VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

FACULTY OF INFORMATION TECHNOLOGY

LDAP Server

AUTOR PRÁCE Ján Gula

AUTHOR

BRNO 2017

Abstrakt

Tento projekt je zameraný na implementáciu paralelného a neblokujúceho LDAP serveru pomocou sieťovej knihovny BSD sockets. Program je implementovaný v jazyku C++ . Server nepodporuje národné znaky a tiež nevyžaduje autentizáciu. Správy týkajúce sa autentizácie server ignoruje.

Abstract

This project is oriented on a implementation of a parallel non-blocking LDAP server using the BSD sockets network library. This program is implemented in the C++ language. Server does not support national characters and also does not require authentication. Messages about authentication are ignored.

Klíčová slova

Sieťové programovanie, Lightweight directory access protocol, server, sockety, C++, regulárne výrazy, port

Keywords

Network programming, Lightweight directory access protocol, server, sockets, C++, regular expressions, port

Contents

1 Introduction **2**

1.1 Task introduction 2

2 Implementation **3**

3 Search request implementation **4**

4 Functions explanation **5**

Literature **7**

Chapter 1

Introduction

Lightweight directory access protocol is an open application protocol for accessing and maintaining directory information over an Internet Protocol (IP). Similar to a telephone directory it stores certain information about users e.g. name surname address email etc. Our implementation expects our input file to contain three different types of information:

1. „cn“ – Represents the name and surname of a user
2. „uid“ – Represents a unique label consisting of eight characters starting with x, the following five characters are the first five characters of the users surname, or if the specific surname is shorter than five, the first letters of the users name are added, depending on the count of characters that are required to be filled to match the summary of five.
3. „mail“ – Represents the mail of a user. In our typical example mail consists of the users uid followed by our school domain.

1.1 Task introduction

The subject of this project is to make our own implementation of a LDAP server. Our task is to implement only specific functions of the original LDAP server used. For example our version does not have to support any authentication that means only **simple bind** needs to be implemented. Our implementation also does not support **national characters**. This is a part of an extension. The input file is expected to be a .csv (comma separated values) file. Messages and variables that are connected to the parts of the project that are not needed to be implemented can be ignored or pointed out by the server. We are not required to expect any other base value other than **scope** that means the whole file to be filtered.

The search request message contains a filter that determines which operators were sent by the client using his ldapsearch command. Brackets „()“ determine the range of each operator. An example of using the operator AND with multiple brackets that determine the association:

(&(cn=Gu*)(&(uid=xg*)(uid=*j00)))

Our assignment specifies only some of the **TLV filter operators**. The TLV filter in a search request message is truncated to five different types:

1. AND – example: (&(uid=xb*)(uid=x*za00))
2. OR – example: (|(uid=xb*)(uid=x*za00))
3. NOT – example: (!(cn=A*))
4. Substrings – example: (uid=xad*a*00)
5. Equality match – examples: (uid=xgulaj00)

Chapter 2

Implementation

This part describes my implementation of our task. The very first issue to solve was to store the input arguments. The expected format of our output binary was:

./myldap {-p <port>} -f <file>

This means that the input file will be set after the „-f“ option and that it is compulsory to fill this argument otherwise we call an error message. Curly brackets around our port arguments mean that we do not have to fill the port argument. The default value, if the port argument is not set is **389**. The –f parameter defines the path to a file ending with .csv (comma separated values) that describes a format where each information is separated by a comma.

My implementation parses input arguments using the function **getopt** that was also the suggest function by our assignment. If an argument value is set, getopt Stores this value into a special variable called **optarg**. To store the port number I have used the function **atoi** with a parameter optarg.

If the file name is empty my implementation prints an error to the stderr and returns a value of -1. The same situation is if the program fails to open the file. My implementation expects the encoding to be set to **UTF-8**.

After the file is opened I separate it with a delimiter „\n“ and store it to an array of data using the function **getline**. After this step we can close the file using function **close**. However having all information in one line did not seem comfortable for me so I have separated each line again but with a delimiter „;“ and stored it to an array. This means that the first line contains the common name(cn) of the user the second line contains the user ID(uid) and the third line contains the mail. However this means a rather complicated structure of data, but a nice way to access each type of information is to use the operator „%“. If we want to cycle through the array and access for example „uid“ we can do it simply by writing „if(i % 3 == 0)“.

To create a socket my implementation uses the **socket**() function with two parameters, first one defines the addres family to be internet(AF_INET) and the second one determines that we want a reliable sequential transmission control protocol(TCP) communication. To secure the fact that our server is non-blocking we have to set flags using the **fcntl**() function with flags **O_NONBLOCK**. Afterwards we set the server configuration to wait on every interface with the parameter **INADDR_ANY**. We the need to bind the server to the port number using the function **bind**. Now everything is set and we can enable the server to listen to incoming connection with the function **listen** with a queue of size two.

Now or server is waiting for incoming connections. This waiting is implemented by an infinite while(1) loop. We catch the incoming connections with the function

accept. When a connection comes we split the process to a parent and a child using the function **fork**. The parent's process is closed and the child's process continues to handle the incoming request.

Now comes the parsing part. We read the message that our binded client sent us on the socket mentioned earlier using the function **read** and store the request to a buffer. This is the tricky part as we have to read a buffer writen in hexadecimal numbers. We know that a simple bind communication starts with a bind request with a hexadecimal code 0x60. After receiving the whole bind request message we have to generate a bind response. This is done by a set of hexadecimal bytes that are mostly static. When we have our response written in the buffer we can use **write** to send it to the socket where our client is.

