BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

SINGLE D-BUS SERVER FOR SSSD

CENTRÁLNÍ D-BUS SERVER PRO SSSD

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **DUŠAN ÚRADNÍK**
AUTOR PRÁCE

SUPERVISOR **Ing. JIŘÍ PAVELA**
VEDOUCÍ PRÁCE

BRNO 2021

Brno University of Technology
Faculty of Information Technology

Department of Intelligent Systems (DITS) Academic year 2021/2022

Bachelor's Thesis Specification

Student: **Úradník Dušan**
Programme: Information Technology
Title: **Single D-Bus Server for SSSD**
Category: Software analysis and testing

Assignment:

1. Study SSSD, LDAP, D-Bus technologies, and the topic of topologies in distributed systems.
2. Get acquainted with the topic of software performance analysis and available tools for conducting such analysis.
3. Extend the SSSD service to use a more efficient topology based on a single D-Bus server.
4. Propose, design and implement new extensions of SSSD that leverage the new topology to improve the SSSD interface.
5. Design and conduct a series of performance tests of the new implementation. Focus on tests that best describe the performance changes in original topology and new extensions implemented in the previous step.
6. Discuss the achieved results with emphasis on notable performance gains and/or regressions and their cause. Propose additional optimizations of the system that focus on further mitigating the identified performance issues.

Recommended literature:

- SSSD documentation: https://sssd.io
- D-Bus specification: https://dbus.freedesktop.org/doc/dbus-specification.html
- LDAP reference materials: https://ldap.com/ldap-reference-materials/
- Gregg, B. (2020). Systems Performance, (2nd ed.). Pearson. ISBN: 9780136821694.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor: **Pavela Jiří, Ing.**
Consultant: Březina Pavel, Mgr., RedHatCZ
Head of Department: Hanáček Petr, doc. Dr. Ing.
Beginning of work: November 1, 2021
Submission deadline: July 29, 2022
Approval date: November 3, 2021

Abstract

This thesis aims to reimplement the current topology of SSSD's inter-process communication. This communication is managed through separate D-Bus message buses to which components connect and send messages. The star topology with a single D-Bus requires to create a central message bus for components to use without affecting the current performance of SSSD. To ensure that, a thorough performance analysis had to be done by measuring response times and monitoring SSSD's behavior under constant stream of requests. Therefore, the tools SystemTap and hyperfine were employed to assemble a performance test suite.

Abstrakt

Cieľom tejto práce je nahradiť aktuálnu topológiu, prostredníctvom ktorej komponenty nástroja SSSD komunikujú. Spomínaná komunikácia je manažovaná viacerými D-Bus zbernicami, ku ktorým sa komponenty pripajajú a posielajú cez ne správy. Požadovaná hviezdicová topológia s jednou D-Bus zbernicou vyžaduje vytvoriť centrálnu zbernicu, ktorú sa bude využívať všetkými komponentami, bez strát na výkone SSSD. Na zaručenie, že táto podmienka bude splnená, je potrebná dôkladná analýza výkonu skrz meranie doby odozvy a monitorovanie SSSD pri prívale stáleho prúdu požiadaviek. A tak bola vytvorená sada testov pomocou nástrojov SystemTap a hyperfine.

Keywords

SSSD, D-Bus, FreeIPA, Active Directory Domain Services, LDAP, performance, performance analysis, SystemTap, hyperfine, topology

Klíčová slova

SSSD, D-Bus, FreeIPA, Active Directory Domain Services, LDAP, výkon, alanýza výkonu, SystemTap, hyperfine, topológia

Reference

ÚRADNÍK, Dušan. *Single D-Bus Server for SSSD*. Brno, 2021. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Jiří Pavela

Rozšířený abstrakt

Cieľom tejto práce bolo navrhnúť a implementovať novú topológiu komunikácie skrz D-Bus zbernicu v rámci produktu SSSD. Na prezentovanie nových možností odomknutých po prechode z pôvodnej topológie na hviezdicovú topológiu sa taktiež práca venovala zmene D-Bus rozhraní. Aby sa mohli tieto modifikácie využiť aj v produkcii, bolo potrebné vytvoriť sadu výkonnostných testov a skontrolovať či nepôsobili na SSSD negatívne.

K pretvoreniu topológie bolo nutné analyzovať algoritmy využívané na tvorbu tej pôvodnej. Ich najväčšou súčasťou bola knižnica sbus, vytvorená tímom, ktorý stojí za vývojom SSSD, a ktorá slúži k jednoduchšej práci s D-Bus knižnicou. S jej pomocou sa podarilo zmeniť pôvodnú zostavu zberníc, kde jedna náležala vždy jednému poskytovateľovi dát, na konfiguráciu s jednou centrálnou zbernicou, zodpovednou za správu všetkej komunikácie. Keďže doteraz sa k príjemcom požiadavky rozdeľovali smerovaním na adresu konkrétnej zbernice, D-Bus signály v SSSD doteraz nemali využitie. Jedná a sa o správy bez konkrétneho príjemcu, ktoré príjmu len komponenty, ktoré sa na ich príjem explicitne prihlásia. Po zmene topológie ich však bolo možné začať využívať nakoľko ich odosielatelia mohli začať vysielať na centrálnu zbernicu kde sa o zvyšok smerovania postaral samotný D-Bus.

Na vykonanie analýzy výkonu boli zvolené programy SystemTap a hyperfine. Pomocou prvého z nich bola vytvorená inštrumetácia priamo v kóde samotného SSSD. Prostredníctvom tejto inštrumetácie bolo možné skrz skript písaný pre SystemTap zachytávať údaje o vnútornom stave jednotlivých komponentov. Toto bolo využité na zachytenie komunikácie medzi bodom, v ktorom procesy odosielateľov posúvajú požiadavku na zbernicu, a bodom, v ktorom prijímajú na ňu prijímajú odpoveď. Druhý vytváral na SSSD stálu záťaž a kontroloval priemerný čas vybavenia požiadavky dát užívateľa pomocou príkazu z príkazového riadku. Dáta boli týmito dvoma scenármi zbierané z obidvoch topológií.

Výsledky z testovania ukázali, že ako celok tieto zmeny neprniesli žiadny negatívny vplyv na výkon SSSD. V niektorých situáciách, ako spracovanie požiadavky odoslanej na LDAP server, sa dokonca preukázalo vo všetkých meraniach zrýchlenie. Nakoľko sa ukázalo, že je využitie týchto zmien možné aj v verejnej verzii SSSD odomyká to nové možnosti znižovania komplexnosti tohto produktu.

Single D-Bus Server for SSSD

Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of Mr. Ing. Jiří Pavela The supplementary information was provided by Mr. Mgr. Pavel Březina. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Dušan Úradník
July 28, 2022

Acknowledgements

I would like to thank my supervisor Mr. Ing. Jiří Pavela, for his guidance and willingness to help with any problem I encountered during the work on this thesis. Furthermore, I want to thank my external supervisor, Mr. Mgr. Pavel Březina, for giving me this opportunity to challenge myself. Additionally, I thank everyone who stood by me this past year for keeping me sane and focused.

Contents

Chapter 1

Introduction

System Security Services Daemon (SSSD) is a client component of a multi-product system used for *centralized identity management*. Its sister project, FreeIPA, together with Active Directory Domain Services (a tool for managing Windows environments) are SSSD's server-side components. Centralized identity management, in short, represents a system that simplifies the management of a large group of computers.

More precisely, SSSD is a tool created for the Linux environment (specifically Fedora), and it incorporates many of its valuable mechanisms as well. One of them, D-Bus, is in charge of the inter-process communication (IPC) between some of SSSD's components, such as back-ends (FreeIPA, AD DS) and responders (NSS, PAM, InfoPipe, ...). These components all live in their separate processes, and every time a message needs to be transmitted, a new D-Bus *message bus* is created.

How the components cooperate is described by a set of instructions, also called a topology. The current topology dictates that the communication between processes is scattered across multiple message buses, which is taxing on the system's resources. Since the original design of SSSD, better options to utilize D-Bus' capabilities have surfaced. The most significant change introduced an upgraded replacement for the original dbus-daemon: a reimplementation, named *dbus-broker*, that brought D-Bus better performance and greater stability.

Consequently, this thesis aims to reimplement the original behavior of SSSD in a way that adopts these new possibilities. Therefore the original form of communication needs to change to one represented by a star topology, thus to a one where all messages go through a single *message bus*. Furthermore, once this configuration is realized, it is possible to reimplement the interfaces so that they can utilize this new approach (transform the appropriate D-Bus methods to D-Bus signals).

The secondary goal is to inspect whether incorporating these changes would preserve, improve, or degrade SSSD's performance. For this purpose, a performance test suite will be created as well. This test suite will assess SSSD under expected and unexpected loads (sending requests to remote identity management servers) using both old and new topologies. Thus, after examining the results of these measurements, a decision on whether these changes are beneficial to SSSD or not can be made.

Outline. This thesis will first delve into SSSD's core features and identity management solutions compatible with SSSD. It will also explain the features of D-Bus and SSSD's wrapper sbus as well as performance testing of system tools. Furthermore, the focus will be on reimplementation of SSSD's topology and the subsequent performance analysis.

