ISA project 2023

LDAP Server implementation in C++

Author: Matyáš Strelec <xstrel03@stud.fit.vutbr.cz>

Description

The assignemnt was to study the LDAP protocol and implement a server that would respond to clients' requests. The server is implemented in C++ in the C++20 standard. The code is in the `/src` subdirectory, the documentation is in the `/doc` subdirectory, and the tests are in the `/tests` subdirectory.

LDAP Protocol

The Lightweight Directory Access Protocol (LDAP) is an open, vendor-neutral, industry standard application protocol for accessing and maintaining distributed directory information services. [3]

A client starts an LDAP session by connecting to an LDAP server. The client then sends an operation request to the server, and a server sends responses in return. With some exceptions, the client does not need to wait for a response before sending the next request, and the server may send the responses in any order. All information is transmitted using Basic Encoding Rules (BER).

The protocol provides an interface with directories that follow the 1993 edition of the X.500 model:

An entry consists of a set of attributes. An attribute has a name (an attribute type or attribute description) and one or more values. The attributes are defined in a schema (see below). Each entry has a unique identifier: its Distinguished Name (DN). [5][6]

BER encoding

LDAP is a binary protocol, which helps make it compact and efficient to parse. The particular binary encoding that it uses is based on ASN.1 (Abstract Syntax Notation One), which is a framework for representing structured data. LDAP uses the Basic Encoding Rules (BER). In ASN.1 BER, each piece of data is called an element.

In BER encoding, different types of data can exist, including primitive types like integers, strings, and booleans, as well as constructed types such as sequences and sets. These data types are encoded using a tag-length-value format.

Tag: It specifies the type of data or the data element. The tag is typically a numerical identifier that indicates the meaning of the data that follows.

Length: It indicates the number of octets (bytes) or elements in the Value field. The length can vary, allowing for flexibility in representing different amounts of data.

Value: This field contains the actual data, which can be of variable length based on the information provided in the Length field. It holds the content associated with the Tag. [4]

Usage

The server can be compiled using `make` command which produces a binary called `isa-ldapserver`. The server can be as follows:

```
isa-ldapserver {-p <port>} -f <file>
```

- `-p <port>` — port to listen on (optional, default `389`)
- `-f <file>` — input file with data in CSV format

Implementation

The server is implemented in C++ in the C++20 standard. The code is structured followingly: - `main.cpp` — main function, parses arguments and calls the server - `server.cpp/h` — server implementation, handles requests, sends responses - `csv.cpp/h` — read and parse a CSV file - `ber.cpp/h` — BER encoding and decoding - `ldap.cpp/h` — LDAP protocol implementation, building and parsing messages - `filter.cpp/h` — LDAP filter implementation, parsing and evaluating filters

Main function

The entry point to the program. It parses command line arguments, handles wrong arguments, gets the input file and calls the server.

CSV

Reads the database from a CSV file separated by semicolons. The file is read line by line, and each line is split into fields separated by the delimiter. The fields are then stored as a vector of vectors of strings - each entry is a string and each line is a vector of strings, the lines make up the file.

Server

The server receives the database and port as arguments. First, it sets up the TCP server. It can be binded on either IPv4 or IPv6, the connection is non-blocking. The server then enters an infinite loop where it waits for incoming connections. When a connection is received, it is accepted and the server is forked to handle the connection.

The server is based on the implementation from IPK subject stubs on the FIT Gitea. [1]

The `ldapserver()` function is the finite state machine that handles the connection. It receives the data from the client, parses it to the class and handles any errors. It then builds a response and sends it back to the client.

First, the `BindRequest` is received. The server checks if the request is valid, if it is, it sends a `BindResponse` with the result code `success`. In a loop, it then receives a `SearchRequest` and sends a `SearchResultEntry` for each entry that matches the search criteria. Every entry is matched against the filter, if present, and if it matches, it is sent to the client. When all entries are sent, a `SearchResultDone` is sent to the client and the connection is closed. The server can now send either an `UnbindRequest` or another `SearchRequest` to the client. If the client sends an `UnbindRequest`, the server closes the connection and exits. If the client sends another `SearchRequest`, the server loops back and handles it.

Other message types are not implemented, and the server exits with an error code, closing the connection.

After the connection is finished, the parent process waits for child processes, and when all child processes are finished, the sockets are closed and the server exits.

BER

The `BERreader` class is defined here. It contains a vector of unsigned chars (bytes) which it reads from and an iterator which keeps track of the current position. The class is used to read BER encoded data. It contains methods to read integers, strings, booleans, sequences, strings, enums, filters and other BER types.

The `read_filter()` method is particularly interesting. It reads a filter from the input and returns a `filter` object. The bytes of the filter are recursively read, depending on the tag of the filter. For every filter, mutliple fields of attributes can be used. Therefore, it reads as much as is actually needed, and parses into a filter object, which can contain other filters in it.

The `BERwriter` class is also defined here. It contains a vector of unsigned chars (bytes) and is used to write BER encoded data. It contains methods to write integers, strings, booleans, sequences, strings, and other BER types.

