VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

SYNCHRONIZATION OF DNS RECORDS BETWEEN LDAP DATABASE AND DNS SERVER

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. MARTIN BAŠTI

BRNO 2015

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

SYNCHRONIZACE DNS ZÁZNAMŮ MEZI LDAP DATABÁZÍ A DNS SERVEREM

SYNCHRONIZATION OF DNS RECORDS BETWEEN LDAP DATABASE AND DNS SERVER

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE Bc. MARTIN BAŠTI
AUTHOR

VEDOUCÍ PRÁCE Ing. RUDOLF ČEJKA
SUPERVISOR

BRNO 2015

Abstrakt

Cílem této práce je prozkoumat možnosti uložení DNS dat v LDAP databázi a jejich oboustranné synchronizace s DNS serverem s využitím standardních technologií, paralelizace a jednoduché rozšiřitelnosti o podporu různých DNS a LDAP serverů s různými DNS schématy. Práce dále popisuje různa schémata pro uložení dat v LDAP databázi a analyzuje problémy spojené se synchronizací a navrhuje jejich řešení. Výsledkem je prototyp synchronizační aplikace s doporučenými nastaveními pro DNS server a LDAP databázi, která je schopna efektivně synchronizovat DNS data.

Abstract

The aim of this thesis is to discover possibilities in storing the DNS data in the LDAP Database and keep them synchronized in two-way manner, using standard technologies, paralelization and possibility of extensions to support various types of LDAP and LDAP servers with various DNS schemas. This thesis also describes various schemas for storing DNS data in an LDAP database and analyzes issues related to synchronization and provides solutions how to solve them. The result of this thesis is a proof-of-concept code of synchronization application with recommended settings for the DNS server and the LDAP database, which will be able effectively to synchronize the DNS data.

Klíčová slova

synchronizace, DNS, doménový server , BIND, LDAP, adresářový server

Keywords

synchronization, DNS, nameserver, BIND, LDAP, directory server

Citace

Martin Bašti: Synchronization of DNS Records between LDAP Database and DNS Server, diplomová práce, Brno, FIT VUT v Brně, 2015

Synchronization of DNS Records between LDAP Database and DNS Server

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Rudolfa Čejky.

....
Martin Bašti
May 26, 2015

Poděkování

Děkuji svému externímu konzultantovi Ing. Petru Špačkovi za poskytnutí odborné pomoci.

Contents

List of Figures

Chapter 1

Introduction

The current trend in information technologies is to provide solutions that consist of several sub systems integrated into one product. These solutions may for example consist of authentication, authorization and accounting services, database, user interface, and host management including DNS services. Integration reduces administration overhead allowing to manage data centrally, specifying privileges and identification for user and services. Example of the integrated product mentioned above is the open source project FreeIPA.

An authoritative DNS server, specifically BIND 9 server, uses mainly zone files as source of DNS data, which are stored locally on the server's file system. As the integration requires some central database for all data, in case of FreeIPA it is an LDAP database, so updating DNS data via zone files is not an option. For this reason a bidirectional synchronization between an LDAP database and a DNS server must be established.

The aim of this thesis is to analyze various schemas and provide solutions for various synchronization challenges. The result is a proof of concept implementation of a synchronization daemon, named `dnssyncd`, which synchronize DNS records between an LDAP database and a DNS server. For initial implementation were chosen the BIND 9 DNS server and 389 Directory Server as the LDAP database.

The synchronization daemon should meet several goals.

The first goal is that the synchronization daemon must support plugins which will allow to extend synchronization for various types LDAP schemas, LDAP and DNS servers. To meet this goal a plugin interface must be created. The interface must require as minimal as possible methods to be implemented in a plugin. On other hand interface may provide optional methods which may improve the synchronization.

The second goal is that as much as possible standardized operations/technologies must be used to ensure the plugins can share some parts of implementation and creation new plugins will be easier. These standardized technologies will minimize differences between particular implementations of plugins.

Chapter 2 Technology Overview provides basic information about DNS and LDAP technologies, and describes standardized methods and configurations which can be used for synchronization daemon implementation.

Various schemas for storing DNS data in LDAP database and advantages of storing data in LDAP database are discussed in chapter 3 Using an LDAP Server as Data Storage for the DNS.

Synchronization challenges which can affect the synchronization daemon and possible solutions are described in chapter 4 Synchronization Issues.

Chapter 5 Implementation contains implementation details of the synchronization daemon, describes synchronization algorithm, the daemon design and plugin interface. This chapter also evaluate differences between the synchronization daemon and `bind-dyndb-ldap` plugin for BIND 9 server that allows read DNS data from LDAP database, but using different approach.

Chapter 2

Technology Overview

This chapter describes DNS, LDAP and related utilities. The description is focused on explaining the standardized technologies suitable to use in the synchronization of DNS resource records with LDAP (directory) server. The methods used to dynamically modify the records and methods used to get resource records from the DNS server are the focus of this chapter. This chapter also covers how the BIND 9 server works, the description how stores resource records, and ways of managing zones.

The end of this chapter covers basic information about LDAP and directory server.

2.1 Domain Name System (DNS)

This section contains information from RFC 1034 [23] and RFC 1035 [24]. Both RFCs describe the domain name system. Relevant updates of these RFCs will be mentioned in the text.

2.1.1 DNS Architecture

The DNS consists of three major components:

- domain name space and resource records in the namespace,
- name servers,
- resolvers.

Resolvers are programs that get information from name servers. A resolver is typically a system routine that is directly accessible without using a protocol. Resolvers are beyond the scope of this thesis and will not be explained further.

Domain Name Space and Resource Records

A domain name space is a tree structure (see figure 2.1 Example of the DNS tree), where each node or leaf contains corresponding resource sets (which may be empty). Every node or leaf has a label. Labels can be 0 to 63 octets length, and the same label can be used for more nodes, except brothers. The null label (zero length label) is reserved for the root of the domain name space tree.

A domain name consists of sequence of labels usually separated by dots (”.”). The two types of domain names are: relative names and fully qualified domain names.

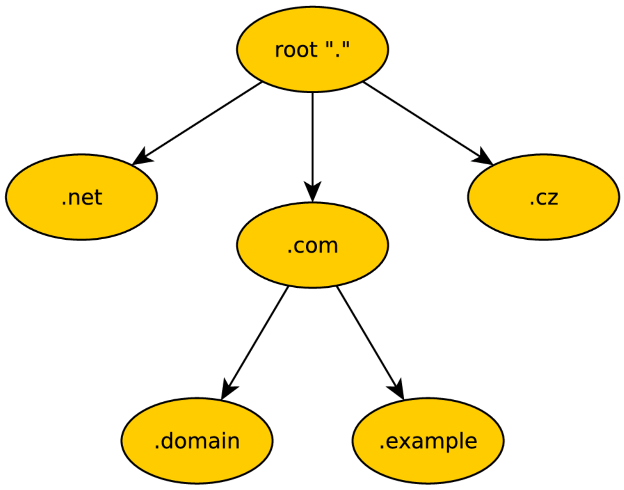

Figure 2.1: Example of the DNS tree

Fully qualified domain names (FQDN), or absolute domain names, are label sequences ending with the root label. These strings in printed form end with a dot. For example, FQDN domain: ”example.com.”.

A character string that represents the starting labels of a domain name is a relative domain name. A relative domain name can not end with the root label (the dot). Relative domain names can be completed to FQDNs by the local system knowledge. For example, relative domain: ”example.com” or only ”example”.

Relative domain names are taken relative to a well known origin. Relative names are mostly used in user interfaces and master zone files, where they are relative to a single origin domain name. For example, two domain names, an FQDN origin name ”com.” and a relative name ”for.example”, can construct the FQDN name ”for.example.com.”.

The maximum length of a domain name is limited to 255 octets.

As RFC 1034 [23, section 3.1] specifies, names in the domain name system are case-insensitive and consist of ASCII value. As RFC 4343 [11] clarifies, DNS names were expected to consist only of valid ACSII characters, however, the individual octets are not limited to valid ASCII character codes. Unprintable ASCII characters have to be mapped in textual representation to the escape sequence with backslash, followed by three digits that represent unsigned decimal value of the ASCII character.

The internationalized domain names, latest specification in RFC5891 [19], are encoded into ASCII form using the punycode algorithm.

As mentioned above, a domain name identifies node, and every node has a set of resource information. The set of resource information consists of particular resource records (RR). Resource records will be mentioned later in this section.

Name Servers

Name servers are programs which hold information about the domain name space. The servers can cache information about any part of the domain tree, but in general, a particular server has complete information about subset of the domain space and pointers to other domain servers. The server is an authority for these subsets (parts), which have complete information. The authoritative information is organized into zones, which are distributed among the servers.

A zone can be available from several name servers. Typically, a given name server supports one or more zones, which give authoritative information about a small section of the domain trees only. The server can have an amount of cached (non-authoritative) data about the other parts of tree.

A primary (or master) server contains authoritative DNS resource records for zone. Secondary (or slave) servers the download zone resource records from the primary server using the zone transfer. Secondary servers are used to distribute load. The synchronization daemon acts as a secondary DNS server from the point of view of the DNS primary server. The zone transfer will be covered later.

Every zone change must be made at the primary server. The changes are then distributed to the secondary servers.

2.1.2 Resource Records (RRs) and Zones

Resource records consists of:

- owner (name),
- type,
- class,
- TTL,
- RDATA.

The owner is the domain name where the RR is found.

The type is an encoded 16 bit value that specifies the type of resource in the RR. Every type has a particular purpose. Some of them are shown in table 2.1. The most important type of record for this thesis and synchronization mechanism is the SOA (Start of a zone of authority) RR. The SOA resource records will be explained later. The other RR types are not important for this thesis and will not be explained any further.

`A`	an IPv4 host address
`AAAA`	an IPv6 host address
`CNAME`	identifies the canonical name of an alias
`PTR`	a pointer to another part of the domain name space
`SOA`	identifies the start of a zone of authority
`NS`	the authoritative name server for the domain

Table 2.1: Examples of resource record types

The class is an encoded 16 bit value which identifies a protocol family or an instance of a protocol. Only the IN class, the Internet system, will be used in this text and in the synchronization program. Other classes are historical and not used.

The TTL value, the time to live value, is a 32 bit integer that represents in seconds for how long a RR can be cached before a resolver should discard it.

The RDATA are type and class dependent data, which describe the resource.

The following textual representation of the resource records will be used in the text and examples later. The RR in the textual representation consists of an owner name (domain name), TTL, class, type and RDATA values separated by space. This is a common way of representing DNS records. TTL and class can be omitted. If the domain name is empty, the previous one will be used.

```
example.com.      3600  IN  NS    ns.example.com.
ns.example.com.   3600  IN  A     192.168.1.0
                  3600  IN  AAAA  2001:db8:0::1
host.example.com. 3600  IN  A     172.16.0.1
                  3600  IN  A     10.0.0.1
```

If the class value is omitted in the examples later, it will be considered that the class is IN, the Internet system.

The resource records may be grouped into a resource record set (RRSet). As RFC 2181 [33, section 5] describes, the RRSet is a group of resource records with the same label (owner), class and type, but with different data.

Standard zone files use the textual representation; the following example shows a part of a zone file used by a master server. Only authoritative information is written in a zone file.

```
$TTL 2d ; zone TTL default = 2 days or 172800 seconds
$ORIGIN example.com.
@       IN   SOA     ns.example.com. root.example.com. (
                       2014010500 ; se = serial number
                       3h         ; ref = refresh
                       15m        ; ret = update retry
                       3w         ; ex = expiry
                       3h         ; min = minimum
                     )
        IN   NS      ns.example.com.
        IN   MX      10  mail.example.com.
server  IN   A       192.168.1.1
www     IN   CNAME   server
```

The zone file above starts with a SOA record, which stores an additional information about the zone. This type of RR will be explained in detail later.

SOA Resource Records

This type of RR controls the detection of changes of the secondary servers. The detection is based on polling (or by the DNS Notify mechanism, see section 2.1.4 DNS Notify).

The RDATA format of a SOA RR specified by RFC 1035 [24]:

- MNAME, the domain name of the name server that is primary for this zone.
- RNAME, a domain name, which specifies the e-mail address of the person responsible for this zone.

- SERIAL, the unsigned 32-bit value. It is the version of the original copy of the zone.
- REFRESH, a 32-bit time interval in seconds before the zone should be refreshed.
- RETRY, a 32-bit time interval in seconds that should elapse before a failed refresh should be retried.
- EXPIRE, a 32-bit time value in seconds that specifies the upper limit on the time interval that can elapse before a zone is no longer authoritative.
- MINIMUM, the unsigned 32 bit minimum TTL that is usead mainly with negative caching as is specified in RFC2308 [9, section 4], which is out of scope of this work.

To detect changes, secondary servers check the SERIAL field of the SOA RR for the zone. If any changes are made, the SERIAL field in the SOA RR of the zone is always advanced.

The periodic polling of the secondary servers is controlled by REFRESH, RETRY, and EXPIRE parameters in the SOA RR for the zone. Whenever a new zone is loaded in a secondary server, the secondary server waits REFRESH seconds before checking with the primary server for a new serial. If this check cannot be completed, new checks are started every RETRY seconds. If the secondary server finds it impossible to perform a serial check for the EXPIRE interval, it must assume that its copy of the zone is obsolete and discard it. The check is a simple query to the primary server for the SOA RR of the zone.

RFC 1982 [32] specifies a way how to check if a new SERIAL number is newer than the actual value. The definition of the result of a comparison of any two SERIAL numbers s_1 and s_2 is as follows:

s_1 is said to be less than s_2 if, and only if, s_1 is not equal to s_2, and

$$\left(s_1 < s_2 \textit{ and } s_2 - s_1 < 2^{(32-1)}\right) \textit{ or } \left(s_1 > s_2 \textit{ and } s_1 - s_2 > 2^{(32-1)}\right)$$

s_1 is said to be greater than s_2 if, and only if, s_1 is not equal to s_2, and

$$\left(s_1 < s_2 \textit{ and } s_2 - s_1 > 2^{(32-1)}\right) \textit{ or } \left(s_1 > s_2 \textit{ and } s_1 - s_2 < 2^{(32-1)}\right)$$

The synchronization daemon must carefully detect any changes in the SOA SERIAL of the particular zone to ensure up to date synchronization.

2.1.3 Zone Transfer

As mentioned above, the zone transfer is a method to get all zone data from the server. Zone transfer is used by secondary servers to get data from primary servers. Two methods of the zone transfer exist: authoritative or full zone transfer (AXFR) and incremental zone transfer (IXFR). A secondary server using zone transfer is called an AXFR or IXFR client.

AXFR

The full zone transfer (AXFR) is explained in detail in RFC 5936 [20].

The objective of the AXFR is to transfer the contents of a zone, in order to permit the AXFR client to reconstruct the zone, as it exists at the primary server for the given zone serial number.

An AXFR client must ensure that only a successfully transferred copy of the zone can be used. If any error occurs (or a DNS notify request is received), the AXFR client must continue to serve the previous version of the zone. This is called a two-stage model, and any implementation of AXFR client must be equivalent to this model.

An AXFR response from a server that is transferring the zone's content consists of a series of DNS messages. The first message in the series must be the SOA resource record, and the last message must conclude with the same SOA RR. The zone content is represented as an unordered collection (set) of resource records.

IXFR

The incremental zone transfer (IXFR) is covered in RFC 1995 [25].

The full zone transfer mechanism is not efficient in propagating minor changes of the zone, as it transfers the entire zone. The incremental zone transfer is more efficient.

An IXFR client sends an IXFR message containing the SOA serial number of its copy of the zone. An IXFR server replies with a message containing only differences required to make that version of zone current. Alternatively, the IXFR server can choose to transfer zone as AXFR (for example, when IXFR is not available).

If the incremental zone transfer is available, one or more difference sequences are returned. The list of difference sequences is preceded and followed by a copy of the server's current version of the SOA. Each difference sequence represents one update to the zone (one SOA serial change) consisting of deleted RRs and added RRs. The first RR of the deleted RRs is the older SOA RR and the first RR of the added RRs is the newer SOA RR. A modification of an RR is performed first by removing the original RR and then adding the modified one. The sequences of differential information are ordered oldest first newest last.

2.1.4 DNS Notify

A slow propagation of DNS data in a zone can be caused by the zone's relatively long REFRESH interval. Using a DNS Notify message, a primary server advises to secondary servers that zone was changed, as explained in RFC 1996 [38].

When the primary server has updated one or more records in the zone, it sends the DNS Notify message to the secondary servers. The notify message is sent only when a SOA serial changes.

When the secondary server receives the notify message, it queries for the zone's SOA record, and if the SOA SERIAL is newer, the server will start the AXFR or IXFR transfer. Due to the DNS architecture more DNS notify messages can be received. For example, the server sends several copies of the DNS notify message via UDP, until the message is considered as delivered through unreliable network.

2.1.5 Dynamic Updates (DDNS)

Originally the DNS was designed to support queries for a statically configured database, where updates were made by manual editing of masterfiles by administrator. The RFC 2136 [37] brings the method to dynamically update the DNS resource records.

A dynamic update is a DNS message, which the client sends to the primary server to add or delete a resource record in a particular zone. The update message must contain a SOA record of a particular zone, where the updated record belongs, and a set of records to change/add/delete. The update operation is atomic.

The server must check the requestor's permissions. For example, in BIND 9 the option `allow-update` has to be configured in configuration file per zone. The option specifies the list of network addresses or keys which are allowed to perform the DDNS. Another, more granular, way in BIND 9 is a configuration of the `update-policy` settings.

If a zone is updated and the SOA record is not changed in the dynamic update, then the primary DNS server shall increment the SOA serial value to ensure that the changes will be distributed properly.