Chapter 2

SSSD and related technologies

SSSD is a client-side component that "*serves and caches the information stored in the remote directory server and provides identity, authentication, and authorization services to the host machine*" [2]. It manages this by connecting to a remote identity manager, retrieving the required data, and creating a local cache of it.

Like any other tool, SSSD can be divided into multiple logical sub-elements. These consist of back-ends, responders, client libraries, clients, monitor, and tools, as can be seen in the Figure 2.1. Each of them has a specific set of capabilities and responsibilities:

- **Back-ends** keep cache consistent and constantly valid. Checks, both automatic (timed) and triggered by events (user authentication), are in place to guarantee this behavior. Back-ends are the only components that can write into the cache as well as communicate with remote servers. This communication consists of requesting data from external sources on behalf of other SSSD components and receiving replies. These replies are then forwarded back to the originators of the requests and stored in a local cache for further use. The terms *back-end* and *SSSD domain* are interchangeable, and both describe an instance of an `sssd_be` process. Each back-end comprises modules (data providers) that implement specific functionalities, such as:
  - *id provider (users, groups, services),*
  - *authentication provider (user authentication),*
  - *access provider (user/host authorization),*
  - *sudo provider (sudo rules from remote servers),*
  - *and many more* [2].
- **Responders** mediate communication between the target application and the local cache, as well as between the target application and the back-end through SSSD's internal message buses. As stated before, each responder lives in its own process (`sssd_nss` for NSS[1], `sssd_pam` for PAM[2], `sssd_sudo` for sudo, ...). The processes start at the same time as all the other components, and after the initial setup, they idle and await requests from an application. When a request arrives, a responder processes it and looks into the cache. If the data is found and it is valid (it has not expired yet), it is returned by the responder. In contrast, whenever the data is either

[1] https://www.gnu.org/software/libc/manual/html_node/Name-Service-Switch.html
[2] http://www.linux-pam.org/

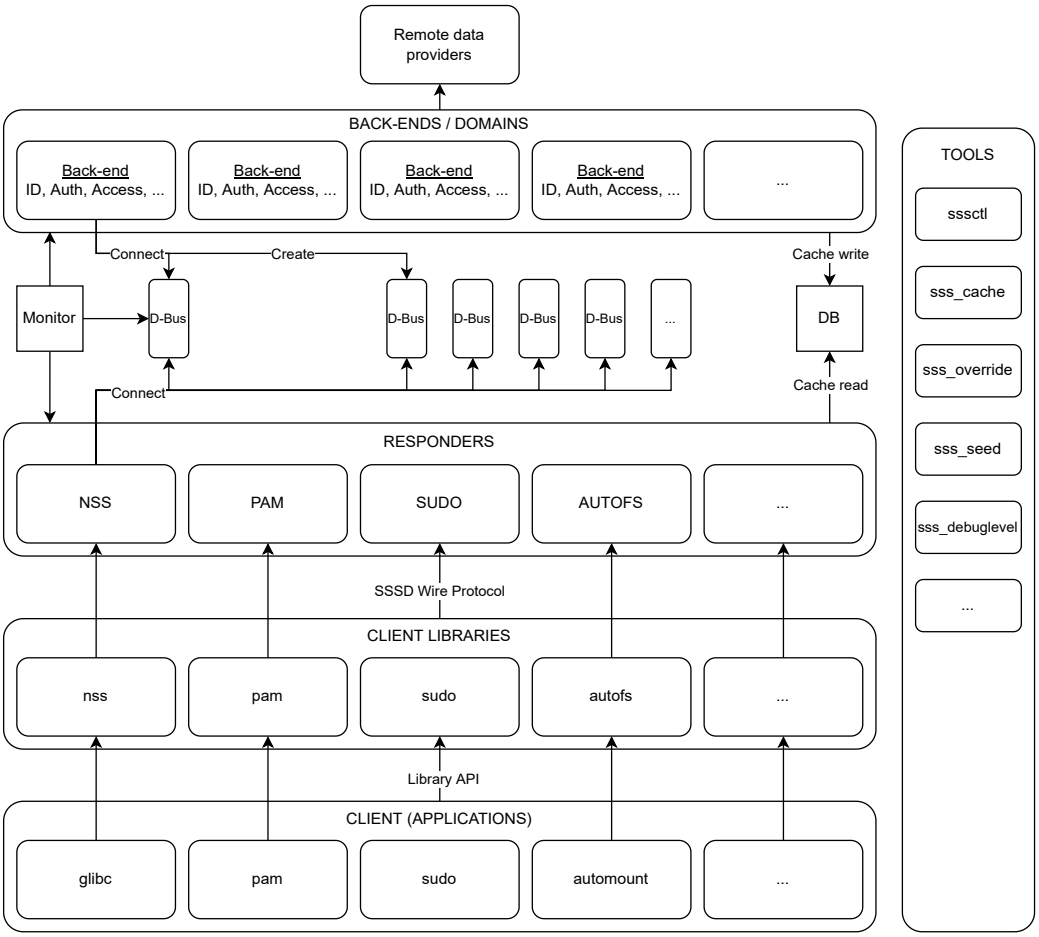

Figure 2.1: SSSD's components and the relations between them. The connections to the D-Bus server are depicted only for the monitor, the first back-end, and the first responder to make the diagram less cluttered. In reality, each responder connects to all message buses in the same way, and all back-ends connect to the monitor's D-Bus and create their own.

not found or invalid, the responder forwards the original request to the back-ends and awaits a response before replying.

- **Client libraries** implement interfaces through which the applications communicate with SSSD. They arrange communication between the applications and the responders.
- **Clients**, also known as applications (id, getent, su, sudo, and more), ask for various data from SSSD through the client libraries. Contrary to those, clients are not a part of SSSD, but their behavior is closely intertwined with it.
- **Monitor** is the primary process that serves as a supervisor over all back-ends and responders. It oversees their behavior, state and starts or stops them when needed (e.g., a new request comes, an error occurs).

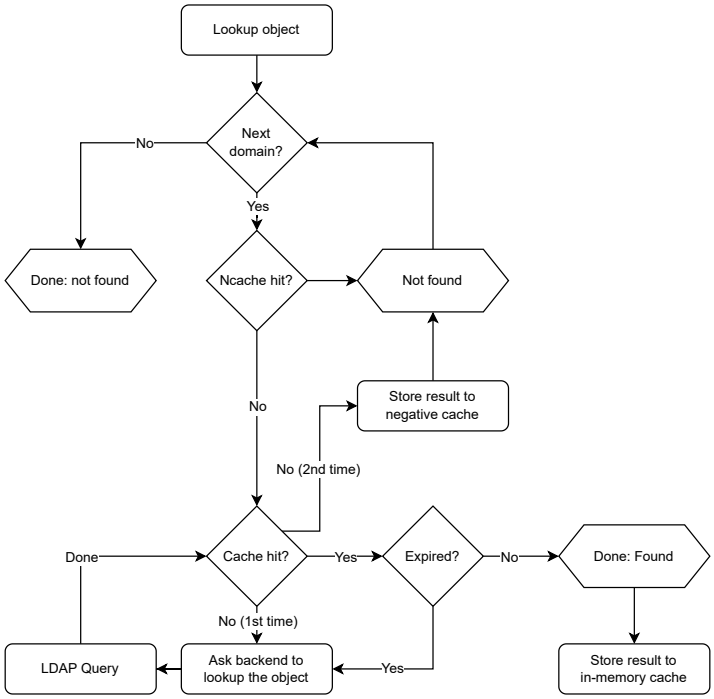

Figure 2.2: This diagram shows how SSSD utilizes its three types of cache: local, negative, and in-memory in its cache lookup process.

- **Tools** are mainly used to monitor SSSD and to manipulate some of it parts like cache, configuration, etc. The most notable mentions are:
  - `sssctl`[3] - used to retrieving SSSD diagnostics and debugging
  - `sss-cache`[4] - used for invalidating SSSD's cache records to initiate a data refresh
  - `sss-debuglevel`[5] - used for changing the debug level of components in a running SSSD instance

The main advantage of using SSSD over working with FreeIPA or AD DS directly is the local caching and offline authentication. Figure 2.2 depicts a simplified version of the cache lookup process. This process provides a more seamless user experience, access to already cached remote data offline and reduces the load on the identity and authentication providers. For example, when a user's connection fails, it does not mean all access to the remote files would be denied. Since the user's data is cached, "*as long as the user has logged in before, they will be able to log in with SSSD even when it is unable to communicate to the servers*" [2]. There are three different types of *cache* in SSSD each with their own purpose and features:

- **Local cache** (`cache`) – a persistent storage for all data currently in SSSD. The data is stored and managed by an LDAP-like embedded Unix tool called the `ldb`[6] database.

