BACHELOR THESIS

Ondřej Hrubý

A Minimalistic Directory Service

Department of Applied Mathematics

Supervisor of the bachelor thesis: Mgr. Martin Mareš, PhD.
Study programme: Computer Science
Study branch: IOI

Prague 2018
I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

Prague, 19th July 2018

signature of the author
Title: A Minimalistic Directory Service

Author: Ondřej Hrubý

Department: Department of Applied Mathematics

Supervisor: Mgr. Martin Mareš, PhD., Department of Applied Mathematics

Abstract: We present a light, simple and secure network protocol for accessing directory services called Feather-weight Directory Access Protocol (FDAP). It is inspired by Lightweight Directory Access Protocol (LDAP), but the concept of a directory service is built from scratch. This decision is supported by analysis of shortcomings of LDAP which has seen widespread use in the past. We provide specifications both of an FDAP service and the protocol, examples of well-tested server implementation, client library and an application as a proof of concept.

Keywords: FDAP, LDAP, X.500, Directory services
At this place, I would like to thank to all people who have been supporting me during my studies and have been helping me with this thesis in various ways. Especially, I would like to thank:

- My mother and family for their constant and faithful support, thanks to it I could freely focus on my studies.

- Mgr. Martin Mareš PhD. for his patience and favor during the assignment of this thesis, as well as during my studies when helping me to solve diverse problems and study affairs. For his never-ending, original and novel ideas which he is always glad to share with others.

- David, as always, for supporting me in the decision to start working on this thesis at all, and to go to the uncharted lands of UNIX and C language. Also thank you for your professional advisement during the working process.

- My friends, Vojtěch and Milan, for pleasant distraction in the times of hard work.

This thesis is dedicated to

F. Liszt’s Transcendental Étude #11 (Harmonies du soir)
– the grand piano piece
which I was working on along with the thesis.

Figure 0: Franz Liszt, Harmonies du soir [Lis51] – main theme
Contents

Introduction .................................................. 5
  Thesis Goals .............................................. 5
  Thesis Layout ............................................ 5

1 Problem Analysis ........................................... 7
  1.1 Task of the Directory Service ....................... 7
  1.2 Network Information Service (NIS) .................. 8
    1.2.1 Service Overview ................................ 8
    1.2.2 Security Issues ................................... 9
  1.3 Lightweight Directory Access Protocol (LDAP) ..... 10
    1.3.1 Hell of Standards ................................ 10
    1.3.2 LDAP Complexity .................................. 12
  1.4 Summary ............................................... 15

2 Feather-weight Directory Access Protocol (FDAP) .... 17
  2.1 Service and Protocol Requirements .................. 17
  2.2 CBOR .................................................. 18
    2.2.1 Brief Introduction ............................... 18
    2.2.2 Reasons for the Choice ......................... 19
  2.3 Directory Data Model ................................ 20
    2.3.1 Entries and Attributes ........................... 20
    2.3.2 Identification of Attributes ..................... 20
    2.3.3 Identification of Entries ......................... 20
    2.3.4 Referencing the Entries .......................... 21
    2.3.5 Typing of Entries ................................ 21
    2.3.6 Authenticable Entries ............................ 21
    2.3.7 Comparison of Attribute Values ................... 23
  2.4 Filters ............................................... 24
  2.5 Communication Protocol .............................. 28
    2.5.1 Transport .......................................... 28
    2.5.2 Request Format .................................... 30
    2.5.3 Response Format .................................. 31
  2.6 Operations ............................................ 32
    2.6.1 Authenticate ....................................... 32
    2.6.2 Search ............................................ 34
    2.6.3 Get ............................................... 35
    2.6.4 Create ............................................ 35
    2.6.5 Update ............................................ 35
    2.6.6 Delete ............................................. 36

3 Implementation ............................................ 37
  3.1 Overall Architecture .................................. 37
  3.2 Source Code ........................................... 39
    3.2.1 Programming Language ............................ 39
    3.2.2 Authorship ........................................ 39
3.2.3 Directory Source Tree ........................................ 40
3.2.4 Building the Source Code .................................. 40
3.3 CBOR Encoder and Decoder ..................................... 41
  3.3.1 Usage ...................................................... 41
  3.3.2 Application ............................................... 42
  3.3.3 Known Limitations ....................................... 42
3.4 FDAP Daemon .................................................. 43
  3.4.1 I/O ....................................................... 43
  3.4.2 Storage Abstraction ...................................... 44
  3.4.3 Usage ..................................................... 45
  3.4.4 Configuration ........................................... 45
3.5 FDAP Client Library ........................................... 48
  3.5.1 Application Programming Interface (API) ............... 48
  3.5.2 Usage ..................................................... 48
  3.5.3 Configuration ........................................... 48
3.6 FDAP Command-line Utility ................................... 50
  3.6.1 Usage ..................................................... 50

4 Client Application Example ..................................... 53
  4.1 Name Service Switch (NSS) ................................ 53
    4.1.1 Databases .............................................. 53
    4.1.2 Services ............................................... 55
    4.1.3 NSS-FDAP Module ...................................... 57
  4.2 Pluggable Authentication Modules (PAM) ................... 59
    4.2.1 Application API ....................................... 59
    4.2.2 Modules ............................................... 61
    4.2.3 PAM-FDAP Module ...................................... 62

Conclusion .......................................................... 63

Appendix A Filter Textual Notation Syntax ....................... 65
  A.1 Non-terminals .............................................. 65
  A.2 Terminals .................................................. 67

Appendix B API Documentation .................................. 69
  B.1 src/cbor.c .................................................. 69
  B.2 src/fdapd.c ............................................... 73
    B.2.1 TCP/IP listening sockets ................................ 74
    B.2.2 UNIX domain listening sockets ......................... 74
    B.2.3 FDAPc (client) sockets ................................ 75
    B.2.4 FDAPd (downstream) sockets ............................ 76
    B.2.5 epoll()-based main loop ................................ 76
  B.3 src/include/array.h ........................................ 76
  B.4 src/include/cbor.h ......................................... 77
    B.4.1 CBOR stream ............................................ 78
    B.4.2 Generic items ......................................... 79
    B.4.3 Integers ............................................... 80
    B.4.4 Text streams .......................................... 82
    B.4.5 Byte streams .......................................... 83
Introduction

Nowadays, Lightweight Directory Access Protocol (LDAP) is a standard network protocol, used to access directory services over the network. Task of a directory server is to maintain a database of entries (for example contact information, such as e-mail addresses), and to serve this content to the end user or application – for example mail user agent (MUA).

LDAP is called lightweight, but, indeed, it is quite complex protocol. And the specification of directory services, which is inherently tied up with LDAP, is even more complex and over-engineered.

Although LDAP directory is capable to store generic data, it is frequently used to store user accounts data with the ability to authenticate the users. We will adopt this use-case and will present a new protocol called Feather-weight Directory Access Protocol (FDAP). In order to do this, we will slightly redefine the meaning of the directory term in context of FDAP. This will lead to design of a service with less general storage capabilities, compared to LDAP, yet simpler, and certainly more lightweight. And what could be more lightweight than a feather?

Thesis Goals

(G1) To provide thorough analysis of LDAP and explain why it is complex, robust and therefore inappropriate solution for a particular (yet common) use-case.

(G2) To design and describe the FDAP protocol to present a simpler and lighter alternative to LDAP, which would fit the mentioned use-case well.

(G3) To implement proof-of-concept FDAP server, FDAP client library and supply nice documentation.

(G4) To illustrate client applications in practice – implement FDAP module for NSS and PAM system facilities.

Thesis Layout

In the first chapter, we will describe directory services in general. We will look at the use-cases, significant directory access protocols, and implementations over last 35 years. At the end of the first chapter, we will provide a brief summary of our research, and we will point out eventual insufficiencies of currently used protocols, and the service definition itself.

In the second chapter, we will propose a new solution of the previously discussed problem. We will define both the service, and the protocol. We will describe the service data model, the functional model and show, how to encode all protocol elements.

Next, we will go through the proof-of-concept implementation of the newly designed protocol. We will try to provide a big picture of what we have implemented, describe the important parts, and explain how do they cooperate together. We
will also focus on pitfalls that we have met during the working process. Detailed API reference of the important modules can be found in appendix B.

In the last chapter, we will give summary of internal architecture of UNIX-specific mechanisms and facilities that will allow us to integrate our directory service with the rest of the operating system. Namely, Name Service Switch (NSS), and Pluggable Authentication Modules (PAM). We will describe installation and usage of our FDAP modules for these two facilities.

Finally, we will present conclusion about what we have done and what we have not.

There are two appendices attached at the end of the thesis. The first is a formal syntax definition of filter textual notation, developed as a part of the protocol. The second appendix is the API Reference, already mentioned.
Chapter 1
Problem Analysis

1.1 Task of the Directory Service

A directory service is a quite old concept. Its origins date back to the beginning of modern Internet. As the Internet started to grow in early 1980s, and multiple networks have been merged into the global, bigger one, people and machines needed to somehow identify and locate resources on the network. The resources were of various kinds – such as computers, printers, other services, user accounts, contact information (e-mail addresses, phone numbers), and so on. The common characteristics of these resources were (and still are) that they are usually small pieces of information needed by other services, placed in an extensive infrastructure.

The directory servers are a part of this infrastructure, and the rest of the infrastructure relies on them in order to be operational. Typical nature of the directory service data is system-like. This contrasts with today’s database servers (such as SQL databases) which, typically, house database applications and hold domain-specific data, accessed by end-user applications. As a consequence, directory services are massively optimized for read operations, at the expense of slower write operations. This is because the directory contents change almost only when the infrastructure changes, which is nearly not at all, compared to the number of read accesses to the directory.

Consider the following situation: A user wants to send an e-mail to some other user. In order to do this, a lot of things must happen in the background. First, the user must login to the machine, from which they want to send the e-mail. Then imagine that the user does not know the destination e-mail address; the only thing they remember is the target user’s surname. They have to search the e-mail address within a directory, prior to sending the e-mail.

![Figure 1.1: Directory server operating inside an infrastructure](image-url)
1. The user wants to login: First, Domain Name System (DNS) look-up for the IP address of the directory server must be done, before the directory server can be contacted.

2. The user wants to login: Second, the directory server is contacted to authenticate the user.

3. The user wants to search for an e-mail address: Again, the DNS server may be contacted to translate directory server’s name.

4. The user wants to search for an e-mail address: The directory server is contacted to search for an e-mail address by surname.

5. The user wants to send an e-mail: Mail is sent by a mail server, using SMTP.

Why have we mentioned DNS? It could be, according to our informal definition, also sort of directory service. And, in fact, it really is. Actually, it can be considered the first directory service ever. First DNS specifications were published in 1983 as RFC 882 [Moc83a], and RFC 883 [Moc83b]. Due to whatever reasons, DNS has developed independently of other directory services. Most likely, the reasons are merely historical, as at that time, there was a need for name resolution service, so IETF has just made one. Further directory services were developed later, and, as it happens, by various initiatives, aimed at different fields of interest, but sharing several common characteristics.

1.2 Network Information Service (NIS)

Network Information Service was developed around 1985 at Sun Microsystems. It is also known as Yellow Pages.

1.2.1 Service Overview

NIS uses a client-server model with potentially multiple servers, ensuring failover redundancy. One of the servers must be a master; other servers are called slaves. Master maintains the primary directory database. All changes to the database are made to master. The slaves just hold a replica of the master’s database. They are notified each time a change in the master occurs.

A key concept to NIS are domains. A domain is identified by a textual identifier. It represents a logical group of computers (both servers and clients), which have set the same identifier. Computers in the same domain are allowed to exchange information, which comprises a serious security risk, as there is no other method of authentication, nor server validation. Note that these domains have nothing to do with DNS domains.

---

1 Until that time, all domain name registrations had to been done manually by changing a file, named HOSTS.TXT, which was handled by John Postel at that time.

2 Because British Telecom PLC has registered Yellow Pages as a trademark for the paper telephone directory, Sun was forced to rename the service to NIS.
Data in the database are stored as key-value pairs, in NDBM format [Ray01]. A set of the key-value pairs is called a map. Each UNIX system database (for now, let us consider a list of system users as an example) corresponds to some map, but the maps are not limited only to system databases. Maps are an inherent way in which NIS performs indexing of the data. If one wants to enable indexing on a new data field, another map must be built with the data fields as the keys. In order to do this, the directory database must be rebuilt, which requires the NIS daemon to go offline for the rebuild time. The same must happen in order to add a new directory entry on master, as the primary data source is not the NIS database itself, but the system database (usually a text file, like /etc/passwd). The change is made to the file (for example by executing useradd command on master), and the database must be rebuilt from the text file.

NIS uses Remote Procedure Call (RPC) to execute its operations. Several operations are specified in the NIS protocol. A port mapper daemon must be installed on server the to demultiplex incoming RPC calls to target applications. The mapper listens, usually on TCP port 111, for incoming RPC requests, that contain an RPC application number. The port mapper then redirects these requests to the port, where target RPC service is listening. In order to do this, the port mapper must be aware of the target service port. This is ensured by that all RPC server applications contact the port mapper during their start-up and a port is assigned to them. Note that, despite this adds an extra layer of complexity to the design of the service, it is also a weak point, as it can be misused by DDoS attackers [net15] to amplify the effects of the attack, because the response of port mapper returns many times bigger amount of data.

1.2.2 Security Issues

There are well-known security issues with NIS [O’B00]. We will try to summarize them here:

Broadcast Mode

When a client needs to retrieve an information from a NIS server, it must first bind to it. This is done by client utility ypbind, and this can be done in two modes. In server-list mode, client simply examines a pre-configured list of servers and it tries to bind to one of them. In broadcast mode (which is more typically used), a suitable NIS server in local-area network is found by issuing a link broadcast. The client binds to the first server which replies to the broadcast, and which is part of the same NIS domain. As the traffic is not encrypted, an attacker can easily read information, such as real NIS server’s IP address, NIS domain name, or sensitive data, such as password hashes.

Absence of Client Authentication

Anybody, who knows NIS server name and NIS domain name, can configure its device as a NIS client. NIS does not enforce any authentication methods that

---

3 Assuming, they do not want to perform sequential search.
4 Assuming, there is not other security mechanism installed at the network level (such as IPSEC).
would verify client’s identity. Once a client establishes a connection to a NIS server, it has unlimited access to data stored in NIS maps within the directory. Note there exist some configuration options by which the access can be restricted, but they are not straightforward, nor set by default configuration options.

Absence of Server Validation

A NIS client does not perform any validation of the server’s identity, other than checking that the server belongs to the same NIS domain as the client. As the result, anyone can act as a fake NIS server, just with knowledge of the NIS domain name. Together with the broadcast mode operation, it is very easy to attract clients in the network and spread false information across the network. In combination with a Denial of Service attack on a real NIS master, which would effectively eliminate the server off its operation, this could lead to takeover of the entire control of the directory infrastructure.

1.3 Lightweight Directory Access Protocol (LDAP)

The Lightweight Directory Access Protocol (LDAP) was developed around 1993 as a simpler and lighter alternative to Directory Access Protocol (DAP), which was defined as a part of X.500 standards by ITU-T\(^5\) in 1988. DAP was built on the top of the ISO/OSI protocol stack as an application layer (L7) protocol. Therefore one of the reasons to design LDAP was to adapt DAP to TCP/IP architecture. Purpose of both protocols is the same—to provide unified interface to X.500 directory server over the network.

Directory services in X.500 standards were primarily intended as a supplement of X.400 recommendations (ITU-T; first issued in 1984), which describe mainly electronic mail exchange services.

1.3.1 Hell of Standards

Because it is far over 20 years since LDAP was first introduced, and this evolution has non-negligible impact on its current shape, it seems appropriate to introduce the reader to concise history of the protocol. We will also give a brief summary of relevant LDAP standards for easier orientation in available literature.

The first LDAP version was meant as protocol proxy to a directory servers already accessible through DAP. Without LDAP, one had to implement whole heavy ISO/OSI protocol stack on the client side to access a DAP server. With widespread use of TCP/IP during late 1980s a need arose to reach already deployed X.500 DAP servers via TCP/IP. LDAP has come to fulfill this need. So DAP servers used LDAP as a front-end API to communicate with clients via TCP/IP, while keeping DAP as a back-end, translating LDAP client requests.

\(^5\)In 1988, the name of the organization used to be CCITT (Comité Consultatif International Téléphonique et Télégraphique). It was renamed in 1993 to ITU-T (International Telecommunication Union—Telecommunication Standardization Sector).
into DAP requests through ISO/OSI stack. There was no one-to-one correspondence kept in the translations, so even one LDAP request could be translated into more DAP requests. This has formed the background, why LDAP is called lightweight\footnote{Compared to DAP it really is.}, as cited from RFC 1487 \cite{YHK93}:

- “Protocol elements are carried directly over TCP or other transport, bypassing much of the session/presentation overhead.”
- “Many protocol data elements are encoding as ordinary strings (e.g., Distinguished Names).”
- “A lightweight BER encoding is used to encode all protocol elements.”

To show, that X.500 standards are not kind of “good night tales” reading, it is enough to look at the amount of pages of the standards, as shown in figure 1.2.

<table>
<thead>
<tr>
<th>Standard name</th>
<th># of pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>X.500: Overview of concepts, models and services</td>
<td>32</td>
</tr>
<tr>
<td>X.501: Models</td>
<td>262</td>
</tr>
<tr>
<td>X.509: Public-key and attribute certificate frameworks</td>
<td>254</td>
</tr>
<tr>
<td>X.511: Abstract service definition</td>
<td>130</td>
</tr>
<tr>
<td>X.518: Procedures for distributed operation</td>
<td>136</td>
</tr>
<tr>
<td>X.519: Protocol specifications</td>
<td>94</td>
</tr>
<tr>
<td>X.520: Selected attribute types</td>
<td>126</td>
</tr>
<tr>
<td>X.521: Selected object classes</td>
<td>41</td>
</tr>
<tr>
<td>X.525: Replication</td>
<td>46</td>
</tr>
<tr>
<td>X.530: Use of systems management for administration of the Directory</td>
<td>88</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>1209</strong></td>
</tr>
</tbody>
</table>

Figure 1.2: X.500 Standards: Number of pages summary

As stated before, X.500 standards were first issued at 1988 by ITU-T. LDAP was first introduced in 1993, in RFC 1487, due to mentioned reasons. In 1993, also X.500 has evolved significantly to encompass many new key features. Finally, in 1996, LDAP Version 3 was defined in several RFCs to allow access to the X.500 (1993) features. We can say that LDAP was tracking X.500. LDAPv3 is the latest version of the protocol. RFC 4510 \cite{Zei06b} is “table of contents” of technical specification of LDAPv3 – it contains links to all relevant RFCs, according to the protocol specification (see figure 1.3).

These RFCs reference X.500 specification, mainly standards X.501 (The Directory: Models) and X.511 (The Directory: Abstract service definition). Despite these references, all important things are redefined (possibly duplicated) in LDAP RFCs from scratch. We should think about LDAP as an interface protocol to an X.500 directory server. The whole idea of directory services was already described in X.500 standards. Hence it is not surprising that many LDAP characteristics have their origin in X.500, especially its data model. These include, for example:
• Hierarchical directory tree structure and Distinguished Names (DN) format
• Typing of entries – the concept of object classes and attribute types
• Idea of directory schemas and its definition syntax.
• Set of operations exposed by the protocol

LDAP: The Protocol [RFC4511]
LDAP: Directory Information Models [RFC4512]
LDAP: Authentication Methods and Security Mechanisms [RFC4513]
LDAP: String Representation of Distinguished Names [RFC4514]
LDAP: String Representation of Search Filters [RFC4515]
LDAP: Uniform Resource Locator [RFC4516]
LDAP: Syntaxes and Matching Rules [RFC4517]
LDAP: Internationalized String Preparation [RFC4518]
LDAP: Schema for User Applications [RFC4519]

Figure 1.3: List of RFCs describing LDAP, taken from RFC 4510

LDAP  
(DAP)  
X.500  
X.501  
X.511

Figure 1.4: LDAP and X.500 standards relationship diagram

1.3.2 LDAP Complexity

We have shown, that LDAP is based upon X.500 standards. Now, we will focus on particular examples of characteristics, that make LDAP too robust and complex.

Schema Complexity

Each entry stored in the LDAP directory, belongs to one or more object classes. An object class is a set of rules describing an entry, which can enforce and define several data fields (attributes), that must, or may, be present in the entry. Many complex relationships and rules can be expressed using object classes, such as data types, rich comparison semantics of data values, object class inheritance, etc. A schema is simply a set of object classes, attributes, and other definitions, with a name assigned. An LDAP server is configured to maintain entries that conform to specific schema. The definition of the schema is a part of the server’s
configuration. Before any LDAP operation, which potentially modifies contents of the directory, is committed, the resulting change is verified against the schema, and if the verification fails, the change is denied, the operation is not executed, and an error response is sent to the client. By this mechanism, it is ensured that illegal entries (e.g., with missing required attributes) are not inserted into the directory.