The `nsupdate` utility [6] submits the DDNS requests to a name server. The utility input is a textual representation of RRs, which is transformed to DDNS message. The example of adding a new A record dynamically using the `nsupdate` utility is shown below.

```
nsupdate
> update add host.example.com A 192.168.1.2
> send
```

This utility can be used with the synchronization program to perform DDNS updates of the DNS master server.

2.1.6 BIND 9 DNS server

The Berkeley Internet Name Domain (BIND) is a widely used free name server software. The BIND was originally written at University of California at Berkeley. It is maintained by the Internet System Consortium (ISC) [16] now. Versions older than BIND 9 are officially deprecated.

Overview

The BIND 9 is running only as a single process, `named`, which is configured by the `named.conf` configuration file. Please remember that different distributions of operating system may have different predefined values in the `named.conf` file.

This version is based strongly on masterfiles, and the configuration file. The configuration file contains zone definitions, paths to master files to the particular zones, and configuration attributes of the zones.

If you change any information in the zone files, the BIND 9 must be reloaded.

RNDC utility

Fortunately, the remote name daemon control (`rndc`) utility allows to dynamically reload zones without the need to restart the whole BIND 9.

The `rndc` utility also allows to dynamically add and remove new zones. This feature can be used to create a new zone and synchronize its records from the LDAP database, or remove the zone when it was removed from LDAP. The example below shows adding and removing the zone using the `rdnc`.

```
 rndc addzone example.com in~'{type master; file "master/example.com";};'
 rndc delzone example.com
```

To allow `rndc` to add new zones, the `allow-new-zones yes;` statement is required in options part of the `named.conf`.

```
options {
    allow-new-zones yes;
};
```

Allowing zone transfer and DDNS

The BIND 9 allows to specify the set of the name servers (addresses optionally with port, TSIG keys) that are allowed to execute an AXFR or IXFR transfer. The transfer has to be allowed globally or particularly per zone in the configuration file `named.conf`.

To allow the IXFR zone transfer, the global option `provide-ixfr` has to be set to `yes` (by default in BIND 9).

Using the same format of addresses as for the zone transfer, BIND 9 allows to specify the sources of dynamic updates, which are allowed to modify records.

The following example shows how to allow the IXFR zone transfer requested by the localhost on the particular port and receiving updates from the localhost.

```
options{
    provide-ixfr yes;
};

zone "example.com." IN {
    type master;
    file "data/example.com";
    allow-transfer {127.0.0.1 port 62123; };
    allow-update {127.0.0.1; };
};
```

2.2 Lightweight Directory Access Protocol (LDAP)

This section explains the LDAP and describes methods and features that are important for understanding. The information in this section will be useful in the next chapters where LDAP methods will be used in synchronization context.

As RFC 4511 [35] specifies, an LDAP is a protocol, which defines the method of how directory data is accessed. Also, the standard RFC 4512 [39] defines and describes how data is represented.

Both RFC 4512 [39] and ZyTrax LDAP book [5] are two of the main sources of information for this section.

The LDAP is designed to support bigger amount of reading than writing data, which means that the directory servers are read-optimized and the information should not be changed often. The question is, why is the LDAP directory server used as a storage for DNS resource records when DNS data can change using dynamic updates? The answer is, because a major part of the DNS system does not change the resource records too often and DNS could be only a subsystem in a bigger system which uses an LDAP server as a data storage." For example the DNS is a subsystem of Microsoft Active Directory [22] and FreeIPA [2], which both use an LDAP directory server as a storage.

Some specific configurations or technical issues in this thesis are related to the 389 Directory Server. Other implementations of the LDAP server may behave differently.

2.2.1 Architecture

As ZyTrax [5, chapter 2] and Oracle [26, chapter 9] manual pages describe, the LDAP defines four models:

- An information model, which defines what kind of information is represented in an LDAP system. This model is focused on data entries. An entry consist of attributes that contain data. Each attribute has a type and a value (or more values). The type of an attribute has an associated syntax that defines what kind of information can be stored in the value of the attribute, and how the value behave during directory operations.

  Attributes are members of object classes. An attribute allowed in the object class can be optional (keyword MAY) or mandratory (keyword MUST) and object classes may inherit characteristics of parent object classes. LDAP directory entries are usually represented in a hierarchical tree structure called the directory information tree (DIT).

- A naming model specifies how entries are named, organized and referenced in an LDAP directory server. Entries are named according to their position in the hierarchy by a distinguished name (DN). The distinguished name consists of components called relative distinguished names (RDNs). The first RDN of a DN is the least significant and the last RDN is the most significant.

  The distinguish name `DN: cn=John Smith, ou=people, dc=example, dc=com` consists of RDN names `RDN: cn=John Smith`, `RDN: ou=people` and `RDN: dc=example, dc=com` (the last one can be splitted into two RDNs).

- A functional model defines what you can do with the information in the LDAP directory and how it can be updated and accessed. This model also covers a search operation, size or time limits of the search operation and search scopes. Search scopes will be explained further in section 2.2.6 Search Scopes.

- A security model, defines how you can secure information in an LDAP directory.

The LDAP architecture supports creating replicas. One or more copies of the directory can be slaved from a master (or masters) to replicas. There is a difference between transactional databases and the LDAP, where in the LDAP context, a temporary lack of synchronization is unimportant and synchronization may take some time. This must be considered in the application design of synchronization between DNS and LDAP. The application has to be aware of this temporary inconsistence, which may occurs between multiple master servers.

Two possible replication configurations exist:

- In the master-slave deployment, mentioned before, the DIT is copied to slave servers. A slave server operates with read-only mode and all changes must be made on the master server.

- In the multi-master deployment, more servers running master DITs may be updated and the updates are propagated to peer masters.

2.2.2 LDAP Data Interchange Format (LDIF)

The LDIF is a file format typically used to import and export LDAP data. The LDIF is specified in RFC 2849 [15].

LDIF files can be used in following cases: to initially construct the DIT structure, to add records into a directory, to import a directory, to export a directory, and to apply bulk changes to a directory.

The most important case, which will be used often in this thesis, is making the DIT structure.

An example of an LDIF file is shown below. There must be at least one blank line between entries. This LDIF file adds person entry named "John Smith" into database, and includes parent entries.

```
# Example of~LDIF #

dn: dc=example,dc=com
dc: example
objectclass: top
objectclass: dcObject

dn: ou=people, dc=example,dc=com
ou: people
description: All people
objectclass: top
objectclass: organizationalunit

dn: cn=John Smith,ou=people,dc=example,dc=com
objectclass: top
objectclass: person
cn: John Smith
cn: J Smith
sn: smith
uid: jsmith
```

2.2.3 Schemas, Object Classes and Attributes

The RFC 4512 [39, section 4] describes schemas as packages that group together related object classes and attributes. The attributes defined in one schema can be used in an object class defined in another schema.

Object classes are sets of attribute defined inside schemas. Object classes may be organized in a hierarchy and inherit all the properties of parents (superior classes) and this hierarchy is terminated with the abstract `top` object class. There are three types of object classes: STRUCTURAL in which a case can be used to create an entry and specify the DIT structure, AUXILIARY are used to augment the characteristics of entries, and ABSTRACT class, which provides a base of characteristics from which other object classes can inherit. An AUXILIARY class cannot inherit from a STRUCTURAL class, and an ABSTRACT class cannot inherit from either STRUCTURAL or ABSTRACT classes. MAY and MUST keywords define optional, respectively mandratory attributes within an object class.

Every object class has a unique Object Identifier (OID), which consist of decimal numbers separated with dots. Enterprise OIDs must be registered by IANA.

The object class definition from RFC 4512 [39, section 4.1.1] is shown below.

```
ObjectClassDescription = "(" WSP
    numericoid                 ; object identifier
    [ SP "NAME" SP qdescrs ]   ; short names (descriptors)
    [ SP "DESC" SP qdstring ]  ; description
    [ SP "OBSOLETE" ]          ; not active
    [ SP "SUP" SP oids ]       ; superior object classes
    [ SP "ABSTRACT"/"STRUCTURAL"/"AUXILIARY"] ; kind of~class
    [ SP "MUST" SP oids ]      ; attribute types
    [ SP "MAY" SP oids ]       ; attribute types
    extensions WSP ")"
```

Where SP and WSP means space, respectively optional space, qdescrs and qdstring are text strings and oids are object identifiers (OID). The example below, taken from ZyTrax [5, section 3.3], shows what an object class definition can look like.

```
objectclass ( 2.5.6.2 NAME ’country’ SUP top STRUCTURAL
  DESC ’2 character iso 3611 assigned country code’
  MUST c
  MAY ( searchGuide $ description ) )
```

Attributes are defined as a part of schema and can be included in one or more object classes. Attributes can be used in object classes as single value or multi value. For example, only one password and more e-mail addresses can be allowed in a schema. Attributes are multi-valued by default.

An attribute definition can include SYNTAX of type and matching rules for comparison, for example rules for a case sensitive or case ignore match.

The attribute definition from RFC 4512 [39, section 4.1.2] is shown below.

```
AttributeTypeDescription = "(" WSP
    numericoid                    ; object identifier
    [ SP "NAME" SP qdescrs ]      ; short names (descriptors)
    [ SP "DESC" SP qdstring ]     ; description
    [ SP "OBSOLETE" ]             ; not active
    [ SP "SUP" SP oid ]           ; supertype
    [ SP "EQUALITY" SP oid ]      ; equality matching rule
    [ SP "ORDERING" SP oid ]      ; ordering matching rule
    [ SP "SUBSTR" SP oid ]        ; substrings matching rule
    [ SP "SYNTAX" SP noidlen ]    ; value syntax
    [ SP "SINGLE-VALUE" ]         ; single-value
    [ SP "COLLECTIVE" ]           ; collective
    [ SP "NO-USER-MODIFICATION" ] ; not user modifiable
    [ SP "USAGE" SP usage ]       ; usage
    extensions WSP ")"         ; extensions
```

The example below is taken from FreeIPA DNS schema [3], and represents how the DNS TTL attribute.

```
attributetype ( 1.3.6.1.4.1.2428.20.0.0
    NAME 'dNSTTL'
    DESC 'An integer denoting time to~live'
    EQUALITY integerMatch
    SYNTAX 1.3.6.1.4.1.1466.115.121.1.27 )
```

Further details about LDAP schemas are beyond of the scope of this thesis.

2.2.4 Syncrepl – Content Synchronization Operation

A syncrepl (Content Synchronization Operation), as RFC 4533[18, section 1] describes, allows a client to request a copy of a fragment of a DIT. Syncrepl supports two modes:

- refreshOnly: polling for changes – the client receives all entries matching the search criteria, and a cookie, represents the state. Client can pass cookie to the server and get only newer data since cookie was received. When data are received, connection is closed.
- refreshAndPersist: listening for changes – same as the refreshOnly, plus the server continues sending updates after the initial poll (both sides must keep their connection open).

The entries returned by syncrepl contain UUID (Universal Unique Identifier), which keeps the same per entry, instead of DN, which can change. UUIDs identify entries uniquely.

The syncrepl in refreshAndPersist mode can be used as notification mechanism to detect any changes in DNS subtree.

2.2.5 Indexation

As is mentioned in the Red Hat Directory Server manual [7, chapter 7], an LDAP server can mantain indexes of attributes to speed up the search, whose filter uses indexed attribute. Implementation of indexes depends on the particular directory server, but there are four common index types:

- Presence index (PRES), which contains a list of entries that contain a particular attribute.
- Equality index (EQ) improves searches for entries containing a specific attribute value.
- Approximate index (APPROX) is used for effective approximate (sounds-like) searches.
- Substring index (SUB) is costly to maintain index, that is used to efficient searching against substrings.

Proper configuration of indexes is crucial to get a good synchronization performance with the DNS server.

2.2.6 Search Scopes

An LDAP server can perform search in several scopes defined in OpenLdap [12], as is shown in figure 2.2 Search scopes:

- BASE indicate searching only the entry at the base DN, resulting in only that entry being returned (if the entry matches the filter).
- ONE indicate searching all entries one level under the base DN, but not including the base DN and not including any entries under that one level under the base DN.
- CHILDREN indicate searching all entries at all levels except the specified base DN.
- SUBTREE indicate searching of all entries at all levels under and including the specified base DN.

Please note, some implementations of LDAP client allows less types of scope, for example python LDAP (base, one, and subtree) [36].

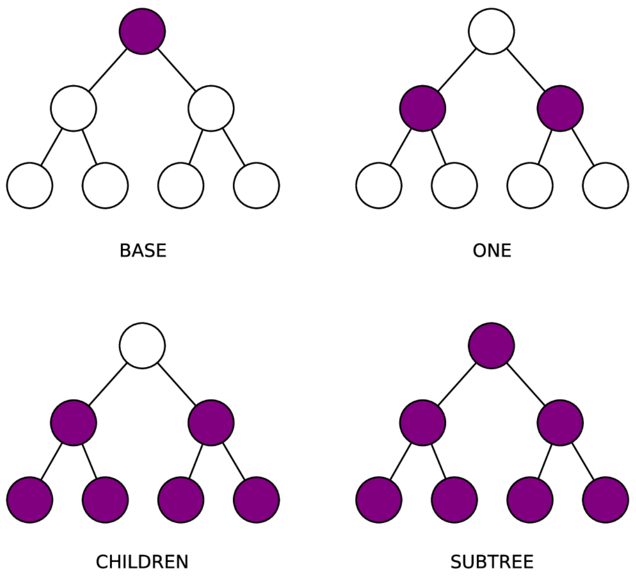

Figure 2.2: Search scopes

Chapter 3

Using an LDAP Server as Data Storage for the DNS

This chapter describes the advantages and disadvantages of using an LDAP server as a storage for DNS data, describe several LDAP schemas which can be used to store DNS data.

3.1 Advantages and Disadvantages of Using an LDAP Server as DNS Data Storage

The LDAP database has several advantages agains storing DNS records in zone files on a DNS server.

Zones and records stored in the LDAP database can contain additional attributes of records which the DNS server can't handle, as the LDAP schema can be easily extended to support new attributes instead of DNS protocol. The new attributes can store additional values lsuch as when new records was added, who adds that record. Also LDAP ACIs (Access Control Instructions) allows to restrict who can modify which zones.

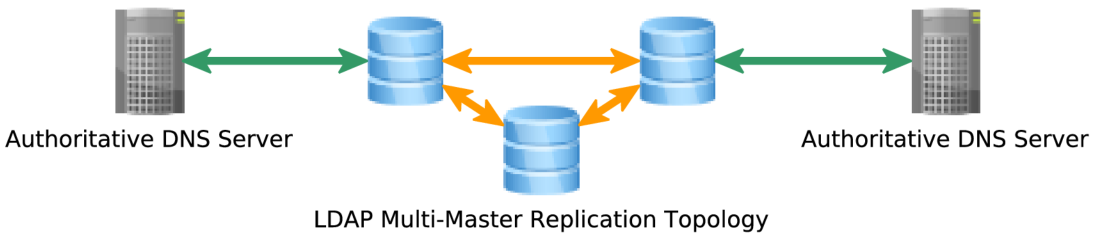

Figure 3.1: Multi-Master DNS Topology

Storing data in the LDAP database together with using multiple DNS servers, which are synchronized directly with this database, leads into multi-master DNS topology where all DNS servers can be authoritative for one zone, as figure 3.1 Multi-Master DNS Topology shows. This eliminates the particular DNS servers as single points of failure, because there are additional authoritative servers with the same copy of DNS sub tree.

Usually the LDAP is used with other services, and the LDAP database is used as backend allows integrate services together, for example as FreeIPA and Microsoft Active Directory, which integrate authentication, authorization service, computer hosts management, user management, DNS and more services together. These integrated services may need to add new records, which are easier to add into the LDAP database using existing code, without the need to implement extra functionality of adding records into the DNS. Also those products usually have a nice (interactive) user interface, which allows to manage zones and records in very comfortable way using the LDAP as data storage.

The main disadvantage is more points of failure than in the classic DNS topology. These additional points of failure can be caused by LDAP replication issues, by synchronization daemon itself or by connection problems between the synchronization daemon and the opposite sides.

3.2 LDAP Schemas for DNS Data

Here is a brief description of some of the LDAP schemas, which may be used to store DNS data. The following schemas will be covered in this section:

- Cosine schema
- dNSZone schema
- BIND-DLZ schema
- FreeIPA schema

These schemas will be compared at the end of this section.

3.2.1 Cosine Schema

The specification for this schema can be found in RFC 1274 [27]. In RFC 4524 [40] were the DNS records related parts of schema removed.

The schema (as defined in RFC 1274 [27]) contains only few DNS related definitions:

- attributes for A, NS, MD, MX, SOA, and CNAME records (RDATA)
- Associated Name attribute – domain name
- DNS domain object class, containing the attributes listed above

The RDATA values of the RRs are stored in the particular attributes of the object class. These attributes represent the type of the RRs.

3.2.2 dNSZone Schema

This schema, as is shown in figure 3.2 dNSZone schema definition, contains only one objectclass (`dNSZone`), which representing both a master zone and records.

The object class requires two mandatory attributes represents a relative name of a record (`relativeDomainName`) and a zone name (`zoneName`). All the other attributes are optional and represent TTL value, CLASS value, and several types of DNS records (radically extended when compared with Cosine schema). The records are stored in textual representation, an attribute represents a type of a RR.

The schema requires to specify the relative domain and the zone name for each entry, which is the main difference from the other schemas described here, that do not allow

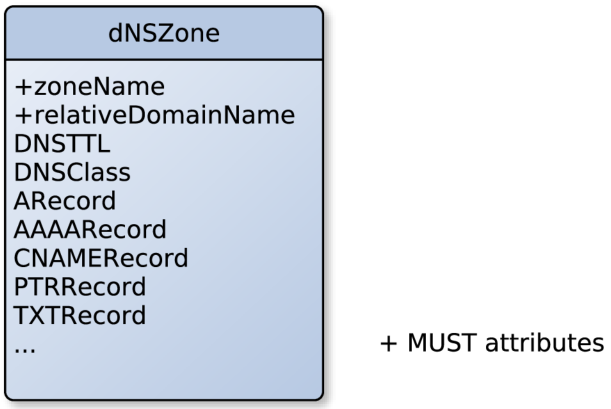

Figure 3.2: dNSZone schema definition

to specify the zone name together with the record name in the same entry. In this manner, the schema does not respect the tree structure of the DNS, because the domain names do not have to construct a tree.