[3] https://jhrozek.fedorapeople.org/sssd/1.14.0/man/sssctl.8.html
[4] https://jhrozek.fedorapeople.org/sssd/1.14.0/man/sss_cache.8.html
[5] https://jhrozek.fedorapeople.org/sssd/1.14.0/man/sss_debuglevel.8.html
[6] https://ldb.samba.org/

Validation is achieved by storing all the data with an expiration time which is read and evaluated every time data is requested. Hence, when a responder finds an expired or invalid value, it sends a request to the back-end, which forwards it to a remote server. After receiving a reply, the back-end updates the original value and sets a new expiration time. In contrast, if the remote servers would not hold any information about the data anymore, the back-end would delete all traces regarding said data from the cache.

- **Negative cache** (`ncache`) – a non-persistent storage present in each responder. When a responder receives a reply from a back-end that some data no longer exists, it stores information about this fact to prevent unnecessary subsequent lookups.
- **In-memory cache** (`memcache`) – a "*volatile memory-mapped cache of selected objects (user, groups, initgroups)*" [2]. Responders store data objects in their cache after a successful lookup, and thus there is no need for a client library to ask a responder again for data requested mere moments ago.

SSSD implements its own *D-Bus*[7] wrapper `sbus` to create a convenient form of communication between back-ends and responders. `Sbus` is based on a reference C implementation protocol `libdbus` combined with `talloc` and `tevent` libraries. More in-depth information about D-Bus can be found in Section 2.2.

`Talloc` is a "*hierarchical, reference-counted memory pool system with destructors*" [13]. In short, `talloc` simplifies memory management as it provides means to arrange allocated space in an orientated graph (n-ary tree). This makes the handling of allocated memory more straightforward, as shown in the Listing 2.1. Since all allocated memory fields are hierarchically grouped under one root node `r`, deallocating all memory allocated within `r` becomes only a matter of calling the `talloc_free(r)` function on the root. In contrast, all `r`'s members would have to be deallocated one by one when not using `talloc`.

```
struct record {
    char *name;
    char *value;
};

struct record *r;

r = talloc_zero(NULL, struct record);
r->name = talloc_strdup(r, "Hello");
r->value = talloc_strdup(r, "World");

talloc_free(r);
```

Listing 2.1: Talloc allows recursive deallocation of resources by calling `talloc_free()` function on the root `struct`. This example was taken from SSSD's documentation.

The Listing 2.1 is just a simple demonstration of `talloc`'s abilities, but its potential is truly realized when working with complex nested data structures. In SSSD, where the

[7] https://www.freedesktop.org/wiki/Software/dbus/

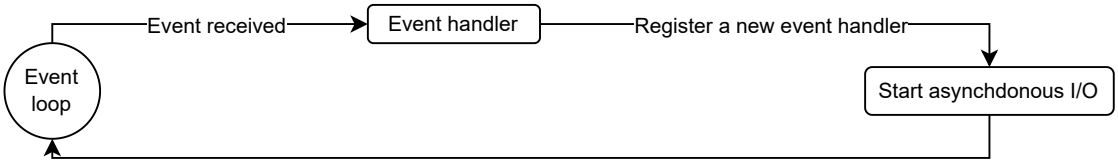

Figure 2.3: The talloc-based event loop spends most of its life cycle waiting for an event trigger. Upon receiving an event, the event loop starts an event handler and returns back to an idle state after the handler finishes. [2]

allocated memory can span through tens of layers of data structures, the option to free everything with one command significantly decreases the code's complexity.

`Talloc` is also a central element of the `tevent` library, an event system with "*support for many event types, including timers, signals, and the classic file descriptor events*" [14]. It brings a more coherent asynchronous programming style to SSSD by enabling parallel request handling through `talloc`-based *event loops*, depicted in Figure 2.3. This provides an easy way of sending, waiting for, and handling requests to remote servers and other processes, all while remaining in a single thread.

The `tevent` library supports a few different types of events: signals, file descriptor (fd) events, timer events, immediate events, and requests. Admittedly, it is the last type that is the most prevalent in the topic of this thesis, and SSSD in general. They form a callback-based API that provides a unified callback interface depicted in Figure 2.4.

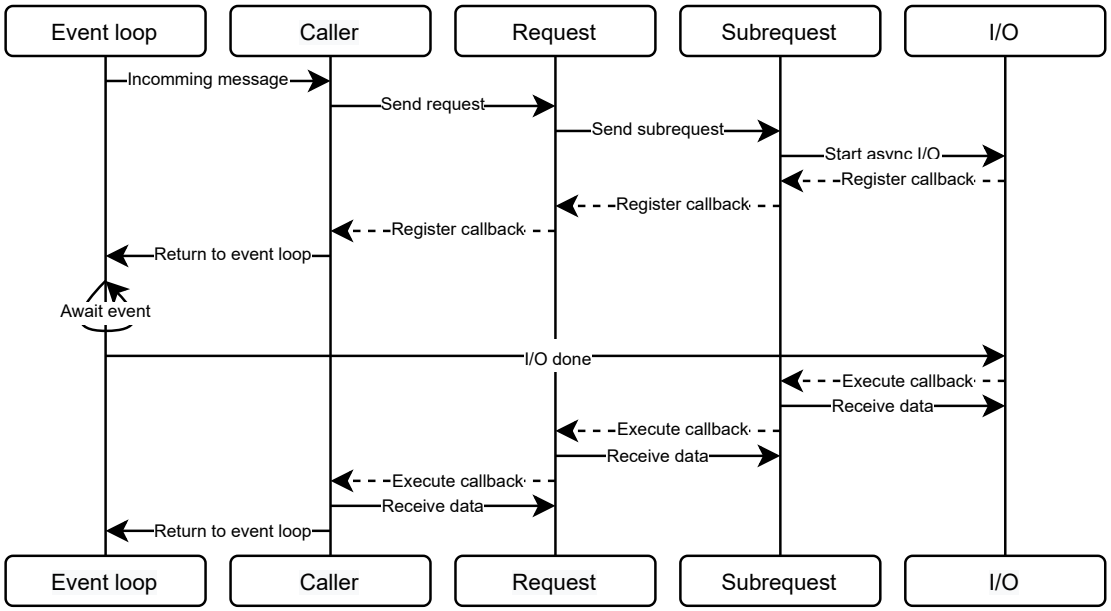

Figure 2.4: The `tevent`'s callback-based API depicted is the main building block of SSSD's request handling. In the top half it describes the process of `tevent` request's set up, while in the bottom half the request gets handled after it is received.

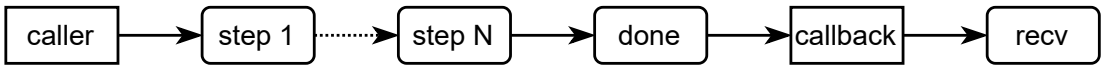

Figure 2.5: The chronologically ordered steps in the life cycle of a request in SSSD.

Each request "*needs to be named, created, finished, and consumed*" [2]. Upon creation, each request initiates an asynchronous operation. Registering of a low-level event (fd, timer, immediate) can either happen at this time, or the responsibility falls upon a request's subsequent sub-request.

In SSSD, the life cycle of a request, as shown in Figure 2.5 consists of a few components:

- **State structure** (`state`) – a data structure containing all information related to the current state of a request. Every request is associated with a state.
- **Send function** (`send`) – a function which initializes and creates the request with `tevent_req_create()`. It also starts the setup of any structure related to the request (back-ends, providers) as well as it "*initializes an asynchronous operation or sends another sub-request after the request is created*" [2].
- **Done function** (`done`) – after a reply is received, this function terminates the original request with `tevent_req_done()`. After this, a callback associated with the current request gets triggered. Depending on the request's result, this function either finishes the data structure's setup or terminates it.
- **Step function** (`step`) – in some cases, it can take multiple steps (e.g., multiple sub-requests, multiple LDAP lookups) to get everything needed to resolve a request.
- **Receiver function** (`recv`) – used to check the result of the request as well as a way to pass along the received data.
- **Caller** – the originator of the request. It arranges everything from sending the request to setting the callback to be triggered when the request finishes.

In SSSD's code, the components follow a set order in which they are written:

```
struct fetch_user_state;
struct tevent_req *fetch_user_send(...);
void fetch_user_done(...);
errno_t fetch_user_recv(...);
```

This helps with understanding the code since it creates a deception of a synchronous flow of the program even though it is not.

2.1 Identity management solutions compatible with SSSD

As stated before, SSSD is a client that users utilize to log in to access their files. To do so, SSSD incorporates and builds upon tools like NSS and PAM as well as connects to any combination of the following external identity and authentication providers:

- **Identity Management** – an open-source solution developed for Linux environment which is officially supported by Red Hat.
- **FreeIPA** – a free alternative implementation of IdM without the official Red Hat support services.
- **AD DS** – developed for Windows Servers by Microsoft.
- **LDAP directory** – server for storing and accessing cached user data.
- **Kerberos realm** – a domain with authority to authenticate a user.

The rest of this section will focus on FreeIPA and AD DS. These two management services have many common features and integrate similar implementations of the same protocols and tools. These include LDAP, DNS management, Certificate Authorities[8], etc.