That is not a bad idea in its roots, but let's look at an example of a concrete schema definition (see figure 1.5). As we can see from the example, the schema definition language is not very straightforward. Many references to external entities are made through OIDs[7] The syntax itself is not clean and allows to define very advanced behavior (syntax of attribute values, matching rules, etc.). It could be interpreted as a feature of the protocol. But in my opinion it is a misfeature. Let us think about consequences.

2.39. 'uid'

The 'uid' ('userid' in RFC 1274) attribute type contains computer system login names associated with the object. Each name is one value of this multi-valued attribute.

(Source: RFC 2798 [RFC2798] and RFC 1274 [RFC1274])

```plaintext
( 0.9.2342.19200300.100.1.1 NAME 'uid'
   EQUALITY caseIgnoreMatch
   SUBSTR caseIgnoreSubstringsMatch
   SYNTAX 1.3.6.1.4.1.1466.115.121.1.15 )
```

1.3.6.1.4.1.1466.115.121.1.15 refers to the Directory String syntax [RFC4517].

Examples: "s9709015", "admin", and "Administrator".

3.4. 'device'

The 'device' object class is the basis of an entry that represents an appliance, computer, or network element.

(Source: X.521 [X.521])

```plaintext
( 2.5.6.14 NAME 'device'
   SUP top
   STRUCTURAL
   MUST cn
   MAY ( serialNumber $ seeAlso $ owner $ ou $ o $ l $ description )
)
```

**Figure 1.5:** LDAP basic schema definition (RFC 4519 [Sci06])

---

[7] For example 1.3.6.1.4.1.343. Object Identifier (OID) format is defined in ITU-T X.660 standard. The OIDs form a tree hierarchy with several roots. Each root is governed by particular organization (ITU-T, ISO, etc.), which is responsible for assigning the OIDs.
If you give people an extension point in the hand, certainly, sooner or later, someone will get the idea to develop an extension. Extensibility, in general, is not an inherently bad thing. But it has to be in the right place. If it is not, it often does more harm than good. First, the difficulty of defining a new schema discourages people from defining new schemas when they are needed. This can lead to recycling inappropriate general schemas, instead. Second, the big degree of freedom can lead to inconsistent behavior at places, where it is not expected and desirable. Consider different matching rules for a string attribute. In one situation, string comparison behaves differently than in another. One have to always think about what will be the result of comparison in the context of each attribute.

There are many “standard schemas” defined. Their purpose is to serve a basis for further user-defined schemas or to be used in general situations. The main problem with them is that they are huge. The base LDAP schema for user applications is presented in RFC 4519. Its fragment is shown in figure 1.5. Plenty of domain-specific schemas inherit from its object classes, so it is really ubiquitous. It contains 43 attributes and 14 object classes. For a schema in a real deployment, this would be probably fine, but for a schema acting as the very base? A lot of elements of those schemas have their origins in X.500 standards. As X.500 was originally designed by telecommunication organization for standardization in 1980s, many of those elements (e.g., attributes) are useless today, such as a telex number, X.121 address, etc.

Distinguished Names

Entries within a LDAP directory are organized in a tree structure, similarly to files are organized within a file-system. The tree is called the Directory Information Tree (DIT). Each its entry holds a set of attributes (data items) of various data types, defined in a schema. Entries within the DIT are identified by Distinguished Names (DNs). For each entry, one can select one or more of its attributes as so-called Relative Distinguished Name (RDN). Almost any of the attributes can be chosen as an RDN attribute. The whole DN of an entry is then the concatenation of its own RDN, and RDNs of all its ancestors upto the DIT root. A textual representation of DNs exists (see figure 1.6), and is specified in the LDAP RFC 4512.

\[
\text{CN=John Doe+telNumber=+1 123-456-7890,OU=People,DC=example,DC=com}
\]

Figure 1.6: LDAP textual Distinguished Name (DN)

At the left side of the ‘=’ sign, there is an abbreviation of the RDN attribute name. At the right side, there is the attribute value. Individual RDNs are separated by a comma. If multiple attributes are selected as an RDN, the attributes are separated by ‘+’ sign.

Not only that DNs are ugly, long, difficult to remember, and annoying to parse, there are also practical aspects which demonstrate that they are bad design

---

8 The address format, used by the old X.25 network protocol suite.
9 The attribute must have a matching rule assigned, and its value must be unique among all entry’s siblings.
choice. DNs are used in LDAP as primary keys to entries. In my own experience, primary keys should be always as simple as possible. First, it is much more easier to perform indexing on data with simple data types (e.g., integer). With string primary key, for example, searching cannot be ever as fast. It is a crucial factor, as the directory service should be optimized for these operations. This can be even worse, if it is allowed to have a primary key glued up from components, and each of them can have a different data type. Second, with a simple primary key, one can implement useful things, such as caching, easily. Requests, like: “Give me last 10 queried entries.” can be satisfied in a much more natural way. With DNs, one has to implement some kind of complex hashing or key translation. Third, a primary key should be immutable. As other entities (even outside the directory) can reference the entries, it can be real pain to fix the integrity if the primary key (user’s name or phone number) changes. Although most database theorists will not agree with me, the best practice is to have a surrogate primary key, which is not bound to any real-world entity. Every time.

BER Encoding

LDAP uses ASN.1—BER encoding to encode all of its protocol elements. BER uses TLV (Tag, Length, Value) encoding scheme. Sometimes it is said that BER encoding is inefficient. There certainly exist data patterns for which BER has non-trivial overhead, but the difference is not in the orders of magnitude, and thus is not that relevant for us.

We have a bigger problem with ASN.1 itself as, along with its encoding rules (BER, DER, CER), it is very complicated and confused. It is not used anymore in protocol design for good reasons.

1.4 Summary

We have made a research about directory services related to UNIX environment, over last 35 years. There are two significant services: NIS and LDAP.

The former, NIS, suffers from serious security issues, and cannot be considered as a possible option today. Nowadays, there is nothing such as insensitive network traffic. Although, an alternative, NIS+, exists, which tried to solve these issues, but NIS+ has not achieved much success. There were support and functional issues [O'B00], and now neither NIS, nor NIS+ are longer officially supported and developed. Although, there are plenty of people still using the old NIS, despite of its security weaknesses.

The later, LDAP, is based on old telecommunication X.500 standards, which burden the protocol with things, that are useless for today needs. This complicates implementation of the protocol, as well as its usage and integration with other services. It is also accompanied by bad design choices, that prevent implementations to be fast and efficient.

Other directory services are less or more based on the previous two, especially on LDAP (e.g., FreeIPA, Active Directory, etc). This approach adds an extra layer of logic above the LDAP logic, which sometimes makes the service even worse. Sometimes it tries to moderate LDAP complexity by additional restrictive rules, but this does not solve discussed inherent problems of LDAP itself.
Due to the mentioned insufficiencies of current protocols, we decided to design a new protocol for accessing directory service. It will be called Feather-Weight Directory Access Protocol (FDAP), in the memory of LDAP. By accessing directory services, we do not mean accessing an X.500 directory. Our goal will be to avoid X.500 definition of the directory service. We will get inspired by directory service example usage, provided at the beginning of chapter. Also note that we will primarily target UNIX-like systems.
Chapter 2

Feather-weight Directory Access Protocol (FDAP)

2.1 Service and Protocol Requirements

Our goal will be to design a new application protocol for accessing a directory service over the network – FDAP. We will interpret the directory as a dataset of fuzzy entries (usually short-length) exposed by FDAP to the public.

We already have some requirements and we know some of the invariants that we would like FDAP to satisfy:

(R1) FDAP will be based upon TCP/IP architecture, namely on the top of the TCP transport protocol.

(R2) The protocol should be as simple as possible. For example, we should:

– Store the data in a simple way.
– Not introduce a new syntax if it is not really necessary.
– Not enforce unnecessary restrictions.

(R3) FDAP should fit a particular use-case. Trying to predict the unpredictable about its future usage is treated as undesirable intent and it is in contradiction with the previous point.

(R4) Design decisions regarding the protocol (encoding, data model, etc.) should not prevent the implementation from being fast and efficient.

(R5) FDAP should be a Directory Access Protocol, therefore it should be, obviously, able to store and maintain dataset of entries – a directory. It should be optimized especially for read (search) operations.

(R6) FDAP should provide built-in, safe authentication support of the entries (most likely users) against the directory.

(R7) FDAP should be consistent with all reasonable assumptions about its safety as one could expect from a modern, secured network protocol (e.g., data authenticity, integrity and encryption).

These requirements are either obvious or rather intuitive ideas than precise constraints; they can naturally contradict each other. We are also inspired by LDAP, so we have an idea what to adopt, and what to avoid.

Next, we will base the proposed solution upon these requirements.

1The term directory will no longer refer to the X.500 specification.
2.2 CBOR

Because we will base our solution on CBOR, its (at least shallow) knowledge is crucial for understanding further text. In FDAP design, we will use CBOR for two purposes:

1. For a definition of the FDAP directory data model,
2. and for encoding all of its protocol elements, to be able to transfer them over the network.

2.2.1 Brief Introduction

CBOR stands for Concise Binary Object Representation. It is a data serialization format, thus it can be used for expressing various data objects in a binary form. CBOR is standardized in RFC 7049 [BH13]. It is loosely based on JSON, so it allows to transfer key-value pairs (besides other useful data types) but more concisely.

CBOR defines several data types of encoded objects. These types are called major types. CBOR defines seven major types: unsigned integers, negative integers, byte strings, text strings, arrays, maps (similar to JSON objects), semantic tags (for data annotation), and simple values (true, false, but also floats).

See figure 2.1 for example of CBOR-encoded data. The same data are shown in the diagnostic notation in figure 2.2.

To become more familiar with CBOR, please read the RFC 7049 [BH13], it is very well-written. Also, there is a nice 5-page crash-course summary, written by David Čepelík in his bachelor thesis, second chapter [Če17].

```plaintext
A3 map(3)
  64 . text(4)
  6E616D65 . "name"
  66 . text(6)
  416E64726577 . "Andrew"
  67 . text(7)
  7375726E616D65 . "surname"
  66 . text(6)
  436F61727365 . "Coarse"
  63 . text(3)
  616765 . "age"
  18 19 . unsigned(25)
```

Figure 2.1: Data serialized in CBOR

```json
{
  "name": "Andrew",
  "surname": "Coarse",
  "age": 25
}
```

Figure 2.2: CBOR diagnostic notation
2.2.2 Reasons for the Choice

We have chosen CBOR because of its following properties:

- CBOR is an economical choice – it saves space and time. This allows the implementations of the encoders and decoders to be fast and efficient.
  
  We can for example hold entries in memory for caching purposes in their binary representation, because we know the representation does not waste space. We also know that when the cache invalidates, we can quickly generate the updated encoded image of the original entry.

- CBOR is self-delimiting. It means that all metadata are present in the encoded data, they are not defined by a context (i.e. pre-agreed order). As a consequence, it is more error-resilient, and the debugging is much easier.

- It is concise in its code size, but still clear and quite readable at byte level due to its subtle design.
  
  See figure 2.1 – for small integers, their value is equal to the byte value. The same applies to the explicit length delimiters of all definite-length elements.

- CBOR is standardized in RFC 7049, thus we have some stability guarantees.

- A tool-set exists for working with CBOR:
  
  - Diagnostic Notation [BH13] (see figure 2.2) for expressing CBOR data in JSON-like, human-readable form.
  - Concise Data Definition Language (CDDL) [BVB18] for expressing schemas that allow to define CBOR data formats.
  - Useful program utilities exist as well.

Finally, we will compare CBOR to some other serialization formats.

Extensible Markup Language (XML)

XML is considered an industry standard. A lot of tools exist. On the other hand, there are not many well-written parsers. It is not an easy task to write our own parser that would be efficient. And the format itself is very, very bulky – this would prevent our implementation to efficiently decode and encode the data.

JavaScript Object Notation (JSON)

JSON is popular serialization format for the obvious reasons: It is simple, clear and human-readable. If it was not so verbose, it would be probably our choice. Again, the JSON would prevent the encoding and decoding from being fast enough. In contrast, CBOR has the advantages of JSON, but solves the efficiency problem.

And yet another fact is annoying about JSON: It does not define integer types. One is forced to use floating-point numbers if wants to encode a 64-bit integer, which reduces the precision, as the mantisa is limited to 52 bits.
2.3 Directory Data Model

A directory consists of a set of entries in a flat structure, or equivalently: a directory is a list collection of entries. A directory entry is the base unit of information held in a directory. Each entry consists of a set of attributes – pieces of information held by and describing an entry. Attributes can hold information of various data types, but they cannot contain another entry as a value, to preserve flat structure of the directory.

This only means that a structural shape of the directory cannot imply or enforce any relations between the entries (such as subordination, etc.), while LDAP Directory Information Tree (DIT) [Zei06a] is designed in that way. Instead, FDAP presents different mechanisms how to express certain relations between the entries (for example references).

2.3.1 Entries and Attributes

FDAP entry is stored, encoded and treated as a CBOR map (major type 5). Entries—the maps—must have solely textual string keys of definite length (major type 3). Values of the entries can be of any CBOR major type. One key-value pair of the entry root map is treated as an attribute. The key is called an attribute name, the value is called an attribute value or just an attribute, interchangeably.

2.3.2 Identification of Attributes

Attributes are identified by a textual name. This name has to be unique across the directory. If two attributes (potentially of different entries) have the same name, their values must be of the same major type. These invariants must be guaranteed by server implementation.

Entry structure can be nested. Some attributes can be of type map (major type 5) and their children can be again of type map, and so on – this is where the nesting occurs. These nested attributes can be identified by dot-chained names (see figure 2.3). This implies, that the dot ‘.’ cannot be used as a regular character in attribute name.

attr.nested1.nested2.nested3

Figure 2.3: Nested attribute dot-chained name

2.3.3 Identification of Entries

Entries are identified by unsigned integer id. This id is used as the entry’s primary key, thus it must be unique across the directory. The value must be a positive number, because the value of 0 is reserved for special purposes (see below). It is the server implementation’s responsibility to assign the ids to new entries and to ensure uniqueness.

Note that id itself is not an attribute. There is nothing like a key-value pair with a key named id in the entry root map. The id is carried in reserved

2Unless someone has not created an attribute with such name, which would be perfectly legitimate.
fields within the protocol operations that somehow work with the \textit{id} (see subsection \ref{subsec:2.5.2}).

This approach, that \textit{ids} are an explicit part of the protocol, has the advantage that the implementations can make smart use of the \textit{ids}. Consider an intermediate cache. If the \textit{id} were an attribute, one would have to query the entries by a filter, like \textit{id} = 42. Then the cache would have to evaluate the filter on the directory which would lose its meaning, or would have to parse the filter and treat it in a special way, which would complicate the implementation. If the \textit{id} is handled separately, the cache can take the advantage of it, and just compare the \textit{id} in the operation with the cached set.

\section*{2.3.4 Referencing the Entries}

A mechanism of references is supported by FDAP. It means that one entry can point to another – it is used in a similar way as foreign keys in relational databases. To create a reference, an attribute of CBOR major type 0 (unsigned integer) must be created on an entry. This attribute must be tagged with CBOR tag \#29. The value itself must be a valid \textit{id} of some entry within the directory. A reference value of 0 is also valid, and it is treated as a null reference. It is a server implementation’s responsibility to check whether all references are valid, before any write operation is committed on the directory. If an invalid reference exists within a request, the error code \texttt{Invalid reference} (see table \ref{tab:2.19}) must be returned in the response.

\section*{2.3.5 Typing of Entries}

Entries itself not have any explicit types assigned, nor any validation of required attributes on the entries is forced by the protocol. It is left entirely to an implementation how much strict it wishes to act.

However, a few attributes may be required to ensure the authentication.

\section*{2.3.6 Authenticable Entries}

One of the protocol requirements was to provide built-in authentication. In order to satisfy the requirement, some authentication data must be stored in the directory. FDAP authentication is bound to entries. This means that if someone wants to access the directory, there must exist an entry through which is the person authenticated.

Authenticable entries must contain a \texttt{read-only} attribute, which must be of major type 5 – map. This attribute (and all of its sub-attributes) can be changed only by administrator (see subsection \ref{subsec:2.6.5} for details).

The entries are identified by the \texttt{read-only.username} attribute, for the authentication purposes. This attribute must be of type string (major type 3). It is the server implementation’s responsibility to ensure that the value is unique across the directory.

Also, \texttt{read-only.admin} flag attribute may be present on an authenticable entry. If present, this attribute must be of major type 7 (simple value), and must

\footnote{Value 29 is reserved for value reference by IANA. But to be rigorous, we should get reserved a dedicated tag value. So far it is an experimental version of the protocol, so let it be for now.}
be either true or false. If set to true, the user’s access rights are elevated after successful authentication, and is granted to perform any operation on the directory. If either set to false, or the attribute is not present, the user is treated as a regular user.

The last piece of data, which is required for authentication, is an attribute named password. It must be of major type 3 – text string.

Both read-only.username and password attributes are required for authentication, however optional for non-authenticable entries. If one of the attributes is in a bad format or the password attribute is missing, authentication fails, and an appropriate error code is returned. All of the four attributes (read-only, read-only.username, read-only.admin, and password), their names and types, are reserved for protocol purposes.

Example of an authenticable entry is shown in figure 2.4 in CBOR diagnostic notation.

```
{
  password: b'6E017B5464F820A6C1BB5E9F6D711A667A80D8EA',
  read-only: {
    username: 'hrubon',
    admin: true,
    uid: 1000,
    gid: 1000, ...
  }, ...
}
```

**Figure 2.4:** An authenticable entry
2.3.7 Comparison of Attribute Values

Sometimes, it is necessary to compare attribute values to each other or to some constant expression (for example when implementing entry filtering – see subsection [2.4]).

Assume two attribute values $a$ and $b$. Then the result of their comparison can be one of the following:

1. $a$ is less than $b$,
2. $a$ is equal to $b$,
3. $a$ is greater than $b$,

If $a$ has different CBOR major type than $b$, then the attribute with the greater value of major type is considered the greater one. Otherwise, FDAP defines comparison semantics for each CBOR major type:

- Major types 0, 1 (integers) and 7 (simple values) are compared according to their numeric values.
- All of the following major types (2, 3, 4 and 5) are compared primarily by the length of the string, array or map, respectively. The longer one is considered the greater one.
- Major types 2 and 3 (byte and text strings): If they are of the same length, they are compared according to the lexicographical order.
- Major type 4 (arrays): If they are of the same length, they are compared value by value, recursively in sequential order. This leads to lexicographical order of array items too.
- Major type 5 (maps): If they are of the same length, they are compared primarily key by key and secondarily value by value, recursively, in sequential order. This leads to lexicographical order of map items, as well.
- Major type 6 (semantic tags) are compared as follows:
  1. If $a$ has a tag, and $b$ has not, then $a$ is greater than $b$, and vice-versa.
  2. Else, if both items have a tag, the items are compared according to the values of their tags (unsigned integers).
  3. Finally, if the values of the tags are equal, the values of the tagged items are compared recursively.

By the described comparison model, all major types are fully comparable to each other, even if they are different. This implies that the comparison model forms a total order on the set of attributes, which is a desirable property (sorting the attributes always produces meaningful results).

4With the exception of CBOR floats, that are not supported by the current protocol version.
5`strcmp` and `memcmp` POSIX functions will exactly do the job.
2.4 Filters

Filters are an integral part of the protocol, and are used together with several FDAP operations. In general, they allow us to express logical formulas that are applied to entries within the directory and produce a subset of the directory as a result. Filtering occurs on entry attributes by querying their values. The FDAP protocol specifies the following aspects of filter usage:

1. Human-readable infix\[6\] textual notation for filter expression,
2. semantics of the filter evaluation on directory entries and attributes,
3. encoding of the filter elements within FDAP requests at the binary level.

As we said, filters express logical formulas. We can write them using the textual infix notation, as one could expect (see figure 2.5). For a formal grammar definition, please refer to appendix A. The usual repertoire of three logical operations is supported (not, and, or) to provide an ability to generate the full set of boolean functions (logical universality).

\[(\neg C_1 \mid C_2) \& \neg (C_3 \& C_4 \& C_5)\]

Where:

1. \& stands for logical binary and.
2. | stands for logical binary or.
3. ! stands for logical unary not.
4. C1...C5 represents atomic filter conditions (logical atoms).
5. Parentheses ( and ) determine the precedence of sub-formulas within the formula. They can be omitted when unary operator precedes an atom (e.g. \neg C_1), and also when a binary operation is chained in a sequence (e.g. C_3 \& C_4 \& C_5).

Figure 2.5: Filter textual notation formula (partially expanded)

The given example formula can be expressed as an Abstract Syntax Tree (AST) (see figure 2.6), which is the recommended output form for textual notation parsers. We will base further ideas about filter evaluation and encoding upon the AST representation.

\[\text{6Compared to LDAP, that uses prefix notation, which is not that much human-readable.}\]
Encoding of filters is simply a projection of the AST to the sequential CBOR stream. The order of AST nodes in the stream follows the DFS pre-order tree traversal, so it is, in fact, binary-encoded expression in prefix notation. We encode two pieces of information about the node: first, its type (leaf, unary, binary) and then the operation (not, and, or). From the first byte we can determine what will follow next in the stream. The second byte (if present) is an operation code and must be interpreted in the context of the previous byte. See table 2.7 for details.