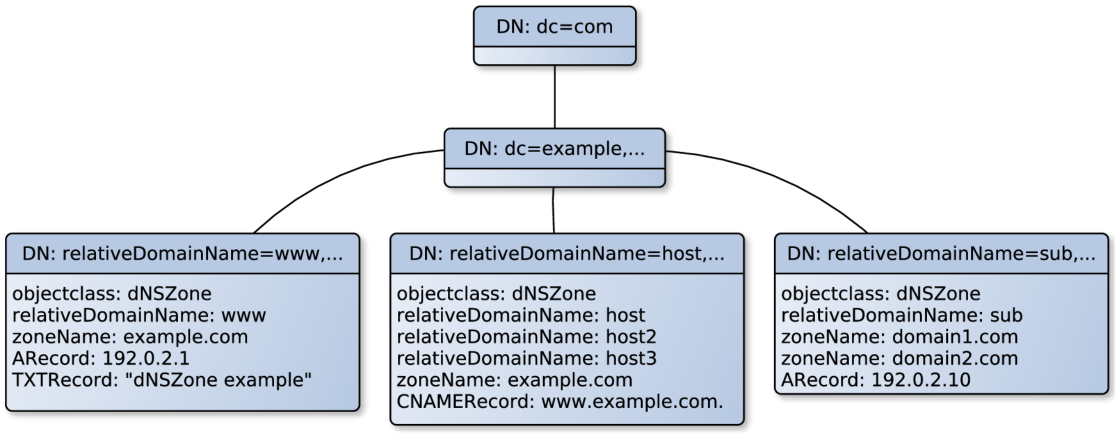

Figure 3.3: dNSZone schema example

The following example in figure 3.3 dNSZone schema example follows a usage described in documentation for BIND 9 SDB backend [4].

As shown in the figure, several ways how to store DNS data exist. The example shows the big freedom which the dNSZone schema allows. An entry could contain just one RRSet (entry `DN: relativeDomainName=www,...`) and follow the tree structure. Also one entry can contain several relative names (as show in entry `DN: relativeDomainName=host,...`), which share one or more record attributes (RDATA), which together create several records with the same data part in a zone. The most interesting combination, which is also allowed, is using the same relative name inside several zones. In this case (entry `DN: relativeDomainName=sub,...`) in each specified domain the A record (`sub.domain1.com`, `sub.domain2.com`) with same RDATA value exists.

3.2.3 BIND-DLZ Schema

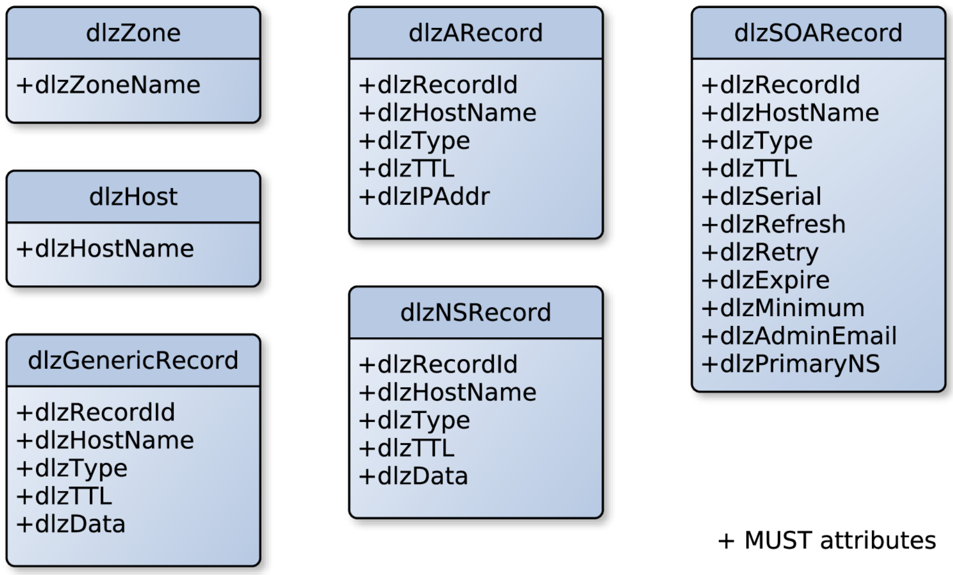

Figure 3.4: DLZ schema definition

In case of the DLZ schema [1], all object classes of the DLZ schema have only mandatory attributes, as is shown in figure 3.4 DLZ schema definition, where several object classes from the schema are shown. The object class representing a zone (`dlzZone`) stores only one attribute – the zone name. To distinguish records into separate subtrees for different owner names, the host objectclass is used (`dlzHost`), which contains just one attribute to specify the relative owner name.

The records are identified by unique ID (`dlzRecordId`), and all records contain a relative owner name (`dlzHostName`), a record type (`dlzType`) and a TTL value (`dlzTTL`). There are three special record types defined as object classes, SOA, MX and A record.

The SOA record is represented as the `dlzSOARecord` object class, and the particular parts of the SOA record are represented as attributes of that object class. The A record (`dlzArecord`) object class definition contains special attribute for IP address. The MX record (`dlzMXRecord`) objectclass definition contains an additional attribute for preference value and inherits data part used to specify exchanger value from the generic object class (see bellow).

The generic record object class (`dlzGenericRecord`) is used for other record types. The schema contains several others objectclass definitions for other records, but these records just inherit all attributes from the generic record object class without adding any new attribute. These additional object classes are for the NS (`dlzNSRecord`), TXT (`dlzTextRecord`), PTR (`dlzPTRRecord`), and CNAME (`dlzCNAMERecord`) records. The record type is always specified by `dlzType` attribute, except for the special records mentioned before, where this attribute must respect the object class, so in this schema the `dlzGenericRecord` object class can be used to store any record types, without the need to define additional object classes or attributes. The attribute `dlzData` is used to store RDATA part of records in textual form, for all generic record types.

This schema allows to store only one record per entry.

The DLZ schema allows two ways how to structure DNS data in the DIT.

The figure 3.5 DLZ schema example shows the first way, which is preferred. The `dlzZone` is the parent entry of a `dlzHost` entry, and the `dlzHost` entry is the parent of `dlz*Record` entries. This structure allows better human readability. The figure shows one zone with two different owner names. The `dlzRecordId=1` represents SOA record in zone apex (zone apex is represented by @ character). As shown in the figure, the owner name `www` contains two records (A and TXT), which are in separated entries, as was mentioned before.

The second way is that the `dlzZone` entry is parent of `dlz*Record` entries directly, without the `dlzHostName` entries layer. Information about owner name is stored inside of every record entry, so technically this layer is not needed.

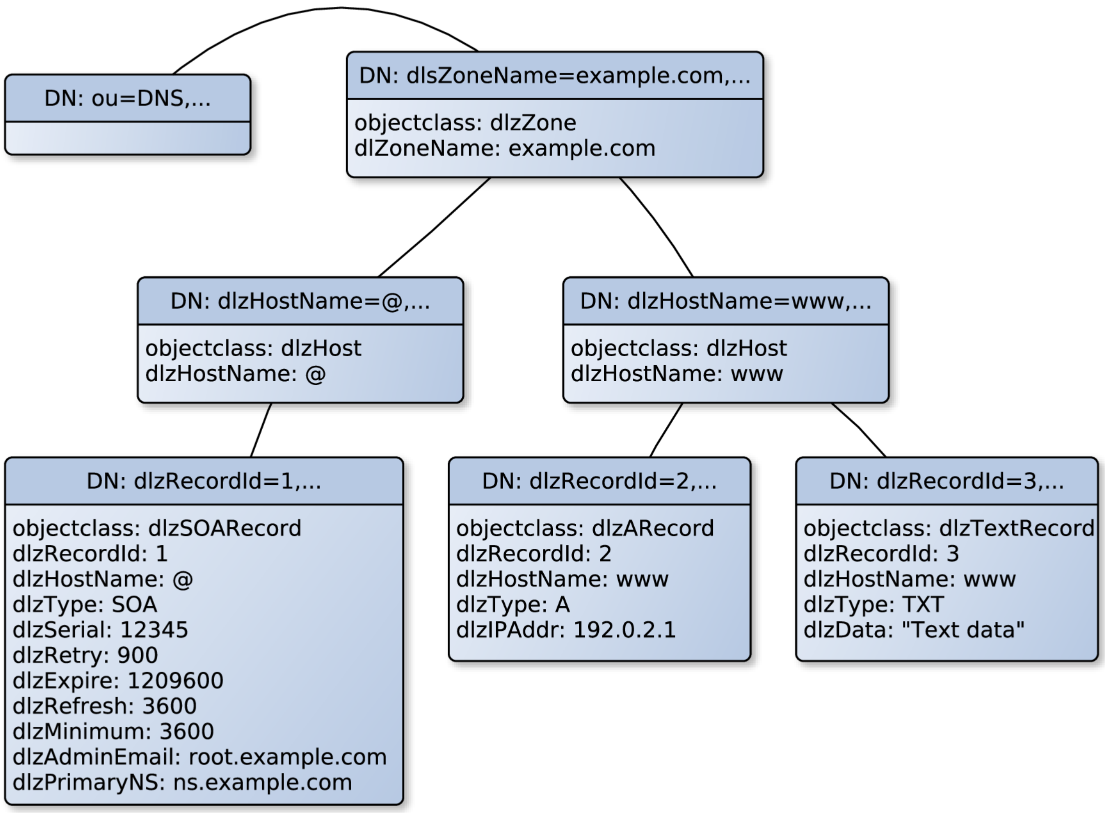

Figure 3.5: DLZ schema example

3.2.4 FreeIPA Schema

The figure 3.6 FreeIPA schema definition shows a simplified definition of the schema. The FreeIPA schema allows to store two types of zones, forward and master, as defined in the BIND 9 server documentation [17, section 6.2.28.1].

A master zone contains records that are authoritative. A forward zone does not contain any records. This type of zone allows to specify forwarding (forward policy and forwarders) for a particular name (instead of global forwarding).

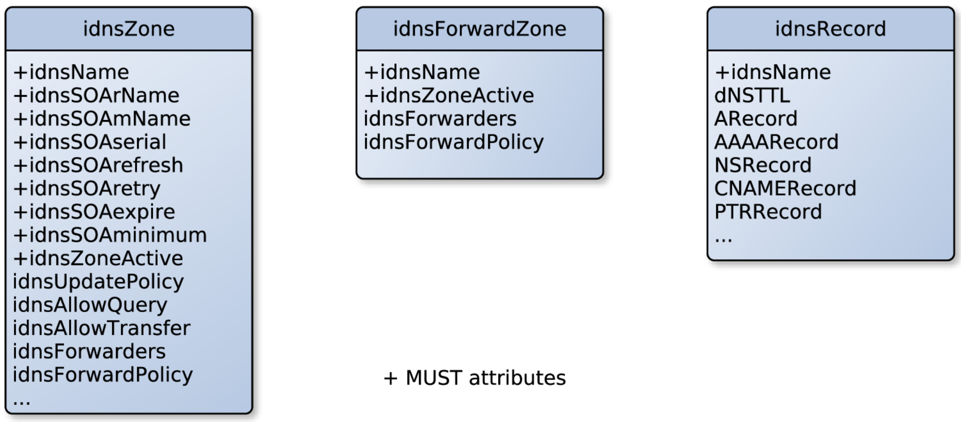

Figure 3.6: FreeIPA schema definition

The master zone definition, represented by the object class `idnsZone`, contains mandatory attributes representing a zone name (`idnsName`), and a flag (`idnsZoneActive`), which can disable the whole zone. Also the SOA record, which is a mandatory attribute as well, is handled differently than the other records. The SOA record is separated into several attributes (`idnsSOArName`, `indsSOAmName`, `idnsSOAserial`, `idnsSOArefresh`, `idnsSOAretry`, `idnsSOAexpire`, `idnsSOAminimum`) representing a particular part of a SOA record.

The object class also contains attributes representing BIND 9 policies for zone (`idnsUpdatePolicy`, `idnsAllowQuery`, `idnsAllowTransfer`) and forwarding settings (`idnsForwarders`, `idnsForwardPolicy`).

The forward zone definition, represented by the object class `idnsForwardZone`, consists only of four attributes. It contains two mandatory attributes representing a zone name (`idnsName`) and a flag (`idnsZoneActive`) showing if the zone is active in name-server, and two optional attributes representing settings for forwarders and forward policy (`idnsForwardZone` and `idnsForwardPolicy`).

The resource records are stored as attributes of the `idnsRecord` object class, which has the only mandatory attribute, representing owner name of RR (`idnsName`). Each type of the RR is represented by an attribute representing the particular type of a RR record (`ARecord`, `AAAARecord`, `NSRecord`, `PTRRecord`, `CNAMERecord`, . . .). The attributes representing RRs are multi-valued, which allows to store multiple RDATA for the particular owner (domain name) in one entry, so one entry can contain multiple RRs, respectively it contains RRSet. Particular RDATA of the records are stored in textual form as defined in particular RFCs for various records.

This schema has implemented most of the standard records types.

The figure 3.7 FreeIPA example, shows a simple example how DNS data are stored in FreeIPA.

Zones of all types and records are part of one DNS subtree in a DIT.

Forward and master zones share the same namespace, so forward and master zone can not coexist together (which is not allowed in BIND name server as well).

The zone `fwzone` in the example is a forward zone. There are no DNS records allowed.

The other zones, `example.com.` and `sub.example.com.`, are master zones. Zone `sub.example.com.` is a subzone of the zone `example.com.`. As the figure shows, there

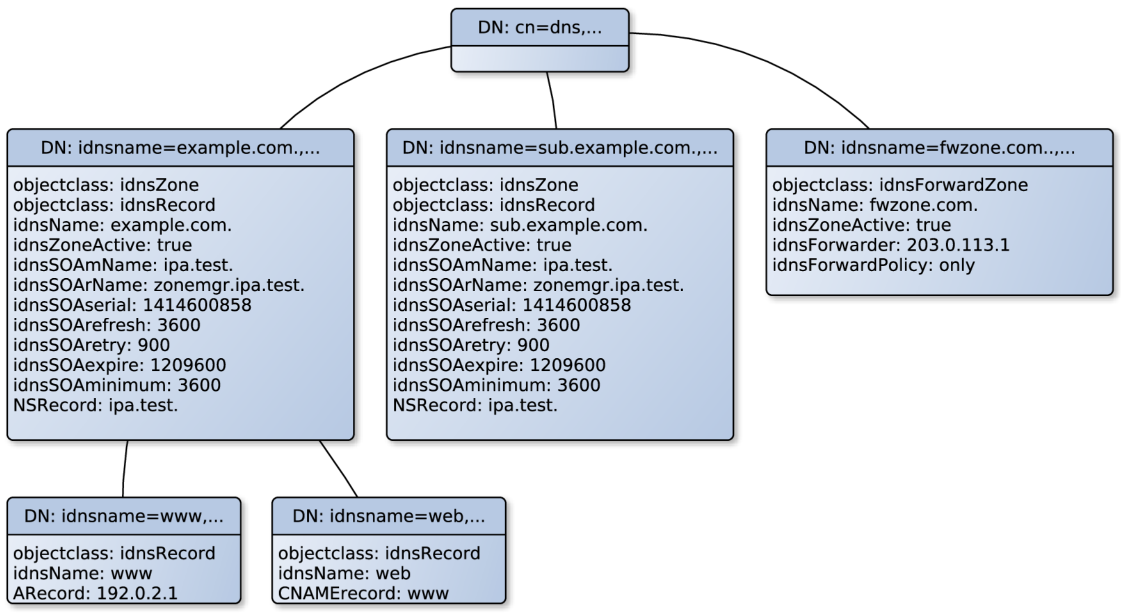

Figure 3.7: FreeIPA example

is no tree hierarchy between child and parent zones in FreeIPA DIT. The tree hierarchy is kept only between master zones and theirs records.

The zone apex records are stored inside zone entries, for example NS record `ipa.test.` in both zones in the example.

Records are represented by a relative domain name (except for zone apex records as was mentioned above). Following records `www` and `web` belong to the `example.com.` zone, so in absolute form records represent domain names `www.example.com` and `web.example.com`.

The other schemas, described before, support only a subset of those RR types.

The FreeIPA schema [3] is part of the `bind-dyndb-ldap` plugin for BIND 9.

3.2.5 Comparison of the LDAP Schemas

Only the LDAP schemas described above are compared in this section, except the Cosine schema that is a very basic schema and will not be covered in this comparison.

Granularity

The BIND-DLZ schema has the highest granularity. This schema allows to specify the TTL for each record separately, instead of FreeIPA or dNSZone schema, where the TTL value can be set only per owner name. As in the DNS system, the TTL is specified per a record, respectively as RFC 2181 [33, section 5.2] describes, the TTL value should be the same for all records in the RRSet. This may be a limitation for the DNS data synchronization.

Number of Record Types

The BIND-DLZ schema allows to store any record types, because it supports generic record object class. The record types are identified by an additional attribute, which allows to use any new record. The other schemas mentioned in this thesis have predefined attributes per record type, which work as identifier for record types.

Enforcing tree structure

The FreeIPA and BIND-DLZ schema definitions require a parent-child structure for zones and records. The dNSZone schema allows any structure of DNS entries, as the information about the zone (or several zones) is stored directly in the entry with records.