The methods have vectors of bytes as outputs, which are then used in LDAP class to create messages.

LDAP

The main class here is the LDAPMessage, which is the parent class for every message. It contains getters for the fields that every message has, such as the message ID. The `LDAPMessage` class is then inherited by the `Request` and `Response` classes.

Requests The `Request` class is used to read messages from the client. It contains methods to read the message using the `BERreader` class. The `Response` class is used to build messages using the `BERwriter` class.

Each of the `BindRequest`, `SearchRequest`, and `UnbindRequest` classes inherit from the `Request` class. They contain getters for their specific fields, and the `parse()` method which parses the message using the `BERreader` class.

The `parse()` method reads the components in the specific order as defined in the LDAP protocol. [2]

Responses The `Response` class is used to build messages. It contains methods to build the message using the `BERwriter` class.

The `BindResponse`, `SearchResultEntry` and `SearchResultDone` classes inherit from the `Response` class. They contain methods to set the messages attributes.

With the attributes filled, `build()` method can be called to build the message using the `BERwriter` class, returning a vector of bytes. The `build()` method writes the components in the specific order as defined in the LDAP protocol. [2]

Filter

The `filter` structure contains the filter type, the data used by the `EqualityMatch` and `Substrings` filters, and a vector of filters used by the `And`, `Or` and `Not` filters. The filter is parsed in the `BERreader` class when creating the class of the request. [2]

Evaluation The `match_filter()` function is used to match a filter against an entry. It takes the filter and the entry, and returns a boolean value whether the entry matches the filter.

The `EqualityMatch` and `Substrings` filters are matched directly against the entry's attributes. For the `EqualityMatch` filter, the attribute must be equal to the filter's value. For the `Substrings`, the attribute is first compared to the `initial` substring, and then to the `final` substring. Then, `any` substrings in the middle are compared to the attribute are matched. If the fitler also contains the `initial` or `final` substrings, they are removed from the attribute before matching, to not create overlapping false positives. Same for every `any` substring, to not match it twice and keep their order.

The `And`, `Or` and `Not` filters are matched recursively. They go through all the filters in their vector, evaluating every one against the entry. For the `And` and `Or` filter, all are compared together and the result is returned. For the `Not` filter, the result is negated.

Limitations

The server handles only the requests that are defined in the assignment. It does not handle any other requests, and exits with an error code if it receives any other request.

The server can only work with databases with data in ASCII format. It does not support any other encoding (such as UTF-8).

The server can only use the `simple` authentication method. It does not support any other authentication method.

The substring filter can sometimes fail, particurarly when using more than 3 `any` substrings. I wasn't yet able to fix this bug.

Testing

The server was tested using the `ldapsearch` command. During development, I was testing the parts that I was writing code for manually, comparing the output of my server with the one of `ldap.fit.vutbr.cz` server.

I was capturing the network traffic with Wireshark and comparing the hexdump packets.

For automated testing, I wrote a Python script that will take the input file located in `tests'/testing.txt`, parse the filter and expected number of entries, and send a request to the server. The server will then send a response, which is parsed by the script and compared to the expected number of entries. The script then prints out the result of the test.

The tests helped me find a bug in the evalutaing of the `Substrings` filter, where the `any` substrings were not matched correctly when more than 3 `any` substrings are present.

The server was tested on `merlin`, GCC 10.5 and macOS 14, clang 15.0.0. It works on both.

References

[1]: VESELÝ, Vladimír. NESFIT/IPK-Projekty/Stubs at master - IPK-Projekty - FIT - VUT Brno - git [online]. [cit. 2023-11-20].

Avaiable at: https://git.fit.vutbr.cz/NESFIT/IPK-Projekty/src/branch/master/Stubs/cpp

[2]: J. SERMERSHEIM, Ed. RFC4511. Lightweight Directory Access Protocol (LDAP): The Protocol [online]. 2006 [cit. 2023-11-20].

Avaiable at: https://www.rfc-editor.org/rfc/rfc4511

[3]: MATOUŠEK, Petr. Síťové aplikace a správa sítí. Poštovní a adresářové služby [online]. 2023 [cit. 2023-11-20].

Available at: https://moodle.vut.cz/pluginfile.php/707865/mod_resource/content/4/isa-mail.pdf

[4]: WILSON, Neil. LDAPv3 Wire Protocol Reference: The ASN.1 Basic Encoding Rules. LDAP.com [online]. 2023 [cit. 2023-11-20].

Avaiable at: https://ldap.com/ldapv3-wire-protocol-reference-asn1-ber/

[5]: ELLINGWOOD, Justin. Understanding the LDAP Protocol, Data Hierarchy, and Entry Components. DigitalOcean [online]. 2023 [cit. 2023-11-20].

Avaiable at: https://www.digitalocean.com/community/tutorials/understanding-the-ldap-protocol...

[6]: Lightweight Directory Access Protocol. Wikipedia.org [online]. [cit. 2023-11-20].

Avaiable at: https://en.wikipedia.org/wiki/Lightweight_Directory_Access_Protocol