<table>
<thead>
<tr>
<th>semantics</th>
<th>textual</th>
<th>CBOR opcode</th>
<th>follows</th>
</tr>
</thead>
<tbody>
<tr>
<td>leaf</td>
<td></td>
<td>0x00</td>
<td>condition</td>
</tr>
<tr>
<td>logical not</td>
<td>!</td>
<td>0x01 0x00</td>
<td>child node</td>
</tr>
<tr>
<td>logical and</td>
<td>&amp;</td>
<td>0x02 0x00</td>
<td>left child, then right child</td>
</tr>
<tr>
<td>logical or</td>
<td></td>
<td>0x02 0x01</td>
<td>left child, then right child</td>
</tr>
</tbody>
</table>

Figure 2.6: Filter AST

Figure 2.7: Encoding of filter AST nodes

Placeholder symbols C1...C5 in the examples are just non-terminals of the grammar. They must be substituted by meaningful filter conditions, that would have some relation to the entry attributes. We will define conditions consisting of three components:

1. Attribute name, conforming to subsection 2.3.2
2. conditional operator (see table 2.8 for list of operators),
3. target CBOR value.

\(^7\)In this case, the CBOR-encoded binary values are equal to the real values of the opcodes, because of the nature of CBOR major type 0 (uint) encoding.
When evaluating a condition on a directory entry, first the attribute value is extracted, based on the condition attribute name. The dot-chained name can be used to reference nested attributes of the entry. Then the extracted CBOR value is compared to the target CBOR value using the conditional operator. Notice that we compare two CBOR values (attribute values are CBOR items). The comparison itself is performed in two steps:

1. If the major type of the attribute is different from the major type of the target value, then the condition node must be always evaluated to false.

2. Else, the condition node is evaluated to true, iff the result of the CBOR values comparison corresponds to the conditional operator semantics\(^8\) (according to subsection 2.3.7), false otherwise.

The same applies to the rest of the AST, which can be evaluated recursively, as the evaluation result of each node is always a boolean value.

<table>
<thead>
<tr>
<th>semantics</th>
<th>textual</th>
<th>CBOR(^8) opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;</td>
<td>&lt;</td>
<td>0x00</td>
</tr>
<tr>
<td>≤</td>
<td>&lt;=</td>
<td>0x01</td>
</tr>
<tr>
<td>=</td>
<td>=</td>
<td>0x02</td>
</tr>
<tr>
<td>≠</td>
<td>!=</td>
<td>0x03</td>
</tr>
<tr>
<td>≥</td>
<td>&gt;=</td>
<td>0x04</td>
</tr>
<tr>
<td>&gt;</td>
<td>&gt;</td>
<td>0x05</td>
</tr>
</tbody>
</table>

**Figure 2.8:** Encoding of filter conditional operators

The last two things, which we did not mention, is an encoding of the conditions to the CBOR stream and textual representation of conditions. We will encode conditions sequentially as follows:

1. First, the attribute name, which is encoded as an definite-length text string (CBOR major type 3). If the attribute name is nested, a dot-chained name is encoded as a string there.

2. Then, the conditional operator opcode (see table 2.8) is encoded as an unsigned integer (CBOR major type 0).

3. Finally, the target CBOR value is encoded as a regular CBOR item, depending on its own major type. Any major type can be supplied there\(^9\).

The textual format follows the same pattern: for the attribute name, the textual dot-chained name takes the place, followed by textual form of the operator. The target CBOR value is written in CBOR diagnostic notation [BH13]. For textual filter notation, we support only a subset of the diag. notation: integers (majors 0 and 1), text strings (major 3) and simple values (major 7, excluding floats). Please refer to appendix A for details.

\(^8\)The semantics of the operators should be obvious – for example: Condition \((\text{age} > 18)\) is evaluated to true iff the value of attribute \text{age} is greater than 18, in conformance with subsection 2.3.7.

\(^9\)With the exception of CBOR floats, which are not supported by current protocol version. Anyway, the builder of the filter should pass a meaningful value (matching the target attribute major type) in their own interest—in order to obtain a non-empty result set.
Finally, we will provide an example of the filter formula from figure 2.5 with expanded non-terminals \( C_1 \ldots C_5 \) to some real filter conditions, and also an example of the corresponding CBOR stream of encoded filter data.

\[(!isAdult = \text{true} \mid \text{age} > 18) \land !(\text{addr.street} \neq 'Diagon Alley' \land \text{addr.number} = 93 \land \text{flag} = \text{simple}(42))\]

**Figure 2.9:** Filter textual notation formula (fully expanded)

```plaintext
02 uint(2) binary node
00 uint(0) & operator
02 uint(2) . binary node
01 uint(1) . | operator
01 uint(1) . . unary node
00 uint(1) . . ! operator
00 uint(0) . . leaf node
67 69 73 41 64 75 6c
74 text(7) . . . "isAdult"
02 uint(2) . . . = operator
f5 simple(21) . . . true
00 uint(0) . . leaf node
63 61 67 65
05 uint(5) . . > operator
12 uint(18) . . 18
01 uint(1) . unary node
00 uint(0) . . ! operator
02 uint(2) . . binary node
00 uint(0) . . & operator
02 uint(2) . . binary node
00 uint(0) . . & operator
00 uint(0) . . leaf node
6b 61 64 64 72 2e 73
74 72 65 65 74 text(11) . . . "addr.street"
03 uint(3) . . . != operator
6c 44 69 61 67 6f 6e
text(12) . . . "Diagon Alley"
00 uint(0) . . leaf node
6b 61 64 64 72 2e 6e
75 6d 62 65 72 text(11) . . . "addr.number"
02 uint(2) . . . = operator
18 5d uint(93) . . . 93
00 uint(0) . . leaf node
64 66 6c 61 67
text(4) . . . "flag"
02 uint(2) . . . = operator
f8 2a simple(42) . . . simple(42)
```

**Figure 2.10:** CBOR-encoded filter data
2.5 Communication Protocol

FDAP is an application layer (L7) protocol for accessing directory services in a client-server model fashion. The altered definition of the directory service is described in sections 2.3 and 2.6. Now, we will describe transport and encoding aspects of the protocol.

2.5.1 Transport

FDAP is designed to run over connection-oriented reliable transports. A canonical example of such transport is the Transmission Control Protocol (TCP). As the FDAP is an experimental protocol so far, it have no official TCP port number assigned by ICANN authority yet. Any TCP port from the dynamic range can be used for now.

Because operating above a raw TCP is not secure, it is strongly recommended to use TLS layer (or similar encryption layer) above the transport layer. FDAP does not provide any STARTTLS command, so the TLS must operate on dedicated transport end-point (port), and both communication sides must be aware of the usage of the TLS layer in advance.

All protocol elements are encoded using CBOR. Request and response formats are defined in the next section. The encoded FDAP messages (request and responses) are mapped directly onto the transport protocol bytestream using the CBOR encoding.

```
+-----------------------------------+
| FDAP message layer (encoded CBOR) |
+-----------------------------------+ > FDAP PDUs
+-----------------------------------+ < data
| TLS layer                        |
Application +------------------------+ > TLS-protected data
+-----------------------------------+ < data
Transport | transport connection |
+-----------------------------------+
```

Figure 2.11: FDAP transport – layer overview

Protocol Transactions

For each request, FDAP server always gives the client feedback by sending exactly one response message. There is no operation specified in the protocol that would require to send more than one request-response transaction in order to be committed and considered finished – there are no handshake operations.

Likewise, no pipelining of requests is allowed by the protocol, thus the order of the requests within the same transport connection is unambiguously determined. If client issues a request, it must wait for a response prior to issuing another request. Reasonable timeout can be implemented at the client side. After expiry of the timeout, the client is permitted to reset the transport connection. Similarly,

Assuming, there is not other security mechanism installed at the network level (such as IPSEC).
the server is permitted to reset the transport connection with a client after some period of client inactivity (idle timeout).

**Communication within the same Host**

Sometimes it can be practical to communicate within the same host (machine). For such purposes, the usage of typical reliable network transport protocols can mean a big overhead. Especially, when querying the directory with many short-length requests, which is the typical traffic pattern. Suppose a file-system listing command (`ls`). For each file in the file-system directory, the name of the user owner and group owner must be queried from the FDAP directory. Initializing a TCP connection, and especially a TLS tunnel for each request would be very inefficient.

Instead, it is much more effective for local-system applications (such as `ls`) to communicate with a local proxy daemon via other IPC mechanisms, for example Unix-Domain Sockets. The proxy daemon may then have a permanent secure connection to a remote upstream server. These transports does not implement any encryption (as the environment within a local machine is considered trusted), and their connection overhead is minimal, compared for example to TCP. Therefore, it is recommended to implement a Unix-Domain Socket (or similar) listener at the server-side.
2.5.2 Request Format

FDAP specifies several read and write operations for interaction with the directory. For each operation, we define a request format within the protocol. All requests are encoded into sequential CBOR stream, as well as the other protocol elements. Each request starts with *length* – unsigned int (CBOR major type 0), followed by *request operation code* (also major type 0). The rest of the data stream is specific for particular operation:

Authenticate

![Authenticate request](image)

Search

![Search request](image)

Get

![Get request](image)

Create

![Create request](image)

Update

![Update request](image)
Prefixing Requests by Length

Thanks to the fact that all messages are prefixed by a length field, the implementations can be more efficient and it can also help them to solve several I/O problems. In particular, it is possible to buffer whole message into memory prior beginning the decoding procedure. This approach can simplify the implementation, compared to the approach of decoding the requests “on the fly”—meaning that the CBOR decoder would call read directly on the underlying network stream on demand, which would result in blocking read calls, as there is not another option if the data are not available yet in the network card buffer.

Blocking I/O can be avoided using several multiplexing mechanisms (such as poll). However their use usually requires to service the clients in an asynchronous fashion. This is not a problem as long as the servicing of clients is simple, and it does not depend on some non-trivial context. Decoding of requests is not that simple, as it is context-dependent. It is an inherently recursive task – the decoded data can be organized in a rich, nested structure. If one had to switch between multiple decoders, each of them working on a different half-decoded request, very non-trivial context switching would have to be implemented. Probably some sort of stack switching or coroutines would have to be involved.

Instead, it is much easier to just continuously buffer the data, without knowledge of data semantics. This is already a simple task to be done asynchronously. After the whole message is buffered (we are able to detect this thanks to the length field), we can decode the buffered data at once, without any context switching, and without being blocked.

Finally, implementation of features like maximum request length restrictions is easily possible with the explicit length field.

2.5.3 Response Format

The response format is generic and it is the same for all operation requests:

First goes a status code, indicating the result of the operation. See the full status code table in figure 2.19. The status codes are global for entire protocol,
however some of them make sense only for specific operations – see concrete operation description below. After the status code, there is a result set in the response. The result set can be empty (no result set present), as again it makes sense only for some of the operations. If it is non-empty, it is a sequence of entries (CBOR maps – major type 5), each entry prefixed with its id (unsigned int – major type 0). The result set (even the empty one) is always terminated by an unsigned integer of zero value.

Notice that unlike requests, the response format does not prefix the message with the length field. It is because at the client side, blocking I/O is not a problem. In fact, it is quite desirable, as the client is often unable to continue until it obtains a complete response to its original request.

<table>
<thead>
<tr>
<th>Status code</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>OK</td>
</tr>
<tr>
<td>101</td>
<td>Not found</td>
</tr>
<tr>
<td>301</td>
<td>Internal error</td>
</tr>
<tr>
<td>302</td>
<td>Not implemented</td>
</tr>
<tr>
<td>303</td>
<td>Request too long</td>
</tr>
<tr>
<td>304</td>
<td>String too long</td>
</tr>
<tr>
<td>401</td>
<td>Invalid filter</td>
</tr>
<tr>
<td>402</td>
<td>Invalid entry</td>
</tr>
<tr>
<td>403</td>
<td>Entry not a map</td>
</tr>
<tr>
<td>404</td>
<td>Invalid reference</td>
</tr>
<tr>
<td>501</td>
<td>Not authenticated</td>
</tr>
<tr>
<td>502</td>
<td>Forbidden</td>
</tr>
</tbody>
</table>

Figure 2.19: FDAP status code table

2.6 Operations

2.6.1 Authenticate

FDAP supports built-in authentication. It is intended to be used in two ways:

1. To secure the directory itself – a user must authenticate prior to accessing the directory,

2. and to support authentication as a service for external applications (e.g., to enable single sign-on).

The Authentication state is local to the transport connection to the FDAP service (e.g., a TCP connection). A connection is either in authenticated state, or in not authenticated state, which is the default state. The Authenticate operation is used to make a transition from the not authenticated state to the authenticated state. The connection is kept authenticated until the transport connection is closed or reset by either party. The implementation is free to impose an idle client time-out and terminate client’s connection once this time-out is exceeded.
Authentication is performed against entries in the directory. In order to enable authentication on an entry, some required attributes must be present on the entry as described in subsection 2.3.6.

The authentication request is handled as follows:

1. An entry with attribute `read-only.username` equal to the `username` field provided in the authenticate request is searched.
   - If no such entry exists, a response with code `Not found` is sent to the client, and the connection is considered `not authenticated`.
   - Otherwise, the first matched entry is taken and the next step is performed.

2. The `password` attribute from the matched entry is then compared to the `password` field in the request.
   - If a match occurs, a response with code `OK` is sent to the client and the connection is considered `authenticated`.
   - Otherwise, a response with code `Not authenticated` is sent to the client and the connection is considered `not authenticated`.

If the authentication was successful, the entry through which the user was authenticated is sent in the response result set. Otherwise an empty result set is sent.

Once successfully authenticated, the authorization needs to be performed. It is implemented in a minimalistic way, but it is necessary to have at least some authorization, because of both security and practical reasons. There are two flags that describe the authorization state:

1. The `read-only.admin` flag from subsection 2.3.6. It should be read from the entry right after the authentication. If the attribute does not exist, `false` value is taken as default. This flag is directory-wide and it is held per connection.

2. The connection is considered the owner of the entry through which it has been authenticated, so this flag is entry-specific. The owner of the entry has some additional access rights when requesting operations on that entry. Possible implementation could be that the `id` of the owned entry is associated with the connection.

When an operation is requested by a client, it must be authorized first. Whether the operation should be authorized or not, is determined both by the authentication state and the combination of the previously mentioned authorization flags.

The authorization schema is shown in figure 2.20. In the leaves, there is the set of operations that a particular client (with the corresponding values of flags) is authorized to perform. If the requested operation is not contained in the set, a response with an empty result set should be sent to the client, with status code `Forbidden` or `Not authenticated`, respectively.
**) Updates are not allowed to the read-only attribute (and its sub-attributes) present at the root level of the entry (see subsection 2.6.5).

2.6.2 Search

Performs search on the directory. The filter in the request is sequentially evaluated on each entry within the directory according to subsection 2.4. If the filter is in a bad format, a response with Invalid filter status code and an empty result set is sent to the client. Otherwise, a response with OK status code and the corresponding result set, which contains entries matched against the filter is sent to the client.

Note that if the result set contains entries whose owner is not the requesting connection, the server should masquerade or remove the password attribute of those entries, if present.

The result set of this operation may be large. Sending such a response at once would result in the temporary denial of service. Therefore, it is recommended to the server implementations to be able to interleave the responding procedure across multiple clients in a round-robin fashion, if the response length would be above some reasonable limit. This is, however, just the implementation aspect—a recommendation—not governed by the protocol.
2.6.3 Get

Fetches an entry from the directory by the entry’s id. As the id is unique, the result set can contain at most one entry. If no entry with the requested id exists within the directory, Not found status code and an empty result set is sent to the client. Otherwise, a response with OK status code and a result set containing exactly one entry is sent to the client.

Note that if the result set contains entries whose owner is not the requesting connection, the server should masquerade or remove the password attribute of those entries, if present.

2.6.4 Create

Creates a new entry within the directory. The entry map is taken from the request as is, and a new id is assigned to it. If the entry contains an attribute with some name and of some type, and another attribute with the same name exists within the directory, major types of both attributes must match, according to subsection 2.3.2. If they do not match, the entry is considered invalid and the operation should not be committed.

The effect of this operation should be persistent to the directory. It means that after restart of the FDAP server, an entry inserted by this operation should appear in the directory.

If this operation fails, a response with the appropriate status code and an empty result set is sent to the client. Otherwise, a response with status code OK and a result set containing the inserted entry with its newly generated id is sent to the client. This operation may fail because of the following reasons:

- If the entry in the request is not of CBOR major type 5 – map, status code Entry not a map should be sent to the client.
- If some string in the entry exceeds allowed size, status code String too long should be sent to the client.
- If a reference attribute points to invalid entry (the attribute value is not equal to an id of any entry, or it is not zero), then the status code Invalid reference should be sent to the client.
- If something else is wrong with the entry (e.g., the attribute name is wrong), status code Invalid entry should be sent to the client.

2.6.5 Update

Updates an existing entry within the directory. First, the requested entry must be fetched from the directory by the id in the request. This is done completely in the same way as the Get operation. If the Get operation would fail, this operation should fail as well with the same status code. Otherwise, the fetched entry is updated with entry data in the request.

Values of existing attributes on the fetched entry should be overwritten by new values from the entry in the request, with one exception: If the value of the attribute in the request is undef (despite of the fetched entry attribute’s real
type), the attribute should be removed from the entry. The rest of the attributes on the fetched entry is kept as is. If the entry in the request contains some extra attributes, which are not contained in the fetched entry, these attributes are added to the entry in the directory. All attributes in the request entry should be checked in the same way as it is done in the Create operation. If the check does not pass, the entry is considered invalid and the operation should not be committed.

The effect of this operation should be persistent to the directory. It means that after restart of the FDAP server, an entry updated by this operation should appear in the directory in the same form.

If this operation fails, a response with the appropriate status code and an empty result set is sent to the client. Otherwise, a response with status code OK and a result set containing the updated entry with its id is sent to the client. This operation may fail because of the same reasons as the Get operation or Create operation.

This operation can be executed either by the administrator, or the owner connection of the entry. In the later case, the read-only attribute (and all of its sub-attributes) is not allowed to be changed by the owner. It is the responsibility of the server implementation to enforce this restriction, and it is up to the implementation how to enforce that.

2.6.6 Delete

Removes an existing entry from the directory. First, the requested entry must be fetched from the directory by the id in the request. This is done completely in the same way as the Get operation. If the Get operation would fail, this operation should fail as well with the same status code. Otherwise, the check must be done if the entry is not referenced from other entries. If so, the delete operation should fail and the Invalid reference status code should be returned to the client.

The effect of this operation should be persistent to the directory. It means that after restart of the FDAP server, an entry removed by this operation should not appear in the directory anymore.

If this operation fails, a response with the appropriate status code and an empty result set is sent to the client. Otherwise, a response with status code OK and a result set containing just the removed entry with its id is sent to the client. This operation may fail because of the same reasons as the Get operation or Create operation.

This operation can be executed only by the administrator.
Chapter 3
Implementation

The goal of the following sections is to provide a high-level overview of the problems solved by various modules of the FDAP suite of programs. We will start with a big picture about overall architecture.

For low-level implementation details, please refer to the API documentation attached in appendix B of this thesis. A brief summary of selected important parts will be mentioned in this chapter.

We are not fans of auto-generated documentations (like Doxygen). But we believe, that clear code and appropriate comments and annotations in the code are a principle of every good programmer. Unfortunately, this is is not often the case. We have developed a script that generates \LaTeX{} documentation from the source code. But unlike other tools, the output is nice and readable. The source comments are written carefully and only selected functions are included in the output. Each module documentation starts with a foreword that summarizes its usage.

In this chapter, we will also provide instructions for installation, configuration and usage of the executable parts.

3.1 Overall Architecture

Our proof-of-concept implementation consists of the following executables and libraries:

- \texttt{fdapd} – FDAP server daemon.
- \texttt{fdapdiag} – Simple FDAP command-line client for debugging and diagnostic purposes.
- \texttt{libfdapc.a} – FDAP client static library for usage in the following modules and client applications.
- \texttt{libnss_fdap.so} – shared library of FDAP module for the NSS service.
- \texttt{pam_fdap.so} – shared library of FDAP module for the PAM framework.

In figure 3.1 we can see how the libraries cooperate on the client machine. The arrows between the modules symbolize calls to the libraries. The direction goes from the caller library (or application) to the called library. The called library is statically linked to the calling one in most situations. The two dashed arrows from the \texttt{libfdapc.so} symbolize FDAP transport connection (e.g., TCP or Unix-Domain Socket) to some \texttt{fdapd} instance.

Note that all memory resources allocated by the code of \texttt{libfdapc.a} are allocated in the address spaces of the corresponding caller applications. Also, each caller has its own transport connection to the \texttt{fdapd} instance.
Figure 3.1: Cooperation of the client-side programs

Figure 3.2: Example FDAP infrastructure setup
Figure 3.2 illustrates an example setup of FDAP infrastructure. At the client PC, the two rectangles symbolize generic applications, which use the FDAP client library somehow. The two rectangles could be expanded to the diagram illustrated in figure 3.1. The two client apps are connected to the local \texttt{fdapd}, using Unix-Domain Sockets transport. The \texttt{fdapd} is operating in the proxy mode – see subsection 3.4.2 for details. The upstream \texttt{fdapd} instance is listening on two sockets. One for the proxy connections, and the second for client connections. Another client, running \texttt{fdapdiag}, is connected through the latter socket to the upstream.