The FreeIPA schema also requires exactly one SOA record, as the SOA record is stored directly in the zone object class.

Storing RDATA values

As it was mentioned before, the BIND-DLZ schema allows to store only one record per entry (the RDATA attribute is defined as single valued). The other schemas store RDATA part of the record as multivalued attributes.

Search for records

The search speed is quite important for fast synchronization. The worst case for search records is the dNSZone, where the search filter must evaluate almost all entries to find a particular zone and record, as this schema allows to specify the same record for several zones, or specify the same RDATA for several records inside the same entry.

Achieving better speed is allowed by FreeIPA schema and BIND-DLZ schema, where to find a record, it is required to just search in the subtree, where the root of this subtree is a zone name. The FreeIPA schema also allows to get all records for the same owner name at once, because all the data related to the same owner are stored in one entry.

Additional features

The FreeIPA schema allows to specify additional DNS configuration:

- two zone types, forward and master,
- forwarders and forwarding policy,
- policies for transfer, query, update.

Those features are bounded to the BIND name server.

Also the FreeIPA schema contains special a zone attribute (`idnsZoneActive`), which allows to temporarily disable zone synchronization, without removing or moving any DNS data in LDAP.

3.2.6 Summary

The various LDAP schemas for DNS data have their advantages and disadvantages. However for purpose of this thesis the FreeIPA schema was chosen as the referential schema. The FreeIPA schema is used for proof of concept implementation of the DNS synchronization daemon.

The schema allows intuitive storage for the DNS data, keeps the tree structure and stores all the records in the same entry, which allows fast and easy access to data.

Chapter 4

Synchronization Issues

This chapter contains information about various synchronization issues that may happen during synchronization of records between an LDAP database and a DNS server. These issues are for example: a detection of when a record change occurs, several types of record conflicts, how data caching influences synchronization, and proper causality resolution of record changes. Some of the issues could be resolved in several ways, whose advantages and disadvantages will be discussed in this chapter.

4.1 Change Detection

A short time delay between a change and its detection is crucial to ensure good usability. A long delay can cause connection issues in dynamic environments, where new hosts, such as virtual machines, are added into topology on demand. In these environments, it is not acceptable to have delay in minutes. As new hosts can be added on demand, there must be a possibility of a dynamic change detection, otherwise synchronization daemon just ends up as a regular cron job.

The detection should not waste system resources but it should keep the delay as short as possible. Doing the synchronization in an infinite loop without any waiting or change detection is not an optimal idea, because it causes high usage of system resources.

There are several methods of change detection, that are suitable for different environments, respectively some of them can be used as failback mechanism.

Notifications should be used with periodical detection as the default detection method should, where the periodical detection can work as failback if any issues happen. This should cover the most use cases. Other use cases are described in the particular sections, which follows.

4.1.1 Periodical Detection

This type of detection is suitable for networks, where the time delay is not so crucial, or where notifications cannot be used. Another advantage of this type is in deployment, where a lot of changes is made on one side, but a system administrator is fine with doing synchronization just once per period of time to save system resources.

This solution might be even better than dynamic notification in environments, where changes are made often and neither of sides supports sending incremental changes. Doing an overall synchronization every five minutes is far more efficient than doing it every ten seconds because one or two records were changed.

Of course, this type of detection is not usable and notifications should be used if the environment needs to have the data synchronized as soon as possible.

The periodical detection of change may be used as failback method if notification mechanism failed (for example a UDP notification message is lost).

4.1.2 Notifications

Both synchronization sides must support notifications to ensure the detection delay will be as small as possible.

Each side may support different notification mechanism, which has its own advantages and disadvantages. Usually, notification mechanism requires additional system resources, such as extra threads, connections, bigger amount of memory or extra CPU on all sides (synchronization daemon, both data source sides).

DNS uses a notification mechanism that was described in section 2.1.4 DNS Notify. This mechanism requires an extra opened port on daemon side, which receives notify messages, and an extra thread, which parses these messages and executes responsive actions. As the DNS notify messages are usually send through the UDP protocol, the synchronization daemon may occasionally miss this event and the synchronization may be delayed.

LDAP may use syncrepl described in section 2.2.4 Syncrepl – Content Synchronization Operation as notification mechanism. These mechanisms require to keep network connections open and extra thread on both sides.

4.1.3 Requested by User/Another System

The synchronization should be allowed on demand, respectively it can be triggered by user or process, which could be the best approach in some specific environments.

This method can be used in environments where an administrator uses some framework built on top of the LDAP database to add DNS records. It is suitable for small companies, where only one person is responsible to keep the DNS system up to date, and he runs synchronization on demand, or in environments where just one synchronization at midnight is required.

There could be another system in environment that keeps information about records with complex logic above the possibilities of synchronization daemon, which may decide when synchronization should be executed. In this case, the system just gives signal to daemon API to execute synchronization.

This on demand synchronization also allows to run synchronization in various time periods, which is a more specific case of periodical detection. For example, during the business hours, detection of changes may be executed more often than during night just using a cron job.

This method can also be used as a failback method, if an administrator detects that the notifications do not work.

4.2 Synchronization Speed

The synchronization speed is affected by various factors, such as the network link speed, the amount of data to be received or send, the amount of data to be processed, the response delay from the remote side, etc.

To reduce the amount of data to be transferred and processed, incremental changes may be used (details are described in section 4.3, Incremental Changes).

A data caching also allows to save some resources by reducing the number of times when getting data from remote side is required. The cache can temporarily keep the results during temporal connection loss instead of processing all data inputs again. Caching is described in section 4.2.1 Data Caching.

As zones are independent, a computing of differences for each zone can be done parallelly. Usually the LDAP server and DNS servers allow to open parallel connections, so getting data and sending updates can be done independently for each zone per remote side. If parallel connections are not supported by remote sides, respectively parallel queries are internally serialized in those servers, it causes a bottleneck, which degrades the possible speed. However, it is still important to know this case, even if this situation cannot be solved by the synchronization daemon, so at least parallel computation can bring a speedup.

4.2.1 Data Caching

The main reason for caching is to make synchronization and computation of differences faster. The records cache stores all sucessfully synchronized records per zone. This cache can be used to detect changes since last event in case when the remote side does not support sending incremental changes. Also, the cache helps to improve performance if both sides support notifications (if we can rely that notifications are received immediately), but if one of remote sides (or both) does not support sending incremental changes, the synchronization process can determine changes without querying opposite side to get data for comparison. The data is compared localy using the cache and no additional data from the opposite side has to be downloaded.

For example, if the DNS server sends notify message (with a higher SOA serial number than the current), but it allows only AXFR transfer, a difference can be computed by using a cache without downloading the full set of records from the second side (LDAP). Less data downloaded and less data compared make synchronization faster than downloading all data and doing full comparison.

The disadvantage is the permanent memory consumption using the cache.

4.3 Incremental Changes

Incremental changes cover both an ability of the remote side to send incremental data and an ability of the opposite side to receive update in incremental format.

In this case, the smallest incremental entry is one DNS record. As was mentioned in section 2.1.2 Resource Records (RRs) and Zones, the record is identified by: owner name, type, class, TTL and RDATA. Any change made to a record can be considered as removing one record and adding a different new record.

The ability to receive incremental updates implies, that the remote side must be able to work on records basis, e.g. must be able to remove a particular record or to add a particular record without any additional information (except the zone name where the record belongs).

The DNS servers naturally work on record basis, it is not an issue for them. However, various LDAP schemas may not fulfill this requirement, for example when a schema stores all RDATA values in one STRING as SINGLE-VALUED attribute per owner. This is an exaggerated example, none of the schemas listed in section 3.2 LDAP Schemas for DNS Data use that format, but is not impossible to create this type of schema.

The special case described in section 4.4 LDAP Schema Granularity affects some of schemas mentioned before, but to solve this special case, there is no need to have the information about other records, so it is considered as an incremental upgrade and a separate issue.

Downloading incremental changes from DNS is allowed by configuring IXFR zone transfer. An LDAP database combined with a proper schema and syncrepl (described in 2.2.4 Syncrepl – Content Synchronization Operation) can be used to receive incremental updates. In case of FreeIPA schema, the whole owner entry where the change occured is returned by syncrepl. There is no possibility to decide which record was changed without caching the original value.

Less records mean lower number of comparisons, so incremental changes allow faster synchronization.

Unfortunately, the availability of incremental changes depends on the DNS server and the LDAP server and in some cases it cannot be enabled. For this reason, the synchronization daemon has to support the overall synchronization by comparing all records on both sides.

4.4 LDAP Schema Granularity

As mentioned before, DNS system naturally works with records as with the smallest entries. Respectively as RFC 2181 [33, section 5.2] describes, to have different TTLs in the RRSet is deprecated, but synchronization cannot rely on it and must still handle the record as the smallest possible block, because it is still possible to have different TTLs in the RRset. However, LDAP schemas may not consider the record or even RRSet as the smallest independent unit, and may use the RRSets, respectively a group of RRSets, which shares some values among records. This is a case of FreeIPA schema as it uses just one entry per owner name, which shares the same value for TTL and CLASS for all records belonging under this owner. The following two sections describe the issues with schema granularity where various values must be mapped into a one for a RRSet or a group of RRSets.

4.4.1 Record Class

Currently only the `IN` (INTERNET) class is commonly used, so this issue can be solved by supporting only `IN` class without decreasing real usability.

4.4.2 TTL

As figure 4.1 FreeIPA schema granularity example shows, the most serious issue with FreeIPA schema is setting the TTL value just per owner name. Multiple records with different TTL values, must be mapped to one TTL value for a whole owner name represented by FreeIPA LDAP entry.

In this case, there are several solutions that can be chosen by system administrators to fulfill their requirements.

Possible solutions for schemas that share multiple TTL values for multiple RR sets:

Ignoring TTL value
: The TTL will be ignored for synchronization, a default TTL value specified by administrator will be used for the LDAP.

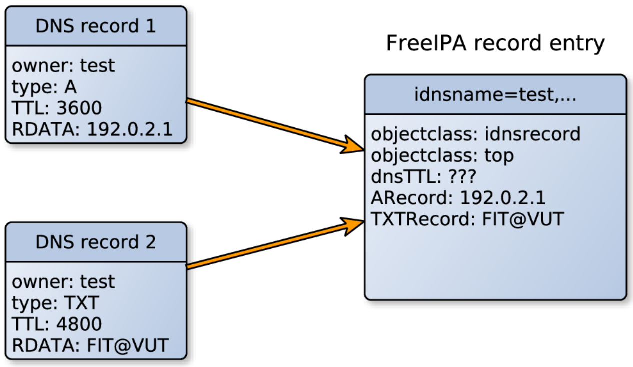

Figure 4.1: FreeIPA schema granularity example

Using the lowest/highest TTL value
The lowest/highest TTL will be used per owner. This solution does not scale well and should not be used, because the TTL value will always decrease/increase for whole owner name.

Using the latest TTL value
In this case administrator must count with fact that one record will change the TTL for the whole owner name.

Ignoring TTL value & using the latest TTL value for LDAP
A TTL value will be ignored during comparison. The records on the DNS side will keep their TTL values and the owner entry on the LDAP side will always rewrite the current TTL value with the value from the latest updated record. The new records from the LDAP side will be synchronized to the DNS with the current TTL value in the LDAP owner entry.

For schemas where records are stored as separate entries, one-to-one mapping for TTL synchronization can be used.

4.5 Causality

The ability to detect which changes happened first, using notify mechanism may be influenced by several factors:

- a notify message is going through network with a bigger delay,
- a notify message can be lost,
- the LDAP multi-master replication can cause the delay, if a record change was made in different a LDAP server than the one directly connected to the synchronization daemon.

All of issues listed above cause the notification that was created first may be delayed so much that notification from opposite the side which was created after the first can, arrive earlier than the first notification message. This may result into a record conflict or replacement of newer data by older data.

As synchronization is not dependent on notifications (periodical check, triggered check), the common mechanism described in section 4.7 Conflicts to solve record conflicts must exist, because in many cases there is no way to detect which change happened first. Notifications can be used to detect causality only for improving synchronization results, the synchronization daemon cannot rely on causality derived from notification mechanism.

The DNS notify messages do not support any timestamps, only an SOA serial number, which determine ordering only on the DNS side, which will not help to determine the overall causality of all messages.

To avoid issues caused by an unreliable causality detection, after receiving a notification from one side, both side should be checked for changes, instead of directly applying the changes to the second side.

Of course, in environments where mainly one way synchronization happens for most of the time, downloading changes just from the side where a notification originated may make the synchronization faster without negative effects.

4.6 Duplicated Notifications

The synchronization process must implement detection of duplicated notifications, as the DNS notify messages can be sent in multiple copies. This detection may be implemented by comparing the SOA numbers and processing only messages with higher numbers.

There is a special case, which can be considered as a duplicate notification: when records are updated on the remote side, then the remote side will generate a notification that references the data that were in update sent by the synchronization daemon. For example, the synchronization daemon sent the update containing the record `test A 192.0.2.1` to a DNS server. Then the DNS server generates a new notify message because it received a new record (`test A 192.0.2.1`), but this records was already processed by the synchronization daemon.

The synchronization must filter this kind of notifications to avoid processing data that were already processed. In case of the DNS notify messages, the SOA serial numbers can be used as described above. In case of the LDAP and syncrepl, the returned entries must be compared with a record cache to detect if the notify message contains any new changes.

4.7 Conflicts

Several types of conflict may happen during synchronization. Some types of conflicts may happen just with incremental data, other just with full data exchange with no history.

Only the conflicts that can be caused by synchronization are discussed. The conflicts caused by improper input data, which violates rules specified in RFC documents, are not handled by synchronization daemon and can lead into error state. This kind of conflicts should be checked and resolved on remote synchronization sides.

This section describes various conflicts that must be solved by the synchronization daemon.

All issues mentioned here can be solved by choosing the authoritative side, where the values from this server will replace the conflicts.

4.7.1 Records Which Cannot Coexist Together

Some record types, such as CNAME [23, section 3.6.2] and DNAME [34, section 2.4], cannot coexists with other records under the same owner name. Respectively as RFC 4035 [31, section 2.5] specifies, for the signed CNAME record, there can be RRSIG and NSEC RR sets for the same owner as for the CNAME RR.

For example, this conflict may easily happen, if a new A record and a new CNAME record were simultaneously added on remote synchronization sides for the same owner name.

Solutions for this situation are to keep those particular records unsynchronized and let an administrator to decide which record should be there, or choose an authoritative server.

4.7.2 Different SOA Values

As the one SOA record only is allowed per zone, this conflict may happen often, as it can not be resolved by keeping both SOA records as RR set.

The easiest part of the SOA synchronization is the SOA serial number. The highest serial number can be safely chosen as the right value, because the serial is used to detect changes in the zone, the higher serial the newer version of zone data available, and thus the highest value should be used to spread changes to other servers. Of course the SOA serial number is increased after each synchronization event.

The other parts of SOA records are not resolvable as easily as the serial number. For example during the initial synchronization, if the SOA records mismatch, there is no way how to decide which value is the right one. The changes after the initial synchronization can be compared per particular value in the RDATA part of the record, and non conflicting changes can be merged.

The solution in this case is to mark one remote side as authoritative for the SOA records, and values from this side will be used for the SOA result record.

4.7.3 Record Changed on Both Sides Simultaneously

This case happens when a record A is on both sides, then simultaneously an A record is replaced with B represented by $A \to B$ record on the one side and C represented by $A \to C$ record on the other side.

The recommended solution in this case is to keep both changes ($A \to [B, C]$).

4.7.4 Record Removed and Added on Both Sides Simultaneously

This case can happen if the synchronization daemon receives contradicting messages from incremental changes. In this case, if data were synchronized before, one side sent wrong data, which may be caused by a malfunction.

This issue can be solved by the synchronization cache, which can detect which side sent the wrong data.

4.7.5 Record Removed and Changed on Both Sides Simultaneously

In fact, this case that could look like a conflict is not a conflict. As the DNS system works with records, this is interpreted as the record was removed on one side and the same record was removed on the other side, plus another record was just added. Removing the same record on both sides is not a conflict, and adding new record is not a conflict either.

4.7.6 Singleton Records which Have More than One Value

Singleton record is a record type that allows to have only one RDATA value per owner. For example the singleton record is the DNAME record type [34, section 2.4].

This case can happen when one side creates a new DNAME record and the other side creates new a DNAME record too but with a different RDATA value. As mentioned above, this cannot be solved by merging into one RR set containing two records. The solution is to keep these records unsynchronized or choose the authoritative side.

4.7.7 Different TTL

If the same record on both sides contains different TTL values, several strategies can be used to solve this issue.

The higher value or the lower value can be used as the resolution, default value, or an authoritative server can be chosen. A possible solution is also not to compare the TTL values, but then the TTL values must be maintained manually on both sides, which miss the goal of the synchronization daemon.

4.8 DNSSEC

DNSSEC – DNS Security Extensions, as RFC 4033 [29, section 3] describes, is a technology that allows to verify the authenticity of record or gives a proof that a record really does not exist, by using public key cryptography. DNSSEC builds a trusted signatures chain from the root zone to the child zones, where the root zone keys are well known and each child zone can be validated using the record in a particular parent zone.

The DNSSEC technology adds several new types of records, which are specified in RFC 4035 [30], RFC 5155 [10, section 3, section 4], and RFC 4431 [21], into the DNS system:

DNSKEY record contains a public key for each private key that was used to sign a zone. DNSKEY record is used to validate RRSIG signatures in the zone.

RRSIG record is created for each authoritative RR set and contains digital signatures for that particular RR set.

NSEC, NSEC3 records are used to "provide authenticated denial of existence" for DNS data, because even replies for a name that does not exist must be signed.

NSEC3PARAM record contains parameters to be used for generating NSEC3 record.

