Université du Québec à Chicoutimi

Mémoire présenté à
L'Université du Québec à Chicoutimi
comme exigence partielle
de la maîtrise en informatique

offerte à

l'Université du Québec à Chicoutimi
en vertu d'un protocole d'entente
avec l'Université du Québec à Montréal

par

GUO-HONGPEI

Design of Uniform Campus Identity Authentication System Based on LDAP

Juin 2006

ABSTRACT

With the development of campus network, many kinds of applications based on campus network get a rapid development in recent years. For management and expansion requirements, many of these applications need functionality provided by a uniform identity authentication system. On the other hand, LDAP (Lightweight Directory Accessing Protocol) and related technologies, taking advantage of distributed Directory Service architecture, organize and manage resources in network effectively and provide availability and security of system accessing, can be used for designing of a uniform identity authentication system.

This thesis discusses a design of uniform campus identity authentication system based on LDAP. The thesis analyzes the common situation of campus network application in China, describes what a uniform identity authentication system is and the necessity of designing a uniform identity authentication system, introduces the LDAP and related technologies in brief and designs a framework of uniform campus identity authentication system.

ACKNOWLEDGE

First of all, I wish to express my sincere gratitude to my research associate director, Prof. Wang-Jinsong, who helped me and guided me towards my academic as well as my professional success. He gave me many useful and important suggestions that help me complete this thesis successfully.

I also wish to thank my research supervisor, Prof. Qiao-Mei. This thesis would not be successful too without her help. Also, a special thank to Prof. Girard for being a member of my thesis committee. He gave me many substantial advices throughout my work.

I would also like to thank all my friends who helped me selflessly in the past year. Special thanks should be given to my parents and my wife for their great supports and encouragement to me.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

API	Application Programming Interface
CCITT	International Telephone and Telegraph Consultative Committee
DAP	Directory Access Protocol
DIT	Directory Information Tree
DN	Distinguished Name
IE	Internet Explorer
J2EE	Java 2 platform, Enterprise Edition
JNDI	Java Naming and Directory Interface
LDAP	Lightweight Directory Accessing Protocol
OASIS	Organization for the Advancement of Structured Information Standards
OSI	Open System Interconnect Reference Model
PDA	Personal Digital Assistant
SOA	Service Oriented Architecture
SQL	Structured Query Language
SSL	Secure Socket Layer

SSO	Single-Sign-On
W3C	World Wide Web Consortium

CHAPTER 1

INTRODUCTION

In this chapter, the reader is invited to analyze the reason that makes up this thesis project. The chapter also presents the contributions and an overview of the outline of the thesis.

1.1 Backgrounds and Motivations

With the development of communication and network technology, the campus network is playing a more and more important role in many fields such as teaching, researching and management aspects. The campus network is a kind of complicated network that integrates with multi-user, multi-system and multi-applications. A typical campus network may include Students Management System, Educational Administration System, Financial Information System, Office Automation System, etc. Most of these systems need to authenticate users and authorize users' operation rights when users access the system.

In traditional implementations, every system has its own users' management and authentication modules. These modules are isolated and used various backend database or protocols to store and manage the users' information. This method will bring some issues listed below:

1. Systems are separated, no way to share users information between different systems.

2. No way to implement a global and centralized management and security policies for controlling resources (users, applications, hardware) in the network.

3. Redundant data store and repetition administration and development for every system's authentication part.

Above problems bring problems to both of developers, administrators and end-users. For developers, they had to repeat the similar developing work for every authentication modules of various applications. For administrators, suppose a user needs to be added or removed from the campus network, administrators had to repeat the same management works in every system in campus network. For end-users, they must remember many user account and password for every system, and if they want to update their personal information, they must do the same work in every system. It seems like a doom of authentication and these drawbacks have greatly influenced the campus network's development. Therefore, the realization of uniform identity management and authentication of the campus users becomes a necessary requirement.

1.2 Thesis objectives and methodology

To solve problems caused by present authentication mode of application systems, the main objective of the thesis project is try to build a Uniform Identity authentication System based on LDAP in campus environment, which would be utilized by other application systems. The research emphasis is placed on building architecture of identity authentication system and designs a feasible authentication policy.

The methodology of research and building of the system is based on J2EE (Java 2 platform, Enterprise Edition) framework, which is an opening and effective development framework. Many open source projects were also adopted in the building of system. The purpose of using open source projects is to take advantage of the fruit of related works and make the research has more availability and compatibility.

There are already a lot of works in identity authentication field in recent years. With the development of Directory Service, especially the maturation of LDAP (Lightweight Directory Accessing Protocol) technology, it is possible to use standardized method to store and manage users' personal information centrally. This point is just one of the core features of a Uniform Identity Authentication System.

A Uniform Identity Authentication System needs a uniform authentication data format and a centralized data store that are flexible for applications' modifications and extensions. With applying centralized data store and management, the system replaced repeated development and management work of identity authentication module in other application systems and also promoted the security level of campus network.

In the present data store solutions, Directory Service is maybe a good solution for the centralized store and management requirement. Directory Service is a type of database, which has more advanced performance than traditional relationship database in data browsing and querying. Directory Service can use a series of attributes to name and describe users and other resources in the network uses different data structure to store data and this type of data structure is more suitable than relationship database to describe hierarchy and semi- structured data.

LDAP, the abbreviation of Lightweight Directory Accessing Protocol, is a communication protocol that defines series of interfaces between client and Directory Service. It is a helper between applications and Directory Service(s). Applications don't need to consider difference between various types of Directory Service by using LDAP. It is becoming the standard of next generation Internet. By using LDAP and Directory Service, we can construct complicated distributed directory architecture, store variant types of data in the directory and provide data accessing with uniform interface and high performance. This thesis project will use OpenLDAP, an open source implementation of LDAP and Directory Service, as the backend database and data accessing protocol.

Some projects about identity authentication often define and use their own standards or criterions. Applications need to be developed for the specific private standard if they want to apply the specific project. If the project was changed to others, the application may need to be developed for the new private standard supplied by other projects. This thesis project will provide interfaces based on Web-Service instead of define private standards so that applications don't need to know details about backend implementation and use uniform interface to access identity authentication system.

1.3 Thesis contributions

The thesis goal is to design a Uniform Campus Identity authentication System based on LDAP. For this purpose, the thesis will contribute to:

- Design and model the structure of campus network directory.

- Propose the uniform authentication policy and unified access control for Identity authentication System.

- Implement the Uniform Identity authentication System based on the proposed policies and directory model. The uniform campus identity authentication system will realize uniform identity management and authentication for most kinds of applications in campus network. Additionally, it will also realize Single Sign On (SSO) functionality. That means, the user only need a single account and sign into the system one time and then they have the right to access all the applications and services based on the directory services in campus network. Consequently, centralized account management, uniform identity authentication and unified access control to the whole network could be realized.

1.4 Thesis organization

The rest of this thesis is organized into the following chapters:

Chapter 2 covers the background information, which include the main technologies involved in the thesis such as Directory Service, LDAP, J2EE and Web-Service, etc.

Chapter 3 presents the analysis and design of authentication system. First, the chapter analyzes the system requirements of identity authentication system. Second, according to the requirements, the chapter describes the design about system architecture, directory structure, authentication policies and common service interface operations.

Chapter 4 details the design and implementation of the system. Finally, a conclusion and some recommendations for future work are presented in Chapter 5.

CHAPTER 2

LITERATURE REVIEW

This chapter will let the user know the background information related to the thesis. For this purpose, the chapter first introduces the Directory Service and LDAP technologies, and then introduces the architecture of J2EE (Java2 platform, Enterprise Edition) and Web-Service application that will be involved in the thesis.

2.1 Main concepts about identity authentication

2.1.1 What is identity authentication?

Identity authentication is the process of verifying whether you are the person or object you declared. In distributed environment, like campus network or Internet, identity authentication is needed for both applications and uses. Applications need to check user's identity to avoid unauthorized accessing or cracking. This is often accomplished by requesting user to input their user name and password when they sign on. Users have the same worry when they access the applications. They need to confirm the application is the real one that they want to access. For instance, in some e-commerce applications, users often are requested to supply their sensitive information like bank account or credit card number and transaction password. When users do so, they need a kind of authentication mechanism to confirm the authenticity of the application so that their important sensitive information is under protecting.

2.1.2 What is uniform identity authentication?

The simplest way to perform identity authentication is verifying username and password provided by user. Today, many applications apply this type of identity authentication. This type of authentication requests application to implement everything related to authentication by application itself. For instance, applications may need to provide private user storing mechanism. This is usually implemented by save username and password and other related information like department or role of the application in a relationship database. Besides, applications also need to provide a set of user management interface like user registration or password changing. Finally, applications must implement identity authentication process by themselves. Generally speaking, with this traditional way, applications had to take care of every thing related to identity authentication.

Besides above-mentioned issues, there are still some other problems caused by this traditional implementation. For example, because of storing users information into two-dimension table of relationship database, it is very difficult to modify or extend data structure of user information. Besides this, it is also difficult to describe hierarchy data with relationship database. On the other hand, developers often spend too much time on developing every functionality of authentication module, but they have no more time to consider the security of the system. They often choose the simplest authentication mechanism like transfer password with clear text in network or build some programs with security bugs. These problems usually incur some attack to applications, which are well known as third man attack or SQL injection attack.

Another type of authentication is called Uniform Identity Authentication. Different from traditional mode, this type of authentication provides user management and identity authentication with centralized mode. User information is easy to be shared between applications by take advantage of centralized management. Other applications only need to forward authentication request from client user to uniform identity authentication system instead of performing authentication themselves.

Uniform identity authentication system often adopts directory service as backend user database instead of relationship database. This type of database can easily describe hierarchy data and can flexibly define data structure of entity. These features are very fit for describing users and other related information in campus network.

2.1.3 SSO

SSO is the abbreviation of Single-Sign-On. It is an authentication process that allows a user to access multiple applications by authenticate stub which is generated when user previously sign-on an application. SSO is should be included in uniform identity authentication system. It requests user to provide identity information (like username/password) and authenticate them, if the authentication passed, a stub was generated and placed in client side, the user do not need to provide identity information again when they try to access other applications which are use the same uniform identity authentication system as their authentication service.

2.1.4 SSL

SSL is the abbreviation of Secure Socket Layer. It is a common used communication protocol for providing the security of information transmission over the network. SSL can guarantee the communication between client and server without wiretapping, modification and fabrication. It is initialized and developed by *Netscape Communications Corporation* and *RSA Data Security, Inc*. SSL uses the public-and-private key encryption algorithm from

RSA, which also includes the use of a digital certificate mechanism. SSL is a transport protocol and is transparent to applications. This thesis supposes all communications involved sensitive information between applications and identity authentication system over the network are built on the SSL.

2.2 Directory Service

2.2.1 What is Directory & Directory Service?

IBM describes directory as "A directory is a listing of information about objects arranged in some order that gives details about each object." [6] in its Redbook named *Understanding LDAP*. In another word, a directory is a kind of database that stores typed and ordered information about objects. Directory is very common in real world. A typical example is a contact list or a campus library card catalog. In computer world, we often use directory to store some hierarchical information and perform online querying for these information. From this point of view, "A directory is a hierarchical collection of objects and the attributes of the objects much like the subdirectories of a file system and the files contained in the subdirectories." [1]

Directory Service is a service that allows users or applications to search some information in a directory for a specified purpose. For example, a directory of contact list could be used to look up a person's phone number or e-mail address. A directory of device list could be used to find a scanner or printer. Or a directory of application servers could be used to find a server that can access customers' information or vendors' information. Directory Service usually provides some interfaces to outside to finish these operations.

There are many different ways to provide a directory service. Different methods allow different kinds of information to be stored in the directory, place different requirements on how that information can be referenced, queried and updated, how it is protected from unauthorized access, etc. Some directory services are local, providing service to a restricted context (e.g., finger service on a single machine). Other services are global, providing service to a much broader context such as DNS service on Internet. [14]

2.2.2 Why use Directory

Today, when we talk about database, we often mean relationship database. We have several choices about relationship database like DB2, Oracle, SQL Server or some other open source projects when we want to utilize a "database". These products has developed for several years and achieved a mature level. So why we still need Directory Service—another type of database?