3.2 Source Code

3.2.1 Programming Language

All code attached to this thesis is written in the C programming language. Besides some of the parsers and lexers, which are written in GNU Bison and Flex.

We wanted our implementation to be fast and efficient, thus we were trying to avoid languages with garbage collection or automatic memory management. We also wanted to have fine control of memory allocation and structure layout, as it can be important fact when designing things like binary decoders, as well as network protocol in whole. Due to these requirements and also according to the fact that we are writing a project for UNIX-like environment, two major options were: C and C++.

With C++ we could choose if we want to design our solution in an object-oriented way, or in the traditional procedural style. We do not believe that this project could benefit from object-oriented approach or design patterns, as this often complicates things, inflate code, and at the end, it becomes unreadable. With the choice of C++ procedural style, we might use smart pointers, templates and all of that cool stuff, but our knowledge of these things is too shallow for us to be confident enough for the choice. So we will rather stick with the good old C language.

3.2.2 Authorship

All code submitted as part of this thesis is original, with the exception of the following modules, which were adapted\footnote{With author’s permission.} from the NetBufs \cite{NetBufs} library for convenience:

- \texttt{objpool} and \texttt{mempool}, simple memory allocators,
- \texttt{strbuf}, growing string buffer,
- \texttt{deebug}, collection of useful debugging macros.

The \texttt{cbor} module used for encoding and decoding of CBOR streams was inspired by a similar module of the NetBufs library.
3.2.3 Directory Source Tree

The attached ZIP archive contains the project root directory, named fdap/.

We will use the term project root directory in further text. We will mean the fdap/ directory.

This directory has the layout described in figure 3.3.

```
fdap/  Project root directory
  +-- build/  
  |     +-- dbg/bin/  Built executables
  |     +-- dbg/lib/  Built shared libraries
  |     +-- tests/  Built test executables
  +-- conf/  Example configuration files
  +-- examples/  Example directory data for fdapdiag
  +-- src/  Source files
  |     +-- include/  Header files
  +-- tests/  Test source files
```

**Figure 3.3:** Directory source tree (selected parts)

3.2.4 Building the Source Code

Before building any part of the project, please install the mbedtls library – the TLS library, which we use. Please use your package manager for installation. The library package should be contained in the package database of most UNIX-like operating systems. For example:

```
apt-get install mbedtls
```

For easy building, there is a Makefile attached to the project. It can be found in the project root directory. To build the project, please cd to the project root directory and type:

```
make
```

Then, the whole project should be built, including all libraries and executables mentioned in section 3.1.

Because our implementation is just a proof-of-concept and is not intended to be deployed in production yet, we provide all binaries only in the debug version. This is also the default build configuration. Built binaries can be found in build/dbg/ directory. Executables in build/dbg/bin/, and the libraries in build/dbg/lib/.

Our implementation is also well tested. It contains around 30 unit tests. The test sources are stored in tests/ and they are built in the build/tests/ directory. The tests are run automatically during the build process. To run them separately, please type:

```
make tests
```
3.3 CBOR Encoder and Decoder

The cbor module is intellectually a descendant of the NetBufs’ CBOR module, but it has been written from scratch. Our implementation has the following advantages compared to the original NetBufs’ implementation:

- Size of struct cbor_item, the structure holding generic decoded cbor items in memory, has been reduced to 16 B from original 24 B. As billions of these items may coexist in memory at the same time, this has profound impact on memory footprint.

- Text streams shorter than 14 B (including the terminating NUL byte) can be embedded directly in the cbor_item structure, cutting back on memory usage and heap allocations. This idea was outlined in, but not implemented by, the original implementation.

- The decoder provides rich API for decoding of byte and text streams, allowing the user to read from the logical rather than physical byte or text stream, fixing various discrepancies in the original API.

- The code is more idiomatic and shorter than the original.

- The related iobuf module (which is of essential importance to the encoding/decoding speed) avoids double-buffering of data.

- The module is well-documented and carefully tested. Negative tests are provided which test most of the modules corner-cases, unlike the original encoder/decoder.

3.3.1 Usage

To use the cbor module, a struct cbor_stream needs to be first obtained which acts as a handle:

```c
struct iobuf *buf = iobuf_new_sock(fd, 1024);
struct cbor_stream c;
cbor_init(&c, buf, cbor_errh_default);
```

Figure 3.4: CBOR stream initialization

The stream is disposed by a call to cbor_stream_free. The buf parameter specifies an iobuf I/O buffer which will satisfy all read/write operations which result from the operations called on the stream. This allows CBOR streams to be encoded and decoded regardless of the underlying I/O mechanism, e.g., read from memory or written to sockets, without a change in the cbor module.

Following the initialization, other operations may be called on the stream, which are either stream-oriented or item-oriented:

- Stream-oriented functions such as cbor_read_u32 or cbor_write_bytes are used when the type of the item to be encoded to or decoded from the
stream is known beforehand. The API is quite low-level and sometimes a little bit tedious to use, but it gives the user absolute control over the encoding/decoding process. Also, it has very good performance as it avoids memory allocation entirely.

- Item-oriented functions `cbor_read_item` and `cbor_write_item` allow the user to decode/encode the next item from/to the stream, respectively, and the type of the item and the necessary steps to decode it will be determined automatically. This allows for decoding of complex items, such as maps or arrays of other complex items, and their re-encoding later on. At the same time, the `cbor_item` structures are quite convenient to work with and their diagnostic notation can be obtained easily by `cbor_item_dump`. This comes at the expense of having little control over the process and may logically necessitate memory allocation. (Items should be freed by a call to `cbor_item_free` when no longer needed.)

The API is very rich and is extensively documented in appendix B.4.

### 3.3.2 Application

The `cbor` module is used between the clients and upstream servers and between upstream and downstream servers to encode all communication. The messages are very concise, yet very fast to encode and decode and most importantly, they provide an external (machine-independent) representation of data.

The `cbor` module is also used by the `storage` module: all documents in a storage are in fact CBOR maps represented as CBOR items. This way, items can be quickly inserted into/retrieved from the storage and converted to a convenient binary form for transfer.

### 3.3.3 Known Limitations

Length of arrays and maps has been limited to $2^{32} - 1$ items, as well as the supported tag range. As this allowed us to make the `cbor_item` structure compact and having more than this amount of items in memory simultaneously is infeasible, we think that this was a good compromise.
3.4 FDAP Daemon

The FDAP daemon (or `fdapd`) is the main component of the FDAP program suite. It provides the following functions:

- Hosts the directory itself and all data it contains, and guarantees their consistency.
- Provides access to the directory through the FDAP protocol over UNIX-domain sockets and TCP/IP. TLS can be configured (and in all practical deployments, should always be) to provide a secure communication channel.
- It can serve as FDAP proxy for local (UNIX-domain) clients to reduce latencies and encryption overhead. UNIX-domain sockets communication is unencrypted, hence faster.

3.4.1 I/O

I/O Multiplexing

I/O multiplexing is provided by `epoll`. We realize that the `epoll` function is Linux-specific (but also implemented by FreeBSD) and that the implementation suffered from bugs in the past, but at the moment, `epoll` is the sole high-performance conservative. Coroutines could be used to solve many problems, and powerful libraries adding coroutine support exist, but would likely cause much more portability problems than `epoll`. I/O multiplexing feature provided by Linux.

In the future, the event loops will be able to work with `poll` and `select` on systems where `epoll` is not supported. Because this implementation is a proof-of-concept of our directory service, and performance is a critical indicator of success, we have traded portability to other UNIX-like systems in exchange for performance.

Non-blocking I/O

Where possible, I/O operations operate in a non-blocking fashion, but not in non-blocking mode (i.e., all sockets operate in blocking mode, but I/O operations are called when they’re known not to delay the caller). Thanks to the `epoll` I/O multiplexer and having all messages prefixed by their length, we can buffer clients’ requests and only start decoding them once we have a complete message in the buffer.

This is especially important given the relative complexity of decoding of CBOR items. Suppose that a client sends a new record to be stored in the directory in an `CREATE` operation. The document is required to be a CBOR map, and the values can be arbitrary CBOR items. Therefore, documents can be very complex internally and the decoding process will reflect this complexity. As a result, it cannot be easily interrupted and then restarted when there are no buffered data half way into the decoding of a message, other than blocking on the underlying `read` operation.

The reasons to avoid blocking I/O are many:
- Time spent waiting on I/O can be better spent by performing other clients’ requests.

- Misbehaving or malicious clients can decrease the throughput of the service significantly by sending incomplete requests.

- Time-outs and rate limits can be difficult to implement properly.

### 3.4.2 Storage Abstraction

In fdapd design, storage is a module, which accesses the directory database. Because the requirements on the storage can be very heterogeneous, we decided to implement it through function pointer abstraction. There are several operations on the storage interface defined, and it is up to concrete storage module how to implement them. This allows adding more storage back-ends in the future, starting from simple file storage, mirrored storage, and ending at a distributed database back-end.

**Acting as a Proxy**

This approach gives us one more possibility. We can implement remote storage, which would allow fdapd to act as a proxy. Instead to handle a request locally, using for example file storage, it would use the remote storage to relay the request to an upstream server. This however slightly complicates design of fdapd, as an fdapd instance must know which storage to use. We decided to bind this per listening socket. This should be variable, based on the configuration. In our proxy example setup (figure 3.2), the local proxy fdapd instance would be configured to open unix socket with remote storage, set on the socket. The upstream instance would be set to open a tls listening socket with some local storage set on the socket.

This design has one big advantage and one disadvantage. The disadvantage is that the protocol in which the remote storage talks to the upstream is not FDAP, as the storage interface is naturally different (it is more similar to some RPC mechanism). This means that if the FDAP protocol changes, we might have to extend the storage interface as well. On the other hand, the remote storage approach brings us a great flexibility in storage design, which sounds to us as a good trade-off.
3.4.3 Usage

The `fdapd` is just a proof-of-concept server implementation. It is not ready to be deployed in production yet. For now, it is able to run only in the foreground console mode. Therefore we do not care about things like correct daemonization of the process, as it is not the main subject of this thesis and it is quite complex procedure if we would want to do it correctly.

The `fdapd` is built along with all other binaries as a product of the build process. Please refer to subsection 3.2.4 for details. After successful build, it can be run from the project root directory.

It is recommended to run the daemon from the project root directory, as the example configuration has all paths set against this directory. So first, please, `cd` to the project root directory, and then run `fdapd` by typing the following command:

```
build/dbg/bin/fdapd -c conf/fdapd.conf
```

The `fdapd` needs root privileges, because `fdapd` tries to bind to low port numbers and to Unix-Domain Sockets in `/var/run/` directory.

The behavior of `fdapd` is determined by configuration, which is described next.

3.4.4 Configuration

The FDAP daemon loads its configuration on start-up from a file, usually named `fdapd.conf`. This file is searched by the following rules in order of precedence:

1. If a `-c` command-line option is present, the daemon loads a config file from that location.
2. Otherwise a config file `fdapd.conf` is searched in the current working directory.
3. Finally, if the previous file does not exists, a path `/etc/fdap/fdapd.conf` is examined.

Now, we will go through the configuration options and provide an example.

The configuration consists of sections. Each section has a type. There are two types of sections: `listen` and `tls`. Each section type implies some options inside the section that are valid for that type. Some of the options are required, some of them are optional (in such case a reasonable default value is set). But each option must appear at most once in a section.

The configuration supports inserting comments, by writing `#` character anywhere. It means that whole text, starting after `#`, till the end of line, is ignored by the configuration parser. Also white-spaces between tokens are ignored. Both comments and white-space characters act as a token separator.
storage {
    file: "conf/fdap.dir";
};

tls {
    cert: "conf/srv.cert.pem";
    pk: "conf/srv.key.pem";
    pk-pwd: "qwerty123";
    ca-certs: "conf/ca.cert.pem";
};

listen tls {
    host: "0.0.0.0";  # Listen on all IP addresses.
    port: 4433;
    limit: 5;
};

listen tcp {
    host: "127.0.0.1";
    port: 2244;
};

listen unix {
    path: "/var/run/fdapd/fdapd.sock";
    limit: 20;
};

Figure 3.5: FDAP Daemon Configuration Example

storage section

Configures a persistent storage for the directory. So far, only file storage is supported. This section is required.

The only available option is:

* file[STRING]: Required option. A path to the file in which the directory will be stored to ensure persistency of write operations.

listen section

Sets up a listening socket. We support three types of sockets:

* tls: Sets up listening TCP socket and establishes a TLS tunnel over each accepted TCP connection. If such socket is defined in a configuration, also tls section must appear in the configuration.

* tcp: Sets up listening TCP socket without any encryption.

* unix: Sets up listening Unix-domain socket for inter-process communication within the same machine.
Available configuration options are:

- **host[STRING]**: Required for tls and tcp. Specifies an IP address on which the daemon should bind for listening. The value can be any of locally-assigned IP addresses, even "127.0.0.1" or the special "0.0.0.0" wildcard address, which means *all assigned addresses.*

- **port[UINT]**: Required for tls and tcp. Specifies a TCP port on which the daemon should bind for listening. The value must be in range between 0 and 65535. When the value of 0 is used, port number is assigned dynamically by the operating system.

- **path[STRING]**: Required for unix. Specifies a file-system path on which the daemon should bind for listening. It is an administrator’s responsibility to ensure that the directory exists. The socket pseudo-file is created automatically; if the file already exists, it will be overwritten. Care must be taken of file-system permissions to be set appropriately.

- **limit[UINT]**: Optional for all listen types. The default value is 10. Implies a limit on the listening socket to restrict maximum number of simultaneously connected clients, that were accepted by that socket. When the limit is triggered, no further connections are accepted on the socket until some other connection is terminated. Pending connections are buffered in a queue in the order they reached the server.

**tls section**

This section must appear in the configuration file if one or more listen tls section(s) are defined in the file. The available configuration options are:

- **cert[STRING]**: Required. Path to a file that contains server certificate in X.509 format, PEM encoding.

- **pk[STRING]**: Required. Path to a file that contains corresponding private key to the certificate, in PKCS format, PEM encoding.

- **pk-pwd[STRING]**: Optional. The contents of the pk file can be encrypted. If so, the password must be supplied in this option for proper decryption.

- **ca-certs[STRING]**: Optional. Path to a file that contains trusted certification authorities certificate chain in X.509 format, PEM encoding. The file usually contains multiple CA certs, chained together. The file must contain a CA certificate that signed the server certificate in order to make the TLS module functional. If this option is omitted, several system default paths are examined to load the CA certificates.

All paths can be either relative to the current working directory or absolute.


3.5  FDAP Client Library

3.5.1  Application Programming Interface (API)

For client library API description, please refer to the source code. Relevant
header files are: fdap.h, request.h, iter.h, and record.h. For use in an ap-
lication, only the first one (fdap.h) needs to be included, as the rest is included
transitively.

For example usage of the client library, please look at the FDAP diagnostic
utility source code.

3.5.2  Usage

libfdapc is built along with all other binaries as a product of the build process.
Please refer to subsection 3.2.4 for details. The behavior of fdapclib is influenced
both by the caller of the library and by the configuration, which is described next.

For an example of client library usage in code, please look at the implement-
tation of the FDAP command-line utility.

3.5.3  Configuration

Client configuration is loaded by the FDAP client library each time the library
is loaded and executed by a caller (another library or application). The configu-
ration is obtained as follows:

1. It can be provided by the caller of the client library, using configuration API
   at runtime. A caller can provide either configuration options or an explicit
   path to a file from which the configuration should be parsed.

2. Otherwise, a default system location is examined: /etc/fdap/fdapc.conf.

Now, we will go through the configuration options and provide an example.

```
upstream tls {
    host: "127.0.0.1";
    port: 4433;
    ca-cert: "conf/ca.cert.pem";
    skip-cn-verify: yes;
    skip-verify: no;
};

upstream tcp {
    host: "localhost";
    port: 2244;
};
```

**Figure 3.6:** FDAP Client Configuration Example

The configuration follows the same syntactic rules as the daemon configuration
in section 3.4.4. It consists of one or more `upstream` sections.
upstream section

Configures upstream server to connect. The upstreams are examined in sequential order and the lookup stops at the first successful upstream. We support three types of upstreams:

- **tls**: Connects to the remote server using TCP and then establishes TLS tunnel with the server.
- **tcp**: Connects to the remote server using TCP without any encryption.
- **unix**: Connects to a local instance of FDAP server daemon running on the same host using Unix-domain sockets.

Available configuration options are:

- **host[STRING]**: Required for **tls** and **tcp**. Specifies an IP address of the server to which the client should connect.
- **port[UINT]**: Required for **tls** and **tcp**. Specifies a TCP remote port to which the client should connect. The value must be in range between 1 and 65535.
- **path[STRING]**: Required for **unix**. Specifies a file-system path to which the client should connect. Care must be taken about file-system permissions to be set appropriately.
- **ca-certs[STRING]**: Optional for **tls**. Path to a file that contains trusted certification authorities certificate chain in X.509 format, PEM encoding. The file usually contains multiple CA certs, chained together. The file must contain a CA certificate, which signed the server certificate in order to make TLS module functional. If this option is omitted, several system default paths are examined to load the CA certificates.
- **skip-cn-verify[yes|no]**: Optional for **tls**. Default is **no**. Specifies whether to skip verification of the common name (CN) field in the server certificate against server hostname, specified by **host** option. Normally, these two must match in order to the TLS handshake be successful. However, sometimes it can be useful to turn off this check, for example when the **host** option is an IP address and the certificate is issued for a hostname.
- **skip-verify[yes|no]**: Optional for **tls**. Default is **no**. Specifies whether to skip verification of the server certificate issuer – if it was signed by trusted CA root certificate. Normally, the CA root certificate must be present in **ca-certs** file in order to be TLS handshake successful. For debugging purposes, it might be useful to turn off this check. Do not do this in production! If set to **yes**, both **ca-certs** and **skip-cn-verify** options will not take any effect.

All paths can be either relative to the current working directory or absolute. Note that current working directory is determined by the actual application, which loaded the library, so using an absolute path is the recommended way in this case.
3.6 FDAP Command-line Utility

3.6.1 Usage

`fdapdiag` is built along with all other binaries as a product of the build process. Please refer to subsection 3.2.4 for details. After a successful build, it can be run from the project root directory.

It is recommended to run the utility from the project root directory, as the example configuration has all paths set against this directory. So first, please, `cd` to the project root directory, and then run `fdapdiag` by typing the following command:

```
build/dbg/bin/fdapdiag -c conf/fdapc.conf
```

Connecting to a Server

When the client starts, it tries to connect to one of the specified FDAP servers, according to the configuration. The configuration can be passed to the utility in two ways.

1. By the `-c` option and the config file, as in the example. Available options of the config file are described in the previous subsection 3.5.3.

2. Directly on the command-line by specifying several options, as described in figure 3.7.

<table>
<thead>
<tr>
<th>Option</th>
<th>Value</th>
<th>Meaning</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-c</code></td>
<td>STRING</td>
<td>Config. file</td>
<td><code>-c conf/fdapc.conf</code></td>
</tr>
<tr>
<td><code>-t</code></td>
<td>tls</td>
<td>Transport type</td>
<td><code>-t tls</code></td>
</tr>
<tr>
<td><code>h</code></td>
<td>STRING</td>
<td>Server name or IP</td>
<td><code>-h 127.0.0.1</code></td>
</tr>
<tr>
<td><code>-p</code></td>
<td>INT</td>
<td>Server TCP port</td>
<td><code>-p 636</code></td>
</tr>
<tr>
<td><code>-P</code></td>
<td>STRING</td>
<td>unix socket path</td>
<td><code>-P /var/run/fdap.sock</code></td>
</tr>
<tr>
<td><code>-a</code></td>
<td>STRING</td>
<td>CA cert. file</td>
<td><code>-a conf/ca.certs.pem</code></td>
</tr>
</tbody>
</table>

Figure 3.7: Utility command-line options

If both `-c` and other options are specified, the config file always takes precedence.

Interactive Mode

After the connection to the server is successfully started, the utility enters interactive mode. This means that the user can enter FDAP commands and the utility will send them to the connected server. When the response arrives, the utility will show the textual representation of the response to the screen.

The utility takes the advantage of filter textual format (see appendix A) and CBOR diagnostic notation (see [BH13]). With slight addition of its own syntax, it forms sufficient language for expressing FDAP requests and responses in

\(^2\)Textual names for FDAP operations, quotes and semicolon.
human-readable format. Available operations are identical to the FDAP operations and their exact format is specified in the figure 3.8.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Arguments</th>
<th>Argument type</th>
</tr>
</thead>
<tbody>
<tr>
<td>auth</td>
<td>&quot;username&quot; &quot;password&quot;</td>
<td>Both strings</td>
</tr>
<tr>
<td>search</td>
<td>&quot;filter&quot;</td>
<td>Filter textual format</td>
</tr>
<tr>
<td>get</td>
<td>&quot;id&quot;</td>
<td>Unsigned Integer</td>
</tr>
<tr>
<td>create</td>
<td>&quot;entry&quot;</td>
<td>CBOR diag. notation</td>
</tr>
<tr>
<td>update</td>
<td>&quot;id&quot; &quot;entry&quot;</td>
<td>Uint. and CBOR diag. notation</td>
</tr>
<tr>
<td>delete</td>
<td>&quot;id&quot;</td>
<td>Unsigned Integer</td>
</tr>
</tbody>
</table>

**Figure 3.8:** Utility supported interactive commands

Note that formally, unsigned integers and strings are also subset of CBOR diag. notation.