DS records are used to verify DNSKEY key records and must be stored in the parent zone.

DLV record allows to validate a DNS record out of normal delegation chain.

The NSEC3PARAM, DS and DLV records can be handled by the synchronization daemon as any other regular records. The other DNSSEC related records (DNSKEY, RRSIG, NSEC, NSEC3) are usually created by a DNS server not by user. The DNS server uses private keys, which are stored (and generated) locally on the DNS server, to generate RRSIG, NSEC and NSEC3 records, and to put public part of the key into DNSKEY record. No private key can be exposed into the DNS system, so there are two approaches how to handle them for synchronization purposes, mainly in DNS multi-master environments.

The first approach is that the only DNS server in the topology has the private keys to generate signatures, and other DNS server will use the same signatures. Then all the generated records (DNSKEY, RRSIG, NSEC, NSEC3) are synchronized to the opposite sides, respectively these records must be synced into the LDAP database. This may require an additional configuration of the DNS servers which should not generate signatures.

The second approach is that all the DNS servers have their own private keys, and each server signs the records by itself. In this case, for each generated KSK key a DS record must be added to the parent zone. The synchronization daemon can ignore all DNSKEY, RRSIG, NSEC and NSEC3 records, because all DS records are stored in the parent zone and any signed record by any key can be successfully validated. This approach should be also used in case when only one DNS server is in topology.

Chapter 5

Implementation

This chapter contains high level design of the synchronization daemon and various designs of particular parts of this daemon.

This chapter also contains information about public interfaces, which can be used to extend functionality by more plugins.

At the end of this chapter, how various settings of the daemon effect the synchronization process is discussed.

5.1 Proposed Synchronization Algorithm

The following section contains high level algorithms for synchronization of DNS records.

5.1.1 Keyword Definitions

This section describes keywords used in further in this text, in description of algorithms or in description of the daemon design.

The keywords are:

incremental data contain information which modifications happened since last time.

all data contains all current DNS data without information about changes made in the past.

DNSDataPackage data structure that stores information about records, including a flag if it contains incremental data.

DNSUpdatePackage data structure that stores information about data that will be updated.

DataCache data structure that contains all current records that were already synchronized.

authoritative value, authoritative source synchronization side chosen by a user whose values are used in case of conflicts.

initial synchronization is the first synchronization per zone, when the synchronization daemon requires to download all DNS records from both sides.

5.1.2 Algorithms

For better understanding, the following algorithms are simplified and implementation details are omitted in this section. Also a part of paralelization, various submodules, user settings and communication routines are omitted or very simplified in the representations. These details are covered in section 5.2 Synchronization Daemon Design, algorithms listed here, give only basic picture of how synchronization is executed.

The descriptions of the particular algorithms follow.

TTL Conflicts Resolution

Algorithm 1 resolves the TTL conflicts, when the records with the same owner, type, class and RDATA have different TTL values for each side. This algorithm compares the records which should be added only. Removing records cannot cause the TTL conflict.

```
Method resolveTTLConflicts(A, B: DNSUpdatePackage; cache: DataCache)
    Data: DNS updates
    Result: Resolved TTL conflicts
    /* Only updates which adds new records must be compared.                */
    for each record $a \in A.add$, $b \in$
    $B.add$ where a and bare the same record with different TTL value do
        if cache then
            if $a \in cache$ then
                /* b is newer, use b                                         */
                Remove a from update A.add;
                continue;
            end
            else if $b \in cache$ then
                /* a is newer, use a                                         */
                Remove b from update B.add;
                continue;
            end
        end
        if authoritative source is specified then
            if source(A) is authoritative then
                /* Do not update authoritative side                          */
                Remove a from A.add;
            end
            else
                Remove b from B.add;
            end
        end
        else
            Remove a and b form updates;
            Add a add b to unresolvable records list;
        end
    end
```

Algorithm 1: Resolving conflicts of TTL

Comparation of SOA records

This method is used for comparing SOA records. As there can be only one SOA record per zone, the resolving conflicts occurs in different manner than for other record types. As shown in algorithm 2, each attribute of the SOA record is compared separately. The algorithm is trying to decide which SOA changes are new by comparing them with the cached SOA record. If there is an unresolvable conflict, an error is returned or authoritative value is used (if authoritative side is configured).

The serial numbers of SOA records are handled in different way, the highest serial number (that will be later incremented) is chosen for the new SOA record to achieve the change will be distributed to all slave servers.

```
Method newSoa(A, B: DNSDataPackage; cache: DataCache)
    Data: DNS data received from zone plugins and DNS cache
    Result: new SOA value
    /* Initialize with values of A                                   */
    soa = SOA(A.soa);
    Compare each SOA attribute between A and B, except serial;
    for each attribute which does not match do
        if cache then
            Compare values against cache and use the one which changed;
            In case both values changed, use authoritative value or raise
            Error(Unable to synchronize SOA);
        end
        else
            Use value that is authoritative, or raise Error(Unable to synchronize
            SOA);
        end
    end
    soa.serial = max(A.soa.serial, B.soa.serial);
    return soa;
```

Algorithm 2: Resolving values of a new SOA record

Diff Algorithm

The following algorithm, algorithm 3, is responsible for returning proper differences that will be used for update. As written in algorithm representation, the first part is responsible to generate a new SOA record, using the previously defined method `newSoa()`.

The method `removeSameRecords()` compares record sets and returns only records that do not match. There is a difference between incremental and non-incremental DNS data.

With non-incremental data (all records), we can only add new records to both sides, because there is no way how to detect which records should be removed. With incremental data, there must be a separate comparison for newly added and removed records. Please note the sides are switched for results from removing the same records on sides ($B,\ A \leftarrow A,\ B$), because the records that were added on the A side must be also added via update on the B side and vice versa.

The method `resolveConflicts()` solve record conflicts (except TTL conflicts), that were discussed in section 4.7 Conflicts. This method must also provide the list of records that cannot be synchronized.

Before the updates are sent to the remote servers, the SOA serial is incremented.

```
Method diff(A, B: DNSDataPackage; cache: DataCache)
    Data: DNS data received from zone plugins and DNS cache
    Result: updateA, updateB: DNSUpdatePackage; list of record that were not
            synchronized
    soa = newSoa(A, B, cache);
    if both A and B are incremental then
        updateB.add, updateA.add = removeSameRecords(A.added, B.added);
        updateB.remove, updateA.remove = removeSameRecords(A.removed,
        B.removed);
    end
    else if both A and B contain all records then
        updateB.add, updateA.add = removeSameRecords(A.all, B.all);
    end
    else
        /* This case is unsupported                                    */
        /* Records should be compared with records in the cache to get
           incremental data, or both sources should send all records   */
    end
    resolveTTLConflicts(updateA, updateB, cache);
    resolveConflicts(updateA, updateB, cache);
    increment (soa.serial);
    updateA.soa = soa, updateB.soa = soa;
    return updateA, updateB
```

Algorithm 3: Computing records differences

Synchronization Algorithm

Algorithm 4 describes the synchronization process for a particular zone. For each zone an extra process and separate data cache are created, which ensures that a synchronization failure of one zone will not affect others.

The method `diff()`, which was previously defined, is used to compute updates. The process always waits for any external event to start a new synchronization loop. The process itself can not decide when it should happen.

```
Method zoneSync(zonename: DNSName)
    while true do
        if initial synchronization then
            A, B ← download all records of zone zonename;
        end
        else
            A, B ← try to download incremental changes of zone zonename;
            A, B ← compute incremental changes using the cache if non-incremental
            data was received;
        end
        updateA, updateB = diff(A, B, cache);
        Send updates (updateA, updateB) to remote sides, if required;
        if initial synchonization then
            Initialize cache for the zone zonename;
        end
        else
            Update cache for the zone zonename;
        end
        Wait for next event;
    end
```

Algorithm 4: Zone synchronization

Algorithm 5 describes the main loop of the dnssyncd synchronization daemon, which is mainly responsible to detect which zones will be synchronized and send events to particular zone processes to run synchronization.

The detection of zones which will be synchronized is made first. This depends on current settings, it can be a static list of zones or a dynamic source of zones can be specified. As previously mentioned, each zone has its own process which performs synchronization of records.

The rest of the time the main loop just waits for any event. The most common events and matching actions are listed in algorithm representation.

```
Method mainLoop()
    Parse settings;
    Detection which zones will be synchronized;
    for each zone to be synchronized do
        Create zone, if needed;
        Create a new zone process;
        /* Executed in a new process                                    */
        zoneSync(zone name)
    end
    while true do
        event ← wait for an event;
        switch event do
            case timer ends
                Send event to all zone processes to start synchronization;
            end
            case records changed notification
                Send event to the particular zone process;
            end
            case new zone notification
                Add new zone;
                Start zone process for the particular zone;
            end
            case zone removed notification
                Stop zone process for the particular zone;
                Remove zone;
            end
            case user event
                Execute actions required by a user;
            end
        endsw
    end
```

Algorithm 5: Main loop of the dnssyncd daemon

5.2 Synchronization Daemon Design

As figure 5.1 Synchronization topology shows, the proof of concept of the synchronization daemon (dnssyncd) is connected in topology between an LDAP Server and a DNS server (represented by 389 DS and BIND 9).

The standard LDAP operations are used to update DNS records (and zones) in the LDAP database, specifically the LDAP search operation is used to get records and zones data and the LDAP modify operation is responsible for updating and creating records and zones in the LDAP database. Syncrepl is used as a notification mechanism.

In case of the DNS server, a zone transfer is used to download records from the server, dynamic dns updates (DDNS) are used to update records, `rndc` util is used for zone creation/deletion and notifications are arranged by DNS Notify messages.

Details are explained in following sections.

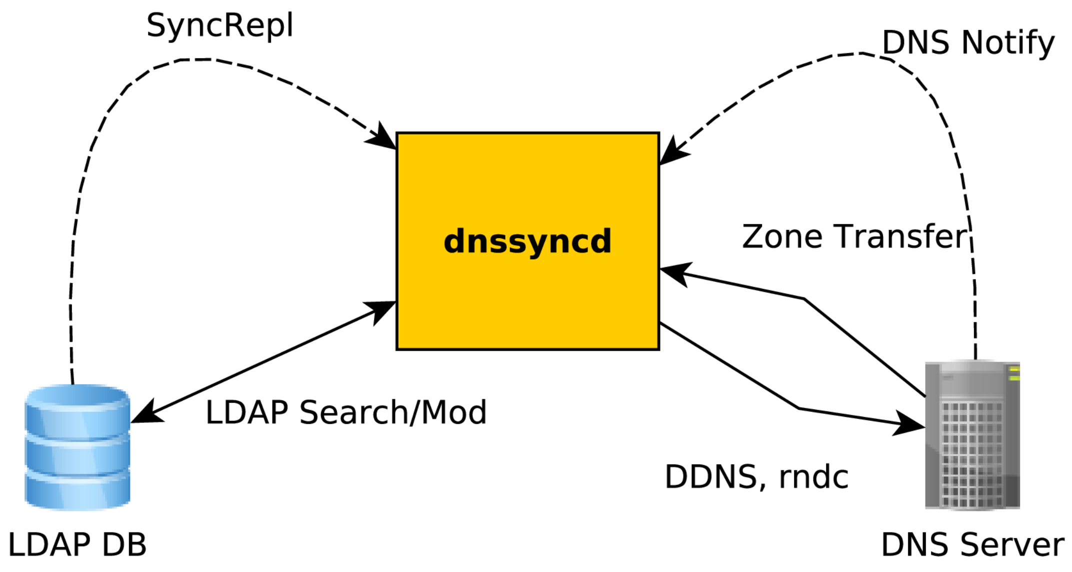

Figure 5.1: Synchronization topology

5.2.1 High Level Overview

The DNS synchronization daemon is a multiprocess application, which consists of the following separate processes as the figure 5.2 dnssyncd diagram shows:

- core module,
- zone module (one per zone),
- management module,
- and notify modules (optional).

The communication between modules (processes) is ensured via messages. (Message flows are represented as arrows in the figure.) The core module works as a central hub, where all the other modules are connected. All messages must pass through the core module, as dashed arrows show in the figure, which is responsible for routing to proper subprocess.

Core Module

Core module is the main process of the synchronization daemon, which exists first and manage the other modules/processes. This module parses settings files, creates other processes on demand, control other modules, destroy processes when they are unneeded. This module is also responsible for adding and removing zones and detection of new zones using core plugins, which are described in detail in section 5.2.3 Core Plugins.

As the synchronization of zone is done on records level, the core module requires to know only zone names. The only actions that core module does with zones are: creating a new zone if it should exists, removing a zone if it should be removed, detect if there is any new zone that should be synchronized, and create a zone module for each zone.

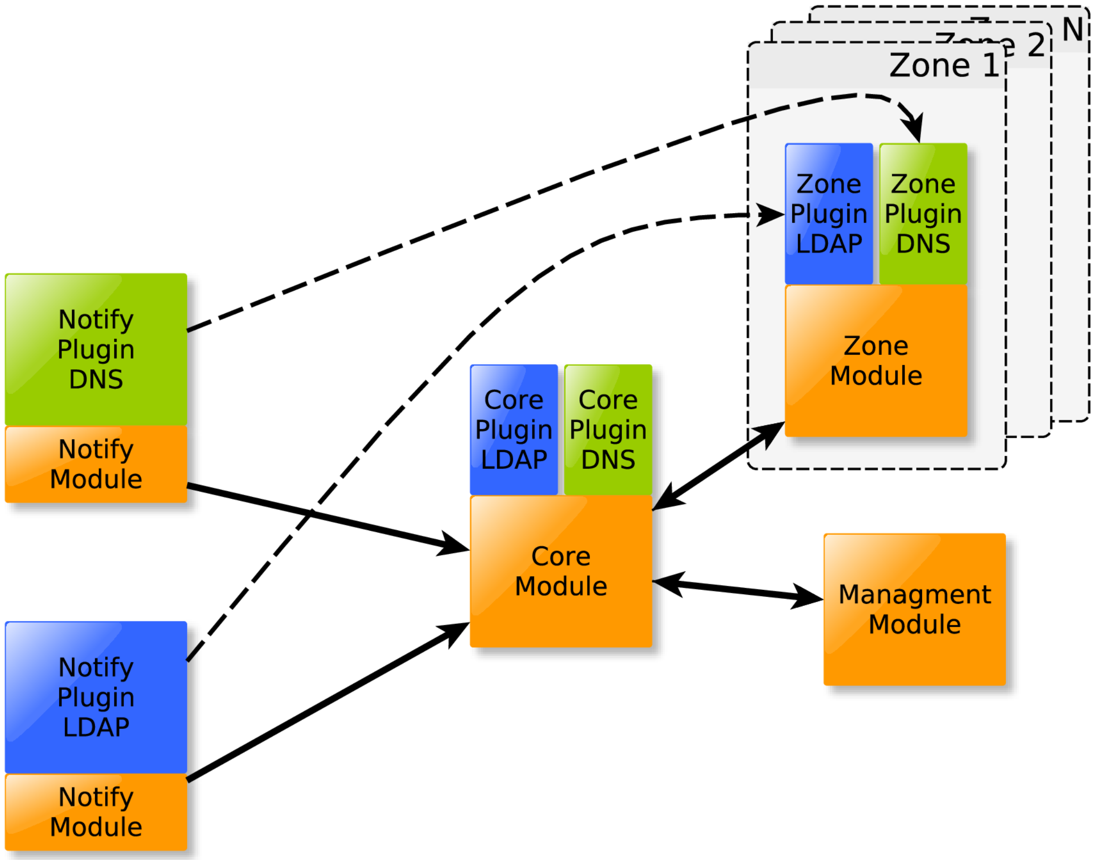

Figure 5.2: dnssyncd diagram

Zone Module(s)

Synchronization of records is done by zone modules. The synchronization algorithm, described in 5.1.2 Algorithms, is executed in this module.

One zone module instance exists for each zone. The zone module is responsible for keeping the particular zone synchronized, for getting record data, updating record data, and resolving conflicts. Zone modules are independent on each other, so if there is an issue with synchronization of a zone, other zones are not affected and will continue to be synchronized.

Zone modules wait until the core module requests to start synchronization by a message. The reason for relying on a message is that the only core module is aware of any external event, like received notification, manual triggered synchronization or timeout event.

Also these modules allocate the highest amount of system resources compared to the other synchronization daemon modules, as they keep lot of DNS data in cache and work with bigger amount of data during synchronization phase which must be stored in memory and of course computation of differences uses a lot of CPU power.

Zone modules use zone plugins described in detail in section 5.2.3 Zone Plugins.

Notify Module(s)

Notify module is responsible to wait for any notification event from a DNS or an LDAP server and then to immediately send a message to the core module. Current implementation allows just one notify module per one side (one for DNS, one for LDAP). Each module is a separate process.

Notify modules are optional part. Plugins for them may not be implemented, or notification modules may be disabled in settings. Then other methods of change detection must be used, such as a periodical detection.

The notification method is strongly dependent on remote service. The plugin interface is described in section 5.2.3 Notify Plugins.

Management Module

Management module allows direct control over the `dnssyncd` daemon. The supported actions are:

- List all zones that are being synchronized with their synchronization status.
- Show a particular zone in detail.
- Add a new zone that will be synchronized.
- Delete zone from synchronization daemon.
- Suspend synchronization for a particular zone.
- Force starting synchronization for a zone.

A user can use `dnssyncd-control` CLI utility, which is part of the daemon implementation. This utility implements all operations mentioned above by sending a requests to the daemon. The CLI utility contains built-in help.

5.2.2 Synchronization States

The figure B.1 Synchronization states diagram shows the state diagram of the DNS synchronization daemon. These states are independent between zones, i.e. only zone modules work with them.

The particular synchronization states in daemon are:

`init` – **plugins initialization** Zone plugins are being initialized, testing configuration and connection to servers.

