such as storage efficiency and database optimizing.
7 References


Distributed Electronic Health Record
System based on Middleware
Xin Zhang

References
2013-05-19


[26] Richard J. Baron, MD; Elizabeth L. Fabens, MD; Melissa Schiffman, MD; and Erica Wolf, MD, “Electronic Health Records: Just around the Corner? Or over the Cliff?,” Annals of Internal Medicine, 2005