The mapping between the operation textual name and FDAP operation should be obvious. Note that each argument must be enclosed in quotes and each command must be terminated by a semicolon. If the argument does not contain whitespace characters, the quotes can be omitted. Example commands are shown in figure 3.9.

```
auth "hrubyon" "qwerty123";
search "age > 30 & position != 'manager'";
```

**Figure 3.9:** Utility example commands

The utility can be quit by pressing Control+D or Control+C.

A set of example commands can be found in the examples/ directory. As the utility reads its stdin, one can take advantage of the cat command as shown in figure 3.10.

```
cat examples/nss.in - | build/dbg/bin/fdapdiag -c conf/fdapc.conf
```

**Figure 3.10:** Putting an instruction file to the utility’s stdin

In the examples/nss.in file, proper commands are stored for initializing a directory suitable for the use with our NSS-FDAP module.
Chapter 4
Client Application Example

4.1 Name Service Switch (NSS)

Name Service Switch (NSS) is a system facility implemented by all common Unix-like operating systems. Its purpose is to manage and control access to various system databases, such as the system user database, user group database, local DNS resolution database (hosts entries), etc. Each time a database is queried, the request is targeted to NSS and then routed by NSS to one or more NSS modules (based on NSS configuration) to handle the request.

NSS was first developed by Sun Microsystems for their Solaris operating system. It was then ported to many other distributions and operating systems, as the previous solution was to hardcode the rules for accessing the databases into the OS source code. Now, it is also a part of the GNU C library.

NSS is based upon the concept of databases and services, which is described further.

4.1.1 Databases

A database is a collection of entries of specific structure (examples were given – users, groups, etc.). For each database there is a corresponding C struct defined within the GNU C library. See figure 4.2 for structure definitions.

Different databases can be queried in their own specific way, based on the data fields that the database holds. NSS exposes standard client API, that is used by applications, which are interested in particular system database(s). There are two approaches in which the database(s) can be queried:

1. Direct query based on a value of a concrete database field.
2. Enumerative query that explores a database in sequential fashion.

The client API consists of a set of C functions. Their names reflect either the query type (direct or enumerative) and either the target database (and in the case of a direct query either the target database field). For direct queries, the function naming pattern varies for different databases, but usually the function name starts with get and is followed by an abbreviation of the database name. For example:

```c
/* Fetches user entry from passwd database by given username. */
struct passwd *getpwnam(const char *name);

/* Fetches DNS entry from hosts database by given hostname. */
struct hostent *gethostbyname(const char *name);
```

Figure 4.1: NSS client API – direct queries naming scheme example
Whereas for enumerative queries, the naming pattern seems to be always the same. There are always three functions `getDBent`, `setDBent` and `endDBent`, where DB is an abbreviation of concrete database name. `getDBent` advances the enumerator to the next position – it is intended to be called in a sequence. `setDBent` rewinds to the beginning of the database and should be called before an enumeration starts. Once it has finished, `endDBent` should be called to close the connection to the database.

```c
struct passwd
{
    char  *pw_name;   // User's login name.
    uid_t pw_uid;     // Numerical user ID.
    gid_t pw_gid;     // Numerical group ID.
    char  *pw_dir;    // Initial working directory.
    char  *pw_shell;  // Program to use as shell.
};

struct spwd
{
    char  *sp_namp;   // Login name.
    char  *sp_pwdp;   // Encrypted password.
    long int sp_lstchg; // Date of last change.
    long int sp_min;  // Minimum number of days between changes.
    long int sp_max;  // Maximum number of days between changes.
    long int sp_warn; // Number of days to warn user to change
    //    the password.
    long int sp_inact; // Number of days the account may be
    //    inactive.
    long int sp_expire; // Number of days since 1970-01-01 until
    //    account expires.
    unsigned long int sp_flag; // Reserved.
};

struct group
{
    char  *gr_name;   // The name of the group.
    gid_t  gr_gid;   // Numerical group ID.
    char  **gr_mem;  // Pointer to a null-terminated array of
    //    character pointers to member names.
};
```

**Figure 4.2:** System structures, representing NSS database entries [POS18]

In the examples given, the resulting structures are always allocated using static storage and the pointer to the storage is returned by the functions. NSS provides also “reentrant” variants of all functions, which allocate the memory in a different way. Names of these functions are ended with `_r` suffix.
void setpwent(void);
struct passwd *getpwent(void);
void endpwent(void);

Figure 4.3: NSS client API – enumerative queries naming scheme example

int getpwnam_r(const char *name, struct passwd *pwd,
                char *buf, size_t buflen, struct passwd **result);

Figure 4.4: NSS client API – reentrant function example

A caller of a reentrant function is responsible for allocating the memory for the result and passing the pointer to the function. All string fields that are allocated along with the structure are stored into the buffer buf also provided by the caller. If the call succeeds, a pointer to the resulting structure is returned in the result out-parameter. The function returns an integer code, which informs the caller whether the call succeeded or not.

<table>
<thead>
<tr>
<th>DB name</th>
<th>Header file</th>
<th>Available NSS API functions</th>
</tr>
</thead>
</table>
| passwd (pw) | <pwd.h> | struct passwd *getpwuid(uid_t uid);
| | | struct passwd *getpwnam(char *name);
| | | struct passwd *getpwent(void); † |
| shadow (sp) | <shadow.h> | struct spwd *getspnam(char *name);
| | | struct spwd *getspent(void); † |
| group (gr) | <grp.h> | struct group *getgrgid(gid_t gid);
| | | struct group *getgrnam(char *name);
| | | struct group *getgrent(void); † |

Figure 4.5: NSS client API – summary of API functions for selected DBs

†) Note that corresponding setDBent and endDBent functions (where DB ∈ {pw, sp, gr}) come together with getDBent and are also part of the API.

4.1.2 Services

A service is a NSS module, a shared library, that NSS uses for querying one or more databases, based on its configuration. A service can be viewed as a data source or a sort of gateway for given database(s). Behavior of NSS is controlled by the /etc/nsswitch.conf configuration file.

Each line starts with a database name, followed by a list of services used as a data sources for the database. Whenever NSS receives a client API call, for each service, which appears in the config file, the corresponding NSS module library is searched by NSS in the /lib directory: If the service name is, for example, fdap, NSS will try to find libnss_fdap.so.2, where trailing 2 is the version (see manual page [Fre18]). If the module library is found, it is dynamically linked to the executing client process and proper functions are called in the module. Also
passwd: files mymachines systemd fdap
group: files mymachines systemd
shadow: files fdap

hosts: files mymachines resolve [!UNAVAIL=return] dns myhostname

**Figure 4.6:** NSS configuration file example (/etc/nsswitch.conf)

the module library must be compiled with the `soname` field (see ELF [ELF95] for
details).

Note that if two independent processes query the same NSS module in parallel, each process
links its own copy of the module library into its address space. Also, all the NSS client API
functions are not guaranteed to be thread-safe as written in the GNU C library manual [Fre18].
Therefore, there is no need for synchronization within the module source code.

The *service* modules are loaded and queried in the order they appear in the
configuration file. Default policy is that if a preceding module returns a valid
entry, the lookup stops there and the entry is returned to the original caller
of the NSS client API. If the lookup fails, querying continues with the next
module. However, this behavior can be overridden by custom rules for each module
independently, as shown in the example ([!UNAVAIL=return] – see the man-
page [Fre18]).

NSS maps client API calls to the module API calls in a straightforward way.
If the *service* name is `fdap`, for example, all `fdap` module function names are pre-
fixed with `_nss_fdap`. Also, module functions always return `enum nss_status`
code which involves the lookup flow control as described.

```c
/* NSS client API function */
int getpwnam_r(const char *name, struct passwd *pwd,
               char *buf, size_t buflen, struct passwd **result);

/* Corresponding NSS fdap module API function */
enum nss_status _nss_fdap_getpwnam_r(const char *name,
                                     struct passwd *pwd, char *buffer, size_t buflen, int *errnop);

/* errnop is passed by NSS as a return value of the first function */
```

**Figure 4.7:** NSS example mapping between client API and module API

As a canonical example, we will mention the `files` module, which comes
with all GNU/Linux distributions. It is, by default, configured as the first
queried module in the `/etc/nsswitch.conf` file. This module uses well-known
files `/etc/passwd`, `/etc/group`, `/etc/shadow`, `/etc/hosts`, etc. as a database
back-end.
4.1.3 NSS-FDAP Module

libnss_fdap.so is built along with all other binaries as a product of the build process. Please refer to subsection 3.2.4 for details. After successful build, the library can be installed (along with pam_fdap.so library), using the command:

\texttt{make install}

This command needs root privileges, as the script copies the libraries to the system location /lib/.

After library installation, the NSS configuration file needs to be modified. Please append fdap module name to the /etc/nsswitch.conf file as shown in figure 4.6. NSS-FDAP module enables FDAP access to the passwd and shadow databases. Support for the group database is experimental, and may not be fully functional.

Then the NSS module can be tested by executing various commands, such as passwd -Sa. Prior testing the NSS module, the contents of the directory can be modified by creating user entries, using the command-line diag. utility.
Figure 4.9: FDAP entries that represent NSS structures

In figure 4.9 we can see directory entries in CBOR diag. notation, in the form, in which they are expected by the NSS module. Please keep this form when adding entries to the directory for NSS purposes. The mapping between the entries and system structures from figure 4.2 should be obvious.

Note that most of the fields of the spwd system structure are omitted in the directory entry, as these fields are not used by most Linux distributions.

Also notice the read-only section. There are attributes that the user should not change in typical deployment, as these attributes should be managed by system administrator.

The FDAP does not enforce any hashing algorithm, as the requirements of client applications may differ. The password here is stored as SHA-512 hash hexadecimal textual string, because of the compatibility with our PAM-FDAP module, which uses SHA-512 hashing algorithm. The hashed string here is “querty123” and it is used as a password in all data samples provided in the examples/ directory.
<table>
<thead>
<tr>
<th>NSS API function</th>
<th>FDAP op.</th>
<th>FDAP parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>getpwuid_r(uid)</td>
<td>search</td>
<td>read-only.uid = 'uid', read-only.username = 'name', read-only.username &gt; ''</td>
</tr>
<tr>
<td>getpwnam_r(name)</td>
<td></td>
<td>read-only.username = 'name', read-only.username &gt; ''</td>
</tr>
<tr>
<td>getpwent_r()</td>
<td></td>
<td>read-only.username &gt; ''</td>
</tr>
<tr>
<td>getspuid_r(uid)</td>
<td>search</td>
<td>read-only.uid = 'uid', read-only.username = 'name', read-only.username &gt; ''</td>
</tr>
<tr>
<td>getspnam_r(name)</td>
<td></td>
<td>read-only.username = 'name', read-only.username &gt; ''</td>
</tr>
<tr>
<td>getspent_r()</td>
<td></td>
<td>read-only.username &gt; ''</td>
</tr>
<tr>
<td>getggrgid_r(gid)</td>
<td>search</td>
<td>gid = 'gid', groupname = 'name', groupname &gt; ''</td>
</tr>
<tr>
<td>getgrnam_r(name)</td>
<td></td>
<td>groupname = 'name', groupname &gt; ''</td>
</tr>
<tr>
<td>getgrent_r()</td>
<td></td>
<td>groupname &gt; ''</td>
</tr>
</tbody>
</table>

Figure 4.10: NSS API functions mapping to FDAP operations

Note that all mentioned function names should be prefixed by _nss_fdap_ in the source code. The prefixes are not written in the table for better clarity.

4.2 Pluggable Authentication Modules (PAM)

Pluggable Authentication Modules (PAM) is an authentication framework for UNIX-like operating systems. Its purpose is to outsource authentication task from applications to the set of authentication modules provided by PAM. Unlike NSS, which provides a system-wide service, PAM must be configured per application. Every time an application requires authentication, it can take advantage of the PAM library instead, and PAM, based on its configuration, routes the authentication request to the specific module(s). The module then performs the authentication operation against some back-end, and informs the original application about the result. Also authentication related tasks, such as password change, are supported and can be handled by PAM.

It is been a very long time since we last mentioned Sun Microsystems company. Again, PAM was first developed in Sun Microsystems in 1995, and it was first released in 1997. That year, the Linux-PAM project was started, and since then it is common part of all GNU/Linux operating systems.

Next, we will discuss the two sides of the PAM coin. First, the PAM application API, and next we will discuss PAM modules.

4.2.1 Application API

PAM’s functionality is divided into four interfaces. Each interface represents a unit of functionality that is exposed by PAM in the form of API functions. In the figure 4.11 the client side of the PAM API is shown together with the division of API functions into interfaces.
Prior to calling any of these functions, the PAM-aware application must start a PAM session first. This is done by calling the `pam_start` function. By the first argument, the application selects the PAM configuration file in `/etc/pam.d/` directory. Suppose that the argument is `my_app`. Then the configuration file `/etc/pam.d/my_app` is selected when calling `pam_start` function. The config file contains a table with three columns and many rows. The rows are examined in sequential order and each row triggers one authentication module to start if certain conditions are met. The conditions are specified by control flags and the flags also allow stacking of the modules. See figure 4.12 for example configuration file.

The PAM-aware application needs to pass the authentication information in some way to the module. The username is passed as a second argument of function `pam_start`. Another way of passing data between application and modules are functions `pam_get_item` and `pam_set_item`. This is how the password can be exchanged.

After the application has called `pam_start` and has passed appropriate data to the PAM using `pam_set_item`, one or more interface functions (see table 4.11) can be called. The function returns a PAM-specific result code. From the result

<table>
<thead>
<tr>
<th>Interface</th>
<th>API function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>auth</td>
<td>pam_authenticate</td>
<td>Authenticate the user</td>
</tr>
<tr>
<td></td>
<td>pam_setcred</td>
<td>Set credentials to the user</td>
</tr>
<tr>
<td>account</td>
<td>pam_acct</td>
<td>Validate account state (i.e. login hours)</td>
</tr>
<tr>
<td>session</td>
<td>pam_open_session</td>
<td>Set-up session params. (i.e. default shell)</td>
</tr>
<tr>
<td></td>
<td>pam_close_session</td>
<td>Close session callback</td>
</tr>
<tr>
<td>password</td>
<td>pam_chauthtok</td>
<td>Change the password</td>
</tr>
</tbody>
</table>

Figure 4.11: PAM API – client side

Figure 4.12: PAM configuration file
code, we can tell if the user was (for example) authenticated or not. After the application has done its PAM business, it should call `pam_end` to terminate the PAM session. Details about PAM API can be found at PAM documentation [RFC95].

There is one more mechanism implemented in PAM: a conversation function. It is a client application's callback – a function pointer that is passed to the module when calling the `pam_start`. It can be called by the module if it needs to pass an information back to the original application or it just needs to pass control to the original application for a while. Consider a GUI application and a module, which reports some ongoing information about the authentication progress. We would like to draw these informations in the GUI as it goes. But the module does not know (and should not know) whether the calling application is a console, GUI application, or whatever it is. By just calling the conversation function, it is left to the original application, what to do with the data from the module.

### 4.2.2 Modules

A PAM module is basically a shared library, which is dynamically linked to the calling application through the PAM framework when needed. PAM module libraries are placed in `/lib/security` directory. The module is loaded when a client calls the `pam_start` function, based on the configuration file. For example: if the client passes `my_app` as the first argument of the `pam_start` function, the configuration file `/etc/pam.d/my_app` is examined. Let us say that there is `pam_fdap.so` in some row, in the third column of the config file. Then the library `/lib/security/pam_fdap.so` will be loaded. The library must be compiled with the `soname` option (see ELF [ELF95] for details).

The function mapping between PAM-aware applications and PAM modules is very simple. Each function mentioned in table 4.11 has its equivalent in the PAM module. We will demonstrate it on the example: if a client calls a function named `pam_authenticate`, a function named `pam_sm_authenticate` is called in the proper module. So the application API functions and the module functions differ only in the prefix, which is always the same. Each PAM module function has the same signature – see figure 4.13. If some module implements one function of an PAM interface, it should also implement all other functions within the same interface, even if the implementation of them would be trivial. For example: if function `pam_sm_authenticate` is implemented by module, `pam_sm_setcred` should be implemented as well.

```c
PAM_EXTERN int pam_sm_authenticate(pam_handle_t *pamh, int flags, int argc, const char **argv)

PAM_EXTERN int pam_sm_acct_mgmt(pam_handle_t *pamh, int flags, int argc, const char **argv)

PAM_EXTERN int pam_sm_chauthtok(pam_handle_t *pamh, int flags, int argc, const char **argv);
```

**Figure 4.13:** PAM module API function examples

The first argument is the PAM handle context object passed to all PAM
functions by the application. The handle is initialized by calling `pam_start` and then passed to all successive calls. The second argument are flags, passed to the functions by the application. The flags are used to parametrize the request of the calling application. Last two arguments are module arguments passed to the function from the PAM configuration file (see figure 4.12).

Well-known system applications that use PAM are for example system `login` process or `su` – switch user command.

### 4.2.3 PAM-FDAP Module

The `pam_fdap.so` is built along with all other binaries as a product of the build process. Please refer to subsection 3.2.4 for details. After successful build, the library can be installed (along with `libnss_fdap.so` library), using the command:

```
make install
```

This command needs root privileges, as the script copies the libraries to the system location `/lib/security/`.

After library installation, the PAM configuration file needs to be modified. Please choose some application\(^1\) on which you would like to test the module. Then append the line shown in figure 4.14 to the beginning of the configuration file that belongs to the application. The name of the configuration file should have the same name as the application (i.e., `su`). All PAM configuration files are stored in `/etc/pam.d/` directory.

```
auth sufficient pam_fdap.so
```

**Figure 4.14:** PAM configuration file fragment

Then the PAM module can be tested by executing the command, for which you have configured the FDAP module. Prior testing the PAM module, the contents of the directory can be modified by creating user entries, using the command-line `diag` utility. For now, the PAM-FDAP module implements only the `auth` PAM interface.

Note that the PAM-FDAP module computes an SHA-512 hash from the user-provided plain text password. This hash is then compared to the SHA-512 hash from the FDAP directory as a part of the authenticate operation. For simplicity, we do not use any kind of cryptographic salt. The module uses the `mbedtls` library to compute the hash.

---

\(^1\) `su` command could be a good choice.
Conclusion

The goal of this work was to explore existing directory services and related network protocols and analyze their functionality. Two significant services exist: the Network Information Service (NIS) and the Lightweight Directory Access Protocol (LDAP). The former suffers from numerous security problems, the latter is the de-facto industry standard solution today despite being over-engineered and complicated, an issue we sought to address.

Following the research, we have decided to design our own directory access protocol which would be much simpler, secure and would work well as a back-end database for user accounts with the ability to authenticate the users, lending itself to applications such as single sign-on.

The protocol was designed to support authentication of user accounts and the storage, modification and retrieval of arbitrarily complex data items. The protocol itself was influenced by, and relies on, the Concise Binary Object Representation.

To demonstrate the practicality of the protocol, a suite of several programs was created. Together they prove that the protocol can be implemented and provide rudimentary authentication capabilities and other directory services to UNIX-like operating systems by plugging into the Pluggable Authentication Modules (PAM) and Name Service Switch (NSS) subsystems. As a whole, the implementation is fairly complex, totalling about 300 KiB of source code; however, the parts of the implementation dealing with the protocol are very simple, implying that our initial goal of designing a simple protocol was met. The implementation provided is well documented and tested and can become a solid foundation for a production-ready version in the future.

During the development phase, several minor insufficiencies were identified in the protocol which do not render it dysfunctional, but could pose a problem for its future development:

1. We decided to include an additional CBOR map at the end of every request which will provide an extension point: if any request fields will be needed in future protocol versions, they can be safely discarded by older implementations and well understood by newer ones.

2. The storage (mentioned in subsection 3.4.2) communication interface is not part of the FDAP specification. We would like to extend the FDAP specification with the storage communication interface in the future to allow for inter-operability with other FDAP proxy implementations.

We will focus on removing these shortcomings in the first stable version of the protocol. At present time, both the protocol and the implementation are considered experimental and subject to change in the future, it’s not recommended to use them in production yet.

Finally, to facilitate development of programs using our protocol, a client library called libfdapc is provided, which should make the integration easy for other developers.
Appendix A
Filter Textual Notation Syntax

In this appendix, we will describe filter textual notation formally, using EBNF [ISO93] syntactic metalanguage. Definition of each symbol of the grammar (terminal or non-terminal) is followed by a syntax diagram, describing the rule. This appendix is divided in two parts: First, all non-terminal symbols are defined; in the second part the terminal symbols.