"A directory is like a database, but tends to contain more descriptive, attribute-based information". [14] For example, the organization or department structure information. This type of information is relative static. The relationship database, on the other hand, generally holds transaction-based information like balances of account or timestamp of transaction, etc. This type of information is dynamic. Besides the type of information, the structure of information is not same too between directory and relationship database. "A directory is not a database. Objects can have varying attributes and numbers of the same attributes, unlike the columnar structure of an SQL database's 'table'". [1] Adam Tauno Williams describes this feature in his article *LDAP AND OPENLDAP*.

Directory is a specialized database with features that differs from general purposed relationship database. The information in a directory is often generally read much more than it is written. Just like many people would look up a person's phone number from a directory but the phone number seldom change. Therefore, Directory Service must be able to support high performance of read requests and be optimized for that purpose. As a consequence, Directory Service doesn't need to provide high volume of write and update operations. On the other hand, a general purposed relationship database usually needs to support application such as airline reservation system or bank transaction system with high update volumes and performance. [6]

One of the most important parts of identity authentication system design is information storing. Some identity information is often read than written and seldom changed (e.g., usernames). From this point, the Directory Service is better than relationship database for identity information storing.

Another important difference between Directory Service and general purposed relationship database is that Directory Service usually doesn't support the complicated transaction that relationship database used for performing high volume complicated operations. Transactions are inseparable list of database operations which must be executed either in its entirety or not at all. General purposed database usually implements transaction mechanism to maintain data integrity and guarantee that the database will always be in a consistent state. But because Directory Service mostly deals with read requests, the complexities of transactions can be omitted.

The type of information stored in a directory usually does not require strict consistency. It might be acceptable if information such as a telephone number is temporarily out of date. Because directories are not transactional, it is not a good idea to use them to store information sensitive to inconsistencies, like bank account balances. [6]

The way that information can be accessed is another important difference between a

Directory Service and a general relationship database. Most relationship databases provide support for SQL (Structured Query Language), a standardized, powerful access method for database operations. SQL allows complex update and query functions to access information stored in database. Directory Service, on the other hand, provides a simplified and optimized access protocol that can be used for relatively simple applications. [6]

In a word, because Directory Service is not intended to provide as many functions as general-purpose relationship database, they can be utilized for more applications with rapid access to directory data in large distributed environments. That means Directory Service is quite suitable for information store of campus uniform identity authentication system.

2.2.3 Directory Security

Because of some sensitive and private information may be stored in directory, the security of information stored in a directory is a major consideration. A directory should hold some contents needed to implement a security policy. Some of these contents may be stored in directory with encrypted format so that they are unreadable to unauthorized accessing, e.g. querying from LDAP client tool, or directly hidden to unauthorized accessing. Besides, the identity authentication system needs to provide an authentication policy to perform identity verifying by utilize contents in directory. For example, the simplest way to do this is verifying username and password when someone wants to sign on the system. The complicated way to perform verifying is an authentication based on stub. This is just the point this thesis would discuss. Once users are authenticated, the authentication system also needs to determine whether they have the authorization or permission to access the application and perform the requested operation.

2.2.4 Directory Service Standard

X.500 is the first standard of Directory Service. The CCITT (International Telephone and Telegraph Consultative Committee) created the X.500 standard in 1988, which became ISO 9594, Data Communications Network Directory, Recommendations X.500-X.521 in 1990, commonly referred to as X.500.

X.500 organizes data objects in a hierarchical structure so that it can support large amounts of information. It also defines powerful search mechanism to make retrieving information easily. "Because of its functionality and scalability, X.500 is often used together with add-on modules for interoperation between incompatible directory services". [18]

X.500 specifies the DAP (Directory Access Protocol) as the communication protocol between the directory client and the directory service. However, as an application layer protocol, the DAP is too big and too heavy to applications which would take advantage of it. "The DAP requires the entire Open Systems Interconnection (OSI) protocol stack to operate." [18] Supporting the whole OSI protocol stack requires too many resources, which are not available in many small environments. Therefore, an interface to an X.500 directory server using a less resource intensive or lightweight protocol was generated.

2.3 LDAP

2.3.1 What's LDAP?

"LDAP (*Light Weight Directory Access Protocol*) is a client-server protocol for accessing a directory service. It was initially used as a front-end to X.500, but can also be used with stand-alone and other kinds of directory servers". [14] Just like its name, it is a lightweight and cross platform protocol.

"LDAP was developed as a lightweight alternative to DAP." [6] LDAP uses the TCP/IP protocol stack instead of OSI protocol stack, which is lighter and more popular than latter. LDAP is a descendent of X.500 but simplifies some X.500 operations and omits some complicated features.

2.3.2 LDAP directory service

LDAP does not define the Directory Service itself; it only defines a communication protocol, which includes the transport and format of messages used by a client to access X.500 directory. Because of the complexity of X.500, people often have not X.500 servers and can't apply X.500 in their environments. On the other hand, with the development of LDAP, people began to build directories that support LDAP operations only (less than X.500) and could be accessed by various LDAP clients also. So the LDAP is extended to include directory service itself, which is known as LDAP directory server or LDAP directory service. Because the LDAP protocol is cross-platform, network-aware, and standards-based, so there are many kinds of LDAP implementations from all kinds of vendors. [3] In this thesis project, we choose OpenLDAP, an open source LDAP project to play as LDAP directory server role.

2.3.3 Overview of LDAP concepts and architecture

The architecture of application based on LDAP server is shown in figure 2.1.

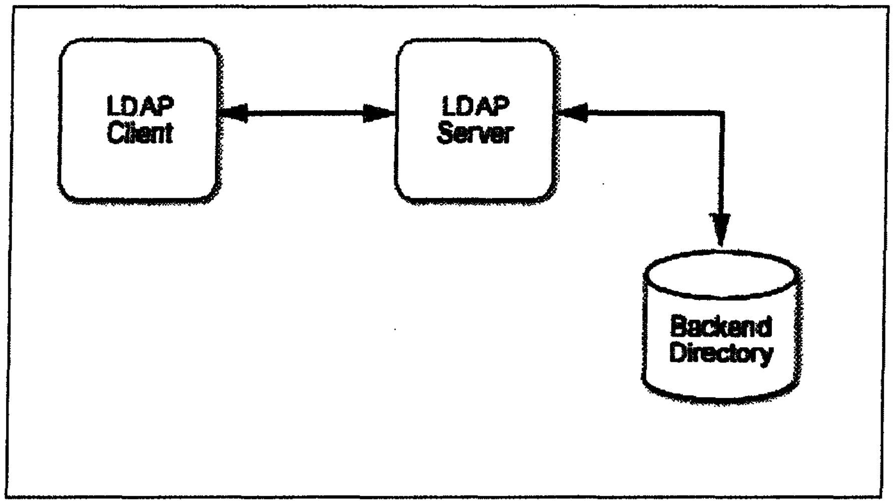

Figure 2.1. LDAP Application Architecture [6]

LDAP defines the content of messages exchanged between an LDAP client and an LDAP server. The messages specify the operations requested by the client, the responses from the server, and the format of data carried in the messages. LDAP messages are carried over TCP/IP, so there are also operations to establish and disconnect a session between the client and server. However, for the designer of an LDAP directory, it is not so much the structure of the messages being sent and received over the wire that is of interest. What is important is the logical model that is defined by these messages and data types, how the directory is organized, what operations are possible, how information is protected, and so forth. [6]

There are several concepts often used in LDAP directory design, which are listed below:

- Distinguished Name (DN)

Distinguished Name is the name of an entry that can identify the entry against others.

It consists of a sequence of parts called relative distinguished names.

- Entry

"A directory entry usually describes an object such as a person, a printer, a server, and so on. Each entry has a distinguished name (DN) that uniquely identifies it". [6] Every entry is corresponding to an object and it consists of a sequence of attributes.

- Attribute

An attribute is the information that describes one aspect of an entry. It consists of one attribute type and several attribute values. These attribute values depend on the type of the attribute. The relationships between values, attribute and entry are listed below.

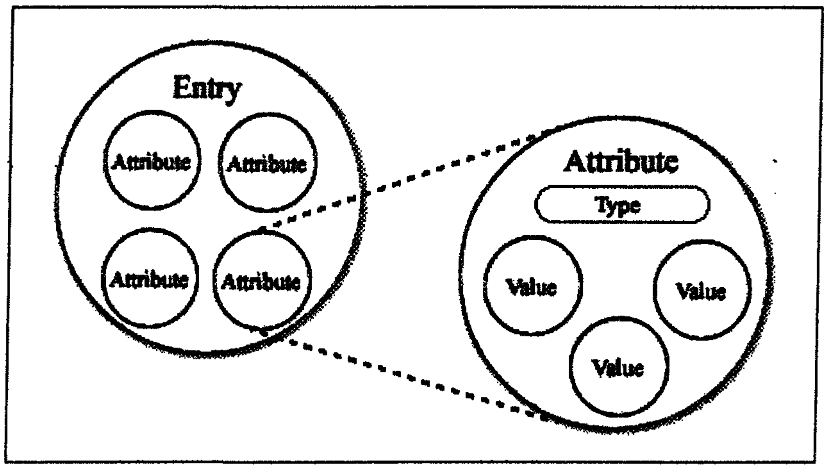

Figure 2.2. Relationships between Values, Attribute and Entry [6]

- DIT (Directory Information Tree)

In a directory service system, all directory sets can be described as a Directory Information Tree (DIT). Every node in the tree is an entry. Every entry can represent an object in real world. A DIT of nations may describe as follows.

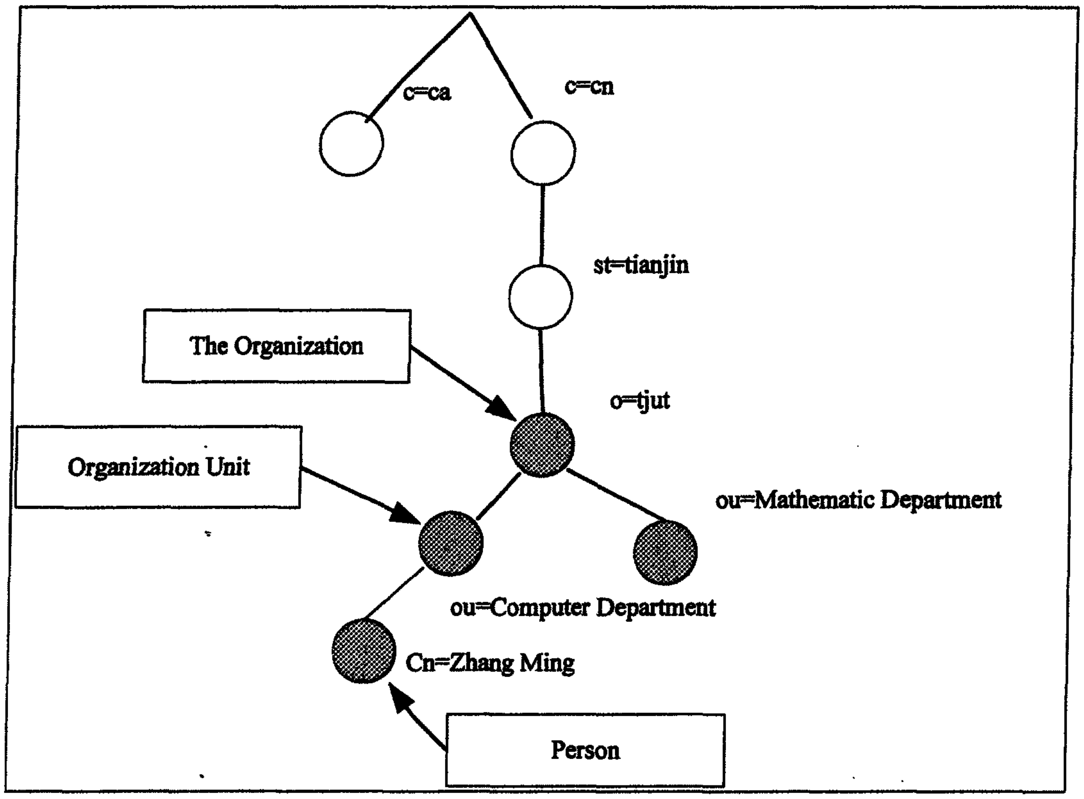

Figure 2.3. A Sample DIT

2.4 Existing authentication modes based on Directory and LDAP

There are already several identity authentication implementations that are based on

LDAP technology, such as *pam_ldap, nss_ldap, Radius, Samba,* and so on. These implementations are focused on the particular service for a given platform or depend on a given programming language. The purpose of this thesis is focused on building an authentication platform and providing authentication by web-service form that can face to many applications and does not limit the programming language on client-side.