Upon closer inspection, a few differences can be spotted as well. The majority of them are related to the two systems running on different platforms. Thus, they take advantage of their respective platform utilities listed in Table 2.1.

Scenario	AD DS	FreeIPA
Authentication	Netlogon	MIT Kerberos[9]
	Kerberos Key Distribution Center	
Time Management	Windows Time	NTP
Component Communication	Intersite Messaging	IPC

Table 2.1: AD DS and FreeIPA do similar things (i.e., identity management) using many different utilities, some of them are listed in this table along with their primary function.

2.1.1 Active Directory Domain Services

This project is used as a service to manage security (authentication and authorization) in a large environment of host machines. The following list consists of the main concepts on which it is built:

- **Schema** – a blueprint that states what attributes to store about any data in AD DS. It is designed to be extendable, which makes it easy to add or remove attributes at any given time without causing any problems.
- **Domain** – a logical group of network objects (e.g., computers, servers) that share the same database [2], security, and replication strategies.
- **Tree** – a group of domains with established implicit, transitive, two-way trusts. The name reflects that all domains in the system have precisely one parent (aside from the top domain) and thus form a tree-like structure. These domains have a shared schema, global catalog, and user/host permissions.

[8] https://en.wikipedia.org/wiki/Certificate_authority
[9] http://web.mit.edu/kerberos/

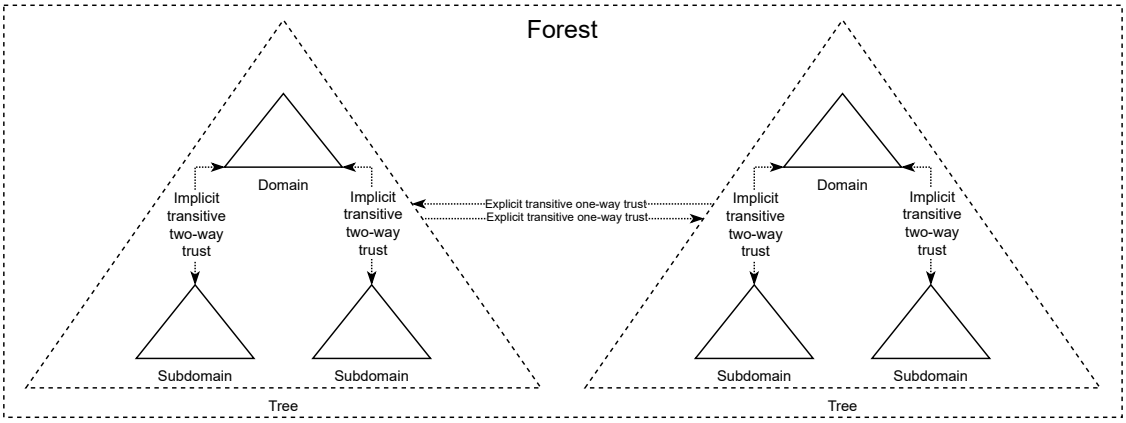

Figure 2.6: An example of an AD DS forest with two trees each with a domain and two subdomains. The trusts between domains and subdomains are implicit because they are automatically created upon starting the connection between them. The trusts between trees on the other hand, have to be explicitly implemented.

- **Forest** – one or multiple trees with different schemas. In a *forest environment*, not every tree has the same level of authority. Therefore, the trees within one forest have explicit transitive *one-way* trusts with each other instead of the two-way trusts mentioned previously.
- **Domain Controller** – a primary server that hosts AD DS. The domain controller (DC) is the central storage of all user/host data and manages all user communication. The standard strategy is to have a group of DCs (cluster) per domain to avoid preventable issues (e.g., data loss, overloading).
- **Global Catalog** – a basic DC stores a complete reference of objects within its domain. In contrast, a global catalog is a DC that collects information about all objects within its forest. This allows users and administrators to find information on anything, regardless of which domain in the directory contains the data [6].
- **Sites** – a network topology that splits domains, trees, and forests into groups to prevent connectivity and network issues. Hosts are grouped if their communication latency will not affect the reliability of the whole system. These groups essentially behave as forests that are interconnected by transitive site links. Through them, replication strategies periodically synchronize data between the respective sites.

Most of the discussed terms are depicted in the schema in Figure 2.6. Granted, it is an oversimplification of the described system, but it is sufficient as a general overview.

SSSD provides seamless integration between Windows clients using AD DS and Linux clients, even though AD DS is officially supported only on Windows servers. It does this through its *ad* provider, which provides SID to uid/gid translation and automatic domain discovery (within a single forest). An alternative is using the *ldap* provider instead of ad, but this comes at the expense of the mentioned automatic discovery and requires a broader knowledge of SSSD configuration.

The support of automatic discovery is inherited from DCs and removes the need to pre-define all DCs in *sssd.conf*. The ad provider searches for DCs, starting with the current AD DS site's primary DC, searching for backups only when this one fails.

The whole process starts with an LDAP ping (a rootDSE[10] search of a `netlogon` attribute) to discover the primary DC. Upon receiving the reply containing information about the site, forest, and domain, SSSD caches all of it. It also employs an alternative CLDAP ping in subsequent attempts of this specific DC's discovery to accelerate the process. It is faster than its LDAP ping counterpart since it searches using UDP instead of TCP, which is vital for SSSD's operation.

If a user needs to integrate domains from multiple forests along with Linux domains, it is recommended employing a strategy of creating trusts between AD DS and FreeIPA/IdM. This is because a direct integration works only within one forest.

2.1.2 FreeIPA

FreeIPA is an integrated security information management framework. It is built up from multiple tools and consists of a web interface and command-line administration tools [7].

Developed as an open-source project, FreeIPA aims to serve as a free Unix/Linux alternative to the Windows Active Directory. From FreeIPA version 3 onward, the contributors implemented features that enabled compatibility between the two. This means that users can use Single Sign-On[11] to log in from a Linux machine into a Windows one and vice versa.

FreeIPA is an extensive system that incorporates many tools and protocols to achieve its functionality, the most crucial core elements are listed below:

- **389 Directory Server**[12] – a service that works as a centralized data storage for all information about users, hosts, and more. It comprises an LDAP server, a web console, and an SNMP agent [10].
- **MIT Kerberos** – a network authentication protocol developed to provide strong authentication for client/server applications using server-key cryptography [15]. FreeIPA utilizes Kerberos for its Single Sign-On scheme, which helps streamline the user database since the system only needs to store one ID per user.
- **Dogtag**[13] – a *Certificate Authority* that enhances the authentication capabilities by unifying the control of a public key infrastructure. This entails validating requests, issuing certificates, storing keys, processing Online Certificate Status Protocol (OCSP) requests, and managing tokens [4].

Each FreeIPA instance uses an LDAP server provided by the 389 Directory Server as its centralized database. LDAP's Access Control Models[14] and Kerberos' ticketing system provide solid authorization and authentication throughout the whole system. Moreover, to further augment the authentication capabilities, FreeIPA employs DogTag, which automates the management of SSL certificates within the domain.

[10] https://ldapwiki.com/wiki/RootDSE
[11] Authentication service which allows user to use a single set of credentials.
[12] https://directory.fedoraproject.org/
[13] https://www.dogtagpki.org/wiki/PKI_Main_Page
[14] https://ldapwiki.com/wiki/Access%20Control%20Models

As a sister project, SSSD provides full feature compatibility to FreeIPA at all times. When it (or any other FreeIPA client) connects to a FreeIPA server through the Single Sign-On, a Kerberos ticket is forwarded to the LDAP server. In this case, the LDAP database also depends on Kerberos to establish the client's identity. As a result, this allows LDAP's Access Control Information to limit what entries can be accessed and what operations can be performed [1].

This approach simplifies the client's implementation because the server can communicate with all providers within its domain in the client's place. In fact, the base capabilities needed to pass as a client of FreeIPA are: the ability to forward a Kerberos ticket, process XML Remote Procedure Call (XML-RPC) or JSON, and the ability to present resulting responses to the user [1].

2.2 D-Bus

D-Bus is a very complex one-to-one (server-client) protocol and one of the main building blocks for almost all Linux distributions. It incorporates features such as interprocess communication (IPC), on-demand startup services, and security policies (more on them later in this chapter).

It is designed to operate on two levels - system and session (per-user). There is always only one system bus which serves mainly as a way for the system to communicate with the user sessions. This communication consists of sending notifications and input requests to a user and connecting all processes of every user. Moreover, thanks to the system bus's single well-known public address, these processes can be connected. In contrast, the session buses can own multiple addresses that belong to the currently logged-in user and are accessible only to those processes belonging to the said user.

D-Bus addresses consist of a transport name followed by a colon and an optional, comma-separated list of keys and values in the form key=value [11] (key = guid, value = UUID[15]). These addresses can be Unix paths or IP addresses with specified ports (e.g., *unix:path=/tmp/dbus-test* or *tcp:host=127.0.0.1,port=4242*). This means addresses do not necessarily have to be unique. If they are not, D-Bus uses the aforementioned key=value pairs to distinguish between them.