Note that the following grammar definitions and diagrams do not cover semantic analysis. It must be done separately after the syntactic analysis.

A.1 Non-terminals

\[
\text{filter ::= and | or | expr} \\
\text{and ::= and ' & ' expr | expr ' & ' expr} \\
\text{or ::= or '|' expr | expr '|' expr}
\]

![Figure A.1: filter](image)

\[
\text{expr ::= '(' filter ')' | '! '(' filter ')'} | \text{cond}
\]

![Figure A.2: expr](image)
cond ::= attrName OPERATOR value

Figure A.3: cond

attrName ::= IDENTIFIER ('.' IDENTIFIER)*

Figure A.4: attrName

value ::= STRING | INT | SIMPLE

Figure A.5: value
A.2 Terminals

 IDENTIFIER ::= [a-zA-Z]\[a-zA-Z0-9-]*

 OPERATOR ::= '<' | '<=' | '=' | '!=' | '>=' | '>'

 Figure A.6: IDENTIFIER

 OPERATOR ::= '='< | '<=' | '=' | '!=' | '>=' | '>'

 Figure A.7: OPERATOR
STRING ::= "'"("|"\"|"\"|\"\"|\"\"|\"\")\"\""
Appendix B
API Documentation

B.1 src/cbor.c

Enum major

Major type of a CBOR item.

```c
enum major {
    MAJOR_UINT,
    MAJOR_NEGINT,
    MAJOR_BYTES,
    MAJOR_TEXT,
    MAJOR_ARRAY,
    MAJOR_MAP,
    MAJOR_TAG,
    MAJOR_7,
    MAJOR_NOTSET,
};
```

Func strmajor

Return a human-readable name for major type major.

```c
static const char *strmajor(byte_t major)
```

Func dump_sval

Return a human-readable name for simple value sval.

```c
static char *dump_sval(enum cbor_sval sval)
```

Func err

Invoke the error handler and set a formatted error message. The error message can later be retrieved by a call to cbor_strerror.

Any further operations on c with the exception of cbor_strerror and cbor_free are undefined and most likely fatal.

```c
noreturn static void err(struct cbor *c, char *msg, ...)
```

Struct scope

Certain CBOR items, namely byte and text streams, arrays and maps are usually not encoded or decoded using a single call to the decoder. For example, to decode a byte stream, one has to start by calling one of the cbor_bytes_read_start_* functions, followed by several calls to cbor_bytes_read calls, followed by a call to cbor_bytes_read_end.

To keep track of the state of the encoding/decoding process, the scope structure is introduced. This allows us to spot common errors, such as reading too little from a byte stream or writing more items to an array than its declared size is, etc.

The u64 and counter members are item-type-specific and are used for this accounting. For byte and text streams, for example, these amount to the number of bytes in the stream and the number of bytes read/written so far, respectively.
The `auto_end` flag is true precisely when the scope was not started by the user (by calling some `cbor.*start_*` function) but rather by internal code, so internal code should call the appropriate function to end the context, too.

```c
struct scope {
    byte_t hdr; /* header of the item which started the scope */
    bool auto_end; /* should this scope be ended automatically? */
    uint64_t u64; /* quantity (number of items, bytes, ...) */
    uint64_t counter; /* counter (read items, written bytes, ...) */
};
```

**Func scope**
Return the currently open scope. If the scope stack is empty, it is an error.

```c
static struct scope *scope(struct cbor *c)
```

**Func no_scope**
Returns true if there’s no scope open.

There’s always one scope: the bottom-most one, which is opened by `cbor_init`. This is an artificial scope with `hdr` set to `BREAK` (because no other scope can have such `hdr` value). It was added so that when we increment the number of items read/written in current scope in `read_hdr` and `write_hdr`, we don’t have to worry about whether the scope stack is empty or not. This makes the process faster, too, because branching on the hot path is reduced.

```c
static bool no_scope(struct cbor *c)
```

**Func start_scope**
Start a new scope for an item with header byte `hdr` and quantity `u64`. If the item uses indefinite-length encoding, set `u64` to 0.

```c
static void start_scope(struct cbor *c, byte_t hdr, uint64_t u64)
```

**Func end_scope**
End current scope. If the scope stack is empty, it is an error.

```c
static void end_scope(struct cbor *c)
```

**Func scope_remains**
Return the number of remaining items in scope `s`.

```c
static uint64_t scope_remains(struct scope *s)
```

**Func scope_check**
Ensure that current scope has major type `major` and return it.

```c
static struct scope *scope_check(struct cbor *c, byte_t major)
```

**Func read_cbor**
Read `nbytes` bytes from the underlying buffer.

If end-of-file is reached and `nbytes > 0`, it is an error. If an error occurs during the read operation on the buffer, `err` will be called.

```c
static void read_cbor(struct cbor *c, void *buf, size_t nbytes)
```
Func read_byte
This function has the same semantics as read, but only reads a single byte which it returns.
static inline byte_t read_byte(struct cbor *c)

Func read_hdr
Read the header byte and increase current scope counter.
static byte_t read_hdr(struct cbor *c)

Func peek
Return the next byte in the CBOR stream without consuming it.
static inline byte_t peek(struct cbor *c)

Func break_follows
Is the next item a break code?
static inline bool break_follows(struct cbor *c)

Func check_type
Check that the given header byte hdr has major type set to major. If indef is either a positive number or zero, check that hdr marks an item which is indefinite- or definite-length, respectively. (If indef is negative, don’t check for definiteness.)
static void check_type(struct cbor *c, byte_t hdr, byte_t major, int indef)

Func read_hdr_u64
Read next item’s header and return it. Also, decode the related quantity and store it into *u64.
If the item uses indefinite-length encoding, *u64 will be set to a 0.
static byte_t read_hdr_u64(struct cbor *c, uint64_t *u64)

Func read_uint
Read an unsigned integer. Require that it is less than or equal to max, otherwise it’s an error.
static uint64_t read_uint(struct cbor *c, uint64_t max)

Func read_int
Read an integer. This can be either an unsigned int (major type 0) or a negative int (major type 1) in the CBOR stream. In any case, require that the resulting integer lies between min and max inclusive, otherwise it’s an error.
static int64_t read_int(struct cbor *c, int64_t min, int64_t max)
**Func read_stream**

Read either a text or a byte stream (depending on major). This function reads the logical stream: if an indefinite-length stream is being read, it correctly decodes the intervening chunks.

```c
static size_t read_stream(struct cbor *c, byte_t major, void *dst, size_t nbytes)
```

**Func read_stream_alloc**

An allocating wrapper for the stream reading functions. Reads a whole logical byte or text stream (depending on major) and stores it in *buf. The total length of the decoded stream is returned. (Excluding the trailing NUL byte in case of text streams.)

This function will NUL-terminate the stream, even if it’s a byte stream. This is necessary for strings and does not do any harm for byte strings. It can however hide some plus one errors in the code that reads the stream from sanitizers.

```c
static uint64_t read_stream_alloc(struct cbor *c, byte_t major, byte_t **buf)
```

**Func write_cbor**

Write nbytes bytes to the underlying buffer. If an error occurs during the read operation on the buffer, err will be called.

```c
static void write_cbor(struct cbor *c, void *buf, size_t nbytes)
```

**Func write_byte**

This function has the same semantics as write, but only writes a single byte.

```c
static inline byte_t write_byte(struct cbor *c, byte_t b)
```

**Func write_hdr**

Write a header and increase current scope counter.

```c
static byte_t write_hdr(struct cbor *c, byte_t major, byte_t minor)
```

**Func cbor_read_item**

Read an item from the stream, store decoded information into item. Please note that this operation allocates further memory as needed and that item should always be disposed with a call to cbor_item_free.

```c
void cbor_read_item(struct cbor *c, struct cbor_item *item)
```

**Func cbor_item_get_text**

Return the textual content of a MAJOR_TEXT item.

```c
char *cbor_item_get_text(struct cbor_item *item)
```

**Func cbor_item_get_text_escaped**

Return an escaped copy of the textual content of a MAJOR_TEXT item.

```c
char *cbor_item_get_text_escaped(struct cbor_item *item)
```
Func cbor_write_item
Write item to the given CBOR stream c.
void cbor_write_item(struct cbor *c, struct cbor_item *item)

Func cbor_item_dump
Dump item to the string buffer str in CBOR diagnostic notation.
void cbor_item_dump(struct cbor_item *item, struct strbuf *str)

Func cbor_item_cmp
Compare given items a and b. A negative value, zero and a positive value are returned, if, respectively, b is less than, equal to or greater than a. The sorting rules are best depicted by the source code itself.
int cbor_item_cmp(struct cbor_item *a, struct cbor_item *b)

B.2 src/fdapd.c

Func new_socket
Allocate new socket structure.
static struct socket *new_socket(void)

Func socket_epoll_ctl
A simple helper around epoll_ctl. Call the op operation on the epoll multiplexer, configuring events for socket.
static void socket_epoll_ctl(struct socket *s, int op, uint32_t events)

Func socket_subscribe
Subscribe socket to events of epoll. The socket has to be registered by a previous EPOLL_CTL_ADD operation.
static void socket_subscribe(struct socket *s, uint32_t events)

Func socket_unsubscribe
Unsubscribe socket from events of epoll. The socket has to be registered by a previous EPOLL_CTL_ADD operation.
static void socket_unsubscribe(struct socket *s, uint32_t events)

Func start_socket
Register s with epoll and start receiving (and handling) events for it.
static void start_socket(struct socket *s)

Func destroy_socket
Unregister s from epoll, invoke a socket-specific tear-down hook and delete s.
static void destroy_socket(struct socket *s)
B.2.1 TCP/IP listening sockets

**Func tcp_accept**

Accept incoming TCP connection on the socket `s` and register it.

```c
static void tcp_accept(struct socket *s)
```

**Func tcp_listen**

Start listening on the socket `s`. Bind to this machine’s `hostname` and `port`. Returns a boolean indicating success.

```c
static bool tcp_listen(struct socket *s, const char *hostname, const char *port)
```

**Func tcp_listen_start**

Start listening for incoming TCP connections on this machine’s `hostname` and `port`. Creates a new TCP listening socket and registers it. Clients accepted on the newly created socket will be configured with `client_ops`. The `use_tls` flag indicates whether TLS will be started on top of the clients’ TCP connections.

```c
static void tcp_listen_start(const char *hostname, const char *port, bool use_tls, struct socket_ops *client_ops)
```

B.2.2 UNIX domain listening sockets

**Func unix_accept**

Accept incoming UNIX-domain socket connection and register it.

```c
static void unix_accept(struct socket *s)
```

**Func unix_listen**

Start listening on the socket `s` for incoming UNIX-domain socket connections. `path` is the name of the UNIX-domain socket.

```c
static bool unix_listen(struct socket *s, const char *path)
```

**Func unix_listen_start**

Start listening for incoming TCP connections on a socket called `path`. Creates a new UNIX-domain listening socket and registers it. Clients accepted on the newly created socket will be configured with `client_ops`.

```c
static void unix_listen_start(const char *path, struct socket_ops *client_ops)
```
B.2.3 FDAPc (client) sockets

Macro SHORT_STRING_MAX

A helper to read a short (shorter than SHORT_STRING_MAX) string into a static buffer.

```
#define SHORT_STRING_MAX 32
static char str[SHORT_STRING_MAX];
static char *read_string_helper(struct cbor *c)
{
    size_t len = cbor_read_text_start_len(c);
    assert(len < SHORT_STRING_MAX);
    size_t l = 0;
    while (!cbor_read_text_end(c))
        l += cbor_read_text(c, str + l, SHORT_STRING_MAX - 1 - l);
    assert(l == len);
    str[l] = '\0';
    return str;
}
```

**Func fdapc_decode_request**

Decode incoming request from client. When this function is called, the RX buffer contains at least as many bytes as the request was declared to be long, and a read limit was set on it. Hence no I/O operation will block, but a CBOR decoding error (unexpected EOF) can be encountered due to the request being longer than declared (which is a violation of the protocol, so be it.)

```
static void fdapc_decode_request(struct socket *s)
```

**Func fdapc_in**

Handles incoming data from an FDAP client and implements a simple finite-state machine. The FSM is used to read client’s request in a non-blocking manner.

All CBOR decoder calls are wrapped in a try/catch construct which emulates simple exceptions. (See except.h.) When exceptions are caught, the client is terminated.

```
static void fdapc_in(struct socket *s)
```

**Func fdapc_setup**

Additional configuration of an FDAP client socket. At this moment, this merely sets some flags.

```
static void fdapc_setup(struct socket *s)
```

**Func fdapc_teardown**

Tear-down FDAP client connection.

```
static void fdapc_teardown(struct socket *s)
```

**Func fdapc_timeout**

Handle FDAP client time-out.

```
static void fdapc_timeout(struct socket *s, enum to_type type)
```
B.2.4 FDAPd (downstream) sockets

B.2.5 epoll()-based main loop

Func setup_listening_sockets
Setup listening sockets according to cfg.

static void setup_listening_sockets(void)

Func main
FDAP daemon entry point.

int main(void)

B.3 src/include/array.h

A growing array of arbitrary items implemented using mostly macros. This allows for a nice and type-safe interface while being quite comfortable to work with.

Struct array_hdr
Header of a growing array. Each time array_new is called, the array_hdr struct is allocated just before the actual pointer returned. This structure holds information needed for automatic resizing of the array.

struct array_hdr
{
    size_t num_items; /* number of items actually contained in the array */
    size_t capacity; /* current capacity of the array */
    size_t item_size; /* size of a single item */
};

Macro ARRAY_SIZE
Return the size of the array (the number of items it contains, not the capacity).

#define ARRAY_SIZE(a) (ARRAY_HDR(a)->num_items)

Macro ARRAY_LAST
The last item of the array. This is an lvalue and can be assigned.

#define ARRAY_LAST(a) ((a)[ARRAY_LAST_INDEX(a)])

Macro ARRAY_RESERVE
Insert an item at the end of the array, return a pointer to it. This is basically an allocation of memory from the array.

#define ARRAY_RESERVE(a) (ARRAY_ENSURE(a, ARRAY_SIZE(a)), ARRAY_SIZE(a)++, &ARRAY_LAST(a))

Macro ARRAY_SET
Set a[idx] = val. If idx is outside of the bounds of the array, grow the array as needed.

#define ARRAY_SET(a, idx, val) (ARRAY_ENSURE(a, idx), (a)[idx] = val, 
ARRAY_SIZE(a) = MAX(ARRAY_SIZE(a), idx + 1))
Macro ARRAY_PUSH
Append val at the end of a.

#define ARRAY_PUSH(a, val) (ARRAY_SET(a, ARRAY_SIZE(a), val))

Macro ARRAY_POP
Remove last item of a and return it.

#define ARRAY_POP(a) ((a)[--ARRAY_SIZE(a)])

Macro ARRAY_EMPTY
Is the array empty?

#define ARRAY_EMPTY(a) (ARRAY_SIZE(a) == 0)

Macro ARRAY_DROP
Replace-drop item at index idx in the array, i.e. override a[idx] with the last item of the array and shrink the array by one. This operation changes order of items! Constant time.

#define ARRAY_DROP(a, idx) (a[idx] = ARRAY_LAST(a), ARRAY_SIZE(a)--)  

Macro ARRAY_DROP_RANDOM
Replace-drop a random item. See ARRAY_DROP. This operation changes order of items! Constant time.

#define ARRAY_DROP_RANDOM(a) (ARRAY_DROP(a, random() % ARRAY_SIZE(a)))

Func array_new
Allocate an array of item_size-sized items.

void *array_new(size_t num_items, size_t item_size);

Func array_destroy
Destroy array a which was previously allocated using array_new. The array will be allocated to have space for num_items initially.

void array_destroy(void *a);

B.4 src/include/cbor.h
This module implements a high-performance CBOR encoder and decoder.

Typedef byte_t
Byte type (one octet, or eight bits).

typedef unsigned char byte_t;
Typedef cbor_tag_t

CBOR tag type. The uint32_t range is not enough to hold all CBOR tags (which can be any number from 0 to $2^{64} - 1$). This is a reasonable trade-off between tag range and size of the cbor_item structure.

typedef uint32_t cbor_tag_t;

Macro CBOR_TAG_MAX

Implementation-defined maximum value of a tag.

#define CBOR_TAG_MAX UINT32_MAX

B.4.1 CBOR stream

Functions for manipulation with a CBOR stream handle.

Typedef cbor_errh_t

Prototype of an error-handler. When called, the error handler is required to guarantee that no further operations are called on c, otherwise the behavior is undefined.

This allows you to build a simple exception-like handling mechanism using the longjmp/setjmp combo. See tests/cbor/error.c for an example.

typedef void (cbor_errh_t)(struct cbor *c);

Func cbor_errh_default

Default error handler implementation. Logs the error using the LOG macro and then calls exit with EXIT_FAILURE as the return code, thereby terminating the calling process.

void cbor_errh_default(struct cbor *c);

Func cbor_errh_throw

When called, this error handler will use the throw macro of except.h to perform a longjmp to a location set by a previous setjmp, which is usually set by the try macro of the same header file. This approach allows us to emulate exceptions nicely.

void cbor_errh_throw(struct cbor *c);

Struct cbor

CBOR stream representation. This is the handle passed to almost all functions of this module. It encapsulates the processing context of the encoding/decoding process.

struct cbor
{
    struct iobuf *buf;  /* the underlying I/O buffer */
    cbor_errh_t *errh;  /* error-handling function */
    struct strbuf errmsg;  /* error message buffer */
    struct scope *scopes;  /* array of open scopes */
};

Func cbor_init

Initialize a CBOR stream pointed to by c. All I/O operations will act upon the underlying buf buffer. If an error occurs, errh will be called to handle it (see cbor_errh_t).

void cbor_init(struct cbor *c, struct iobuf *buf, cbor_errh_t *errh);
Func cbor_free
Free all resources held by c.
void cbor_free(struct cbor *c);

Func cbor_strerror
Get a textual description of the last error which occurred. The returned memory is owned by c, don’t pass it to free. The returned string is only valid until any operation other than cbor_strerror is called on c.
char *cbor_strerror(struct cbor *c);

B.4.2 Generic items
Generic item-based encoding and decoding API.

Enum cbor_flag
CBOR item flags.
enum cbor_flag
{
    CBORF_ETEXT = 1,
    CBORF_ETAG = 2,
};

Enum cbor_type
Logical type of a CBOR item. This is not the major type of the item, though there’s a 1–1 correspondence in many cases.
enum cbor_type
{
    CBOR_TYPE_UINT,
    CBOR_TYPE_INT,
    CBOR_TYPE_ARRAY,
    CBOR_TYPE_MAP,
    CBOR_TYPE_TEXT,
    CBOR_TYPE_BYTES,
    CBOR_TYPE_TAG,
    CBOR_TYPE_SVAL,
};

Enum cbor_sval
CBOR simple value. A "value with no content" the meaning of which is given by the RFC. (Major type 7.)
enum cbor_sval
{
    CBOR_SVAL_FALSE = 20,
    CBOR_SVAL_TRUE,
    CBOR_SVAL_NULL,
    CBOR_SVAL_UNDEF,
};
Struct cbor_item

In-memory representation of a generic CBOR item. The meaning of the u16 field is user-defined, this module does not touch it.

```c
struct cbor_item
{
    byte_t type;    /* type of this item */
    byte_t flags;   /* various flags */
    uint16_t u16;   /* user-defined field */
    union
    {
        char etext[12]; /* embedded text */
        struct
        {
            uint32_t u32; /* size information or a tag */
            union
            {
                uint64_t u64; /* CBOR_UINT */
                int64_t i64;  /* CBOR_INT */
                char *text;  /* CBOR_TEXT */
                byte_t *bytes; /* CBOR_BYTES */
                enum cbor_sval sval; /* CBOR_SVAL */
                struct cbor_item *items; /* CBOR_ARRAY */
                struct cbor_pair *pairs; /* CBOR_MAP */
                struct cbor_item *tagged; /* CBOR_TYPE_TAG */
            }
        }
    }
};
```

Struct cbor_pair

A key-value pair.

```c
struct cbor_pair
{
    struct cbor_item key; /* key item */
    struct cbor_item value; /* value item */
};
```

B.4.3 Integers

Encoding and decoding of unsigned and negative integers (major types 0 and 1).