At this point we successfully managed to bind with the client and can now proceed to the hardest part of our whole project which is the search request message.

Chapter 3

Search request implementation

Search request determines a lot of information many of which we do not need in our implementation of a LDAP server. Unused attributes are for example baseObject, scope or typesOnly. BaseObject and scope are not required because our specification defines that the scope is our whole file. For our purposes we need only the sizeLimit attribute and Filter. Size limit is used in one of our functions that will be described later. Filter is the main source of information that we need to collect. It divides to ten different messages from which we are implementing five. To determine the first message we stored the hexadecimal value of the first byte before the filter itself. This value is followed by the length of the message. We have to store both of these information because they are used as parameters for the **logicalOperators()** function that I have implemented.

After we have read the filter value we sent it to a switch that decides which message was sent. For the three logical operators we have one function mentioned earlier. For **equalityMatch** and **substrings** we have separate functions but with the same parameters and return value. Input parameters for these functions are the loaded buffer that stores our hexadecimal data we want to read , data that we want to filter and the socket on which the client is listening.

Chapter 4

Functions explanation

searchResEntryGenerator()

This function is used to cycle through the array of results and to write each element of the array to a new packet and then send it to the client. In this function we also have to sort the result array alphabetically because in cases where we first wanted all users starting with „b“ and then used OR to add all users starting with „a“ the program did not sort the result array and would not act as the original LDAP server. This function is using a very bad sorting alternative, due to the fact that each user has three elements in the array that have to be switched. This is why we used the least complicated but most uneffective variant of sorting algorithms.

One entry consists of two most important strings, first one is the „cn“ of the user that is included in the result and the second one is the „mail“ of the user included. My implementation **does not include the „uid“ value** of the user because the assignment did not specify it as compulsory. With these information we also send the type of our information that is one of the three mentioned above. To do this we defined two strings with the types names. And then print them character by character into the result entry.

The hardest part of this function was to store exact lengths of each string and of each byte that the result entry includes since the client is very picky for buffer lengths and would throw an error message if the result entry was even a byte longer than expected.

Debugging of this function included tracking byte by byte in the wireshark application to find our missing hexadecimal value.

This function also checks if the size limit is not exceeded. By default we expect the **size limit to be zero** but it can be changed if the client specifies so in the search request. If the size limit is exceeded a global error flag is set.

This function returns void because it has only one purpose and that is to print the results to the client.

searchResDoneGenerator()

This function is simple, it generates the result done it differs only if a global error flag is set by searchResEntry.

equalityMatch()

This function reads the search request attribute description and the assertion value and cycles through the comma separated array comparing both attribute description that is one of the three information we collect and assertion value that is the value of our attribute that we compare to each element in our array data. After we successfully found the same value as assertion value we add three lines to the result array one for every information about the user that fulfills our query.

substrings()
This function is separated to three different states:

1. Initial : this part has to be at the start of the string uid=**xb***urza
2. Any : this part can be anywhere in the string uid=xb***u***rza
3. Final : this part has to be on the end of the string uid=xb*u***rza**

When we construct the regular expression defining the searched string we use **regex_match**() to find the matching string. This required to include a special header filed **<regex>**.
A problematic part of this function was to secure that the common name would be compared correctly using the regex function. The problem was that the regex would fail on a space between the name and the surname of the user. My implementation fixed this by removing the space between the names and creating a new array of common names that would be compared. We had to create a new array because we wanted to preserve the space in the result array. My implementation also transforms the data to lower case because the ldapsearch is case insensitive.

logicalOperators()
This functions was the hardest part of the project and is still not functioning absolutely correctly. I have implemented this as a while that cycles until the operator length is zero, this means we loaded every character belonging to the operator. In this while cycle we then pass the value of the filter to a switch that decides what comes next. Each case is separated into three parts each for every operator AND, OR, NOT. We can determine which operator was used to call this function by a function parameter opFlag, 0 if AND, 1 if OR, 2 if NOT.
The most complicated part is the merging of result arrays. In this function we have three different types of arrays:

1. **helpArr** = this array is contains the merged result of two functions for example if we have xb* and xa* with an AND operator the helpArr stores everyone starting with xb OR xa. This is needed later. The other usage of this array is to store the result of a logicalOperator that was called recursively. When the logicalOperators function ends it stores the result array to the helpArr.
2. **mergeResArr** = this array has only one purpose and that is to store the results of equalityMatch or substrings. Afterwards we compare each element in this array with each element in the helpArr and if we find two with the same value we know that this value should be the result of our current querry. Before we write it to the defResArr though, we have to check if it is not already there to avoid duplicates. This is why we cycle through the defResArr and check if we found it there if we did not we add it to the defResArr
3. **defResArr** = this is a definite result array we store the values that are same for the help array and the mergeResArr, if we have more than two consecutive and queries we have to clear this array because it could contain elements from the last query. Also if the mergeResArr size is zero we have to clear it too because anything AND nothing is always nothing.

Literature

[1] https://cwiki.apache.org/confluence/display/DIRxSRVx10/Ldap+ASN.1+Codec#LdapASN.1Codec-BindResponse

[2] https://wis.fit.vutbr.cz/FIT/st/course-files-st.php?file=%2Fcourse%2FISA-IT%2Ftexts%2Fkapitola2.pdf&cid=12191

[3] http://www.cplusplus.com/articles/