`active_read` – **synchronization is getting DNS data** Synchronization process started, zone plugins are gathering DNS data required by synchronization algorithm.

`active` – **synchronization active** Synchronization algorithm is computing differences and creating DNS updates.

`active_write` – **sending updates** Zone plugins are sending DNS updates to servers.

`unsynced_readfailed` – **failed to get DNS data** Unsynchronized because at least one of zone plugins cannot gather the DNS data from the server.

`unsynced_writefailed` – **failed to send updates** Unsynchronized because at least one of zone plugins cannot successfully send the data to the server.

`unsynced` – **usynchronized** Synchronization algorithm is not able to compute differences, or any internal error during the synchronization.

`sync` – **synchronized** The DNS zone was successfully synchronized. The process will sleep until new synchronization is required.

`part_synced` – **partially synchronized** Some of DNS records cannot be successfully synchronized. These records are recorded and user can show them.

`suspended` – **synchronization suspended** – A critical error happened, synchronization was disabled by user, or synchronization process got stuck between unsynchronized states for too long. This means the process does not exist, just the core module keeps its status. Synchronization must be manually re-enabled for this zone.

5.2.3 Plugin Interface

Plugins work as a bridge between remote servers and synchronization daemon. Each server type has different API, different methods how to access data, how to modify data or how to receive notifications.

A plugin must do mapping between different data format: the format required by synchronization daemon and data served by remote server. Also plugins must map daemon calls to appropriate actions. Each type of plugin has a different subset of actions.

All required operations by synchronization daemon which plugins must implement are mentioned in following sections.

Common Interface

All plugin types have to provide a list of **mandatory** and **optional** (with default values) arguments which must/may be used in synchronization daemon configuration file for that particular plugin. Also the plugin must provide static method to validate options, which raise exceptions if requirements were not met.

Core Plugins

As mentioned before, core plugins are required for zone operations, so all methods this type of plugin implements are zone related.

The methods required for this plugin are:

add_zone(zonename, records) – this method is called by the daemon when a new zone, specified in parameter `zonename`, must be added. The zone is created with records specified in parameter `records`

del_zone(zonename) – this method is called by daemon when a zone, specified in parameter `zonename`, must be removed.

zone_exists(zonename) – called by daemon for detection if the zone with name specified in the parameter `zonename` exists.

get_zone_records(zonename) – called by the daemon to get all the records from the zone that is specified in parameter `zonename`, usually during creating this zone on opposite synchronization side.

get_zones() (optional) – if this method is implemented, a plugin may serve as a source of all zones that should be synchronized. Otherwise, if none of plugins implements this method, a user have to specify which zones should be synchronized manually in configuration file (or via the management module).

Zone Plugins

For each zone a new instance of a zone plugin is created to ensure the individual zones are processed separately. The methods implemented by the zone plugin work with the DNS records of the particular zone.

Method required for this type plugin:

get_zone_data(all_records=False, rr_owners) – returns records from zone. By default the synchronization daemon expects records in incremental form. The option `all_records` forces the plugin to return all actual records. The options `rr_owners` pushes the plugin to return only the records for specified owners. If no owners are specified, records from all owners must be returned.

update(zonedata) – method is called when the update, specified in `zonedata`, is ready to be sent to the remote server.

process_notify_message(message) – method is called when a notify message, passed through parameter `message`, for this plugin was received from the core module.

process_message(message) – this method is called when a message different from the notify message is received. This method allows custom implementation for custom messages send from core or notify plugins belonging to the same group as the zone plugin.

Notify Plugins

Notify plugins must communicate only via messages, as they are mostly independent of the core module and they just wait for notifications from remote servers. The only method that has to be implemented is `run()`, where the code is executed as a separated process.

Messages

Messages allow to communicate between different parts of synchronization daemon and plugins.

Notify plugins can send following messages:

NotifyRecordMsg – destination for this message is a zone plugin. This message may contain information about new records, removed records or just information if synchronization for particular zone should start.

NotifyNewZoneMsg – destination is the core module, which decides if a new zone should be added to opposite side and a new zone module created.

NotifyRemovedZoneMsg – destination is the core module, which decides if it is allowed to remove zone.

As mentioned before, notification plugins are optional, so not all of these messages must be used.

5.3 Implementation Details

This section describes implementation details of the synchronization daemon and both BIND 9 and LDAP with FreeIPA schema plugins.

5.3.1 Implementation Details of the Synchronization Daemon

Configuration

The synchronization daemon is configured via configuration file. This file is in JSON format, consists of three parts: global daemon configuration, synchronization configuration and plugins configuration. For details please read appendix C.3 Configuration File.

Normalization of Records

All received records are derelativized to absolute domain name, before they can be compared via synchronization algorithm. This must be done for RDATA part of records where a domain name is used too.

DNSSEC

The DNSSEC records `RRSIG`, `DNSKEY`, `NSEC` and `NSEC3` are ignored by synchronization daemon. The approach where all DNS servers generate their own signatures for records is used.

Paralelization

Even tough the threads are sufficient for the synchronization daemon, the implementation of CPython threads [13] is using a global lock, which means that the particular threads are executed only one by one. To avoid this limitation, the implementation via processes was used to support parallel computing.

Management module

The current implementation of management module uses the D-BUS as the way to receive requests from outside. As is written on the official page [14], D-BUS is a message bus system, which allows different applications to communicate. Additional information about D-BUS is out of scope of this thesis and will not be discussed more.

The management module of the `dnssyncd` daemon is waiting for any request over D-BUS from the `dnssyncd-control` utility. The current implementation allows to control the daemon only as a root user.

5.3.2 Implementation Details of BIND 9 Plugin

This section explains how the required operations are implemented for BIND 9 DNS server in the plugin.

Getting Records

Records are downloaded from the DNS server using AXFR/IXFR mechanism, which was explained in sections 2.1.3 AXFR and 2.1.3 IXFR. The preferred mechanism is IXFR except, for the initial synchronization.

DNS data downloaded through IXFR is parsed by plugin, into result consists of lists of added records and removed records since the last download.

The plugin internally keeps the latest SOA serial to be able to get just the new changes next time, and is also responsible for marking data as incremental if IXFR were successful. In case the IXFR is not supported by the DNS server, the DNS will return AXFR answer and the plugin must detect this situation and properly mark data as non-incremental.

Updating Records

Records are updated on the DNS server using DDNS that was described in section 2.1.5 Dynamic Updates (DDNS).

One DDNS message is handled by the DNS server as one transaction, all or none records are updated.

Adding a New Zone

All new zones are added by plugin into BIND 9 server via RNDC utility, mentioned in section 2.1.6 RNDC utility.

The following template is used to create a new zone via RNDC:

```
rndc addzone <zone name> '{
    type master;
    file <path to~zone file>;
    allow-transfer { 127.0.0.1; ::1; };
    allow-update { 127.0.0.1; ::1; };
    also-notify { 127.0.0.1 port <port>; ::1 port <port>; };
};'
```

Because the zone file must be stored locally, the current implementation does not allow to use remote DNS server and synchronization daemon must run on the same machine as the DNS server. This file was created with plugin before the `rndc` command was executed. The zone files contains records that has been downloaded from opposite synchronization side, as the zone is new and no record conflicts are possible.

The template shown above also shows settings for update, zone transfer and notifications, which are needed for proper functionality of the DNS BIND 9 synchronization plugin.

To make the new zone active, the `rndc reconfig` command must be executed after the zone was added.

Removing a Zone

As the adding of a new zone, the removing a zone is also done by RNDC utility.

The following template shows how zone can be removed using RDNC:

```
rndc delzone <zone name>
```

Also the `rndc reconfig` command must be executed after that to make the new zone configuration active on the server.

Notifications

The notification plugin for BIND 9 is permanently waiting at localhost for both UDP and TCP notify messages sent by server.

If a notification is received, the SOA serial of the particular zone from notify messages is compared with the last SOA serial known by the zone plugin, and if the received SOA is newer then the synchronization for this zone should start. The received notification message from the BIND 9 server can contain a list of changed records, but as RFC 1996 [38, section 3.7] defines, these hints are unsecured and thus the list of records is ignored by plugin.

Detection of New Zones

Current implementation does not support this feature.

5.3.3 Implementation Details of LDAP Plugin

Only the FreeIPA schema is currently supported by current plugin implementation. The DNS related parts of the FreeIPA schema are listed in the appendix D FreeIPA LDAP Schema.

The standard LDAP operations, `ldapadd`, `ldapmod` and `ldapdelete` allows to implement all required operations except notifications.

The major issue here is mapping the DNS records to the DIT structure and vice versa. This means especially mapping SOA record into several attributes in zone object, creating record with proper TTL value, adding RDATA of records to proper attributes per record type.

For the LDAP entries where TTL is not specified the default TTL value, `8400` is used. This TTL value can be modified in the settings file.

The FreeIPA schema stores zone apex records in the zone LDAP entry. This must be handled in a special way because other records belong to the subtree of the zone entry.

Instead of the DNS plugin, the LDAP plugin does not need to have an LDAP server on the same machine as the dnssyncd daemon is running, all operation can be done remotely.

Notifications

The syncrepl in refreshAndPersist mode is used as the notification mechanism.

The data received via syncrepl is not used for synchronization. The data is just used to decide if synchronization is needed, and if yes then standard synchronization cycle start.

For the removed entries, the syncrepl returns only uuid, which is the unique identifier for every entry. The notification plugin must store uuid and belongings owner and zone names, to be able to distinguish which record or zone was removed. After the first connection, the LDAP server returns via syncrepl all entries with uuid in a particular subtree, which are used to initialize plugin database of uuids.

As mentioned before, the syncrepl returns whole entry, which in case of FreeIPA means the several records are returned. In this case the data cache is used to determine which records were changed.

Detection of New Zones

Detection of new zones is solved in two ways. The first is the notification plugin, which via the syncrepl returns new zone entries. The second is a standard LDAP search done periodically.

This approach is also used for detection of removed zones.

5.4 System Requirements

This section describes the requirements to be able to run the synchronization daemon.

5.4.1 Synchronization Daemon

Requirements of the synchronization daemon:

- the daemon must be executed under the root user,
- it requires `python 2.7` interpreter (It is not compatible with `python 3.x` due to used libraries.),
- required packages: `python-ldap`[1], `python-dns`[2], and `python-dbus`[3].

5.4.2 BIND 9

This section contains requirements of the BIND 9 DNS server to be able to work with the dnssyncd daemon. The following information is applicable for BIND version 9.9.

General Settings

To be able to add new zones, the following configuration must be added into the BIND 9 configuration file (usual path `/etc/named.conf`).

```
options {
    allow-new-zone yes;
};
```

Also, to support creating a new zone dynamically, the permissions of the BIND 9 working directory must be changed.

```
chmod g+w /var/named/
```

A disabled IXFR zone transfer can be enabled by:

[1] http://www.python-ldap.org/
[2] http://www.dnspython.org/, currently (may 19th 2015) is master branch required
[3] http://www.freedesktop.org/wiki/Software/dbus/

```
  options {
    provide-ixfr yes;
  };
```

Setting Up `rndc` Utility

The `rndc` utility and BIND 9 must be properly configured with shared secret key. This configuration can be done by executing the following command line utility:

```
  sudo rndc-confgen -a
```

Setting Up Zones Configurations

As previously mentioned, the zones must contain following configuration lines that allows them to be synchronized by the synchronization daemon:

```
zone "example.com." IN {
  allow-transfer { 127.0.0.1; ::1; };
  allow-update { 127.0.0.1; ::1; };
  also-notify { 127.0.0.1 port <port>; ::1 port <port>;
};
```

In case the zones are configured manually by the zone files, the following line will enable IXFR from files:

```
zone "example.com." IN {
  ixfr-from-differences yes;
}:
```

5.4.3 389 Directory Server

This section contains requirements of the 389 Directory Server required by the synchronization daemon. These settings are applicable for 389 Directory Server version 1.3.3.

5.4.4 Basic Settings

Basic settings requires:

- To have the 389 Directory Server installed.
- Created a user, whose credential will be used by the synchronization daemon to communicate with the LDAP database.
- Create the new container where DNS zones and records will be stored.
- Verify the user is allowed to read, write and modify entries in the container.

This configuration is only directory server related and it is out of scope of this work. Details can be found in the official documentation.

Adding FreeIPA Schema

The FreeIPA schema must be added into the directory server. This can be done by copying the attribute types and object classes listed in appendix D FreeIPA LDAP Schema into the directory with schemas (usually `/etc/dirsrv/schema/`) and restarting the LDAP server.

Syncrepl Configuration

Syncrepl on the 389 Directory Server can be enabled by the following modification of the server configuration:

```
# ldapmodify ...
  dn: cn=Content Synchronization,cn=plugins,cn=config
  changetype: modify
  replace: nsslapd-pluginEnabled
  nsslapd-pluginEnabled: on
  -

  dn: cn=Retro Changelog Plugin,cn=plugins,cn=config
  changetype: modify
  replace:  nsslapd-pluginEnabled
  nsslapd-pluginEnabled: on
  -
  replace nsslapd-attribute
  nsslapd-attribute: nsuniqueid:targetUniqueId
  -
  replace nsslapd-changelogmaxage
  nsslapd-changelogmaxage: 2d
  -
```

5.5 Limitations of the Current Implementation

As this proof of concept of the synchronization daemon runs only at localhost, some negative effects caused by a computer network are suppressed. This covers: almost no delay caused by network connection, no loss of a notification message, no slowdown of data transfer caused by a slow network connection.

The synchronization speed is negatively affected by high level programming language, which is slow by itself. The computation and synchronization itself has not been optimized and may cause a serious delay. As having better readability and less lines of code is better for the proof of concept, some optimization and rewriting to low level code for critical parts may be needed for production implementation to achieve speed needs.

The synchronization daemon implements support only for `IN` zone class, all records without the specified class are by default considered as belonging into the `IN` class. Records with other classes are ignored.

Case when one side sent data in incremental format and the second side sent all data is not supported because of the internal data cache, which returns incremental update from all data, so this case never happen in the implementation. So, the synchronization algorithm is implemented only for the cases when both sides sent incremental updates or both sides send all records.

5.6 Evaluation of the Synchronization Daemon Implementation

As the synchronization daemon it the universal solution for several types of DNS and LDAP servers, the opposite approach is to have specialized solution for specific servers. As the example of specialized solution the `bind-dyndb-ldap` plugin was chosen. Detailed information and compison follows in the following subsection.

Also, the synchronization daemon can be improved as discussed later in this section.

5.6.1 Comparation with `bind-dyndb-ldap` Plugin

The `bind-dyndb-ldap`, as is written on the official page [8], is a plugin for the BIND 9 server that allows to read and write DNS data to an LDAP backend. Currently, this plugin is used in the FreeIPA project to synchronize records between an LDAP server and the BIND 9 server.

As the `bind-dyndb-ldap` uses internal BIND 9 API, it has higher possibilities to control the DNS server than `dnssyncd`. The `bind-dyndb-ldap` plugin has instant access to the record database on the DNS server, it does not need to keep own cache. The `bind-dyndb-ldap` plugin knows about each new record added on the DNS server immediately, but the `dnssyncd` daemon must rely on the notification mechanism.

The `bind-dyndb-ldap` plugin also allows to directly configure the settings for zones and BIND 9 settings, which BIND 9 internal API allows. It also means forwarders and forwarding policy, policies for dynamic updates, transfers and DNS queries can be configured using the LDAP. Also, the `bind-dyndb-ldap` plugin supports the forward zones synchronization. The synchronization daemon can not achieve this functionality without an available public API on the BIND 9 DNS server. However, the `bind-dyndb-ldap` plugin must adapt to changes in the internal BIND 9 API, which can be changed rapidly, instead of the standardized methods and the public API, which the synchronization daemon uses.

Also as the `bind-dyndb-ldap` plugin is running in the BIND 9 namespace, the error in that plugin can cause termination of the DNS server. In the multi-master LDAP topology, error caused by a LDAP entry may cause the failure of all DNS servers which are synchronized with LDAP database. In case the dnssyncd daemon, the daemon is separate process, and termination of the synchronization daemon due an error, does not affect the DNS servers.

On the other hand, the `bind-dyndb-ldap` plugin is tightly bounded with the BIND 9, which prevents it to be used for other DNS servers. The synchronization daemon, with plugin interface, allows to extend functionality for several types of DNS servers. Respectively, is ready to be used in only records synchronization mode with standardized operations with any DNS server.

5.6.2 Possible Future Improvements

To allow work with BIND 9 remotely, the SSH transport of zone file templates for zones that will be added can be implemented. This will allow the new zone to be added by the calling rndc utility remotely, which is not possible now because rndc requires a zone file on a local file system for new zones.

Also, there is a possible solution, how to get list of all zones from BIND 9 server, but it is not preferred. The rndc utility allows to export BIND 9 records database into a file. This file can be parsed and a list of zones will be extracted. Another solution can be to write a plugin for the BIND 9 server that can provide a list of zones, through D-BUS interface for example.

The current implementation uses the python programming language, which is interpreted and slow. There is a possibility to rewrite the performance critical parts of the daemon into python C-extension modules, where the code is compiled and much faster.

The currently supported method to bind into LDAP database is only the simple bind (username and password). The LDAP plugin can be extended with new authentication methods, such as kerberos authentication.

It is possible to implement the DNSSEC zone signing in the synchronization daemon to have zones signed on the fly for DNS servers that do not allow to sign zones dynamically.

Security

As the proof of concept, the synchronization daemon does implement only the basic security, which is not enough to be used in production environment.

For any LDAP connection TLS/SSL or SASL layer should be used. Also an access control should be applied for the synchronization daemon to be able to read and modify only the DNS subtree of an LDAP database.