Similar to addresses, buses can have multiple names assigned to them, although only one is *unique* and remains with them during their entire lifetimes. Unique names must always start with a colon to distinguish them from others since this is forbidden for all other names. All bus names comprise two or multiple elements separated by a period but can not start with one. Only the characters *"[a-Z][0-9]_-"* can be used, with dash being discouraged in new names. Only elements that are part of a unique connection name may begin with a digit; elements in other bus names must not begin with a digit [11].

Buses must also implement at least one interface. D-bus interfaces are similar to classes or interfaces in programming languages and define each bus's methods, signals, and properties.

The naming conventions for interfaces are almost identical to bus names, with a few exceptions. Their names must not start with a number, and the dash character is also forbidden, which is commonly replaced with an underscore. A few interfaces should be present on every system, such as **org.freedesktop.DBus**.

[15] Universally unique identifier.

Information about naming schemes for other D-Bus elements can be found in the *D-Bus specification*[16], but these do not relate as closely to the subject of this thesis.

The communication between various users' processes is carried out by D-Bus messages. These fall into two main categories, which are:

- **Method call** is a message invoked to change or get information about a remote object. It has a single recipient and expects a reply[17]. The reply can be a method return when the invocation succeeds or an error when it does not. Methods are defined with zero or more arguments, where each argument can be of type *in* or *out*. Both types of arguments are supplied by the caller but the *out* are expected to return modified by the recipient accordingly.
  - **Method return** is a message reply to the method calls. If the sender does not specify otherwise, it must be sent out even if it will not contain any return values. The reason is that the sender would not know whether its method call succeeded or not if it did not.
  - **Error** is a reply to a method call upon failure on the receiver's end. It can also be used as a reply from a message bus when it does not have enough memory to forward the signal further. Errors can not have any arguments.
- **Signal** is different from a method call since it never requires a reply, plus it is required that their caller specifies the interface in its header. They are broadcast messages, therefore received by anyone who listens to them. Clients who want to start listening for the event signal must explicitly register to the message bus, which adds them to its internal list of listeners. Signals can not be implemented with *in* arguments as the caller does not require nor listen for the recipient's reply.

2.3 Topologies in distributed systems

Topologies in networking are a more prevalent topic of studies and articles than in distributed systems. However, the authors describe topologies as an architecture of nodes and connections by which a system operates. The communication between nodes in a network is analogous to distributed systems (described in Table 2.2). Consequently, this thesis will apply the principles valid in networking to them.

Component	Networking	Distributed Systems
Nodes	Servers Computers	Objects
Connections	Cables Routers	APIs Protocols

Table 2.2: Comparison of nodes and connections in networking and distributed systems.

There are five metrics to compare the main topology types relevant to this thesis:

[16] https://dbus.freedesktop.org/doc/dbus-specification.html
[17] This is only a preset behavior and can be changed if needed.

- **Manageability** – the level of complexity of a system. The more complex it is, the more it "*requires management: updating, repairing, and logging*" [9].
- **Information coherence** – the authoritativeness of a system. Takes into account whether all data found anywhere on a system is accurate.
- **Extensibility** – how hard (or easy) it is to extend the system or add new resources to it in future revisions.
- **Fault tolerance** – measure of the system's ability to deal with errors or failures.
- **Scalability** – tells us "*how large can any given system grow*" [9].

The rest of this section will describe each respective type of topologies, shown in the Figure 2.7, and evaluate them in regards to the mentioned characteristics.

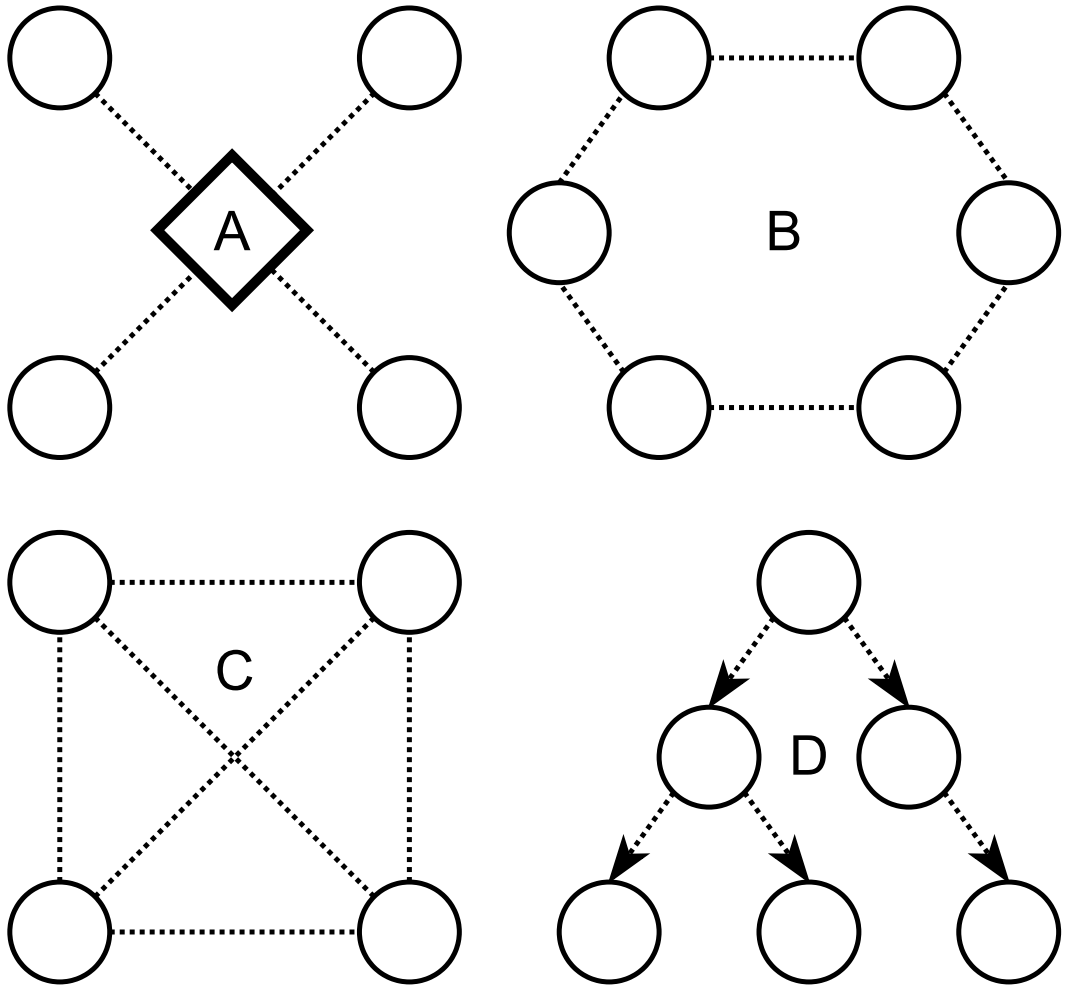

Figure 2.7: In distributed systems, topologies can be divided into multiple groups; star (A), ring (B), mesh (C), and tree (D) topologies are just a few of them.

2.3.1 Point-to-point topology

A topology for systems with the lowest number of nodes, which connects two nodes together. The bandwidth is shared between these two nodes only, it has low latency, and it is effortless to maintain since there are always only two nodes to handle. Depending on the system needs, the data flow between the nodes can be managed in three ways:

- *Simplex* – one of the nodes transmits the data, and the other receives it, the direction is set explicitly and never reverses.
- *Half-duplex* – the data can flow in both directions, but the nodes must take turns in transmitting.
- *Full-duplex* – the nodes are allowed to transmit data simultaneously in both directions.

2.3.2 Star topology

A centralized topology with a single central node that manages all of the other nodes (clients) in the system. These *client* nodes only connect to this central node and use it as a router whenever they need to communicate with other parts of a system. Having one central node has its positives; locating a problem becomes very simple when everything in the system is stored and managed from one place (data consistency is guaranteed). However, for this exact reason, the system is prone to crashes if the central node should ever fail. It also has a limited ability to scale since the total number of client nodes is limited by the central node's capacity; thus, it is suitable for systems with a fewer number of nodes.

2.3.3 Ring topology

In this topology, all nodes connect to another two in the system and together create a circular link. The data flows sequentially from one node to the other until reaching its target destination. It is tough to detect if an error happens between the sender and recipient. On the other hand, thanks to its failover logic, if a node should fail, another would take its place to cover up the problem. Moreover, it enables excellent scalability. Ring topologies can be implemented unidirectional or bidirectional (less common).

2.3.4 Mesh topology

A decentralized topology in which nodes are connected to multiple other nodes in the system. Its scalability is considered unlimited, but its scale is directly proportional to the overhead in its management algorithms. Since all data is stored on every node, it is very resource and time-demanding to keep it constantly consistent and coherent. The total number of connections depends on the implementation:

- *Full mesh* – every node is connected, number of connections is defined by $n(n-1)/2$, where n is the number of nodes.
- *Partially-connected mesh* – at least two nodes are connected to multiple other nodes in the system.