2.5 Applications based on Web-Service

A definition from Wikipedia describes Web-Service as follows:

> A Web Service is a collection of protocols and standards used for exchanging data between applications or systems. Software applications written in various programming languages and running on various platforms can use web services to exchange data over computer networks like the Internet in a manner similar to inter-process communication on a single computer. This interoperability (e.g., between Java and Python, or Windows and Linux applications) is due to the use of open standards. OASIS and the W3C are the steering committees responsible for the architecture and standardization of web services. [22]

Web Service is an implementation of SOA (Service Oriented Architecture). It can be deemed to a software deployment method and has some features listed below.

- Web services use open standards and protocols, support rich data formats, which are text-based, that make it easy to be understood by human and also by computer.

- Web services are loosely coincidence architecture. The service provider and service requester are communicated or "separated" by service interface. Any internal change of each side won't affect another side. This feature makes web services have more flexibility and stability.

- Web services provide interoperability between various software applications running on different platforms and developed by various programming languages.

- Web services can easily integrate software and services from different companies and organizations as a bigger service.

- By utilizing the universality of HTTP protocol, web services can work through many firewall security policies without requiring changes to the firewall filtering rules.

- Web services emphasize the reuse of services and components.

In a word, we can integrate above advantages in our application to provide a reusable identity authentication service to other applications running on different platforms and written with various programming languages in campus network. That will reduce redundant work in every application system and promote holistic application security of campus network.

CHAPTER 3

ARCHITECTURE FOR CAMPUS IDENTITY AUTHENTICATION SYSTEM

In the preview chapters, we have studied the situation about campus network and involved techniques about the thesis project. In this chapter, we will analyze the detailed system requirements and propose architecture for the identity authentication system based on LDAP.

3.1 System requirements analysis

In this section, we will present the requirements that we believe need to be provided by a campus identity authentication system.

Basically, an identity authentication system should be able to verify whether the user is the right person when he signs in the system. This verification is often implemented by checking the user-id and password input by the user and which is also stored in a database. The authentication system also should be able to verify the authority when a user requests to access some applications in the network.

When a user sign into the system and pass the identity verification, the system should give a token to the user. This token will be checked again when the user accesses any other application in the network. So the token should be reusable. With this mechanism, a user only needs to sign in one time and then he can use all granted applications in the network.

Additionally, the identity authentication system should be maintainable. Common users could update their own personal information such as password or e-mail address. The administrator could adjust system's configuration and manage access control policy of the system. All above functionalities are listed in the Figure 3.1.

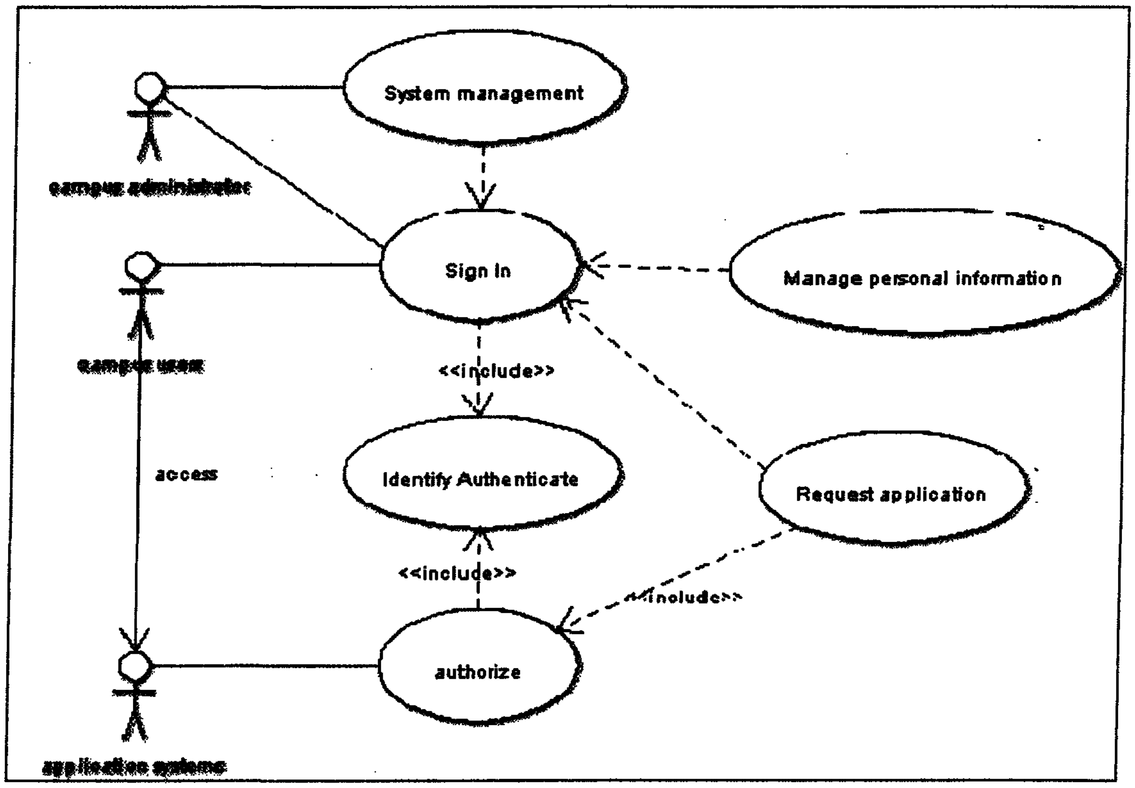

Figure 3.1. Use-Case Diagram of System

According to above analysis, detail requirements are described as follows.

- Provide identity authenticate service

The identity authentication system should be able to provide identity authentication service to other applications or systems running in campus network. These applications may include students' registry management system, interlibrary system, e-mail system, and so on. The authentication service should be able to provide a uniform and security identity authentication to all these applications.

- Implement uniform authenticate policy

The authentication system should be able to implement a service policy for providing identity authentication and avoid unauthorized accessing to identity authentication system itself and other applications protected by authentication system.

- Centralized management for user information

The information of users in campus network should be stored in a centralized directory database. User information may include ID, name, password, e-mail address, etc.

The authentication system should be able to provide a centralized management interface for this information.

- Centralized management for application information

Like user information, the information about applications in campus network also should be store in the same centralized directory database and the authentication system also should be able to provide a centralized management for this information. Application information may include the name of application, application's password (used in authentication system), the location of application, etc. In fact, from system's point of view, all users and applications are resources of system, so the system should be able to provide a uniform management interface for these resources.

- Provide access control based on role

There are many types of users in the campus network (*see Figure 3.2*). Every type of users has their own accessing right and their own activities in the system. For example, a student can view his own score of exam but cannot edit it. A teacher can edit students' scores but cannot modify system's configuration because only administrator is allowed to do this. We called this a role. The role represents the group of persons who has the same rights or activities in the system. The identity authentication system should be able to provide the access control management based on role because this kind of management can facilitate maintainability and consistency of system.

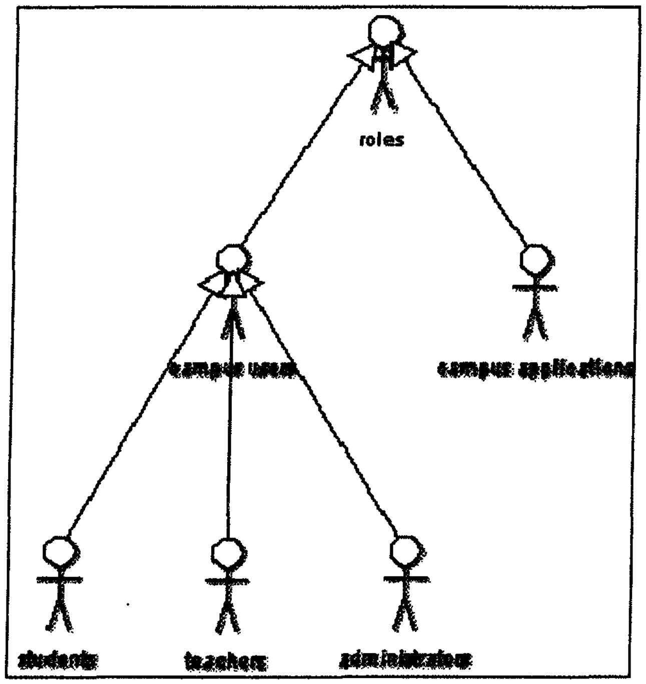

Figure 3.2. Roles Relationships Diagram of System

- Implement single sign-on for users

The authentication system should be able to provide Single Sign-On (SSO) for users. With this function, the user only needs to remember one password and sign into the system only one time, and then he can roam around all systems in campus network without signing in again.

- Integration for existing application systems

Maybe there are several application systems in campus network, which were built before uniform identity authentication system. These systems have their own user management and authentication modules, so modifications for these existing systems are difficult. An integration method for these systems is needed. In the thesis project, the integration is implemented by a user simulation.

- Provide access interface based on Web-Service

The authentication system should be able to provide accessible interfaces based on Web-Service technologies. Web-Service is an open standard in Internet environment. It can provide an interface on web server and any client access this interface can get service provided by the web application without knowing about detail implementation of the application. Because of using HTTP protocol, the communication between client and Web-Service can go through the firewalls in the network and by using the XML data exchange format, the Web-Service facilitates integration with other systems.

3.2 System architecture design

3.2.1 General architecture design

Our proposed architecture is depicted in Figure 3.3. In this architecture, there are three parts: Campus applications systems, campus clients, and Identity authentication System. They are connected to each other within the campus network.

Campus applications systems include applications or services distributing in the campus network such as student registry system, financial report system, e-mail service, and so on. There are two types of these applications in the network. One is the application who hasn't its own identity authentication sub-system. We called this type of application as internal application relative to the Campus Identity authentication System. The other type is the application who already has its own identity authentication sub-system. We called this type as external application relative to our authentication system.

Actually, campus clients are the people who use the campus network, but in the architecture figure, it represents the programs they use to connect their computers or other equipments to the network. Campus users may use variant types of equipments to connect the network. These equipments may include PC, laptop, or even PDA (Personal Digital

Assistant). Some of these equipments are using Windows system, UNIX/LINUX or other operating systems. From a software's point of view, there are also variant types of program that users apply. Users may use thin client program such as browser (ex. *IE, Netscape*) or fat client program such as client program based on Client/Server architecture. The authentication system must be adapted to variant requests sent by all these clients.

The campus network is a kind of complicated network. It may include many sub-networks and for security consideration, it has many firewalls or routers between these sub-networks. So the communication protocol that authentication system use must be able to go through these firewalls and routers in the network.

The authentication system consists of there parts: Authentication Server, LDAP Server and Directory Database. The Authentication Server is the interface to other applications and clients. The clients and applications interact with Directory by accessing the service that Authentication Server provided. Inside the architecture, the Authentication Server plays LDAP-client role (*see Figure 2.1*) and communicate with LDAP-Server by LDAP protocol. The Directory Database provides directory storing for the system. We store information about users, applications and access control, etc. in the Directory Database.

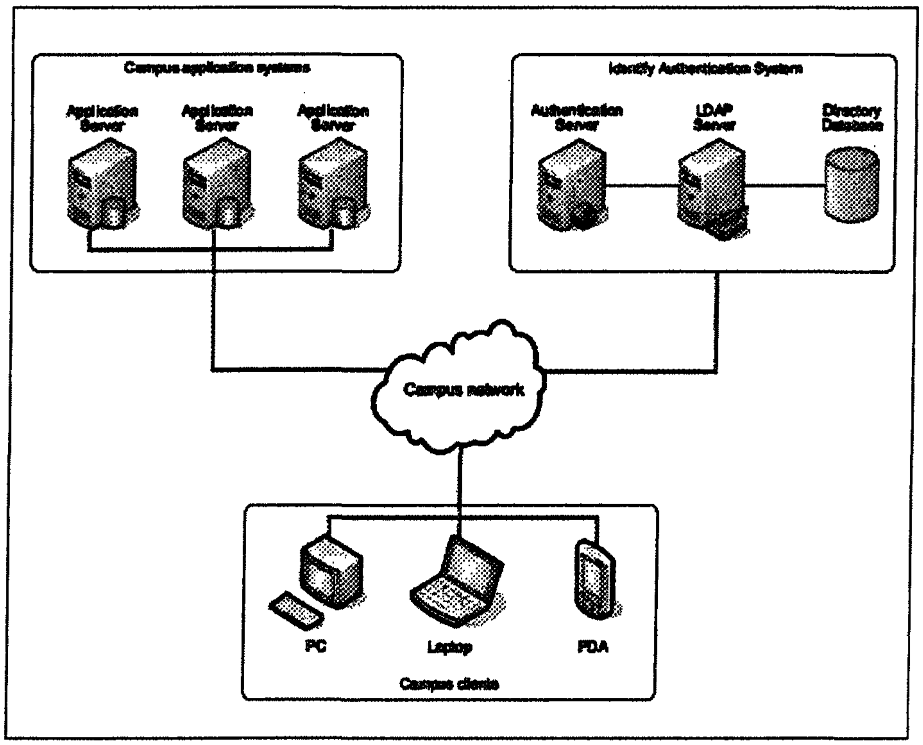

Figure 3.3. System Architecture

When a user tries to sign into the system, the Authentication Server will verify the user's identity. If the user passes the verification, the Authentication Server will record the signing of the user. When the user accesses any application in the network, the application will access the Authentication Server because it need to check the signing of this user. The Authentication Server exposes an interface to application to accomplish the checking. If the checking passed, the application will allow the user to access, or it will deny doing this.