Func cbor_read_u8

Read an 8-bit unsigned integer from c.

```c
uint8_t cbor_read_u8(struct cbor *c);
```

Func cbor_read_u16

Read a 16-bit unsigned integer from c.

```c
uint16_t cbor_read_u16(struct cbor *c);
```

Func cbor_read_u32

Read a 32-bit unsigned integer from c.

```c
uint32_t cbor_read_u32(struct cbor *c);
```
Func cbor_read_u64
Read a 64-bit unsigned integer from c.
uint64_t cbor_read_u64(struct cbor *c);

Func cbor_read_i8
Read an 8-bit signed integer from c.
int8_t cbor_read_i8(struct cbor *c);

Func cbor_read_i16
Read an 16-bit signed integer from c.
int16_t cbor_read_i16(struct cbor *c);

Func cbor_read_i32
Read an 32-bit signed integer from c.
int32_t cbor_read_i32(struct cbor *c);

Func cbor_read_i64
Read an 64-bit signed integer from c.
int64_t cbor_read_i64(struct cbor *c);

Func cbor_write_u8
Write an 8-bit unsigned integer to c.
void cbor_write_u8(struct cbor *c, uint8_t u8);

Func cbor_write_u16
Write a 16-bit unsigned integer to c.
void cbor_write_u16(struct cbor *c, uint16_t u16);

Func cbor_write_u32
Write a 32-bit unsigned integer to c.
void cbor_write_u32(struct cbor *c, uint32_t u32);

Func cbor_write_u64
Write a 64-bit unsigned integer to c.
void cbor_write_u64(struct cbor *c, uint64_t u64);

Func cbor_write_i8
Write an 8-bit signed integer to c.
void cbor_write_i8(struct cbor *c, int8_t i8);
Func cbor_write_i16

Write an 16-bit signed integer to c.

```c
void cbor_write_i16(struct cbor *c, int16_t i16);
```

Func cbor_write_i32

Write an 32-bit signed integer to c.

```c
void cbor_write_i32(struct cbor *c, int32_t i32);
```

Func cbor_write_i64

Write a 64-bit signed integer to c.

```c
void cbor_write_i64(struct cbor *c, int64_t i64);
```

B.4.4 Text streams

Encoding and decoding of text streams (major type 2).

Func cbor_read_text

Read up to nbytes bytes from a logical text-stream into the user-provided buffer dst. Return the number of bytes actually written into dst. If 0 is returned, end of the text stream was reached.

One of the cbor_read_text_start.* functions must be called first to initialize the reading process. cbor_read_text then reads from the logical rather than physical text stream, i.e. it does not matter whether the stream was encoded as a definite- or indefinite-length stream, how many chunks there are, etc.

Please see tests/cbor/text.c for advanced usage examples.

```c
size_t cbor_read_text(struct cbor *c, char *dst, size_t nbytes);
```

Func cbor_read_text_start

Start reading a text stream (definite- or indefinite-length).

```c
void cbor_read_text_start(struct cbor *c);
```

Func cbor_read_text_start_len

Start reading a definite-length text stream. The length of the stream is returned.

```c
uint64_t cbor_read_text_start_len(struct cbor *c);
```

Func cbor_read_text_start_indef

Start reading an indefinite-length text stream.

```c
void cbor_read_text_start_indef(struct cbor *c);
```

Func cbor_read_text_end

Stop reading a text stream. Call to this function will succeed (returning true) only if the whole stream was read; otherwise, it has no side-effects besides wasting time.

```c
bool cbor_read_text_end(struct cbor *c);
```
Func `cbor_read_text_alloc`  
Read a definite- or indefinite-length text stream and save its contents into a newly allocated string `str` in memory. Total length of the string in bytes is returned. The string is NUL-terminated.

When no longer needed, `str` must be disposed by a call to `cbor_text_destroy`. Alternatively, all strings which were not freed by `cbor_text_destroy` will be freed once `cbor_free` is called on `c`.

```c
uint64_t cbor_read_text_alloc(struct cbor *c, char **str);
```

Func `cbor_text_destroy`  
Destroy the string `str` allocated by a call to `cbor_read_text_alloc`.

```c
void cbor_text_destroy(struct cbor *c, char *str);
```

Func `cbor_write_text`  
Write the string `str` to `c` as a definite-length text stream.

```c
void cbor_write_text(struct cbor *c, char *str);
```

Func `cbor_write_text_start_indef`  
Start writing an indefinite-length text stream to `c`. The individual chunks are to be written using `cbor_write_text`.

```c
void cbor_write_text_start_indef(struct cbor *c);
```

Func `cbor_write_text_end`  
Stop writing an indefinite-length text stream.

```c
void cbor_write_text_end(struct cbor *c);
```

### B.4.5 Byte streams

Encoding and decoding of byte streams (major type 3).

Func `cbor_read_bytes`  
Read up to `nbytes` bytes from a logical byte-stream into the user-provided buffer `dst`. Return the number of bytes actually written into `dst`. If 0 is returned, end of the byte stream was reached.

One of the `cbor_read_bytes_start.*` functions must be called first to initialize the reading process. `cbor_read_bytes` then reads from the logical rather than physical byte stream, i.e. it does not matter whether the stream was encoded as a definite- or indefinite-length stream, how many chunks there are, etc.

Please see `tests/cbor/bytes.c` for advanced usage examples.

```c
size_t cbor_read_bytes(struct cbor *c, byte_t *dst, size_t nbytes);
```

Func `cbor_read_bytes_start`  
Start reading a byte stream (definite- or indefinite-length).

```c
void cbor_read_bytes_start(struct cbor *c);
```
Func `cbor_read_bytes_start_len`
Start reading a definite-length byte stream. The length of the stream is returned.

```c
uint64_t cbor_read_bytes_start_len(struct cbor *c);
```

Func `cbor_read_bytes_start_indef`
Start reading an indefinite-length byte stream.

```c
void cbor_read_bytes_start_indef(struct cbor *c);
```

Func `cbor_read_bytes_end`
Stop reading a byte stream. Call to this function will succeed (returning `true`) only if the whole stream was read; otherwise, it has no side-effects besides wasting time.

```c
bool cbor_read_bytes_end(struct cbor *c);
```

Func `cbor阅读_bytes_alloc`
Read a definite- or indefinite-length byte stream and save its contents into a newly allocated buffer `buf` in memory. Total length of the stream in bytes is returned.

When no longer needed, `buf` must be disposed by a call to `cbor_bytes_destroy`. Alternatively, all byte streams which were not freed by `cbor_bytes_destroy` will be freed once `cbor_free` is called on `c`.

```c
uint64_t cbor_read_bytes_alloc(struct cbor *c, byte_t **buf);
```

Func `cbor_bytes_destroy`
Destroy the buffer `buf` allocated by a call to `cbor_read_bytes_alloc`.

```c
void cbor_bytes_destroy(struct cbor *c, byte_t *buf);
```

Func `cbor_write_bytes`
Write `nbytes` bytes of the byte stream contained in buffer `buf` to `c`. Encode it as a definite-length byte stream.

```c
void cbor_write_bytes(struct cbor *c, byte_t *buf, size_t nbytes);
```

Func `cbor_write_bytes_start_indef`
Start writing an indefinite-length byte-stream to `c`. The individual chunks are to be written using `cbor_write_bytes`.

```c
void cbor_write_bytes_start_indef(struct cbor *c);
```

Func `cbor_write_bytes_end`
Stop writing an indefinite-length byte-stream.

```c
void cbor_write_bytes_end(struct cbor *c);
```

### B.4.6 Arrays

Encoding and decoding of arrays (major type 4).
**Func cbor_read_array_start**
Start reading an indefinite- or definite-length array.
```c
void cbor_read_array_start(struct cbor *c);
```

**Func cbor_read_array_start_size**
Start reading a definite-length array, return the number of items it contains.
```c
uint64_t cbor_read_array_start_size(struct cbor *c);
```

**Func cbor_read_array_start_indef**
Start reading an indefinite-length array.
```c
void cbor_read_array_start_indef(struct cbor *c);
```

**Func cbor_read_array_end**
Stop reading an array. This function only succeeds if all items in the array have been read, otherwise it is a no-op.
```c
bool cbor_read_array_end(struct cbor *c);
```

**Func cbor_write_array_start_size**
Start writing an array which will contain `len` items.
```c
void cbor_write_array_start_size(struct cbor *c, uint64_t len);
```

**Func cbor_write_array_start_indef**
Start writing an indefinite-length array.
```c
void cbor_write_array_start_indef(struct cbor *c);
```

**Func cbor_write_array_end**
Stop writing an array.
```c
void cbor_write_array_end(struct cbor *c);
```

**B.4.7 Maps**
Encoding and decoding of maps (major type 5).

**Func cbor_read_map_start**
Start reading an indefinite- or definite-length map.
```c
void cbor_read_map_start(struct cbor *c);
```

**Func cbor_read_map_start_size**
Start reading a definite-length map, return the number of pairs it contains.
```c
uint64_t cbor_read_map_start_size(struct cbor *c);
```
Func cbor_read_map_start_indef
Start reading an indefinite-length map.
void cbor_read_map_start_indef(struct cbor *c);

Func cbor_read_map_end
Stop reading a map. This function only succeeds if all items in the map have been read, otherwise it is a no-op.
bool cbor_read_map_end(struct cbor *c);

Func cbor_write_map_start_size
Start writing a map which will contain size pairs.
void cbor_write_map_start_size(struct cbor *c, uint64_t size);

Func cbor_write_map_start_indef
Start writing an indefinite-length map.
void cbor_write_map_start_indef(struct cbor *c);

Func cbor_write_map_end
Stop writing a map.
void cbor_write_map_end(struct cbor *c);

B.4.8 Tags
Encoding and decoding of semantic tags (major type 6).

Func cbor_read_tag
Read a tag.
cbor_tag_t cbor_read_tag(struct cbor *c);

Func cbor_write_tag
Write a tag.
void cbor_write_tag(struct cbor *c, cbor_tag_t tag);

B.4.9 Simple values
Func cbor_read_sval
Read a simple value from c.
uint8_t cbor_read_sval(struct cbor *c);

Func cbor_write_sval
Write the simple value sval to c.
void cbor_write_sval(struct cbor *c, uint8_t sval);
Func cbor_write_bool

Write the bool b to c. Write either CBOR_SVAL_TRUE or CBOR_SVAL_FALSE.

void cbor_write_bool(struct cbor *c, bool b);

Func cbor_read_bool

Read a bool from c. Assumes the item is either CBOR_SVAL_TRUE or CBOR_SVAL_FALSE, otherwise it’s an error.

bool cbor_read_bool(struct cbor *c);

B.5 src/include/iobuf.h

I/O buffers are useful in situations when you want to read or write large amounts of data without having to worry about the underlying storage mechanism and data buffering.

This generic iobuf module provides a handle and a set of related operations that act upon a buffer. While your code can depend only on this interface, concrete implementations exist (such as iobuf_sock or iobuf_str) which take care of the actual reading and writing of data.

Macro IOBUF_NOLIMIT

If given as argument to iobuf_rlimit, cancel all read limits.

#define IOBUF_NOLIMIT (-1)

Struct iobuf

An I/O buffer.

struct iobuf {
    struct iobuf_ops *ops; /* buffer operations */
    byte_t *buf; /* the I/O buffer */
    byte_t *pos; /* read/write position within the buffer */
    byte_t *bptr; /* back-end position */
    byte_t *bend; /* end of the buffer */
    bool debug; /* enable debugging of reads and writes? */
    struct strbuf dumpbuf; /* string buffer for debugging */
    ssize_t rlimit; /* read limit or 'IOBUF_NOLIMIT' */
};

Struct iobuf_ops

I/O buffer operations.

struct iobuf_ops {
    ssize_t (*fill)(struct iobuf *buf);
    int (*flush)(struct iobuf *buf);
    size_t (*tell)(struct iobuf *buf);
    void (*seek)(struct iobuf *buf, size_t pos);
    void (*destroy)(struct iobuf *buf);
};
Func iobuf_read

Read precisely nbytes bytes from the I/O buffer buf into the user-provided buffer dst. Return the number of bytes actually read. This number will be less than nbytes if and only if EOF was reached. On error, −1 is returned.

ssize_t iobuf_read(struct iobuf *buf, byte_t *dst, size_t nbytes);

Func iobuf_write

Write nbytes bytes from the user-provided buffer src to the I/O buffer buf. Returns 0 on success and −1 otherwise.

int iobuf_write(struct iobuf *buf, byte_t *src, size_t nbytes);

Func iobuf_seek

If the I/O buffer buf supports it, move current read-write position to the absolute position pos within the stream.

void iobuf_seek(struct iobuf *buf, size_t pos);

Func iobuf_tell

An ftell-like function, currently can only return the length of the whole stream (if supported).

size_t iobuf_tell(struct iobuf *buf);

Func iobuf_flush

Flush the buffer. If the buffer contains read data, this merely prepares it for following writes. If the buffer contains written data, this operation forces a write call to the underlying I/O mechanism (or any equivalent).

int iobuf_flush(struct iobuf *buf);

Func iobuf_fill_bg

Fill the buffer in the background. This operation can be used to gradually fill the buffer with a sequence of (short) non-blocking reads (using the underlying I/O mechanism), typically in conjunction with an I/O multiplexer such as epoll. The buffer can be processed later when enough data was accrued.

The operation may trigger a memmove inside the buffer if the unread portion of data does not start at the beginning of the buffer in order to make more space.

Returns the number of newly buffered bytes (can be 0 if the buffer is full, despite that there’s something to read) and −1 is returned on error.

ssize_t iobuf_fill_bg(struct iobuf *buf);

Func iobuf_rlimit

Set read limit limit on the buffer buf, making it impossible to read a total of more than limit bytes in subsequent read operations. To disable the limit, set limit to IOBUF_NOLIMIT.

void iobuf_rlimit(struct iobuf *buf, ssize_t limit);

88
Func iobuf_avail
Returns the number of bytes ready for ready to be read from the buffer without having to call the underlying I/O mechanism and thus, without blocking.

size_t iobuf_avail(struct iobuf *buf);

Func iobuf_getc
Get next character from the buffer. Returns −1 on error.

int iobuf_getc(struct iobuf *buf);

Func iobuf_ungetc
Return last read character back to the buffer. Please note that this operation is only valid immediately after a previous call to iobuf_getc (i.e. there must be no interleaving operations performed on this iobuf).

void iobuf_ungetc(struct iobuf *buf);

Func iobuf_peek
Get next character from the buffer without consuming it. Returns −1 on error.

int iobuf_peek(struct iobuf *buf);

Func iobuf_copy
Copy the contents of buffer src into the buffer dst. The src buffer has to support seek and tell operations.

int iobuf_copy(struct iobuf *dst, struct iobuf *src);

Func iobuf_set_debug
Enable or disable the debug mode. If enabled, all data read from/written to this I/O buffer will be logged.

void iobuf_set_debug(struct iobuf *buf, int debug);

Func iobuf_str_new
Create a new in-memory buffer with initial size init_size bytes. Any data written into this buffer will be stored as a contiguous string in memory. The buffer will grow as needed to accommodate for the writes.

struct iobuf *iobuf_str_new(size_t init_size);

Func iobuf_sock_new
Create a new socket-backed buffer. fd is the file descriptor of the socket and size is the size of the buffer.

struct iobuf *iobuf_sock_new(int fd, size_t size);

Func iobuf_tls_new
Create a new buffer which uses the abstractions of the tls module to read and write encrypted data. tls is the TLS context for the buffer and size is the size of the buffer.

struct iobuf *iobuf_tls_new(struct tls_peer *tls, size_t size);
**Func iobuf_destroy**

Destroy the I/O buffer `buf` obtained by calling one of the constructors above.

```c
void iobuf_destroy(struct iobuf *buf);
```

### B.6 src/include/keystore.h

**Typedef key_id_t**

ID of a key in the keystore.

```c
typedef uint16_t key_id_t;
```

**Macro KEY_ID_MAX**

Maximum value of `key_id_t`.

```c
#define KEY_ID_MAX UINT16_MAX
```

**Struct keystore**

Keystore is a simple data structure which is supposed to provide fast translation from a dynamic set of strings to a (mostly) contiguous range of small integers (called IDs, see `key_id_t`) and vice versa.

This allows for fast string de-duplication (because the integers can be used instead of the strings everywhere) and the IDs can be used to directly index an array of data related to the string (for example to find an index data structure given a column name).

```c
struct keystore
{
    struct list key_to_id;    /* TODO cuckoo */
    struct keystore_entry **id_to_entry; /* ID to keystore entry array */
    key_id_t id_seq;            /* current value of ID sequence */
    key_id_t *free_ids;        /* free IDs (for which refcount fell below 0) */
    struct objpool entry_pool; /* memory pool for entry objects */
};
```

**Struct keystore_entry**

A single entry in the keystore.

```c
struct keystore_entry
{
    struct lnode n;    /* TODO cuckoo */
    key_id_t id;        /* ID of the key in the keystore */
    char *key;          /* the key */
    uint32_t refcnt;    /* reference counter */
};
```

**Func keystore_init**

Initialize the keystore `store`.

```c
void keystore_init(struct keystore *store);
```

**Func keystore_free**

Free the keystore `store`.

```c
void keystore_free(struct keystore *store);
```
Func `keystore_key_to_id`

Use `store` to translate `key` to an ID. If `key` is not present in `store`, it will be assigned a new unused ID.

```c
key_id_t keystore_key_to_id(struct keystore *store, const char *key);
```

### B.7 src/include/log.h

Logging utilities.

**Macro `MBEDTLSDBG_LEVEL_NODEBUG`**

MbedTLS debug log levels.

```c
#define MBEDTLS_DBG_LEVEL_NODEBUG 0
#define MBEDTLS_DBG_LEVEL_ERROR 1
#define MBEDTLS_DBG_LEVEL_STATE_CHANGE 2
#define MBEDTLS_DBG_LEVEL_INFORMATIONAL 3
#define MBEDTLS_DBG_LEVEL_VERBOSE 4
```

**Macro `MBEDTLS_DBG_TARGET_LEVEL`**

MbedTLS target debug level.

```c
#define MBEDTLS_DBG_TARGET_LEVEL MBEDTLS_DBG_LEVEL_NODEBUG
```

**Macro `LOG_TARGET_LEVEL`**

Syslog target logging level.

```c
#define LOG_TARGET_LEVEL LOG_DEBUG
```

**Func `log_helper`**

Set a printf-like `fmt`-formatted message. `flags` is any valid combination of syslog flags, see `syslog(3)`. In debug mode, include `file`, `line` and `func` in the message.

```c
void log_helper(int flags, char *file, size_t line, const char *func, char *fmt, ...);
```

**Func `log_vhelper`**

Set a vprintf-like `fmt`-formatted message with args given in `args`. `flags` is any valid combination of syslog flags, see `syslog(3)`. In debug mode, include `file`, `line` and `func` in the message.

```c
void log_vhelper(int flags, char *file, size_t line, const char *func, char *fmt, va_list va);
```

**Macro `LOG`**

Set a log message `msg`. `flags` is any valid combination of syslog flags, see `syslog(3)`.

```c
#define LOG(flags, msg) \  
  log_helper(flags, __FILE__, __LINE__, __func__, msg)
```

**Macro `LOGF`**

Set a printf-like `fmt`-formatted message. `flags` is any valid combination of syslog flags, see `syslog(3)`.

```c
#define LOGF(flags, fmt, ...) \  
  log_helper(flags, __FILE__, __LINE__, __func__, fmt, __VA_ARGS__)
```
Macro VLOGF

Set a vprintf-like fmt-formatted message with args given in args. flags is any valid combination of syslog flags, see syslog(3).

```c
#define VLOGF(flags, fmt, args) \
    log_vhelper(flags, __FILE__, __LINE__, __func__, fmt, args)
```

B.8 src/include/objpool.h

Simple fixed-size memory allocator.

Struct objpool

Object pool execution context.

```c
struct objpool
{
    struct objpool_block *first_block; /* first block in a list of all blocks */
    struct objpool_unused *first_unused; /* first unused object in a list of unused */
    size_t obj_size; /* size of a single allocation */
    size_t objs_per_block; /* number of objects per block */
    size_t block_size; /* (real) calculated size of a block */
    size_t num_objs; /* number of allocated objects */
    size_t num_blocks; /* number of allocated blocks */
};
```

Struct objpool_block

Represents a memory block in the pool.

```c
struct objpool_block
{
    struct objpool_block *next; /* next block */
};
```

Struct objpool_unused

Represents an unused object within a memory block.

```c
struct objpool_unused
{
    struct objpool_unused *next; /* next unused object */
};
```

Func objpool_init

Initialize objpool to handle obj_size-sized objects, allocating space for objs_per_block of them at once.

```c
void objpool_init(struct objpool *pool, size_t obj_size, size_t objs_per_block);
```

Func objpool_reset

Reset the objpool to its initial state.

    NOTE: No memory will be freed.

```c
void objpool_reset(struct objpool *pool);
```
Func objpool_alloc
Allocate a single object from the pool. The memory returned will be valid until either
objpool_dealloc or objpool_free is called.
void *objpool_alloc(struct objpool *pool);

Func objpool_dealloc
De-allocate the memory mem, returning it to pool to satisfy further allocation requests.
void objpool_dealloc(struct objpool *pool, void *mem);

Func objpool_free
Free all memory held by pool. This renders all memory handles previously returned
from the pool as invalid.
void objpool_free(struct objpool *pool);

B.9 src/include/storage.h
Storage is a device which provides a set of operations for storing and loading of records
and record searching.