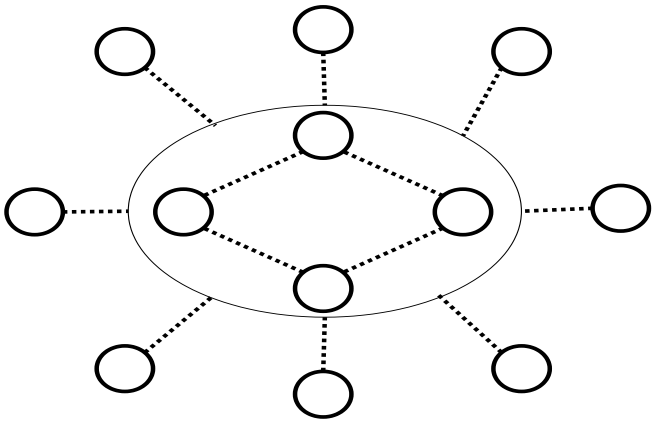

Figure 2.8: A hybrid topology which consists of a ring topology utilized as the star topology's central node.

Characteristic	Star	Ring	Mesh	Tree
Manageable	Yes	Yes	No	Partially
Coherent	Yes	Yes	No	Partially
Extensible	No	No	Yes	Partially
Fault tolerant	No	Yes	Yes	Partially
Scalable	?	Yes	?	Yes

Table 2.3: Summary of topology characteristics adapted from Nelson Minar's comparison of topology categories [9]. The *?* is used for topologies whose scalability can not be clearly judged.

2.3.5 Tree topology

A topology where nodes are organized in a tree, with a node between any two other nodes in the system and a single root node. The management, maintenance, and error detection are simple since there is a clear chain of action thanks to its hierarchical approach. It is a lot more fault-tolerant than other counterparts but shares a problem with the star topology; the whole system is brought down whenever the root node fails. The tree is best utilized in very large systems, because of its incredible scalability compared to the other topologies.

In practice, the majority of systems comprise combinations of these topologies, resulting in the so-called *hybrid topologies*. The topologies used can bring out each other strengths and suppress each other's weaknesses. Star mixed with a ring topology is a great example, shown in Figure 2.8. In this scenario, the disadvantage of a system dependent on one central node is overcome by replacing the node by a ring system.

Chapter 3

Performance analysis

Software tools and systems go through their life cycle with constant updates, bug fixes, and revisions. This is an entirely standard and needed approach; however, it is not without its drawbacks. A small change can create an unpredictable product behavior which can sometimes translate into big loading times, crashes, etc.

It is an excellent practice to have a test suite developed alongside the product and measure its behaviors and performance with every new build. By aggregating and comparing this data regressions or improvements in performance can be captured along with many other valuable findings.

Software performance includes many concepts and can be grasped in many different ways. Firstly, the following list contains some of the most important metrics and concepts that have considerable importance when talking about a product's performance:

- **IOPS** – This abbreviation stands for *Input/Output Operations Per Second*. It is a "*measure of the rate of data transfer operations*" [8] (e.g., IOPS of disc operations stand for read and writes per second).
- **Throughput** – The measurement of the rate at which a particular operation is performed. There are a few different rates that this metric can measure. When a test measures the application's ability to transfer data in, e.g., bytes per second, it measures its *data rate*. In the context of databases, the value of interest is the *operation rate* which determines how many operations per second they can perform.
- **Response time** – This is an evaluation of the entire time user spends waiting to get serviced. The whole time, from their initial request to the moment they receive a reply.
- **Latency** – In many situations, this term can mean a range of things. In one situation, it can stand for the whole time of the operation, in which case it equals the application's throughput. In other cases, for example, in a network service request, it can stand for the time it takes the network to establish a connection. Because of this variability, it is crucial to mention the context in which the latency is measured.
- **Bottleneck** – In the contect of distributed systems, it is a resource that limits their performance.
- **Cache** – It is "*a fast storage area that can duplicate or buffer a limited amount of data.*" [8]. To speed up the process of repeatedly accessing the same data on a slower

data storage, the data is stored in a faster one which can hold only a limited amount of data.

Understanding what needs to be measured, and why, is fundamental when creating a test suite. After that, the discussions about what gets assessed, when and how these metrics are implemented can be held. There are a few well-known approaches to determining the performance of a system or an application, some of which are mentioned in the following list:

- **Load testing** – The process of thoroughly exerting a system or an application to measure and gain knowledge about its behavior in extreme conditions. This process consists of simulating users with many concurrent requests based on a prediction of what the system should handle during production. Through these assessments, it is possible to learn about the maximum load a system or an application can handle.
  - **Benchmarking** – a type of a load test which "*tests performance in a controlled manner*" [8]. It employs metrics and workload specifications to produce comparisons between many performance measurements of a system under test.
- **Tracing** – Recording of a specific event happening in a system and storing the data for later use by an analysis or consumed during the test's run for custom summaries. There are multiple tools for tracing events happening in a system: `strace`, `tcpdump`, `SystemTap`, `bpftrace`, etc. Tracing can be done through two different approaches:
  - *Static* – some preparations must be done before these tests start running. Static tracing works by inserting instrumentation points to specific spots in the product's code. During the actual testing, these instrumentation points serve as a way to get a hold of data about what is going on, parse values being used, etc. In the release builds, the instrumentation is either omitted or substituted by *No Operation* (NOP) instructions to avoid any needles overhead.
  - *Dynamic* – this analysis creates the points and modifies instructions only after the application is already running. One of the essential tools for dynamic instrumentation is `DTrace`[1], initially created in 2005.

Some of the performance analysis tools were already mentioned in the list above; those are just a few of the many that currently exist. Two of them are used for performance analysis in this thesis; their features, advantages, and disadvantages will be discussed in the following sections.

3.1 `SystemTap`

The first of the two tools is used for tracing and probing, which can be used to study and monitor activities of an operating system (kernel) or in user-space programs[2]. Its benefit over other kernel-oriented tools, e.g., `netstat`, `ps`, `top`, is that `SystemTap` provides a level of flexibility not available through these built-in tools. With `SystemTap`, it is possible to specify the exact points to measure and investigate how the acquired data should be processed and how the data should be presented.

[1] https://github.com/opendtrace/opendtrace
[2] Starting from the version 0.6, `SystemTap` expanded into user-space probing too.

In order to operate, `SystemTap` requires the packages used for debugging, `kernel-devel` and `kernel-debuginfo(-common)`, to work correctly. Unfortunately, these packages are not tested with every single version of the kernel; thus, the reliability depends on the current version installed on the system. `SystemTap` works best on RHEL and Fedora since it is developed by Red Hat[3], but in some cases, it can also be set up on other distributions. The process can differ significantly; sometimes, it can be acquired by installation from the specific package manager, through some workaround, or, in the worst-case scenario, it can not be set up at all.

Similar to kernel-space probing based on *tracepoints*, markers can be used in the user-space as well. These *predefined markers* (sometimes called User-space Statically Defined Tracepoints or USDT too) can be created by inserting them into the target's source code, which will be discussed more in Subsection 3.1.3. The reliability of these markers working as expected is also tied to the kernel support, with many of the kernel's versions not supporting it at all.

The tests can be run by using the `stap` command and passing it a script[4], containing all probes and functions related to the program. Probes can also be supplied from the command-line by writing them after the `-e` flag, as shown in Listing 3.1.

```
stap -v -e 'probe vfs.read {printf("read performed\n"); exit()}'

Pass 1: parsed user script and 45 library script(s)
Pass 2: analyzed script
Pass 3: translated to C into "/tmp/stapiArgLX/stap_.*.c"
Pass 4: compiled C into "stap_.*.ko"
Pass 5: starting run.
read
Pass 5: run completed in 10usr/40sys/73real ms.
```

Listing 3.1: An example taken from the `SystemTap` documentation of a `stap` command's run. The `.*` replaces the hashes that are assigned to each generated temporary file. This is an modified example taken from the `SystemTap`'s documentation.

The process in which `stap` runs the tests and how `SystemTap` works is neatly illustrated in Listing 3.1. It is split into five steps (called *passes*), where each step depends on the success of the previous one. Thus, all subsequent steps are canceled when one fails, and an error message is printed out. The following list explains each of the steps in order that they happen:

1. `SystemTap` reads and parses the probe passed from the command-line. As there is a `vfs.read` alias supplied (more on tapsets and aliases in 3.1.1), it also searches through its tapset library and uses the corresponding probe or function to process the supplied *script* properly.

2. This is a confirmation that the semantic analysis has finished without any problems.

3. In this step, `SystemTap` *converted* the supplied probe to a code in `C` language and stored the file containing the code in a temporary file.

[3] https://www.redhat.com/en
[4] `SystemTap` script have the *.stp* suffix.

4. After the successful conversion, the resulting code gets compiled and is ready for execution.

5. Multiple things happen in this step. `SystemTap` loads the compiled kernel module, starts it and waits for the specified function to be called. After the function gets triggered, the probe's body executes; hence, the message `read` appears. After this, `SystemTap` stops the tracing and prints out the final message since the `exit()` function is specified in the probe body. In other situations, `SystemTap` would keep waiting for another `vfs.read` call and would keep running until the process is halted by the user `^C` signal.