The detailed architecture of Authentication Service is depicted in Figure 3.4. This is n-tiers architecture. The top layer is Client Tier that includes campus users and campus applications. From Authentication Service's view, all campus applications and users are clients of the Authentication System. These clients use HTTP or SOAP protocol to communicate with service. SOAP protocol is an encapsulation of HTTP protocol so that it can go through the firewall like HTTP. When client or application communicate with server, a XML data stream will be exchanged between client tier and service tier for message passing requirement.

The middle tier is Authentication Service tier. There are several components in this layer. The Servlets component is a set of java programs running in middleware container. In this architecture, it is responsible for communicating with campus applications. When an application's request arrives at Authentication Service, the Servlets first will accept the data stream with XML format, and then process the XML request from the stream. After that, the Servlets will call JavaBean component to process data. The JavaBean will return the results to Servlets after processing finished and then Servlets will assemble the results into XML document and reply this document as response to the application.

JSPs component is similar to Servlets. It is responsible for interacting with users but it is not a real java program. It is a set of dynamic web pages interfacing with users. The interfaces may include user sign in/out page, personal information management page, available applications list, etc. JSPs can dynamically create web pages according to the result that JavaBean returns.

JavaBeans component is a set of java objects that can perform complicated program logics such as authentication, encrypt, decrypt, access control etc. It is the kernel of authentication system. We can package often-used functions into JavaBeans that will be provided to Servlets and JSPs for their calling.

JNDI (Java Naming and Directory Interface) connector provides a standard interface used to access different kinds of naming and directory services, such as LDAP server. [4] By using these APIs, we can perform basic LDAP operations such as add, delete, query, etc.

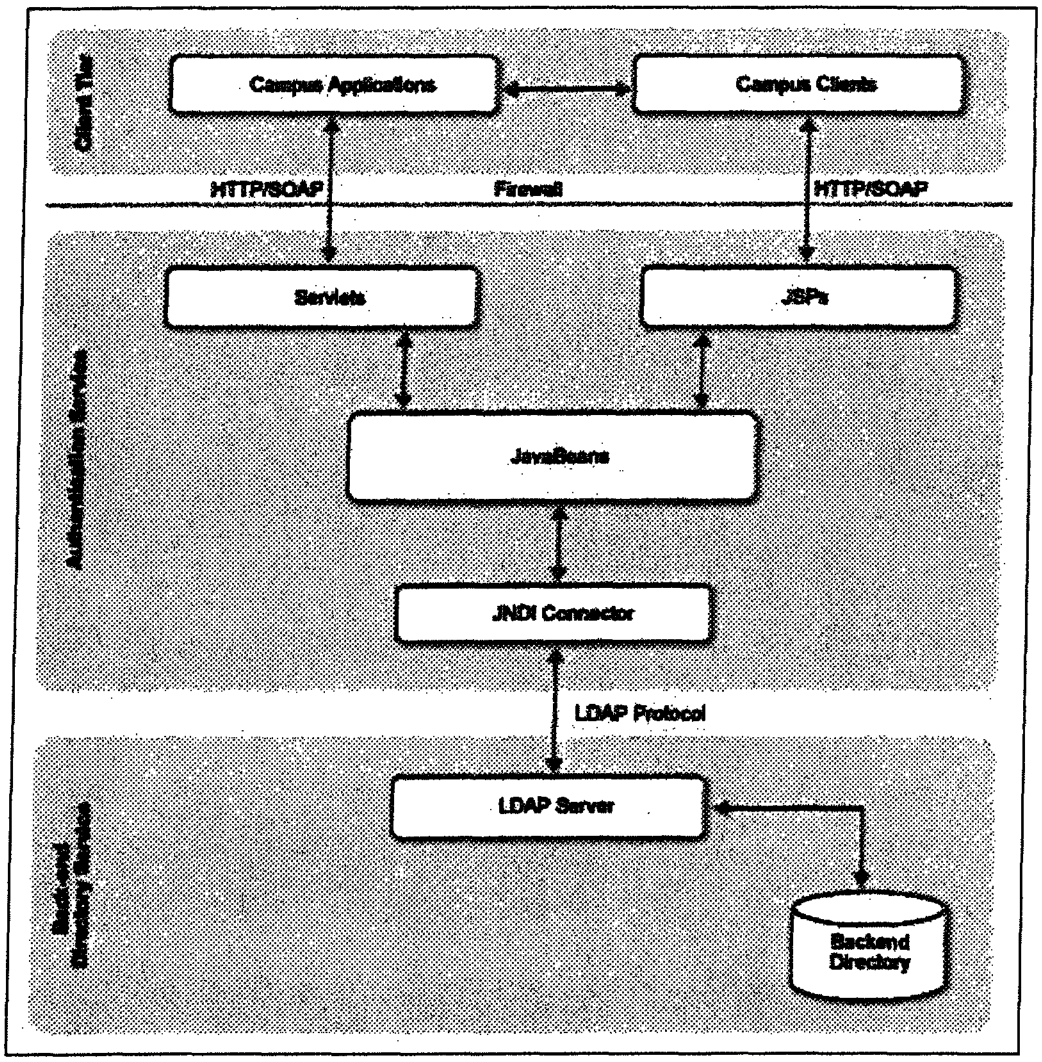

Figure 3.4. Detail Architecture of Authentication Server

3.2.2 Directory structure design

The first consideration about Directory design is the schema. Schema is the set of rules about variant aspects of database storing. The core.schema file provided by *openldap* contains many useful pre-defined object and data type definitions. We can use this file as our schema of Directory and also can add our own customization objects in it. The full schema file will be listed in the appendix.

Our proposed directory structure of campus network is depicted in *Figure 3.5*. In this figure, we only listed main attributes of Directory Entry here. The full structure's description is listed in the appendix.

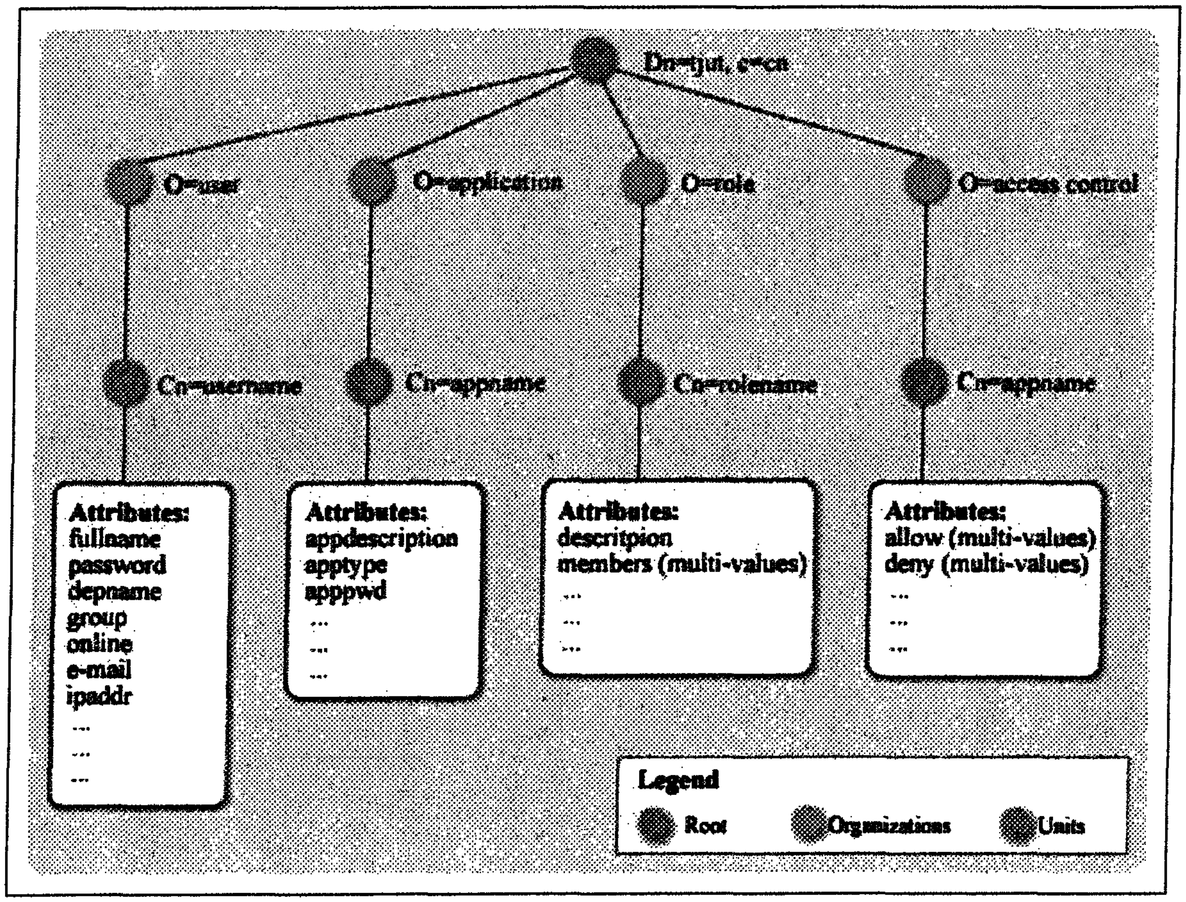
Dn=tjut, c=cn
O=user
O=application
O=role
O=access control
Cn=username
Cn=appname
Cn=rolename
Cn=appname
Attributes: fullname password depname group online e-mail ipaddr ... ...
Attributes: appdescription apptype apppwd ... ...
Attributes: descritpion members (multi-values) ... ...
Attributes: allow (multi-values) deny (multi-values) ... ...
Legend
Root
Organizations
Units

Figure 3.5. Structure of Campus Network Directory

The root entry represents the whole campus network of TJUT (Tianjin University of Technology) and includes four sub-entries. Their RDN name are *o=user, o=application, o=role and o=access control.*

The *user* organization is the group of all registered users on the campus network. Every user is attached with an entry which has a "cn=username" RDN and distinguish with others by this name. This type of entry has some important attributes listed in the figure. The password attribute stores the value of the user's password that will be used for identity authentication. The group attribute represents the role which the user belongs to. The online attribute's value represents the status of user. The value will be 1 when the user signs in, or 0 when the user signs out. The application needs to check this value when a user requests to access the application. The *ipaddr* attribute stores the value of client IP address from where the user signs in. This entry's information is maintained for the registered user and updated after every signing.

The *application* organization is the group of all used applications in the campus network. Every application has an entry with a "cn=appname" RDN and can be distinguished with other applications using the same name. The *appname* is the unique name of the application. The *apptype* attribute represents the type of the application. If the application uses the authentication, which the Campus Identity Authentication System provided, the *apptype* attribute's value will be "internal". If the application has its own authentication processes, the value will be "external" and the Identity Authentication System will simulate a signing interface for this application. The *apppwd* attribute stores the password of application. Like a user, every application has its own password in the Identity Authentication System. The application's password will be used by the authentication mechanism that will be discussed in the following section. The information about the application entry will be maintained by the administrator or by the application's owner. (Who built the application or has management right to the application)

The role organization is the group of user's roles. User's role is the collection of users who have same identities or activities. Every entry in the group has a unique role name that distinguished with other roles. The *members* attribute is a multi-values list that contains all users' name that belongs to this role. The administrator maintains the information on this role.