The DNS dynamic updates, zone transfers and notify messages are currently configured with simple IP address filtration, which is not enough. To improve security, the TSIG mechanism specified in RFC 2845 [28] can be used. TSIG allows authentication using shared secrets and one way hashing. TSIG is out of scope of this thesis and will not be explained in more detail.

Chapter 6

Summary

The objective of this study is to analyze various schemas how to store DNS data in LDAP, provide solution to synchronizing records and create a proof-of-concept implementation of the synchronization daemon that synchronizes the DNS records between a DNS server and an LDAP database. The current proof-of-concept is implemented for the BIND 9 DNS server and the 389 Directory Server LDAP database with FreeIPA schema. This schema was compared in text with several other schemas, where each schema implements a different model of how to storage DNS data.

This daemon was designed as a multiprocess application that allows to synchronize DNS zones independently and in parallel. For each zone there is a local cache containing belonging records kept in each process. The records help to resolve conflicts of DNS records that could appear. Several types of conflicts and synchronization issues that must be solved by synchronization are described in the text.

The synchronization daemon, as was defined in goals, implements an interface that allows to add new plugins to extend the ability to synchronize from/to different types of DNS servers, LDAP databases using various schemas, or even a database different than LDAP. As was defined in second goal, the synchronization daemon also uses the standardized mechanism, for example, zone transfers, notification messages and dynamic DNS updates, which should be supported by all DNS servers, and thus implementation of new plugins for different server is simplified.

The result of this work is the working application, the synchronization daemon, which synchronize the DNS records between the LDAP database and DNS server, using the standardized methods and allowing easily extend functionality for more types of LDAP and DNS servers and LDAP schemas, using the defined interface for the synchronization plugins.

Bibliography

[1] DLZ LDAP schema. [Online; cit. 2014-11-10].
URL http://bind-dlz.sourceforge.net/ldap_example.html

[2] FreeIPA: About FreeIPA. [Online; cit. 2014-11-05].
URL http://www.freeipa.org/page/About#What_is_FreeIPA.3F

[3] GIT repostiry of bind-dyndb-ldap - FreeIPA DNS schema. [Online; cit. 2014-12-10].
URL
https://git.fedorahosted.org/cgit/bind-dyndb-ldap.git/tree/doc/schema

[4] How to use dnsZone with the BIND 9 sdb back-end. [Online; cit. 2015-03-09].
URL http://bind9-ldap.bayour.com/dnszonehowto.html

[5] LDAP for Rocket Scientists. [Online; cit. 2014-10-06].
URL http://www.zytrax.com/books/ldap/

[6] *NSUPDATE manpage*. [Online; cit. 2014-11-07].
URL http://www.freebsd.org/cgi/man.cgi?query=nsupdate&sektion=1

[7] Red Hat Directory Server 8 - Managing Indexes. [Online; cit. 2014-11-12].
URL https://access.redhat.com/site/documentation/en-US/Red_Hat_Directory_Server/8.2/html/Administration_Guide/Managing_Indexes.html

[8] Bind DynDB LDAP. December 2014, [Online; cit. 2015-05-20].
URL https://fedorahosted.org/bind-dyndb-ldap/

[9] Andrews, M.: Negative Caching of DNS Queries (DNS NCACHE). March 1998, [Online; cit. 2015-04-05].
URL http://tools.ietf.org/html/rfc2308

[10] B. Laurie, R. A., G. Sisson: RFC 5155: DNS Security (DNSSEC) Hashed Authenticated Denial of Existence. March 2008, [Online; cit. 2015-05-07].
URL https://tools.ietf.org/html/rfc5155

[11] Eastlake, D.: RFC 4343: Domain Name System (DNS) Case Insensitivity Clarification. January 2006, [Online; cit. 2015-02-02].
URL http://tools.ietf.org/html/rfc4343

[12] Foundation, O.: OpenLDAP Software 2.4 Administrator's Guide. September 2014, [Online; cit. 2015-01-12].
URL http://www.openldap.org/doc/admin24/access-control.html

[13] Foundation, P. S.: The Python Standard Library: threading – Higher-level threading interface. May 2015, [Online; cit. 2015-05-18].
URL https://docs.python.org/2/library/threading.html

[14] freedesktop.org: D-BUS: What is D-Bus? Jan 2014, [Online; cit. 2015-05-06].
URL http://www.freedesktop.org/wiki/Software/dbus/

[15] Good, G.: RFC 2849: The LDAP Data Interchange Format (LDIF) - Technical Specification. June 2000, [Online; cit. 2014-11-12].
URL http://tools.ietf.org/html/rfc2849

[16] ISC: BIND. [Online; cit. 2014-10-07].
URL http://www.isc.org/downloads/bind/

[17] ISC: *BIND 9 Administrator Reference Manual*. [Online; cit. 2015-03-06].
URL http://www.bind9.net/arm99.pdf

[18] K. Zeilenga, J. C.: RFC 4533: The Lightweight Directory Access Protocol (LDAP) Content Synchronization Operation. June 2006, [Online; cit. 2015-03-02].
URL http://tools.ietf.org/html/rfc4533

[19] Klensin, J.: RFC 5891: Internationalized Domain Names in Applications (IDNA): Protocol. August 2010, [Online; cit. 2015-02-10].
URL http://tools.ietf.org/html/rfc5891

[20] Lewis, E.; Hoenes, A.: RFC 5936: DNS Zone Transfer Protocol (AXFR). June 2010, [Online; cit. 2014-10-10].
URL http://tools.ietf.org/html/rfc5936

[21] M. Andrews, S. W.: RFC 4431: The DNSSEC Lookaside Validation (DLV) DNS Resource Record. February 2006, [Online; cit. 2015-05-07].
URL https://tools.ietf.org/html/rfc4431

[22] Microsoft: Protocols and Interfaces to Active Directory. [Online; cit. 2014-11-06].
URL http://technet.microsoft.com/en-us/library/cc961766.aspx

[23] Mockapetris, P.: RFC 1034: DOMAIN NAMES - CONCEPTS AND FACILITIES. November 1987, [Online; cit. 2014-09-12].
URL http://tools.ietf.org/html/rfc1034

[24] Mockapetris, P.: RFC 1035: DOMAIN NAMES - IMPLEMENTATION AND SPECIFICATION. November 1987, [Online; cit. 2014-09-12].
URL http://tools.ietf.org/html/rfc1035

[25] Ohta, M.: RFC 1995: Incremental Zone Transfer in DNS. August 1996, [Online; cit. 2014-10-10].
URL http://tools.ietf.org/search/rfc1995

[26] Oracle: *Oracle8i Integration Server Overview*. [Online; cit. 2014-11-12].
URL http://docs.oracle.com/cd/A87860_01/doc/ois.817/a83729/adois09.htm

[27] P. Barker, S. K.: The COSINE and Internet X.500 Schema. November 1991, [Online; cit. 2015-03-05].
URL http://tools.ietf.org/html/rfc1274

[28] P. Vixie, D. E. r., O. Gudmundsson: RFC 2845: Secret Key Transaction Authentication for DNS (TSIG). May 2000, [Online; cit. 2015-05-20].
URL http://tools.ietf.org/html/rfc2845

[29] R. Arends, M. L., R. Austein: RFC 4033: DNS Security Introduction and Requirements. March 2005, [Online; cit. 2015-05-07].
URL https://tools.ietf.org/html/rfc4033

[30] R. Arends, M. L., R. Austein: RFC 4034: Resource Records for the DNS Security Extensions. March 2005, [Online; cit. 2015-05-07].
URL https://tools.ietf.org/html/rfc4034

[31] R. Arends, M. L., R. Austein: RFC 4035: Protocol Modifications for the DNS Security Extensions. March 2005, [Online; cit. 2015-05-06].
URL https://tools.ietf.org/html/rfc4035

[32] R. Elz, R. B.: RFC 1982: Serial Number Arithmetic. August 1996, [Online; cit. 2015-02-01].
URL http://tools.ietf.org/search/rfc1982

[33] R. Elz, R. B.: RFC 2181: Clarifications to the DNS Specification. July 1997, [Online; cit. 2015-03-10].
URL http://tools.ietf.org/search/rfc2181

[34] S. Rose, W. W.: RFC 6672: DNAME Redirection in the DNS. June 2012, [Online; cit. 2015-05-06].
URL https://tools.ietf.org/html/rfc6672

[35] Sermersheim, J.: RFC 4511: Lightweight Directory Access Protocol (LDAP): The Protocol. June 2006, [Online; cit. 2014-11-12].
URL http://tools.ietf.org/html/rfc4511

[36] python-ldap project team: python-ldap Documentation. September 2014, [Online; cit. 2015-01-12].
URL http://www.python-ldap.org/doc/html/ldap.html#ldap.LDAPObject.search_ext_s

[37] Thomson, S.; Rekhter, Y.; Bound, J.: RFC 2136: Dynamic Updates in the Domain Name System (DNS UPDATE). April 1997, [Online; cit. 2014-10-12].
URL http://tools.ietf.org/html/rfc2136

[38] Vixie, P.: RFC 1996: A Mechanism for Prompt Notification of Zone Changes (DNS NOTIFY). August 1996, [Online; cit. 2015-10-09].
URL http://tools.ietf.org/html/rfc1996

[39] Zeilenga, K.: RFC 4512: Lightweight Directory Access Protocol (LDAP): Directory Information Models. June 2006, [Online; cit. 2014-11-12].
URL http://tools.ietf.org/html/rfc4512

[40] Zeilenga, K.: RFC 4524: COSINE LDAP/X.500 Schema. June 2006, [Online; cit. 2015-02-10].
URL https://tools.ietf.org/html/rfc4524

Appendix A

Content of DVD

The attached DVD contains:

- Electronic version of this thesis.
- Source code of the thesis (text).
- Source code of the synchronization daemon.
- FreeIPA DNS schema LDIF file.
- Virtual image of the test environment with preinstalled and preconfigured LDAP database, DNS server and the synchronization daemon.
- README file that contains basic information.

Appendix B

States Diagram of the Synchronization Daemon

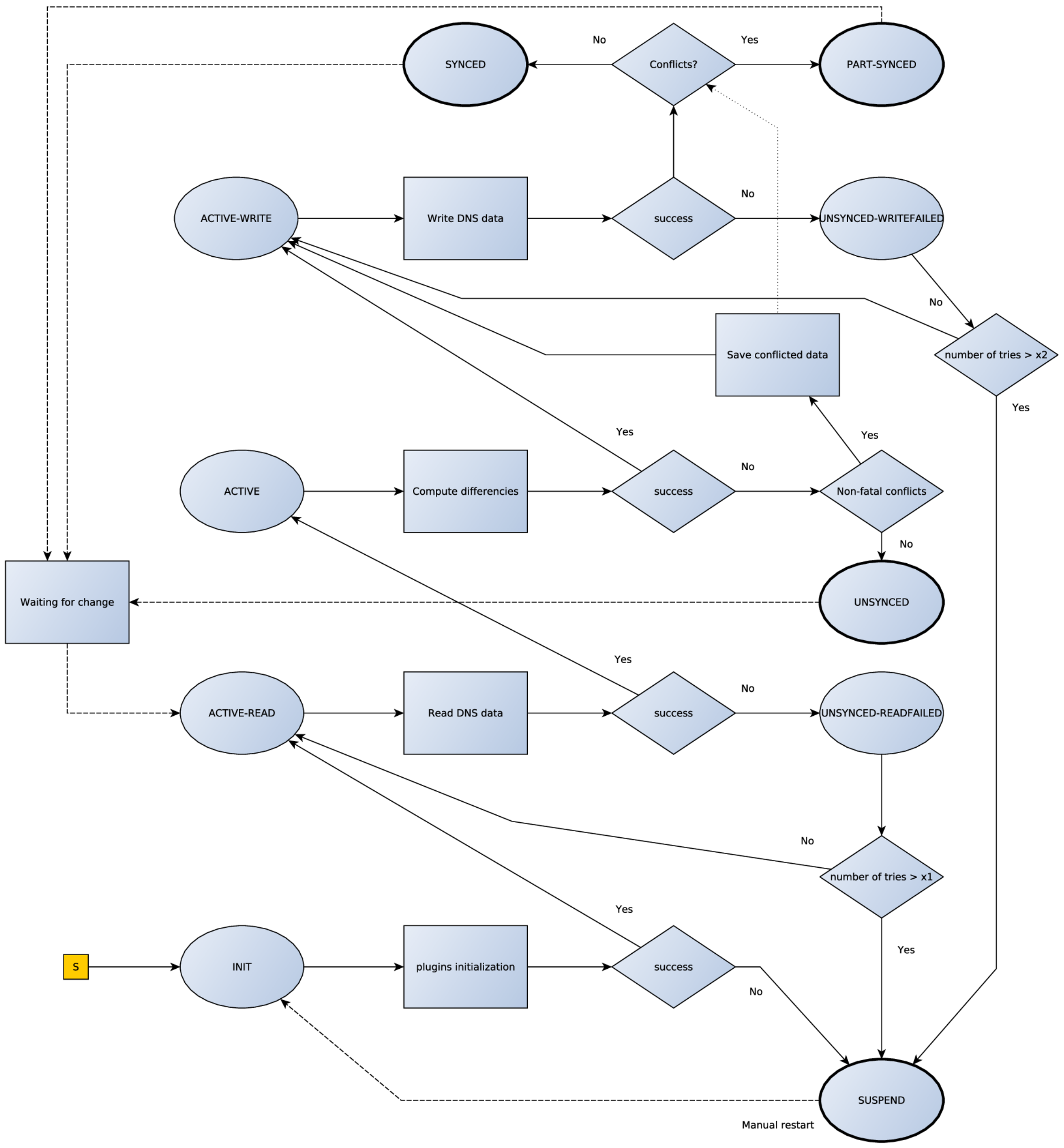

Figure B.1: Synchronization states diagram

Appendix C

User Manual

C.1 Installation

To install the synchronization daemon on the Fedora 21 system, please use the `dnssyncd-install-f21.sh` shell script.

For other distributions please read instructions inside the installation script.

C.2 Usage

C.2.1 `dnssyncd`

To run synchronization daemon, please execute command `dnssyncd`, which will run the daemon in foreground. Daemon requires the configuration file specified in `/etc/dnssyncd.conf`.

C.2.2 `dnssyncd-control`

To run control utility of the synchronization daemon, please execute command `dnssyncd-control`. The built-in help can be shown by executing `dnssyncd-control --help`.

C.3 Configuration File

The configuration file is expected in JSON format. The following example shows the default configuration file.

```
{
  "global":{
    "control_enabled": true,
    "notify_plugins_enabled": true
  },

  "synchronization":{
    "periodical_sync": 300,
    "zones_to_sync":["zone.test."],
    "zones_source": "ldap",
    "ignore_ttl": false,
    "authoritative": "ldap",
    "plugins_conf":{
      "bind":{
        "plugin": "bind9",
        "dns_server_ip":"127.0.0.1",
        "dns_server_port":53,
        "listen_on_notify_port":29458
      },
      "ldap":{
        "plugin": "ldap-ipa",
        "ldap_server_address":"ldap://server.example.com",
        "ldap_server_port":389,
        "ldap_user_dn":"uid=admin,cn=users,cn=accounts,dc=example,dc=com",
        "ldap_password":"Secret123",
        "ldap_base_dn":"cn=dns,dc=example,dc=com",
        "ttl_sync_method": "latest_lowest"
      }
    }
  }
}
```

C.3.1 Global Configuration Options

control_enabled : true|false (default: true), enables the management module to allows comunicate with the `dnssyncd-control` utility.

notify_plugins_enabled : true|false (default: true), enables the notifications plugins

Synchronization Options

periodical_sync : time period in seconds (default: 300 sec), specifies the period time when new synchronization cycle should be executed

zones_to_sync : list of zones, specifies the zones which will be synchronized

zones_source : configuration name, specifies the source of new zones

ignore_ttl : true|false (default: false), specifies if the TTL values should be ignored during detection of changes

authoritative : configuration name, specifies which remote side is authoritative

plugins_conf : configuration of plugins in dictionary „¡configuration name¿“: plugin options, contains the configuration for plugins

389 DS with IPA Schema Plugin

plugin : name of the plugin

ldap_server_address : ldap URI

ldap_server_port : port number (default: 389)

ldap_user_dn : user DN to be used for connection

ldap_password : password to be used for connection

ldap_base_dn : DN of the subtree where the DNS data are located

ttl_sync_method : method how TTL values will be mapped into LDAP owner entry

BIND 9 Plugin

plugin : name of the plugin

dns_server_ip : ip address of the DNS server, please note the current implementation allows to manage zones only at localhost

dns_server_port : port number, the port for DNS queries, AXFR/IXFR transfers

listen_on_notify_port : port number, the port where notifications from the DNS server will be sent