The `SystemTap` script language contains many interesting features. One of them, the *embedded `C` code*, is useful when the script's capabilities are insufficient for certain types of analysis. The code is inserted into the script by defining its bounds by `%{ %}`. These bounds can be used outside `SystemTap`'s primary probes and functions to declare, define, or assign variables or create complete `C` functions with the syntax: `function name:type(arg1:type, ...) % <C code> %`. To successfully run this type of probe script, the `stap` command needs to be run in a *guru* mode triggered by specifying the `-g` flag.

The following subsections will discuss the other interesting and essential features of `SystemTap`.

3.1.1 Tapsets

Tapsets are best described as "*scripts that form a library of pre-written probes and functions to be used in `SystemTap` scripts.*" [3]. There are many tapsets predefined in `SystemTap`'s library at */usr/share/systemtap/tapset/*, but custom tapsets can also be created. As mentioned earlier in this chapter, during *Pass 1*, shown in Listing 3.1, when an alias from a tapset is supplied, `SystemTap` looks for its definition inside its tapset library and replaces them with the corresponding probes or functions. An example of such an alias is shown in Listing 3.4. Using them makes it easier for developers to remember useful functions than using the "*specific kernel functions that might vary between kernel versions.*" [3].

3.1.2 Associative arrays

To enhance its ability to produce statistics about the monitored processes, `SystemTap` supports associative arrays. These arrays are "*collections of unique keys; each key in the array has a value associated with it.*" [3].

The usual use case for these arrays is compactly representing a collection of global variables for aggregation of collected data. To capture the data, the scripting language enables the use of one of the two operators `=` or `<<<`. While the former assigns value to the array at a specific key, the latter creates a list and appends values to it as necessary.

It is possible to state up to nine different values as keys for the arrays: `arr[k1, ..., k9]`. Multiple operations can be executed on the values in the arrays. For example, there can be an array whose values will be incremented with the `++` operator. With this setup, it is possible to count the number of times a function gets triggered and store the counts separately for each process that has done so.

It is possible to use the `foreach` statement to iterate over the existing keys, which works in the same way as in many other programming languages. However, it has some additional

features, which were developed to enhance the processing of the collected data. It is possible to choose whether to start iterating through the values in ascending or descending order by adding a `+` or `-` sign at the end of the array's name. Furthermore, it is possible to set how many values to iterate over using the `limit x` function. So to iterate over the top ten captured values in the array, one can achieve it: `foreach (x in reads- limit 10)`.

`SystemTap` also implements the concept of statistical aggregates, which are created with the aforementioned `<<<` operator. The captured and saved data can then be run through the native statistical functions to produce an output either during the test's rest run, or after it is finished. These functions, listed below, are used in combination with the `foreach` statement to iterate over each key of the array to run calculation on the stored values:

- `@count` returns the number of the stored data,
- `@sum` returns the sum of the stored data,
- `@min` returns the smallest value,
- `@max` returns the largest value,
- and `@avg` returns the average of the stored data.

Having these functions is excellent for producing some statistical aggregates, but if there is a need to use the values to compute some other result, `SystemTap` does not allow it. The statistical aggregate arrays can be used only as intended by the creators and there is no other way to access the values stored under any of the keys.

3.1.3 User-space probing

The user-space probing in `SystemTap` allows to latch on any function, statement, or marker in the program's code. The following list describes the different methods of instrumenting on user-space program:

- `.function("`*name*`")` – This is an instrumentation of the user-space function called *name*, which allows wildcards[5] and the `.return` suffix.
- `.statement("`*location*`")` – This method points to the statement at the line *location*.
- `.mark("`*name*`")` – This is a statically fixed marker point called *name*. The benefit of using this method is that, unlike the first two, markers can pass pre-determined parameters to the script. This enables a closer look at what is going on inside the program.
- `.syscall` – Captures an user-space system call. When defined with the `.return` suffix, the probe gets placed at the `syscall`'s return subsequence.
- `.begin`, `.thread.begin`, `.end`, `.thread.end` – Suffixes that are available for the first three items in this list. The probe is *listening* for a different part of the method trigger depending on the suffix that is chosen.

[5] To investigate multiple functions at ones, the asterisk (*) symbol can be used.

The syntax that defines how these methods are used is shown in Listing 3.2.

In a small project, adding a marker requires including the `sys/std.h` library and adding `DTRACE_PROBEx(provider, name, arg1, ..., argn)` to the source file. It is suggested to use *compile-time configurable instrumentation* to add these probes to a larger project.

```
probe process("<binary path>").method("<label>").suffix {
    <function body>
}
```

Listing 3.2: The template of probe definitions in a `SystemTap` script.

To start with `SystemTap` user-space markers, a switch, shown in Listing 3.3, has to be inserted into the project's *configuration.ac* file. This switch, `--enable-systemtap`, enables or disables `SystemTap` configuration in the project with the `configure` *autotool*[6], which then "*checks to determine if the* `dtrace` *script and the* sdt.h *header are available*" [5]. After that, `dtrace` can generate the required header files with probe macro declarations used by the program during its run.

```
AC_MSG_CHECKING([whether to include systemtap tracing support])
AC_ARG_ENABLE([systemtap],
  [AS_HELP_STRING([--enable-systemtap],
    [Enable inclusion of systemtap trace support])],
    [ENABLE_SYSTEMTAP="${enableval}"], [ENABLE_SYSTEMTAP='no'])
AM_CONDITIONAL([ENABLE_SYSTEMTAP],[test x$ENABLE_SYSTEMTAP=xyes])
AC_MSG_RESULT(${ENABLE_SYSTEMTAP})

if test "x${ENABLE_SYSTEMTAP}" = xyes; then
  # Additional configuration for --enable-systemtap is HERE
fi
```

Listing 3.3: An example of *configure.ac* configuration, taken from `SystemTap`'s documentation, for compile-time instrumentation for enabling `SystemTap` user-space markers.

The next step is to declare each marker in a declaration file (e.g. *probes.d*) that contains a structure-like `provider` of functions (representing markers). In these functions, it is possible to declare which variables are expected to be passed to the probe from the marker's location.

To be able to use these declarations in a program, it is required that the file which contains them is coupled with *trace.h* header is added to the `automake`'s configuration file *Makefile.am*. The trace header contains `TRACE(probe)` macro templates to which values from the `provider` structure are sent during the program's build. Thus, through compilation, all this results in a probe header file (e.g., *probes.h*).

For example, let us say there is a `test(int)` probe in *probes.d* declared in the `provider my_probes`. After the compilation, the header file (e.g., *probes.h* created by `dtrace` would contain the following macro: `#define MY_PROBES_TEST(arg1) STAP_PROBE1(provider, test, arg1)`.

[6] https://www.gnu.org/savannah-checkouts/gnu/autoconf/manual/autoconf-2.71/autoconf.html

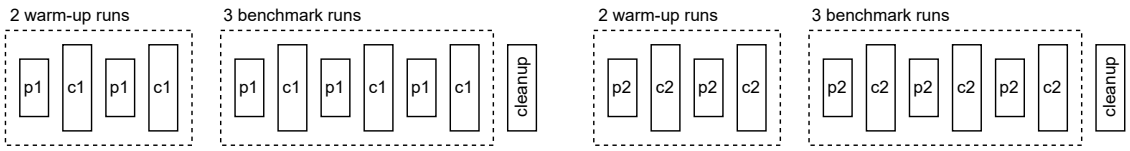

Figure 3.1: The chart shows how hyperfine's different phases go in subsequence in regards to the original command: `hyperfine -warmup 2 -runs 3 -prepare <p1> <c1> -prepare <p2> <c2> -cleanup <cleanup>`.

With all these preparations in place, it is possible to create the markers in the actual code by including the *probes.h* and *trace.h* headers. One of the macros in the trace header is used for cleaner marker inserts. Continuing with the previous example, adding a marker would require to use `TRACE(MY_PROBE_TEST())`, and the header files would take care of the rest. On the `SystemTap`'s side, this probe can be accessed by the tapset shown in Listing 3.4.

```
probe my_probe_test = process("my_probes").mark("test")
{
  string = $arg1;
  probestr = sprintf("%s", string);
}
```

Listing 3.4: An example of an alias declared in a `SystemTap`'s tapset library that prints the value of its string-type argument.

3.2 Hyperfine

`Hyperfine` is a lightweight "*command-line benchmarking tool*"[12] written in Rust[7]. Some of the `hyperfine`'s features include statistical analysis, support for basic shell commands, warm-up and preparation runs, etc. Its workflow and a few of these features are depicted in Figure 3.1.

It can be configured through flags to run the target command a specific number of times. Furthermore, it enables one to choose whether to run several warm-up rounds before the tests. The `--prepare` flag is another helpful feature that allows running a command before the test's run. This is beneficial to reach the desired state of the system the benchmarks are running on (e.g., clearing the cache so that request always comes from the source).

To carry out various tests within a single run, this tool also provides the ability to enter a list of values as a variable. The values are then iterated over during `hyperfine`'s run and each value is then passed to a defined variable as its value.