The access control organization is the group of access control policies of the system. Every item in the group is identified itself by a unique application name. The *allow* attribute's value is a list containing name of roles which the particular application allows to access. The *deny* attribute's value contains the name of roles which the application denies accessing. The *access control* entry's information is maintained by the administrator also.

3.2.3 Authentication policy design

The authentication policy is the kernel of authentication system. Our design for the authentication policy is depicted in *Figure 3.6.*

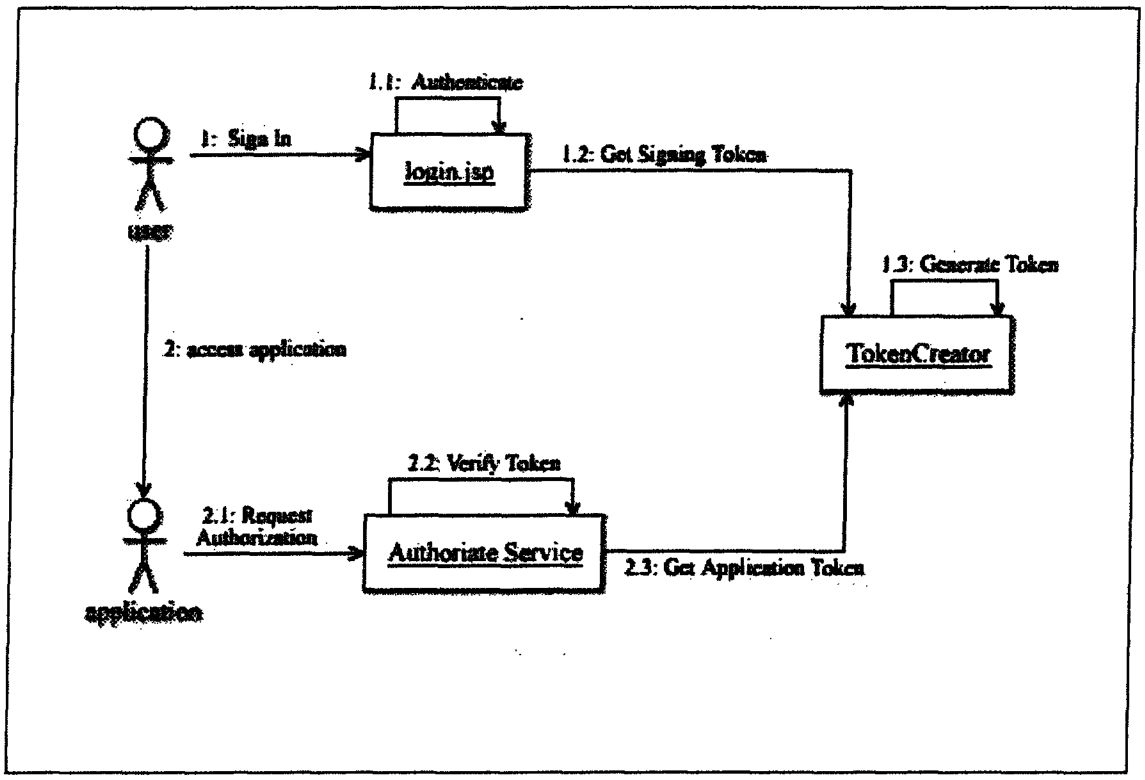

Figure 3.6. System Collaboration Diagram

In above figure, the user and the application are all actors relative to the authentication system. There are three main objects in the model and we will describe the functions of them and the system's workflow as follows.

First the user access login.jsp to sign into the system. The login.jsp is the entry of the system. It can query the LDAP Directory Service and finish the verification of user's ID and password. If the verification passed, it will call TokenCreator to generate the Signing Token. The TokenCreator is a servlet that can create token with encryption format. After token generating, the login.jsp will send the Signing Token and user-id to the user within a cookie.

Now the user may try to access an application. The application will read user's tokens from cookies and send these tokens to *Authorization Service* and attaches its own name, which must equal to the name recorded in the back-end Directory database. The *Authorization Service* is the web-service interface to applications. First, if there is no Application Token, the service will check the use's Signing Token and get username from the token. In this process, it may involve several servlets such as decryption, which are not listed in the figure. If the checking passed, *Authorization Service* will also call *TokenCreator* to generate the Application Token. After doing this, the *Authorization Service* will also generate an encrypted authenticator, then return the Application Token and the authenticator to the application. The application receives them and sends them to user with cookie format, and then allows user's accessing at the same time. Note the Application Token is reusable. That means when the user re-access the application before expire date, the *Authorization Service* has no need to re-generate Application Token. It will verify the Application Token and the authenticator directly.

3.2.4 Service interface design

The uniform identity authentication system should provide some common operation interfaces with web services. Main interfaces are listed in following table.

Common Interface	Interface Description
authenticate()	Check user's identity.
createSignToken()	Create a token by user's information.
createAppToken()	Create a token by application's and user's information.
verifySignToken()	Check the validity of token created by createSignToken()
verifyAppToken()	Check the validity of token created by createAppToken()

Table 3.1. Main Service Interfaces of System

Detail information about interfaces are described as follows:

- authenticate

This operation checks the validity of user's identity. It accepts *userid* and *password* as parameters and return true if they are same with what store in directory database, otherwise return false.

- createSignToken

This operation generates a token with string format. A token is indicator that indicates something happened already. A sign-token usually indicates a user entered the system and passed identity authentication (called by authenticateUser).

- createAppToken

This operation generates a token that indicates the user used the specified application already. There is no need to re-create the token when the user access the application next time if he/she holds the token for this application.

- verifySignToken

This operation checks the validity of token which was created by createSignToken operation. It returns true if the token is right, or false if the token is not one that system needed.

- verifyAppToken

This operation checks the validity of token that created by createAppToken interface. It is usually called by authentication system when user has a application token already and re-access the same application.

3.2.5 System security analysis

There are two preconditions about system security consideration. The first is we consider the clients' computer systems are secure. In other words, we have no responsibility for client-side security. We are only responsible for the security of content transferred in the network. This point is very important to system security because we will store important information in cookie on client-side. For example, if a hacker program is installed on client computer, which can capture the password that user typing, then we have no way to do with it.

The second precondition is the communications between authentication system and users or authentication system and applications are must be performed in the SSL channel. The SSL channel can provide transport layer security and can guarantee the content of communication with no capture and modification.

The authentication policy uses cookie to store token information. Cookie is a section of text stored in the client-side. Then how about cookie's security, is cookie secure enough? The answer is yes, but inadequacy use of cookie maybe led to leak of important information. So the cookie must be encrypted.

In fact, the authentication policy is based on Kerberos [2][12][13], which is a classical authentication mechanism. Kerberos policy applies symmetrical encryption algorithm in processing, which is not very secure. From this point, the SSL channel can promote the security level in a way.

CHAPTER 4

IMPLEMENTATION FOR CAMPUS IDENTITY AUTHENTICATION SYSTEM

In the previous chapter, we proposed the architecture of a Campus Identity Authentication System. In this chapter, we will present the implementation process based on the architecture we proposed.

4.1 Implementation for authentication process

The process for authentication policy has been discussed in the previous chapter (see *Figure 3.6*). In this section, we will present more details about the policy.

When a user tries to sign in the identity authentication system, he must submit his user-id and password that will be sent in the SSL channel. The authentication system will verify the password with what is stored in the Directory database. If the verification passes, the authentication system will generate a Signing Token for the user by the TokenCreator. The token can be described as following:

token={user-id, user-IP, ExpireTime } tokencreatorpwd

Table 4.1. Sign-Token Generation Formulation

Parameters in the formulation are described in following table:

Parameter	Description
user-id	The string of user identifier that user inputted when he signed in.
user-ip	The IP address of user.
ExpireTime	The value of time that the token will be expired at that time.
tokencreatorpwd	The password used to encrypt token.

Table 4.2. Sign-Token Parameters Descriptions

This formulation represents that the token is the string which consists of user-id, IP address of user's client computer, and expire time for the token, and the string itself will be encrypted with the TokenCreator's password. The token and user-id will be sent back to user's computer with cookie form by the authentication system. After doing this, the signing process is done.

Now the user will access an application. From the analysis in chapter 3, we know there are two types of application, internal and external. First, we discuss the situation about internal application. When a signed user accesses an internal application for the first time, the application reads all cookies that include the Signing Token and the user-id from client side, and then sends these cookies to a Web-Service interface provided by the authentication system and attaches its own application name. The service decrypts the token by TokenCreator's password. Because the TokenCreator is actually a part of authentication system, so the service knows its password and can perform the decryption. After the decryption, the service get the user's ID and user's IP from the token and compare them with user's id got from cookie and actual IP address of user, then check the period of validity by expire time. Next the service checks access control in Directory database. If there is any problem, the service will return failed authorization result to application, if there is no problem, the service will first retrieve the password of application from Directory by attached application name, then create a random shared password and then call TokenCreator to generate another token named Application Token. The token can be described as following:

token={SPWD, Appname, user-id, user-IP, ExpireTime } AppPwd

Table 4.3. App-Token Generation Formulation

Parameters in the formulation are described in following table:

Parameter	Description
SPWD	The shared password generated randomly.
Appname	The name of application that is being accessed.
user-id	The string of user identifier that read from cookie.
user-ip	The IP address of user.
ExpireTime	The value of time that the token will be expired at that time.
AppPwd	The password for the application that assigned and stored in directory database.

Table 4.4. App-Token Parameters Descriptions

This formulation means the token is the string encrypted with application's password, which consists of random shared password, name of the application, user's Id, user's IP address, and expire time for the token. Additionally, the service also generates an *Authenticator* consists of use's Id and user's IP, which is encrypted with the shared password. After generating, the service returns successful authentication result and the Application Token and the Authenticator to the application that once submitted the requisition with standard XML document. The application parses the XML document and gets the result, the token and the authenticator. If the result says OK, the application will send the token and authenticator to user with cookie form and allow user's accessing.

If the user accesses the application again in the token's period of validity, the application will read the Application Token and the Authenticator from client cookies, and then send them and its name to the Web-Service. The service first decrypts the Application Token with password of the application, and then gets the shared password from the decrypted token. Next the service uses the shared password to decrypt the authenticator and get user' Id and user's IP from it, and then compares the Id and IP with what is stored in Application Token. After doing this, the service will return the result of authorization to the application. According to the result, the application decides whether to allow the user's accessing or not.

For the second type of applications, external application, there are two ways to access them. The first is direct access and the second is access by checking *Available Applications List*. For the internal applications, there are no differences between two accessing ways. For the external applications, if the user accesses the application by directly connecting to it, the authentication system has no responsibility for authenticating. The authenticating is done by the applications' own authentication module. The user needs to input the password prepared for the particular application, not for the Identity Authentication System. If the user uses the second way to access the application, the Identity Authentication System will generate a login page to simulate a user's sign in. In this manner, the user has no need to input his password again. The authentication system will retrieve the password for the user from the Directory database, which pre-defined when the user customizes the application.