Appendix D

FreeIPA LDAP Schema

Attributes and object classes related to DNSSEC keys were omitted.

```
## IPA Base OID:      2.16.840.1.113730.3.8
##
## Attributes:        2.16.840.1.113730.3.8.5 - V2 DNS related attributres
## ObjectClasses:     2.16.840.1.113730.3.8.6 - V2 DNS related objectclasses
##
dn: cn=schema
attributeTypes: (1.3.6.1.4.1.2428.20.0.0 NAME 'dNSTTL' DESC 'An integer denoting
    time to live' EQUALITY integerMatch SYNTAX 1.3.6.1.4.1.1466.115.121.1.27 )
attributeTypes: (1.3.6.1.4.1.2428.20.0.1 NAME 'dNSClass' DESC 'The class of a
    resource record' EQUALITY caseIgnoreIA5Match SYNTAX
    1.3.6.1.4.1.1466.115.121.1.26 )
attributeTypes: (1.3.6.1.4.1.2428.20.1.12 NAME 'pTRRecord' DESC 'domain name
    pointer, RFC 1035' EQUALITY caseIgnoreIA5Match SUBSTR
    caseIgnoreIA5SubstringsMatch SYNTAX 1.3.6.1.4.1.1466.115.121.1.26 )
attributeTypes: (1.3.6.1.4.1.2428.20.1.13 NAME 'hInfoRecord' DESC 'host
    information, RFC 1035' EQUALITY caseIgnoreIA5Match SUBSTR
    caseIgnoreIA5SubstringsMatch SYNTAX 1.3.6.1.4.1.1466.115.121.1.26 )
attributeTypes: (1.3.6.1.4.1.2428.20.1.14 NAME 'mInfoRecord' DESC 'mailbox or mail
     list information, RFC 1035' EQUALITY caseIgnoreIA5Match SUBSTR
    caseIgnoreIA5SubstringsMatch SYNTAX 1.3.6.1.4.1.1466.115.121.1.26 )
attributeTypes: (1.3.6.1.4.1.2428.20.1.16 NAME 'tXTRecord' DESC 'text string, RFC
    1035' EQUALITY caseIgnoreIA5Match SUBSTR caseIgnoreIA5SubstringsMatch SYNTAX
    1.3.6.1.4.1.1466.115.121.1.26 )
attributeTypes: (1.3.6.1.4.1.2428.20.1.18 NAME 'aFSDBRecord' DESC 'for AFS Data
    Base location, RFC 1183' EQUALITY caseIgnoreIA5Match SUBSTR
    caseIgnoreIA5SubstringsMatch SYNTAX 1.3.6.1.4.1.1466.115.121.1.26 )
attributeTypes: (1.3.6.1.4.1.2428.20.1.24 NAME 'SigRecord' DESC 'Signature, RFC
    2535' EQUALITY caseIgnoreIA5Match SUBSTR caseIgnoreIA5SubstringsMatch SYNTAX
    1.3.6.1.4.1.1466.115.121.1.26 )
attributeTypes: (1.3.6.1.4.1.2428.20.1.25 NAME 'KeyRecord' DESC 'Key, RFC 2535'
    EQUALITY caseIgnoreIA5Match SUBSTR caseIgnoreIA5SubstringsMatch SYNTAX
    1.3.6.1.4.1.1466.115.121.1.26 )
attributeTypes: (1.3.6.1.4.1.2428.20.1.28 NAME 'aAAARecord' DESC 'IPv6 address,
    RFC 1886' EQUALITY caseIgnoreIA5Match SUBSTR caseIgnoreIA5SubstringsMatch
    SYNTAX 1.3.6.1.4.1.1466.115.121.1.26 )
attributeTypes: (1.3.6.1.4.1.2428.20.1.29 NAME 'LocRecord' DESC 'Location, RFC
    1876' EQUALITY caseIgnoreIA5Match SUBSTR caseIgnoreIA5SubstringsMatch SYNTAX
    1.3.6.1.4.1.1466.115.121.1.26 )
```

```
attributeTypes: (1.3.6.1.4.1.2428.20.1.30 NAME 'nXTRecord' DESC 'non-existant, RFC
     2535' EQUALITY caseIgnoreIA5Match SUBSTR caseIgnoreIA5SubstringsMatch SYNTAX
    1.3.6.1.4.1.1466.115.121.1.26 )
attributeTypes: (1.3.6.1.4.1.2428.20.1.33 NAME 'sRVRecord' DESC 'service location,
     RFC 2782' EQUALITY caseIgnoreIA5Match SUBSTR caseIgnoreIA5SubstringsMatch
    SYNTAX 1.3.6.1.4.1.1466.115.121.1.26 )
attributeTypes: (1.3.6.1.4.1.2428.20.1.35 NAME 'nAPTRRecord' DESC 'Naming
    Authority Pointer, RFC 2915' EQUALITY caseIgnoreIA5Match SUBSTR
    caseIgnoreIA5SubstringsMatch SYNTAX 1.3.6.1.4.1.1466.115.121.1.26 )
attributeTypes: (1.3.6.1.4.1.2428.20.1.36 NAME 'kXRecord' DESC 'Key Exchange
    Delegation, RFC 2230' EQUALITY caseIgnoreIA5Match SUBSTR
    caseIgnoreIA5SubstringsMatch SYNTAX 1.3.6.1.4.1.1466.115.121.1.26 )
attributeTypes: (1.3.6.1.4.1.2428.20.1.37 NAME 'certRecord' DESC 'certificate, RFC
     2538' EQUALITY caseIgnoreIA5Match SUBSTR caseIgnoreIA5SubstringsMatch SYNTAX
    1.3.6.1.4.1.1466.115.121.1.26 )
attributeTypes: (1.3.6.1.4.1.2428.20.1.38 NAME 'a6Record' DESC 'A6 Record Type,
    RFC 2874' EQUALITY caseIgnoreIA5Match SUBSTR caseIgnoreIA5SubstringsMatch
    SYNTAX 1.3.6.1.4.1.1466.115.121.1.26 )
attributeTypes: (1.3.6.1.4.1.2428.20.1.39 NAME 'dNameRecord' DESC 'Non-Terminal
    DNS Name Redirection, RFC 2672' EQUALITY caseIgnoreIA5Match SUBSTR
    caseIgnoreIA5SubstringsMatch SYNTAX 1.3.6.1.4.1.1466.115.121.1.26 SINGLE-VALUE
     )
attributeTypes: (1.3.6.1.4.1.2428.20.1.43 NAME 'dSRecord' DESC 'Delegation Signer,
     RFC 3658' EQUALITY caseIgnoreIA5Match SUBSTR caseIgnoreIA5SubstringsMatch
    SYNTAX 1.3.6.1.4.1.1466.115.121.1.26 )
attributeTypes: (1.3.6.1.4.1.2428.20.1.44 NAME 'sSHFPRecord' DESC 'SSH Key
    Fingerprint, draft-ietf-secsh-dns-05.txt' EQUALITY caseIgnoreIA5Match SUBSTR
    caseIgnoreIA5SubstringsMatch SYNTAX 1.3.6.1.4.1.1466.115.121.1.26 )
attributeTypes: (1.3.6.1.4.1.2428.20.1.46 NAME 'rRSIGRecord' DESC 'RRSIG, RFC
    3755' EQUALITY caseIgnoreIA5Match SUBSTR caseIgnoreIA5SubstringsMatch SYNTAX
    1.3.6.1.4.1.1466.115.121.1.26 )
attributeTypes: (1.3.6.1.4.1.2428.20.1.47 NAME 'nSECRecord' DESC 'NSEC, RFC 3755'
    EQUALITY caseIgnoreIA5Match SUBSTR caseIgnoreIA5SubstringsMatch SYNTAX
    1.3.6.1.4.1.1466.115.121.1.26 )
attributeTypes: (1.3.6.1.4.1.2428.20.1.51 NAME 'nSEC3PARAMRecord' DESC 'RFC 5155'
    EQUALITY caseIgnoreIA5Match SUBSTR caseIgnoreIA5SubstringsMatch SYNTAX
    1.3.6.1.4.1.1466.115.121.1.26 SINGLE-VALUE )
attributeTypes: (1.3.6.1.4.1.2428.20.1.52 NAME 'TLSARecord' DESC 'DNS-Based
    Authentication of Named Entities - Transport Layer Security Protocol, RFC
    6698' EQUALITY caseIgnoreIA5Match SUBSTR caseIgnoreIA5SubstringsMatch SYNTAX
    1.3.6.1.4.1.1466.115.121.1.26 )
attributeTypes: (1.3.6.1.4.1.2428.20.1.32769 NAME 'DLVRecord' DESC 'DNSSEC
    Lookaside Validation, RFC 4431' EQUALITY caseIgnoreIA5Match SUBSTR
    caseIgnoreIA5SubstringsMatch SYNTAX 1.3.6.1.4.1.1466.115.121.1.26 )
attributeTypes: (0.9.2342.19200300.100.1.26 NAME 'aRecord' EQUALITY
    caseIgnoreIA5Match SYNTAX 1.3.6.1.4.1.1466.115.121.1.26 )
attributeTypes: (0.9.2342.19200300.100.1.29 NAME 'nSRecord' EQUALITY
    caseIgnoreIA5Match SYNTAX 1.3.6.1.4.1.1466.115.121.1.26 )
attributeTypes: (0.9.2342.19200300.100.1.31 NAME 'cNAMERecord' EQUALITY
    caseIgnoreIA5Match SYNTAX 1.3.6.1.4.1.1466.115.121.1.26 SINGLE-VALUE )
attributeTypes: (0.9.2342.19200300.100.1.28 NAME 'mXRecord' EQUALITY
    caseIgnoreIA5Match SYNTAX 1.3.6.1.4.1.1466.115.121.1.26 )
attributeTypes: (0.9.2342.19200300.100.1.27 NAME 'mDRecord' EQUALITY
    caseIgnoreIA5Match SYNTAX 1.3.6.1.4.1.1466.115.121.1.26 )
```

```
attributeTypes: (2.16.840.1.113730.3.8.5.0 NAME 'idnsName' DESC 'DNS FQDN'
    EQUALITY caseIgnoreIA5Match SUBSTR caseIgnoreIA5SubstringsMatch SYNTAX
    1.3.6.1.4.1.1466.115.121.1.26 SINGLE-VALUE X-ORIGIN 'IPA v2' )
attributeTypes: (2.16.840.1.113730.3.8.5.1 NAME 'idnsAllowDynUpdate' DESC 'permit
    dynamic updates on this zone' EQUALITY booleanMatch SYNTAX
    1.3.6.1.4.1.1466.115.121.1.7 SINGLE-VALUE X-ORIGIN 'IPA v2' )
attributeTypes: (2.16.840.1.113730.3.8.5.2 NAME 'idnsZoneActive' DESC 'define if
    the zone is considered in use' EQUALITY booleanMatch SYNTAX
    1.3.6.1.4.1.1466.115.121.1.7 SINGLE-VALUE X-ORIGIN 'IPA v2' )
attributeTypes: (2.16.840.1.113730.3.8.5.3 NAME 'idnsSOAmName' DESC 'SOA Name'
    EQUALITY caseIgnoreIA5Match SUBSTR caseIgnoreIA5SubstringsMatch SYNTAX
    1.3.6.1.4.1.1466.115.121.1.26 SINGLE-VALUE X-ORIGIN 'IPA v2' )
attributeTypes: (2.16.840.1.113730.3.8.5.4 NAME 'idnsSOArName' DESC 'SOA root Name
    ' EQUALITY caseIgnoreIA5Match SUBSTR caseIgnoreIA5SubstringsMatch SYNTAX
    1.3.6.1.4.1.1466.115.121.1.26 SINGLE-VALUE X-ORIGIN 'IPA v2' )
attributeTypes: (2.16.840.1.113730.3.8.5.5 NAME 'idnsSOAserial' DESC 'SOA serial
    number' EQUALITY caseIgnoreIA5Match SUBSTR caseIgnoreIA5SubstringsMatch SYNTAX
     1.3.6.1.4.1.1466.115.121.1.26 SINGLE-VALUE X-ORIGIN 'IPA v2' )
attributeTypes: (2.16.840.1.113730.3.8.5.6 NAME 'idnsSOArefresh' DESC 'SOA refresh
     value' EQUALITY caseIgnoreIA5Match SUBSTR caseIgnoreIA5SubstringsMatch SYNTAX
     1.3.6.1.4.1.1466.115.121.1.26 SINGLE-VALUE X-ORIGIN 'IPA v2' )
attributeTypes: (2.16.840.1.113730.3.8.5.7 NAME 'idnsSOAretry' DESC 'SOA retry
    value' EQUALITY caseIgnoreIA5Match SUBSTR caseIgnoreIA5SubstringsMatch SYNTAX
    1.3.6.1.4.1.1466.115.121.1.26 SINGLE-VALUE X-ORIGIN 'IPA v2' )
attributeTypes: (2.16.840.1.113730.3.8.5.8 NAME 'idnsSOAexpire' DESC 'SOA expire
    value' EQUALITY caseIgnoreIA5Match SUBSTR caseIgnoreIA5SubstringsMatch SYNTAX
    1.3.6.1.4.1.1466.115.121.1.26 SINGLE-VALUE X-ORIGIN 'IPA v2' )
attributeTypes: (2.16.840.1.113730.3.8.5.9 NAME 'idnsSOAminimum' DESC 'SOA minimum
     value' EQUALITY caseIgnoreIA5Match SUBSTR caseIgnoreIA5SubstringsMatch SYNTAX
     1.3.6.1.4.1.1466.115.121.1.26 SINGLE-VALUE X-ORIGIN 'IPA v2' )
attributeTypes: (2.16.840.1.113730.3.8.5.10 NAME 'idnsUpdatePolicy' DESC 'DNS
    dynamic updates policy' EQUALITY caseIgnoreIA5Match SUBSTR
    caseIgnoreIA5SubstringsMatch SYNTAX 1.3.6.1.4.1.1466.115.121.1.26 SINGLE-VALUE
     X-ORIGIN 'IPA v2' )
attributeTypes: (2.16.840.1.113730.3.8.5.11 NAME 'idnsAllowQuery' DESC 'BIND9
    allow-query ACL element' EQUALITY caseIgnoreIA5Match SYNTAX
    1.3.6.1.4.1.1466.115.121.1.26 SINGLE-VALUE X-ORIGIN 'IPA v2' )
attributeTypes: (2.16.840.1.113730.3.8.5.12 NAME 'idnsAllowTransfer' DESC 'BIND9
    allow-transfer ACL element' EQUALITY caseIgnoreIA5Match SYNTAX
    1.3.6.1.4.1.1466.115.121.1.26 SINGLE-VALUE X-ORIGIN 'IPA v2' )
attributeTypes: (2.16.840.1.113730.3.8.5.13 NAME 'idnsAllowSyncPTR' DESC 'permit
    synchronization of PTR records' EQUALITY booleanMatch SYNTAX
    1.3.6.1.4.1.1466.115.121.1.7 SINGLE-VALUE X-ORIGIN 'IPA v2' )
attributeTypes: (2.16.840.1.113730.3.8.5.14 NAME 'idnsForwardPolicy' DESC 'forward
     policy: only or first' EQUALITY caseIgnoreIA5Match SUBSTR
    caseIgnoreIA5SubstringsMatch SYNTAX 1.3.6.1.4.1.1466.115.121.1.26 SINGLE-VALUE
     X-ORIGIN 'IPA v2' )
attributeTypes: (2.16.840.1.113730.3.8.5.15 NAME 'idnsForwarders' DESC 'list of
    forwarders' EQUALITY caseIgnoreIA5Match SUBSTR caseIgnoreIA5SubstringsMatch
    SYNTAX 1.3.6.1.4.1.1466.115.121.1.26 X-ORIGIN 'IPA v2' )
attributeTypes: (2.16.840.1.113730.3.8.5.16 NAME 'idnsZoneRefresh' DESC 'zone
    refresh interval' EQUALITY integerMatch SYNTAX 1.3.6.1.4.1.1466.115.121.1.27
    SINGLE-VALUE X-ORIGIN 'IPA v2' )
```

```
attributeTypes: (2.16.840.1.113730.3.8.5.17 NAME 'idnsPersistentSearch' DESC '
    allow persistent searches' EQUALITY booleanMatch SYNTAX
    1.3.6.1.4.1.1466.115.121.1.7 SINGLE-VALUE X-ORIGIN 'IPA v2' )
attributeTypes: (2.16.840.1.113730.3.8.5.18 NAME 'idnsSecInlineSigning' DESC '
    allow inline DNSSEC signing' EQUALITY booleanMatch SYNTAX
    1.3.6.1.4.1.1466.115.121.1.7 SINGLE-VALUE X-ORIGIN 'IPA v4.0' )
objectClasses: (2.16.840.1.113730.3.8.6.0 NAME 'idnsRecord' DESC 'dns Record,
    usually a host' SUP top STRUCTURAL MUST idnsName MAY ( cn $ idnsAllowDynUpdate
     $ dNSTTL $ dNSClass $ aRecord $ aAAARecord $ a6Record $ nSRecord $
    cNAMERecord $ pTRRecord $ sRVRecord $ tXTRecord $ mXRecord $ mDRecord $
    hInfoRecord $ mInfoRecord $ aFSDBRecord $ SigRecord $ KeyRecord $ LocRecord $
    nXTRecord $ nAPTRRecord $ kXRecord $ certRecord $ dNameRecord $ dSRecord $
    sSHFPRecord $ rRSIGRecord $ nSECRecord $ DLVRecord $ TLSARecord ) )
objectClasses: (2.16.840.1.113730.3.8.6.1 NAME 'idnsZone' DESC 'Zone class' SUP
    idnsRecord STRUCTURAL MUST ( idnsZoneActive $ idnsSOAmName $ idnsSOArName $
    idnsSOAserial $ idnsSOArefresh $ idnsSOAretry $ idnsSOAexpire $ idnsSOAminimum
     ) MAY ( idnsUpdatePolicy $ idnsAllowQuery $ idnsAllowTransfer $
    idnsAllowSyncPTR $ idnsForwardPolicy $ idnsForwarders $ idnsSecInlineSigning $
     nSEC3PARAMRecord ) )
objectClasses: (2.16.840.1.113730.3.8.6.2 NAME 'idnsConfigObject' DESC 'DNS global
     config options' STRUCTURAL MAY ( idnsForwardPolicy $ idnsForwarders $
    idnsAllowSyncPTR $ idnsZoneRefresh $ idnsPersistentSearch ) )
objectClasses: (2.16.840.1.113730.3.8.12.18 NAME 'ipaDNSZone' SUP top AUXILIARY
    MUST idnsName MAY managedBy X-ORIGIN 'IPA v3' )
objectClasses: (2.16.840.1.113730.3.8.6.3 NAME 'idnsForwardZone' DESC 'Forward
    Zone class' SUP top STRUCTURAL MUST ( idnsName $ idnsZoneActive ) MAY (
    idnsForwarders $ idnsForwardPolicy ) )
```