Enum storage_result
Result of a storage operation.
enum storage_result
{
    STOR_OK,    /* all good */
    STOR_NXREC, /* record does not exist */
};

Typedef storage_result_t
Type of a storage result operation.
typedef enum storage_result storage_result_t;

Struct storage_ops
Operations of a persistent storage.
struct storage_ops
{
    storage_result_t (*update)(struct storage *stor, struct record *record);
    struct record *(*get)(struct storage *stor, record_id_t id);
    storage_result_t (*remove)(struct storage *stor, record_id_t id);
    struct iter *(*walk)(struct storage *stor);
    struct iter *(*search)(struct storage *stor, struct filter *f);
    void (*destroy)(struct storage *stor);
};

Struct storage
A persistent storage.
struct storage
{
    struct storage_ops *ops;    /* storage operations */
    struct objpool record_pool; /* memory pool for record objects */
    struct keystore attrs_store; /* attributes keystore */
    record_id_t last_id;        /* last used ID */
};
Func storage_init

Initialize the storage stor, configure ops as storage operations.

Do not call this function directly, it is used by the various storage implementations only.

```c
void storage_init(struct storage *stor, struct storage_ops *ops);
```

Func storage_get

Retrieve a record with ID id from storage stor, or NULL if record with this ID does not exist.

```c
struct record *storage_get(struct storage *stor, record_id_t id);
```

Func storage_update

Update the record rec in storage stor.

If the storage does not contain record with an ID same as the updated record’s, the record will be inserted into the storage with a new ID. (This is always the case when record’s ID is set to RECORD_ID_NONE.) Otherwise, a record with matching ID is sought and its attributes will be updated according to the values of rec.

```c
storage_result_t storage_update(struct storage *stor, struct record *rec);
```

Func storage_walk

Return an iterator which iterates over all records in the storage.

```c
struct iter *storage_walk(struct storage *stor);
```

Func storage_remove

Delete record with ID id from storage stor.

```c
storage_result_t storage_remove(struct storage *stor, record_id_t id);
```

Func storage_search

Search storage stor for records matching filter f.

```c
struct iter *storage_search(struct storage *stor, struct filter *f);
```

Func storage_destroy

Destroy the storage object stor.

```c
void storage_destroy(struct storage *stor);
```

B.10  src/include/record.h

Typedef record_id_t

Type capable of holding record IDs.

```c
typedef uint32_t record_id_t;
```
Macro RECORD_ID_NONE
A reserved ID value meaning nothing.
#define RECORD_ID_NONE 0

Typedef version_t
Type capable of holding record version.
typedef uint32_t version_t;

Typedef nattrs_t
Type capable of holding the number of attributes a record has (nattrs) and the number of attributes a record can hold (size).
typedef uint16_t nattrs_t;

Struct record
A valid FDAP document is a definite-length map with definite-length text keys, whose values are arbitrary CBOR items. This structure represents FDAP documents in a fairly compact way (16 bytes for the header and 16 bytes for every key-value pair assuming x64), while being relatively simple and efficient to work with.

The key-value pairs which make up the document’s root map are called attributes. An attribute is an ID (an integer equivalent of the key) and the value, which is a generic CBOR item. The ID of the attribute is stored in the u16 field of the corresponding CBOR item, simplifying memory layout.

In memory, an array attrs of attributes follows every record. The array has capacity for size attributes and holds nattrs of them. It is always sorted by attribute ID, providing Θ(log k) operations, where k is the number of attributes before the operation.

This structure is designed to be used in a copy-on-write manner, i.e. when changes are to be made to an existing record, changes are made to a copy of the record instead (with clean-up and propagation of the changes happening later). The number of attributes of the resulting record is either known in advance or fairly simple to calculate; hence, most of the time, this structure is expected to have nattrs ≈ size, wasting only very little space on the attrs array.

struct record
{
    struct lnode n; /* TODO delete */
    version_t version; /* this record’s version */
    record_id_t id; /* this record’s ID */
    nattrs_t nattrs; /* number of attributes */
    nattrs_t size; /* size of the record */
    uint8_t flags; /* various flags */
    uint8_t reserved[3]; /* reserved, must be zeroed */
    struct cbor_item attrs[1]; /* the attributes */
};

Func record_new
Allocate a new record in storage stor, with size size.
struct record *record_new(struct storage *stor, nattrs_t size);

Func record_destroy
Destroy the record r which was previously allocated in storage stor.
void record_destroy(struct storage *stor, struct record *r);
Func record_has

Does the record r have an attribute with ID id?

bool record_has(struct record *r, key_id_t id);

Func record_get

Get the attribute with ID id of record r.

struct cbor_item *record_get(struct record *r, key_id_t id);

Func record_insert

Insert attribute with ID id to the record r. This operation costs \( \Theta(\log k) \) where \( k \) is the number of attributes in the record before the insertion.

If the record has no space for the new attribute (i.e. its nattrs equals its size), NULL is returned. If the record already contains attribute with ID id, it will be returned instead.

struct cbor_item *record_insert(struct record *r, key_id_t id);

Func record_remove

Delete attribute with ID id from the record r. This operation costs \( \Theta(\log k) \) where \( k \) is the number of attributes in the record before the removal.

This function will return true if r was found to have an attribute with ID id. Otherwise, false will be returned and no attribute will be removed.

bool record_remove(struct record *r, key_id_t id);

Func record_dump

Write a human-readable representation of rec into buf.

void record_dump(struct record *rec, struct strbuf *buf);

B.11 src/include/strbuf.h

Automatically growing string buffer with support for formatted printing.

Struct strbuf

The string buffer. Holds accounting data.

struct strbuf
{
    char *str; /* the buffer itself */
    size_t size; /* current size of the buffer */
    size_t len; /* length of the string */
};

Func strbuf_init

Initialize the string buffer buf. The size of the memory used to store the string will initially be init_size bytes.

void strbuf_init(struct strbuf *buf, size_t init_size);
Func strbuf_free
Free resources held by buf. This includes any strings to which pointers have been obtained by calling strbuf_get_string.

void strbuf_free(struct strbuf *buf);

Func strbuf_reset
Reset buffer buf to its initial state when it holds an empty string.

void strbuf_reset(struct strbuf *buf);

Func strbuf_putchar
Append a single character to the string contained in buf.

void strbuf_putchar(struct strbuf *buf, char c);

Func strbuf_strlen
Return length of the string held in buf.

size_t strbuf_strlen(struct strbuf *buf);

Func strbuf_get_string
Get a pointer to the string held in buf. Don’t free or write this memory, it’s managed by buf.

char *strbuf_get_string(struct strbuf *buf);

Func strbuf_strcpy
Get a copy of the string held in buf. It’s the caller’s responsibility to free this string later.

char *strbuf_strcpy(struct strbuf *buf);

Func strbuf_printf
This function is semantically equivalent to printf(3) with the exception that the resulting string is stored in buf, which is resized as needed to fit the appended formatted string.

size_t strbuf_printf(struct strbuf *buf, char *fmt, ...);

Func strbuf_vprintf_at
This function is equivalent to strbuf_printf, but accepts a va_list of arguments. Instead of appending to the string in buf, print at offset given by offset.

size_t strbuf_vprintf_at(struct strbuf *buf, size_t offset, char *fmt, va_list args);

Func strbuf_prepend
This function is equivalent to strbuf_printf, but prepends the string to the content of buf (not overwriting anything that the buffer contains). This function is slow and should be avoided unless performance is not an issue.

size_t strbuf_prepend(struct strbuf *buf, char *fmt, ...);
B.12 src/include/timeout.h

Typedef expiry_t

Represents expiration time in milliseconds.

typedef uint32_t expiry_t;

Macro EXPIRY_NONE

Expiration value which is interpreted as no expiration.

#define EXPIRY_NONE 0

Macro EXPIRY_MAX

Maximum expiration value.

#define EXPIRY_MAX UINT32_MAX

Enum to_type

Type of a time-out.

enum to_type
{
    TO_IDLE,
    TO_TX,
    TO_RX,
    TO_DOWN,
    TO_TYPE_MAX,
};

Func strto

Get a human-readable name for time-out type type.

const char *strto(enum to_type type);

Struct to

A time-out. The exp_s and exp_ms fields are the seconds and milliseconds part of the expiration time, respectively. This is not the wall-clock time, but rather the time determined by clock_gettime at the time of expiration (using CLOCK_MONOTONIC).

struct to
{
    struct lnode n; /* node in the time-out list */
    struct socket *socket; /* socket which configured the time-out */
    uint32_t exp_s; /* expiration seconds */
    uint16_t exp_ms; /* expiration milliseconds */
    byte_t type; /* type of the time-out, see 'enum to_type' */
};

Struct toset

Set of time-outs. This data structure can manipulate large collections of time-outs configured by many different sockets efficiently, provided that the set of different time-out types is small (and, like here, known at compile-time). All operations it provides are \( O(1) \) unless otherwise noted.

struct toset
{
    struct objpool to_pool; /* pool for ‘to’ structures */
    struct list tos[TO_TYPE_MAX]; /* lists of time-outs grouped by type */
    expiry_t expiry[TO_TYPE_MAX]; /* expiration per time-out type */
};
Func toset_init
Initialize new time-out set set.

void toset_init(struct toset *set);

Func toset_free
Free the time-out set set.

void toset_free(struct toset *set);

Func toset_set_expiry
Configure the time-out type to ms milliseconds. This operation should be called before any time-out of type type is set, otherwise time-outs may expire later than desired.

void toset_set_expiry(struct toset *set, enum to_type type, expiry_t ms);

Func toset_reset
Configure the time-out type type on socket in set. If the time-out is already configured, the behaviour is the same as if the time-out was canceled first with toset_cancel and then configured again.

void toset_reset(struct toset *set, struct socket *socket, enum to_type type);

Func toset_cancel
Cancel the time-out type type configured for socket in set. It’s safe to call this operation even if no such time-out is set.

void toset_cancel(struct toset *set, struct socket *socket, enum to_type type);

Func toset_cancel_all
Cancel all time-outs that socket has configured in set.

void toset_cancel_all(struct toset *set, struct socket *socket);

Func toset_check_expired
Find all expired time-outs, notify the sockets which configured them and cancel them.

void toset_check_expired(struct toset *set);

Func toset_nearest_expiry
Return the number of milliseconds till the nearest configured time-out in set expires.

expiry_t toset_nearest_expiry(struct toset *set);
B.13 src/include/tls.h

TLS module. Provides an abstract API to encapsulate implementation details of a specific TLS library. We use mbedtls as a TLS library.

TLS module consists of three parts: Server, Client and Peer. tls_srv structure with corresponding set of functions are intended for use on server, and tls_cli on client. Both hold TLS-related configuration, certificates, including functions that handles initialization and TLS tunnel establishment. Both server and client have non-trivial functionality in common, so tls_ctx structure represents the common core with corresponding set of functions inside the module. This is just an internal implementation detail that does not affect public API of the module.

tls_peer structure represents a remote TLS end-point that is fully initialized for encrypted communication (one end of a TLS tunnel). On the client, tls_peer represents a server, whereas on the server, tls_peer represents a client – for each accepted TLS client one peer. tls_peer is a product of tls_srv-establish function; tls_cli-establish function, respectively.

Struct tls_ctx

TLS context (common core used by both the server and the client implementation).

```c
struct tls_ctx
{  mbedtls_entropy_context entropy; /* Entropy source used by mbedtls */
   mbedtls_ctr_drbg_context ctr_drbg; /* Random generator used by mbedtls */
   mbedtls_ssl_config conf; /* TLS configuration used by mbedtls */
};
```

Struct tls_srv

TLS server context.

```c
struct tls_srv
{  struct tls_ctx tls; /* Common TLS context used by both client and server */
   mbedtls_x509_crt srvcert; /* Loaded TLS cert. chain
                        (CA cert. <-> CA immediate cert. <-> server cert.) */
   mbedtls_pk_context pkey; /* Loaded private key to the server TLS cert. */
};
```

Struct tls_cli

TLS client context.

```c
struct tls_cli
{  struct tls_ctx tls; /* Common TLS context used by both client and server */
   mbedtls_x509_crt cacert; /* Loaded trusted CA certificates chain */
};
```

Struct tls_peer

TLS context per (accepted) client.

```c
struct tls_peer
{  mbedtls_ssl_context ssl; /* per client context -- product of a TLS handshake */
};
```
Typedef tls_err_t
A return value of several TLS functions. 0 indicates success, failure otherwise.

typedef int tls_err_t;

**Func tls_srv_init**
Initialize server-side TLS context.

void tls_srv_init(struct tls_srv *ctx);

**Func tls_srv_config**
Configure server TLS context with cacert_path, srvcert_path and pk_path to be ready to use.

If the private key file is protected by password, pk_pwd can be supplied to decrypt the file. Can be NULL if the decryption is not desirable.

tls_err_t tls_srv_config(struct tls_srv *ctx, const char *cacert_path,
const char *srvcert_path, const char *pk_path, const char *pk_pwd);

**Func tls_srv_reseed**
Reseed server random generator used by TLS core. It is recommended to call this function after each fork in parent if the server TLS context is inherited by children.

tls_err_t tls_srv_reseed(struct tls_srv *ctx);

**Func tls_srv_establish**
Perform TLS handshake with the remote peer identified by peer_fd and establish a TLS tunnel.

peer_fd file descriptor can be obtained for example by accept function. If the handshake is successful, the remote peer, represented by supplied pctx argument, is ready to use for encrypted communication.

tls_err_t tls_srv_establish(struct tls_srv *ctx, int *peer_fd, struct tls_peer *pctx);

**Func tls_srv_free**
Free resources associated with the TLS context structure.

void tls_srv_free(struct tls_srv *ctx);

**Func tls_cli_init**
Initialize client-side TLS context.

void tls_cli_init(struct tls_cli *ctx);

**Func tls_cli_config**
Configure client TLS context with CA cert. path to be ready to use.

tls_err_t tls_cli_config(struct tls_cli *ctx, const char *cacert_path);
**Func tls_cli_establish**

Perform TLS handshake with the remote peer identified by peer_fd and establish a TLS tunnel.

peer_fd file descriptor can be obtained for example by connect function. If the handshake is successful, the remote peer, represented by supplied pctx argument, is ready to use for encrypted communication.

If verify_srvcert is set to true, a server certificate, that is presented by a server during the handshake, will be verified against trusted CA certificates, configured by previous call to tls_cli_config function. The authority that signed the certificate must have a certificate in a trusted CA root file in order to be TLS handshake successful.

If verify_hostname is set to true, a server common name (CN) field in a server certificate will be verified against hostname supplied in the next argument. It is the recommended option, to turn this on, and supply valid hostname of the server. However, it can be sometimes useful to turn this off; for example when connecting to an IP address of the server, instead of a hostname.

```c
tls_err_t tls_cli_establish(struct tls_cli *ctx, int *peer_fd, struct tls_peer *pctx,
    bool verify_srvcert, bool verify_hostname, char *hostname);
```

**Func tls_cli_free**

Free resources associated with the TLS context structure.

```c
void tls_cli_free(struct tls_cli *ctx);
```

**Func tls_peer_read**

Read at most len data bytes from TLS peer ctx and decrypt them to buffer buf. The real number of read bytes is returned.

```c
int tls_peer_read(struct tls_peer *ctx, unsigned char *buf, size_t len);
```

**Func tls_peer_write**

Try to write exactly len data bytes from buffer buf to TLS peer ctx and encrypt them. The real number of written bytes is returned.

```c
int tls_peer_write(struct tls_peer *ctx, const unsigned char *buf, size_t len);
```

**Func tls_peer_free**

Notify the remote peer that the TLS tunnel is going to be closed. Note, that this does not close the underlying transport connection (e.g. TCP).

```c
void tls_peer_free(struct tls_peer *ctx);
```

### B.14 src/include/trie.h

**Struct trie**

Compressed trie.

```c
struct trie
{
    struct tnode *fake_root;    /* fake root node to simplify code */
};
```
Func trie_init
Init the trie t.
void trie_init(struct trie *t);

Func trie_find
Find node with key key and return it. If the key is not present in t, return NULL.
struct tnode *trie_find(struct trie *t, char *key);

Func trie_contains
Does the trie t contain the string key?
bool trie_contains(struct trie *t, char *key);

Func trie_insert
Insert the key key into the trie t.
struct tnode *trie_insert(struct trie *t, char *key);

Func trie_remove
Remove the key key from the trie t.
struct tnode *trie_remove(struct trie *t, char *key);

Func trie_free
Free the trie t.
void trie_free(struct trie *t);

B.15 src/timeout.c

Func ns_to_ms
Convert nanoseconds to milliseconds.
static uint16_t ns_to_ms(uint64_t ns)

Func ms_to_expire
Return the number of milliseconds for the time-out to expire since s. If to expired in the past, 0 is returned.
static expiry_t ms_to_expire(struct *to, struct timespec *s)

Func expired
Is the time-out to expired at the time at?
static bool expired(struct to *to, struct timespec *at)

Func cancel
Cancel the time-out to and free the to structure.
static void cancel(struct toset *set, struct to *to)
Func nearest_to

Return the nearest time-out of type type in set.

static struct to *nearest_to(struct toset *set, enum to_type type)

B.16 src/trie.c

A compressed trie data structure implementation aiming at high performance (i.e., fast insertion and searching) while maintaining reasonably low memory consumption.

Macro LABEL_SIZE

Size of the label field in the tnode structure. This must be enough to hold a char *. If the label is shorter than this value, it will be embedded in the field directly, thus saving the overhead of having to create a heap copy of the string.

#define LABEL_SIZE 13

Struct tnode

A trie node, or more precisely its header. In memory, two arrays (possibly of size 0) always follow: an array of child pointers and a sorted array of characters. The latter array is used to find child index for given input character.

The node itself is capable of storing size children at any given moment. It is resized to fulfill storage requirements as they change. This ensures that the trie is reasonably compact at all times.

struct tnode
{
    char label[LABEL_SIZE]; /* short-enough label or a pointer to a label */
    byte_t flags; /* various flags */
    byte_t size; /* capacity of the 'child' array */
    byte_t nchild; /* number of children */
    struct tnode *child[1]; /* child pointers start here */
};

Func get_label

Return pointer to the label of node n. This is either pointer to the label content if the label was embedded in the tnode directly, or the pointer stored in label if the label was allocated externally.

static char *get_label(struct tnode *n)

Func set_label

Set label of node n to label label. If the label is short enough, label will be copied into the label field of the node. If it’s longer, create a copy of the string given using strdup.

static void set_label(struct tnode *n, char *label)

Func alloc_size

Return the number of bytes needed to allocate a node with size size.

static size_t alloc_size(size_t size)
Func char_array

Return a pointer to the character array of the node \( n \). The character array is allocated just after the array `child` of child pointers.

```c
static char *char_array(struct tnode *n)
```

Func resize

Resize the node \( n \) to size `new_size`.

```c
static struct tnode *resize(struct tnode *n, size_t new_size)
```

Func new_node

Allocate a new node, making sure that it’s initial size will be at least `min_size`, i.e. that `min_size` children can subsequently be inserted without the need to reallocate the node.

```c
static struct tnode *new_node(size_t min_size)
```
Bibliography


List of Figures

1.1 Directory server operating inside an infrastructure ............................. 7
1.2 X.500 Standards: Number of pages summary ...................................... 11
1.3 List of RFCs describing LDAP, taken from RFC 4510 ............................ 12
1.4 LDAP and X.500 standards relationship diagram .................................. 12
1.5 LDAP basic schema definition (RFC 4519 [Sci06]) ............................. 13
1.6 LDAP textual Distinguished Name (DN) .............................................. 14

2.1 Data serialized in CBOR ................................................................. 18
2.2 CBOR diagnostic notation .............................................................. 18
2.3 Nested attribute dot-chained name ................................................... 20
2.4 An authenticable entry ................................................................. 22
2.5 Filter textual notation formula (partially expanded) ............................ 24
2.6 Filter AST .................................................................................. 25
2.7 Encoding of filter AST nodes .......................................................... 25
2.8 Encoding of filter conditional operators .......................................... 26
2.9 Filter textual notation formula (fully expanded) .................................. 27
2.10 CBOR-encoded filter data ............................................................ 27
2.11 FDAP transport – layer overview .................................................... 28
2.12 Authenticate request ................................................................. 30
2.13 Search request ................................................................. 30
2.14 Get request ................................................................. 30
2.15 Create request ................................................................. 30
2.16 Update request ................................................................. 30
2.17 Delete request ................................................................. 31
2.18 Generic response ................................................................. 31
2.19 FDAP status code table ............................................................. 32
2.20 Authorization schema flow diagram ............................................... 34

3.1 Cooperation of the client-side programs .......................................... 38
3.2 Example FDAP infrastructure setup .............................................. 38
3.3 Directory source tree (selected parts) ............................................ 40
3.4 CBOR stream initialization .......................................................... 41
3.5 FDAP Daemon Configuration Example .......................................... 46
3.6 FDAP Client Configuration Example ............................................ 48
3.7 Utility command-line options ....................................................... 50
3.8 Utility supported interactive commands .......................................... 51
3.9 Utility example commands ........................................................... 51
3.10 Putting an instruction file to the utility’s stdin .................................. 51

4.1 NSS client API – direct queries naming scheme example .................... 53
4.2 System structures, representing NSS database entries [POS18] ............... 54
4.3 NSS client API – enumerative queries naming scheme example ............ 55
4.4 NSS client API – reentrant function example ................................... 55
4.5 NSS client API – summary of API functions for selected DBs .............. 55
4.6 NSS configuration file example (/etc/nsswitch.conf) ......................... 56
4.7 NSS example mapping between client API and module API ................. 56