The *Classes Diagram* of authentication module is listed below:

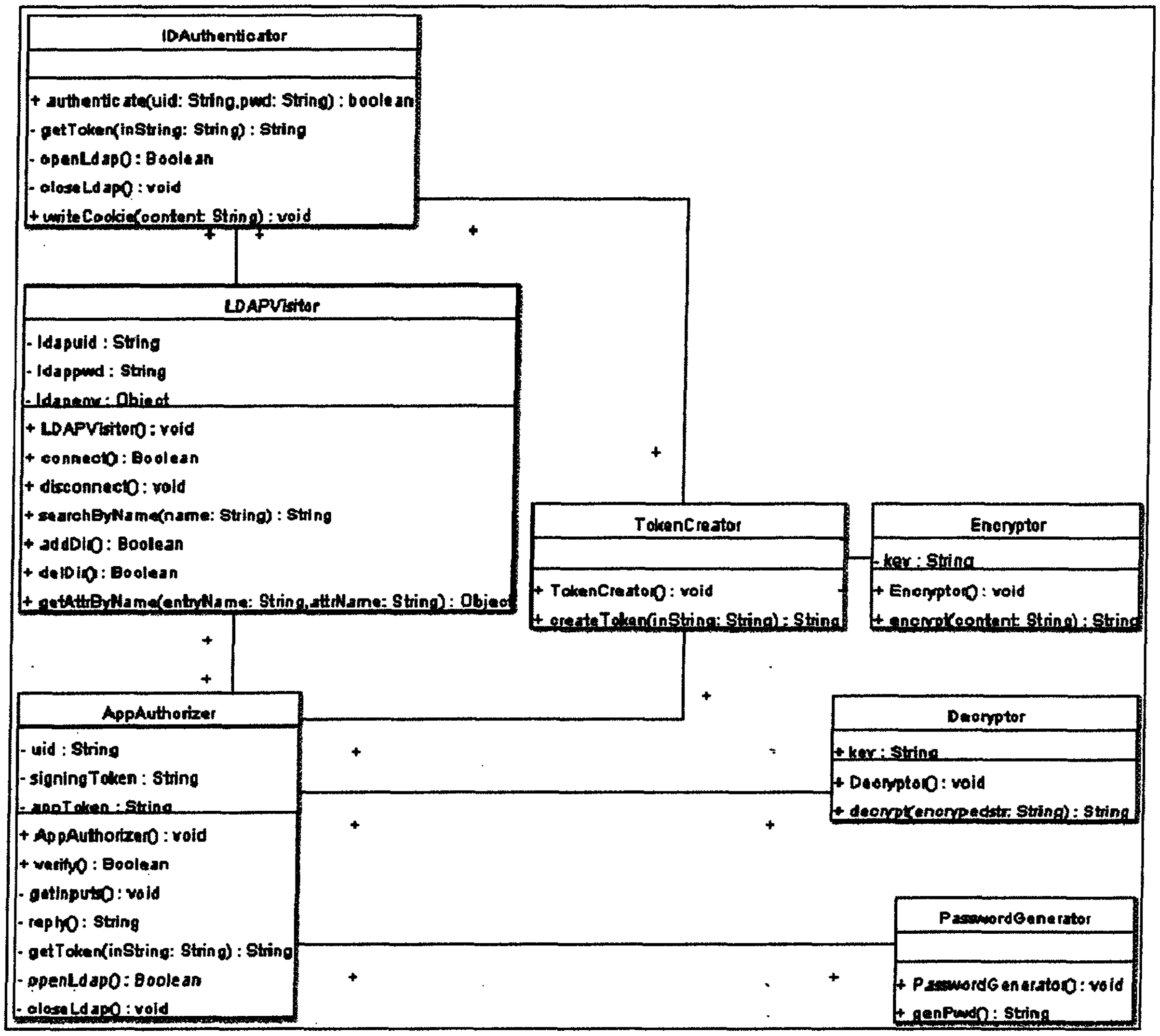

Figure 4.1. Class Diagram of Authentication Module

The main important methods we implemented in each class are listed below:

<u>IDAuthenticator:</u>

This class's main task is dealing with the verification for user's identity and it grants signing tokens to authenticated users.

+ authenticate()

This method performs authentication for user by *uid* and *pwd* in parameters. It returns TRUE if the authentication passed, or FALSE if not passed.

+ getToken()

This method retrieves the token from *TokenCreator*. In fact, it gets the encrypted string of the *inString* parameter.

- openLdap()

This method connects the LDAP server by calls *connect* member function of LDAPVisitor class. It is a private method and called by *authenticate()* method inside the class.

- closeLdap()

This method close the connection between the LDAP client and the LDAP server by calls *disconnect()* method of LDAPVisitor class.

+ writeCookie()

This method sends information in *content* parameter to users of Identity Authentication System by cookie format.

<u>LDAPVisitor</u>:

This class's main task is communicating with LDAP server. Most of its implementations depend on JNDI APIs provided by SUN Microsystems.

- getConfig()

This method gets the configuration value of the specified item provided in parameter from configuration file. The LDAPVisitor uses this function to get address, uid and password of the LDAP server and put them in ldapaddr, ldapuid and ldappwd member variables of the class.

+ connect()

This method connects the LDAP server with member variables by JNDI APIs. It returns TRUE if connected successfully and set *ldapenv* member variable's value to the reference of the connection object.

+ disconnect()

This method closes the connection built between authentication system and LDAP server.

+ getAttrByName()

This method retrieves the value of specified entry's attribute. The attribute is specified by the *entryName* and *attrName* parameters. The *entryName* is the DN string of the specified entry. The *attrName* is the string of the specified attribute's name.

<u>AppAuthorizer</u>:

This main task of this class is checking the accessing right of the specified application for specified user.

- getToken()

This function retrieves token generated by TokenCreator class.

+ verify()

This function is the most important function of authorizing. First, it will decrypt the Signing Token by using *Decryptor*, second, query the access control directory in LDAP database, last, it will generate *Application Token* by calling *TokenCretor* and determine the authorizing result according to the result in first and second steps.

- reply()

This function will send authorizing result and the *Application Token* to user side with HTTP protocol.

<u>TokenCreator</u>:

This class will generate *Signing Token* and *Application Token* for user.

+ createToken()

This function generates encrypted token string with *inString* parameter by calling *Encryptor* class. It will transform the parameter to *Encryptor* and get the token from return value.

4.2 Implementation for user operation

This section is focused on the implementation of user's operation for using the authentication system. The user's operation includes two main aspects: user's registration and user's customization.

4.2.1 User's Registration

The user's registration occurs when a new user first access the authentication system. For the purpose of using the system, he must register a user name for himself. Typically, he will fill some forms from a web page and submit the form to the authentication service. After his registration, the status of this user is pending because it needs the system administrator to approve the registration.

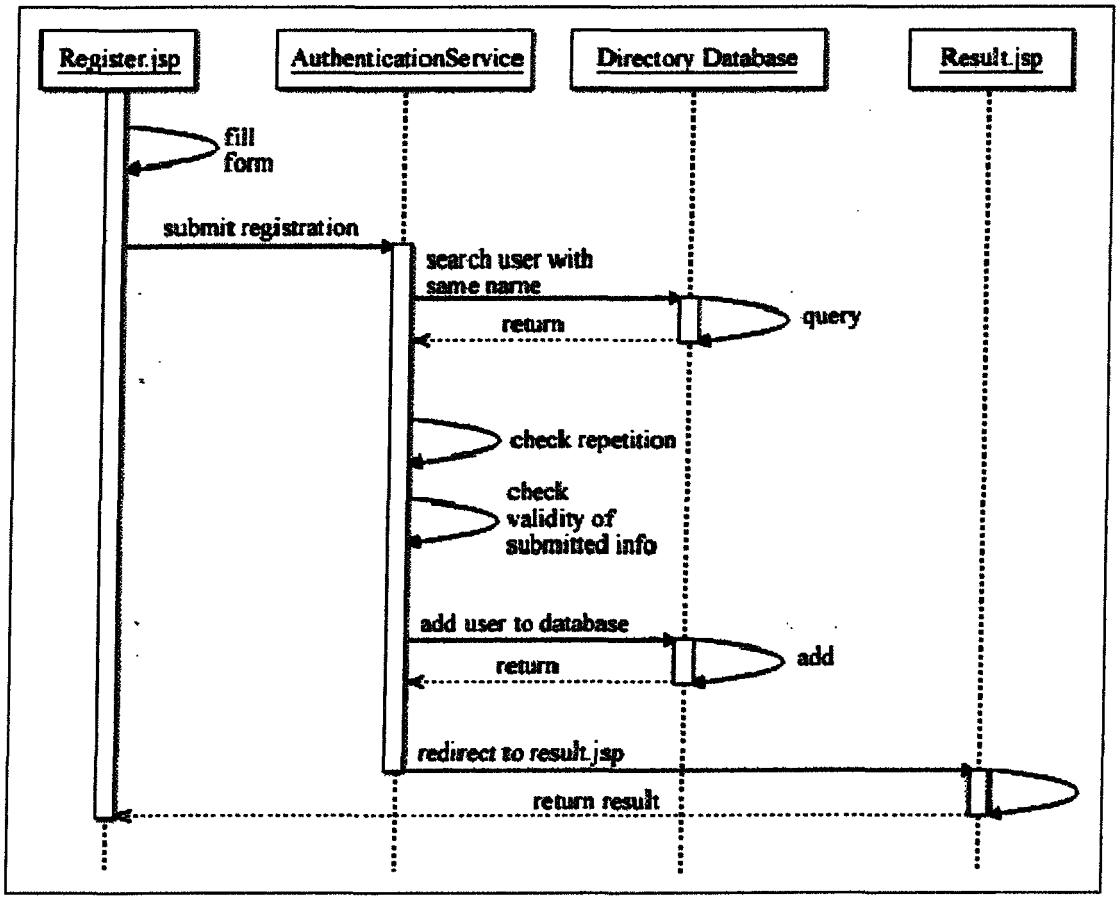

Figure 4.2. Register Sequence Diagram

4.2.2 User's Customization

A registered user can modify or update his personal information stored in the authentication database. This operation is called user's customization. The use's personal information may includes user's password or user's available applications list. As the same manner as user's registration, this operation also finishes with web pages and the result of the operation is stored into the authentication database.

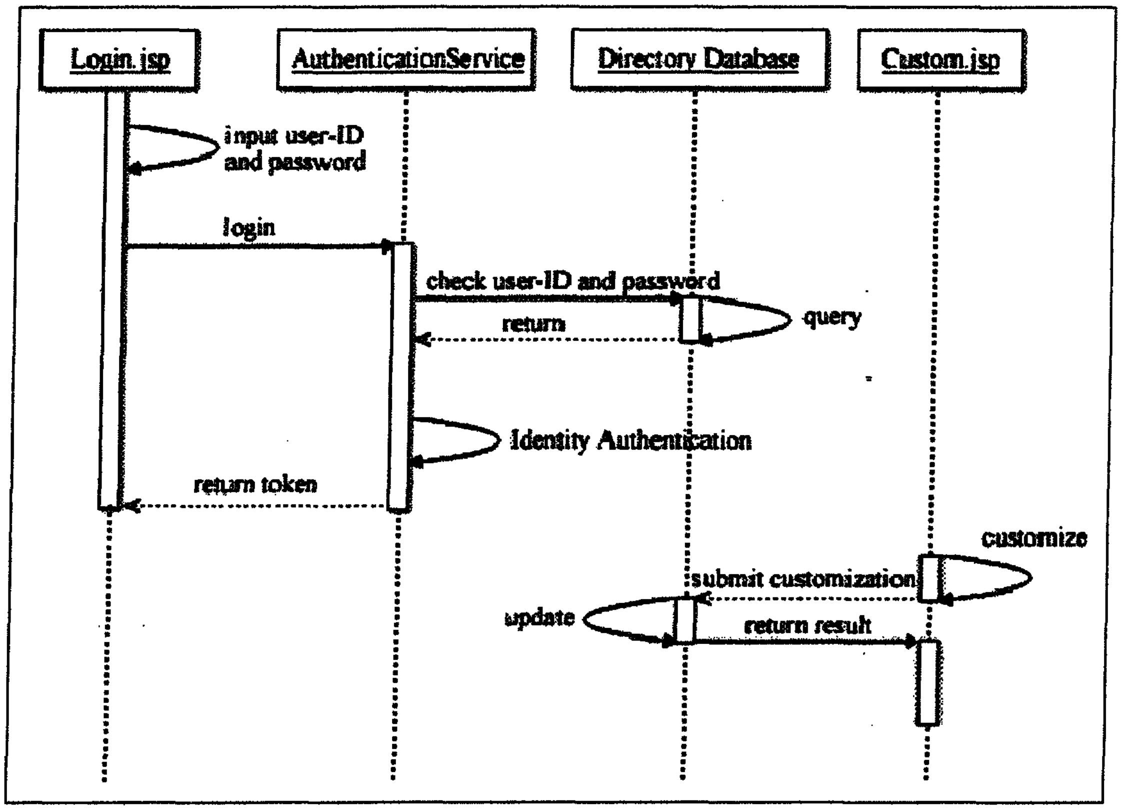

Figure 4.3. User Customization Sequence Diagram

4.3 Implementation for system administration

System administration includes user management, application management and role management aspects. Most of these managements are implemented with web applications.

4.3.1 User Management

User management includes user audit and user deletion operations. These operations are described as below.

When a new user of system submits his registration request, the system administrator would receive the request and must check the information that the user inputted in the registration page. If the information has no problem, the administrator would accept the user by performing audit-pass operation and assign accessing right to this user, or he must cancel the user's registration. If he accepted the user's status would be updated from a pending status to an activated status. Only an activated user can use the authentication system.

When a user no longer exists, the administrator must delete him/her from the system. This purpose is accomplished with an update of the user status from an activate to an abolished status.

4.3.2 Application Management

The application management is defined as a series of operations for other applications (except authentication system) in the campus network. The administrator must maintain application's information stored in database when the status of application is changed. For example, when a new application is available, the administrator must add a new application definition in the authentication database so that users of campus network could later access the application with the identity authentication system.

4.3.3 Role Management

For access control management convenience, this authentication system supplies access management based on role to the administrator. A role is a group of users. The administrator can separate campus network users into various groups and assign different accessing rights to these groups. In the system, there is a dynamic web page supplied for administrator to manage roles and in application management, the administrator can assign the current application to these pre-defined roles.

4.4 Client Applications need to do

If we regard the Identity Authentication System as a server application, so other applications such as student management system or financial information system in the campus network are the clients of the server. There are also some operations the client applications must implement for authentication processing. Because there are too many techniques and developing languages for client applications, we only give our proposals for client implementation. The detailed implementation could refer to the beginning of this chapter.

- Cookies compatible:

Because most information exchanged between server and client are stored with a cookie format, the client application must have reading and writing capabilities of cookies. If the application is implemented with web form and uses browser for its container, this request is easy to accomplish.

- HTTP communication protocol:

Because the authentication service is provided with web form, so the client application must at lease uses HTTP protocol to communicate with server. If the client implements the SOAP protocol, it is also acceptable.

- Web Service/XML supported:

Because we used web-service techniques to implement the service, it assumed that the client has the capability of finding and negotiating with this service. It also requests the client could parse information from the XML stream that is sent by the service.

- Decryption algorithm

Because the encryption algorithm implemented in the system is a symmetrical algorithm, the client application must implement the same decryption algorithm with its own developing language.

CHAPTER 5

CONCLUSION

In this thesis we proposed an architecture for a Uniform Campus Identity Authentication System Based on LDAP. A prototype implementation of identity authentication system using J2EE technology is presented to show how the JAVA platform can be integrated into the authentication system. From the above discussion, we come to some conclusions and items for future work.

5.1 Summary of results

The aim of this thesis is to design a Uniform Campus Identity Authentication System Based on LDAP. We have proposed an architecture for the authentication system. We have also studied the LDAP and J2EE technologies. In the thesis we have defined the kernel authentication process and the interface between service and other applications. In addition, we have implemented the administration requirements based on the web.

The system architecture proposed in thesis can implement a uniform identity authentication for various application systems in the campus network. It also implements

Single Sign On (SSO) functionality for end-user and provides central administration for system administrator. These features provide facility not only for users but also for administrators, and also make the system more maintainable.

Because of using Web-Service and J2EE technology within the design, the system becomes more flexible. The system can provide service for various client environments such as thin client which using browser or fat client which applying C/S architecture. Specially, by using HTTP as the basic communication protocol, the system can be easily applied in various environments in campus network.

5.2 Future work

Future work could focus on promoting the system's security level. This can be considered from several aspects. For example, the encryption algorithm now used in system is a kind of symmetrical algorithm. It could be promoted to non-symmetrical algorithm and apply a public key authentication mechanism.

In addition, with the development of the campus network, the scale of the network would become larger and larger. In this case, distributed directory architecture may be required for identity authentication system. This would be also an interesting goal to pursue.

BIBLIOGRAPHY OF THE REFERENCE

[1] Adam Tauno Williams, "LDAP and OpenLDAP", May 2001.

[2] Bill Bryant, "Designing an Authentication System: a Dialogue in Four Scenes", Massachusetts Institute of Technology, February 1988.

[3] Carla Schroder, "Building an LDAP Server on Linux", November 2003.

[4] Daniel Would, "Navigate the JNDI maze", November 2003.

[5] Eric Anderson, "Unite your Linux and Active Directory authentication", December 2004.

[6] Heinz Johner, Larry Brown, Franz-Stefan Hinner, Wolfgang Reis, Johan Westman, "Understanding LDAP", International Technical Support Organization, June 1998.

[7] Howes and Smith, "LDAP: Programming Directory Enabled Applications with Lightweight Directory Access Protocol"

[8] Howes, Smith, and Good, "Understanding and Deploying LDAP Directory Servers"

[9] James Kao, "Developer's Guide to Building XML-based Web Services with the Java 2 Platform, Enterprise Edition (J2EE), June 2001.

[10] James Snell, "The Web services insider ", May 2001.

[11] Jason Garman, "Single Sign-on for Your Web Applications with Apache and Kerberos", November 2003.

[12] Jason Graman, "Kerberos: The Definitive Guide", August 2003.

[13] Jim Rome, "How to Kerberize your site"

[14] Luiz Ernesto and Pinheiro Malere, "LDAP Linux HOWTO", September 2000.

[15] M. Wahl, T.Howes, S.Kille, "RFC 2251-Lightwegith Directory Access Protocol v3", December 1997.

[16] Mark Wilcox, "Implementing LDAP"

[17] Roel van Meer, Gisueppe Lo Biondo, "LDAP Implementation HOWTO", March 2001.

[18] Steven Tuttle, Kedar Godbole, Grant McCarthy, "Using LDAP for Directory Integration", IBM RedBooks, International Technical Support Organization, February 2004.

[19] Wangming Ye, "Web services programming tips and tricks: Improve interoperability between J2EE technology and .NET, Part 3", February 2005.

[20] "How the Client Obtains a TGT for the User", http://support.entegrity.com/private/doclib/docs/osfhtm/develop/appdev/Appde633.htm.

[21] "Berkeley DB Reference Guide: Building Berkeley DB for UNIX/POSIX systems"

[22] Wikipedia, "Web-Service Introduction", http://en.wikipedia.org/wiki/Web_Service

APPENDICES - SOURCE CODE OF PROGRAMS

IDAuthenticator.java

```
package org.tut.uia;
import java.io.IOException;
import java.security.Principal;
import java.text.MessageFormat;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.HashSet;
import java.util.Hashtable;
import java.util.List;
import java.util.Map;
import java.util.Set;

import javax.naming.AuthenticationException;
import javax.naming.CommunicationException;
import javax.naming.Context;
import javax.naming.Name;
import javax.naming.NameParser;
import javax.naming.NamingEnumeration;
import javax.naming.NamingException;
import javax.naming.directory.Attribute;
import javax.naming.directory.Attributes;
import javax.naming.directory.DirContext;
import javax.naming.directory.InitialDirContext;
import javax.naming.directory.SearchControls;
import javax.naming.directory.SearchResult;
import javax.security.auth.Subject;
import javax.security.auth.callback.Callback;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.callback.NameCallback;
import javax.security.auth.callback.PasswordCallback;
import javax.security.auth.callback.UnsupportedCallbackException;
import javax.security.auth.login.FailedLoginException;
import javax.security.auth.login.LoginException;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

import org.apache.geronimo.security.jaas.JaasLoginModuleUse;
import org.apache.geronimo.security.jaas.WrappingLoginModule;
import org.apache.geronimo.security.realm.providers.GeronimoGroupPrincipal;
import org.apache.geronimo.security.realm.providers.GeronimoUserPrincipal;

public class IDAuthenticator {
   protected Log logger = LogFactory.getLog(this.getClass().getName());

    private Subject subject;
    private CallbackHandler handler;

    private String cbUsername;
    private String cbPassword;

    private boolean loginSucceeded;
    private final Set<String> groups = new HashSet<String>();
    private final Set<Principal> allPrincipals = new HashSet<Principal>();
    private LdapVisitor ldapVisitor = null;

    public IDAuthenticator(){
        ldapVisitor = new LdapVisitor();
    }

    public boolean login() throws LoginException {
        loginSucceeded = false;
        Callback[] callbacks = new Callback[2];

        callbacks[0] = new NameCallback("User name");
        callbacks[1] = new PasswordCallback("Password", false);
        try {
            handler.handle(callbacks);
        } catch (IOException ioe) {
            throw (LoginException) new LoginException().initCause(ioe);
        } catch (UnsupportedCallbackException uce) {
            throw (LoginException) new LoginException().initCause(uce);
        }
        cbUsername = ((NameCallback) callbacks[0]).getName();
        cbPassword = new String(((PasswordCallback) callbacks[1]).getPassword());

        if (cbUsername == null || "".equals(cbUsername)
                || cbPassword == null || "".equals(cbPassword)) {
            // Clear out the private state
            cbUsername = null;
            cbPassword = null;
            groups.clear();
            throw new FailedLoginException();
        }

        try {
            boolean result = authenticate(cbUsername, cbPassword);
            if (!result) {
                throw new FailedLoginException();
            }
        } catch (LoginException e) {
            // Clear out the private state
            cbUsername = null;
            cbPassword = null;
            groups.clear();
            throw e;
        } catch (Exception e) {
            // Clear out the private state
            cbUsername = null;
            cbPassword = null;
            groups.clear();
            throw (LoginException) new LoginException("LDAP Error").initCause(e);
        }

        loginSucceeded = true;
        return true;
    }

    public boolean commit() throws LoginException {
        if(loginSucceeded) {
            if(cbUsername != null) {
                allPrincipals.add(new GeronimoUserPrincipal(cbUsername));
            }
            for(String group: groups) {
                allPrincipals.add(new GeronimoGroupPrincipal(group));
            }
            subject.getPrincipals().addAll(allPrincipals);
        }
        // Clear out the private state
        cbUsername = null;
        cbPassword = null;
        groups.clear();

        return loginSucceeded;
    }

    public boolean abort() throws LoginException {
        if(loginSucceeded) {
            // Clear out the private state
            cbUsername = null;
            cbPassword = null;
            groups.clear();
            allPrincipals.clear();
        }
        return loginSucceeded;
    }

    public boolean logout() throws LoginException {
        // Clear out the private state
        loginSucceeded = false;
        cbUsername = null;
        cbPassword = null;
        groups.clear();
        if(!subject.isReadOnly()) {
            // Remove principals added by this LoginModule
            subject.getPrincipals().removeAll(allPrincipals);
        }
        allPrincipals.clear();
        return true;
```

```
}

protected void closeLdap(DirContext context) {
    try {
        if (ldapVisitor!=null)
            ldapVisitor.disconnect(context);
    } catch (Exception e) {
        logger.error(e);
    }
}

protected boolean authenticate(String username, String password) throws Exception {
    DirContext context = openLdap();
    try {
        NamingEnumeration results = ldapVisitor.search(context, username);

        if (results == null || !results.hasMore()) {
            return false;
        }

        SearchResult result = (SearchResult) results.next();

        if (results.hasMore()) {
            //ignore for now
        }
        String dn = ldapVisitor.getDn(context, result);

        ArrayList<String> roles = null;

        //check the credentials by binding to server
        ldapVisitor.bindUser(context, dn, username, password);
```

```
                roles = ldapVisitor.getRoles(context, result, dn, username, roles);
                for (String role : roles) {
                    groups.add(role);
                }
            } catch (CommunicationException e) {
                closeLdap(context);
                throw (LoginException) new FailedLoginException().initCause(e);
            } catch (NamingException e) {
                closeLdap(context);
                throw (LoginException) new FailedLoginException().initCause(e);
            }

            return true;
        }

        protected InitialDirContext openLdap() throws NamingException {
            return ldapVisitor.connect();
        }
}
```

<u>LdapVisitor.java:</u>

```
package org.tut.uia;
import java.text.MessageFormat;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.Hashtable;
import java.util.List;
import java.util.Map;
import javax.naming.AuthenticationException;
import javax.naming.Context;
import javax.naming.Name;
import javax.naming.NameParser;
import javax.naming.NamingEnumeration;
import javax.naming.NamingException;
import javax.naming.directory.Attribute;
import javax.naming.directory.Attributes;
import javax.naming.directory.DirContext;
import javax.naming.directory.InitialDirContext;
import javax.naming.directory.SearchControls;
import javax.naming.directory.SearchResult;
import javax.security.auth.Subject;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.login.FailedLoginException;

import org.apache.log4j.Logger;

public class LdapVisitor {

    private String initialContextFactory;

    private String connectionURL;

    private String connectionUsername;

    private String connectionPassword;

    private String connectionProtocol;

    private String authentication;

    private String userBase;
private String roleBase;

private String userRoleName;

private String qop;

private String roleName;

private static final String INITIAL_CONTEXT_FACTORY = "initialContextFactory";

private static final String CONNECTION_URL = "connectionURL";

private static final String CONNECTION_USERNAME = "connectionUsername";

private static final String CONNECTION_PASSWORD = "connectionPassword";

private static final String CONNECTION_PROTOCOL = "connectionProtocol";

private static final String AUTHENTICATION = "authentication";

private static final String USER_BASE = "userBase";

private static final String USER_SEARCH_MATCHING = "userSearchMatching";

private static final String USER_SEARCH_SUBTREE = "userSearchSubtree";

private static final String ROLE_BASE = "roleBase";

private static final String ROLE_NAME = "roleName";

private static final String ROLE_SEARCH_MATCHING = "roleSearchMatching";
```

```
    private static final String ROLE_SEARCH_SUBTREE = "roleSearchSubtree";

    private static final String USER_ROLE_NAME = "userRoleName";

    private static final String QOP = "qop";

    public final static List<String> supportedOptions = Collections
            .unmodifiableList(Arrays.asList(INITIAL_CONTEXT_FACTORY,
                    CONNECTION_URL,              CONNECTION_USERNAME,
CONNECTION_PASSWORD,
                    CONNECTION_PROTOCOL, AUTHENTICATION, USER_BASE,
                    USER_SEARCH_MATCHING,        USER_SEARCH_SUBTREE,
ROLE_BASE,
                    ROLE_NAME,                   ROLE_SEARCH_MATCHING,
ROLE_SEARCH_SUBTREE,
                    USER_ROLE_NAME, QOP));

    private Subject subject;

    private CallbackHandler handler;

    private MessageFormat userSearchMatchingFormat;

    private MessageFormat roleSearchMatchingFormat;

    private boolean userSearchSubtreeBool = false;

    private boolean roleSearchSubtreeBool = false;

    protected Logger logger = Logger.getLogger(this.getClass().getName());
protected InitialDirContext connect() throws NamingException {
    InitialDirContext context = null;
    try {
        Hashtable<String, Object> env = new Hashtable<String, Object>();
        env.put(Context.INITIAL_CONTEXT_FACTORY, initialContextFactory);
        if (connectionUsername != null || !"".equals(connectionUsername)) {
            env.put(Context.SECURITY_PRINCIPAL, connectionUsername);
        }
        if (connectionPassword != null || !"".equals(connectionPassword)) {
            env.put(Context.SECURITY_CREDENTIALS, connectionPassword);
        }
        env.put(Context.SECURITY_PROTOCOL, connectionProtocol == null ? ""
                : connectionProtocol);
        env.put(Context.PROVIDER_URL, connectionURL == null ? ""
                : connectionURL);
        if (authentication.indexOf("GSS") > 0) {

            authentication = "DIGEST-MD5"; // 强制使用 DIGEST-MD5 机制

        }
        //   env.put(Context.SECURITY_AUTHENTICATION,   authentication   ==
null ?
        // "" : authentication);
        env.put(Context.SECURITY_AUTHENTICATION, authentication);
        qop = qop == null ? System.getProperty("javax.security.sasl.qop",
                "auth") : qop;
        if (qop.indexOf("auth-conf") > 0) {

            qop = "auth-int"; // 强制降级为 auth-int

        }
        env.put("javax.security.sasl.qop", qop);
        // env.put("com.sun.jndi.ldap.trace.ber", System.err); //debug trace
        context = new InitialDirContext(env);
        } catch (NamingException e) {
            logger.error(e);
            throw e;
        }
        return context;
    }

    protected void disconnect(DirContext context) {
        try {
            context.close();
        } catch (Exception e) {
            logger.error(e);
        }
    }

    public void initialize(Subject subject, CallbackHandler callbackHandler,
                Map sharedState, Map options) {
        this.subject = subject;
        this.handler = callbackHandler;
        for (Object option : options.keySet()) {
            if (!supportedOptions.contains(option)) {
                logger.warn("Ignoring option: " + option + ". Not supported.");
            }
        }
        initialContextFactory = (String) options.get(INITIAL_CONTEXT_FACTORY);

        getConfig(options);

    }
    private void getConfig(Map options) {
        connectionURL = (String) options.get(CONNECTION_URL);
        connectionUsername = (String) options.get(CONNECTION_USERNAME);
        connectionPassword = (String) options.get(CONNECTION_PASSWORD);
        connectionProtocol = (String) options.get(CONNECTION_PROTOCOL);
        authentication = (String) options.get(AUTHENTICATION);
        if (authentication == null) {
            authentication = "";
        }
        userBase = (String) options.get(USER_BASE);
        String          userSearchMatching          =          (String)
  options.get(USER_SEARCH_MATCHING);
        String userSearchSubtree = (String) options.get(USER_SEARCH_SUBTREE);
        roleBase = (String) options.get(ROLE_BASE);
        roleName = (String) options.get(ROLE_NAME);
        String          roleSearchMatching          =          (String)
  options.get(ROLE_SEARCH_MATCHING);
        String roleSearchSubtree = (String) options.get(ROLE_SEARCH_SUBTREE);
        userRoleName = (String) options.get(USER_ROLE_NAME);
        qop = (String) options.get(QOP);
        userSearchMatchingFormat = new MessageFormat(userSearchMatching);
        roleSearchMatchingFormat = new MessageFormat(roleSearchMatching);
        userSearchSubtreeBool = Boolean.valueOf(userSearchSubtree);
        roleSearchSubtreeBool = Boolean.valueOf(roleSearchSubtree);
    }

    public void bindUser(DirContext context, String dn, String username, String password)
throws NamingException, FailedLoginException {
        if (this.authentication.indexOf("GSS")>0)
            context.addToEnvironment(Context.SECURITY_PRINCIPAL, dn);
        else
            context.addToEnvironment(Context.SECURITY_PRINCIPAL, username);
            context.addToEnvironment(Context.SECURITY_CREDENTIALS, password);
            try {
                context.getAttributes("", null);
            } catch (AuthenticationException e) {
                e.printStackTrace();
                logger.debug("Authentication failed for dn=" + dn);
                throw new FailedLoginException();
            } finally {

                if (connectionUsername != null) {
                    context.addToEnvironment(Context.SECURITY_PRINCIPAL,
                            connectionUsername);
                } else {
                    context.removeFromEnvironment(Context.SECURITY_PRINCIPAL);
                }

                if (connectionPassword != null) {
                    context.addToEnvironment(Context.SECURITY_CREDENTIALS,
                            connectionPassword);
                } else {

context.removeFromEnvironment(Context.SECURITY_CREDENTIALS);
                }
            }
    }

    public NamingEnumeration search(DirContext context, String username) throws
NamingException{
        String[] attribs;
        String filter = userSearchMatchingFormat.format(new String[]{username});
        SearchControls constraints = new SearchControls();
        if (userSearchSubtreeBool) {
            constraints.setSearchScope(SearchControls.SUBTREE_SCOPE);
        } else {
            constraints.setSearchScope(SearchControls.ONELEVEL_SCOPE);
        }
        if (userRoleName == null) {
            attribs = new String[]{};
        } else {
            attribs = new String[]{userRoleName};
        }
        constraints.setReturningAttributes(attribs);
        return context.search(userBase, filter, constraints);
    }

    public String getDn(DirContext context, SearchResult result) throws
NamingException{
        NameParser parser = context.getNameParser("");
        Name contextName = parser.parse(context.getNameInNamespace());
        Name baseName = parser.parse(userBase);
        Name entryName = parser.parse(result.getName());
        Name name = contextName.addAll(baseName);
        name = name.addAll(entryName);
        String dn = name.toString();
        return dn;
    }

    public ArrayList<String> getRoles(DirContext context, SearchResult result,
            String dn, String username, ArrayList<String> roles)
            throws NamingException {
        Attributes attrs = result.getAttributes();
        if (attrs == null) {
            throw new NamingException("SearchResult has no attribute.");
        }
        if (userRoleName != null) {
            roles = addAttributeValues(userRoleName, attrs, roles);
        }
        if (roles == null) {
            roles = new ArrayList<String>();
        }
        if (roleName == null || "".equals(roleName)) {
            return roles;
        }
        String         filter         =         roleSearchMatchingFormat.format(new
String[]{doRFC2254Encoding(dn), username});

        SearchControls constraints = new SearchControls();
        if (roleSearchSubtreeBool) {
            constraints.setSearchScope(SearchControls.SUBTREE_SCOPE);
        } else {
            constraints.setSearchScope(SearchControls.ONELEVEL_SCOPE);
        }
        NamingEnumeration roleResults =
                context.search(roleBase, filter, constraints);
        while (roleResults.hasMore()) {
            SearchResult roleResult = (SearchResult) roleResults.next();
            Attributes roleAttrs = roleResult.getAttributes();
            if (roleAttrs == null) {
                continue;
            }
            roles = addAttributeValues(roleName, roleAttrs, roles);
        }
        return roles;

    }
```

```
private ArrayList<String> addAttributeValues(String attrId,
        Attributes attrs, ArrayList<String> values) throws NamingException {

    if (attrId == null || attrs == null) {
        return values;
    }
    if (values == null) {
        values = new ArrayList<String>();
    }
    Attribute attr = attrs.get(attrId);
    if (attr == null) {
        return (values);
    }
    NamingEnumeration e = attr.getAll();
    while (e.hasMore()) {
        String value = (String) e.next();
        values.add(value);
    }
    return values;
}

protected String doRFC2254Encoding(String inputString) {
    StringBuffer buf = new StringBuffer(inputString.length());
    for (int i = 0; i < inputString.length(); i++) {
        char c = inputString.charAt(i);
        switch (c) {
            case '\\':
                buf.append("\\5c");
                break;
            case '*':
                buf.append("\\2a");
```

```
                                    break;
                              case'(':
                                    buf.append("\\28");
                                    break;
                              case')':
                                    buf.append("\\29");
                                    break;
                              case'\0':
                                    buf.append("\\00");
                                    break;
                              default:
                                    buf.append(c);
                                    break;
                        }
                  }
                  return buf.toString();
            }
      }
```

ExtAuthFilter.java:

```
package org.tut.uia;
import java.io.IOException;
import javax.servlet.Filter;
import javax.servlet.FilterChain;
import javax.servlet.FilterConfig;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import javax.servlet.http.Cookie;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.apache.log4j.Logger;

import java.util.Map;
import java.util.List;
import java.util.Iterator;
import javax.sql.DataSource;

public class ExtAuthFilter implements Filter {
    protected Logger logger = Logger.getLogger(this.getClass().getName());

    private FilterConfig filterConfig = null;

    /* (non-Java-doc)
     * @see java.lang.Object#Object()
     */
    public ExtAuthFilter() {
        super();
    }

    /* (non-Java-doc)
     * @see javax.servlet.Filter#init(FilterConfig filterConfig)
     */
    public void init(FilterConfig filterConfig) throws ServletException {
        this.filterConfig = filterConfig;
    }

    /* (non-Java-doc)
     * @see javax.servlet.Filter#doFilter(ServletRequest arg0, ServletResponse arg1,
FilterChain arg2)
     */
    public void doFilter(ServletRequest request, ServletResponse response,
            FilterChain chain) throws IOException, ServletException {
```

```
        HttpServletRequest httpRequest = (HttpServletRequest) request;
        HttpServletResponse httpResponse = (HttpServletResponse) response;
        Cookie[] cookies = httpRequest.getCookies();

        String username = httpRequest.getRemoteUser();
        String ipaddr = getIpAddr(httpRequest);
        TokenCreator tokenCreator = new TokenCreator();
        tokenCreator.setUsername(username);
        tokenCreator.setClientIpAddr(ipaddr);
        String token = tokenCreator.createToken();
         writeCookie(httpResponse, token);
        request.setAttribute("username", username);
        chain.doFilter(request, response);
    }

    private String getIpAddr(HttpServletRequest request){
        String ip = request.getHeader("x-forwarded-for");
            if(ip == null || ip.length() == 0 || "unknown".equalsIgnoreCase(ip)) {
                ip = request.getHeader("Proxy-Client-IP");
          }
            if(ip == null || ip.length() == 0 || "unknown".equalsIgnoreCase(ip)) {
                ip = request.getHeader("WL-Proxy-Client-IP");
           }
          if(ip == null || ip.length() == 0 || "unknown".equalsIgnoreCase(ip)) {
                ip = request.getRemoteAddr();
          }
          return ip;
    }

    private void writeCookie(HttpServletResponse httpResponse, String token){
            Cookie cookie = null;
            cookie = new Cookie("user.auth-token", token);
            httpResponse.addCookie(cookie);
        }

        /* (non-Java-doc)
         * @see javax.servlet.Filter#destroy()
         */
        public void destroy() {
        }
}
```