The tool can export the results to a file in either Markdown or JSON. Markdown stores values like the captured times' mean, max, and min values. On the other hand, JSON stores the time of every command run and is easier to use in performance analysis.

[7] https://www.rust-lang.org/

Chapter 4

Design and implementation

This chapter will go through the design and implementation phases step by step, solving the problems in a systematic manner. Starting with SSSD's current way of dealing with D-Bus communication and the possible solutions while developing the new topology. It will discuss the processes which had to be changed and the steps that lead to the implementation of the centralized communication through a single message bus.

Furthermore, following the bus reimplementation, it will look closely at the proof of concept solution of SSSD's interfaces defined with signals. With a single bus topology, it would be possible to utilize the broadcast abilities of signals in place of methods that do not require method replies.

4.1 Message bus reimplementation

All significant components of SSSD implement *context structures* that manage all data relevant to the each of them, and improve cooperation between the respective components in SSSD. Most of the data related to the D-Bus server is also available to responders and back-ends through various doubly linked lists. Examples of such structures which are relevant to the task at hand are in the list below:

- `sss_domain_info` – a doubly linked list that contains data about all currently active remote providers (domains). Most data is assigned to the list during the start-up in the `confdb_get_domains()` function. The member I have focused on the most in this structure was the domain's `conn_name`. In SSSD, this value represented the bus' well-known name and was originally used only for message routing and other operations concerning message buses. The `sss_domain_info` also contains a member `name` that held the same value but served a different purpose: it was used for the initialization and management of the back-end itself.

- `sbus_connection` – this structure holds all information about a running message bus, i.e. its address, well-known name, the actual D-Bus connection[1], etc.

- `resp_ctx` – each responder stores its info in this structure. Most notably, it contains the `sbus_connection` data about buses to which it is connected and the list of active domains `sss_domain_info`.

[1] `DBusConnection` implemented in the `dbus` library

- `be_conn` – a doubly linked list whose items are created by the back-ends for responders. It tracks details about all responders connected to the back-ends, the buses through which the connections exist (e.g., well-known names, address), the matching `resp_ctx`, etc. Whenever a responder needs to contact a particular provider, it searches for an item containing the bus address through `be_conn` stored in their `resp_ctx`.

These are just a few building blocks that comprise the current architecture in which components communicate through message buses. Using these structures, as depicted in Figure 2.1, SSSD coordinates its current IPC between responders and back-ends in the form of multiple star topologies. Their central nodes are the D-Bus message buses spawned by the monitor and the back-ends. As stated in Chapter 2, the monitor keeps track of everything that goes on in SSSD, including communication. As such, the responders and back-ends are the clients of the monitor's bus in the current topology, with responders being the clients of the back-end owned buses as well.

In contrast, the proposed architecture will instead contain only a single star topology. SSSD will spawn a single instance of the message bus responsible for all communication, and all components (monitor, responders, back-ends) will connect to it. To produce this result, changes in different parts of SSSD's implementation must occur. The process can be divided into three steps:

1. Change how the monitor assigns names to the `sss_domain_info->conn_name` to enable routing using well-known names instead of addresses.

2. Modify SSSD's *spam filter*, which is in place to halt any repeated request on any given bus, to facilitate single-bus communication.

3. Create a central bus that all components will connect to and communicate through.

The following sections will discuss how I have decided to approach these changes with a few details about the current state in contrast.

4.1.1 Prepare SSSD for routing by domains' connection name

The original topology's request routing consists of having the responders wait for a request from a user or other part of SSSD. Upon receiving said request, responders use the data to find which domain to contact and search for the correct D-Bus address to forward the request along to the data providers.

The original topology contained multiple buses, so only their addresses needed to be unique. Therefore, the two members of `sss_domain_info`, `name`, and `conn_name`, held the same value: the name of the domain (e.g., *ipa.vm*, *ldap.test*, ...). The structure `sss_domain_info` did not need to contain `conn_name` yet, but it helped to logically differentiate between the two uses of the value, as described in Section 4.1.

One of the functions, `sss_process_init()`, gets called from the responders to build up their `be_conn` *entry*. SSSD, through the `sss_iface_domain_bus()` function stores the transformed `conn_name` value in the format `sssd.domain_<conn_name>`. This value is then put into the header of a D-Bus message so it can be processed appropriately. To achieve the required implementation, this value needed to be modified in a way that it could be used as a key component in routing messages inside SSSD.

Format	old	`-:%u:%s.%s:%s:%s`
	new	`-:%u:`*`%s:`*`%s.%s:%s:%s`
Legend	old	sender:type:interface.member:path[:arg0:arg1:...]
	new	sender:type:*destination:*interface.member:path[:arg0:arg1:...]
Example	old	-:0:sssd.dataprovider.getDomains:/sssd:arg0
	new	-:0:*sssd.domain_ipa_2etest:*sssd.dataprovider.getDomains:/sssd:arg0

Table 4.1: The original (old) and modified (new) formats of the request keys. The addition of the request's destination to the format is displayed in italics.

The function which was responsible for the formatting, `sss_iface_domain_bus()`, was located in *sss_iface.c*. This file is part of the category responsible for setting up D-Bus interfaces. One of the solutions would be to include the *sss_iface_async.h* header, which declares the function, in *confdb.c* to acquire the required format during domain initialization. Nevertheless, the cleaner alternative was to move the whole function to *confdb.c* since after all changes were done, this function was no longer needed anywhere else in SSSD. After that, I have also renamed the function to `confdb_get_domain_bus()`, so it follows the pattern of the existing functions in the file.

4.1.2 Modify the request keys

SSSD has a system to stop unnecessary repeated requests to the back-end providers. Each request, during its configuration, creates a unique *request key* through the `sbus` wrapper. This key is a concatenated string built up from various pieces of data regarding the request itself:

- the **type** of the request,
- the target **method**,
- the **interface** which implements the method,
- the bus' **path**,
- a list of **arguments** required by the chosen method (the length of the list is $0 - n$).

As the list's last entry suggests, the keys' format is slightly different for every D-Bus method, but the majority of the string stays the same. When the request originates from a specific remote client, the key also contains its UID at the beginning; otherwise, it only includes a character '-' as shown in Table 4.1.

To keep track of the requests, SSSD utilizes a doubly linked list `sbus_request_list`. A new list is created every time a new request key appears, and every subsequent request with a corresponding request key gets appended. With these lists, SSSD can *chain* incoming requests and prevent unnecessary repeated requests of the same data. Therefore, SSSD sends only one message to the chosen destination, waits for an answer, replies to all requests in the list, and clears the list.

This system worked very well for the original topology, but a problem would arise when the number of message buses would be scaled down to one. These problems would be caused by SSSD's inability to separate the same requests for different providers.

Let's say a user requests data about the user admin with the `id` command in an environment where SSSD operates with these data providers: a FreeIPA server – `ipa` and LDAP directory – `ldap`. When the user does not explicitly state the provider from which SSSD should fetch the data, SSSD will try all active ones. This means the NSS responder sends a message using the back-end's `getAccountInfo` method to all providers stored in `sss_domain_info` i.e., to both `ipa` and `ldap` in our case. When there are unique buses for every back-end, each request is handled separately: the responder gets its reply, and the user receives the requested data (if it exists). However, when there is only a single bus, SSSD would be able to send out only the first request to a single provider, and the rest would be incorrectly treated as repeated requests. So if the first request would go to `ipa`, but the requested data would exist in `ldap`, then the user would not get the requested data even though they should.

Hence, the original format needs to be extended with an additional element. In this case, the best solution for the task was to add the destination's bus name to the key. That way, SSSD could differentiate between the requests mentioned in the previous paragraph so that the proposal would go to both providers.

The management of the request keys belonged under one of many parts of the code generated by the `sbus` library. Thus these changes which needed to be made in template file: *keygens.c.tpl*. Here I found out that all of the parts of the request key were taken from the `sbus_request` structure. This structure conveniently already included a member `destination` with the exact data I needed to solve this issue. All that was left to do was to add it to the `talloc_asprintf()` function in every function that generated the keys and this step in the process has been finished.

4.1.3 Centralize communication to a single D-Bus server

The next problem was finding out which component should own the bus, which would be the central point for all D-Bus communication in SSSD (as shown in Figure 4.1). As previously mentioned, the monitor, along with the back-ends, starts and manages its own bus. Since the monitor is a central component responsible for monitoring everything inside the SSSD, it is the perfect place to have its monitor be the central bus. To do so, I firstly made a *cosmetic* change and renamed the bus from `sbus-monitor` to `sbus-master`. Then I split up the problem at hand into two smaller tasks to simplify it:

1. Stop the back-ends from creating their own buses and the responders from connecting to them. Make them all connect to the monitor's `sbus_master` instead.

2. Change the process by which responders search for the D-Bus connection of the requested back-end.

To start with the first task, place which is used to creation of the back-ends' buses had to be located. At SSSD's start up, during the last steps of monitor process' initialization, the function `add_new_provider()` gets called in a loop to start all providers mentioned in `sssd.conf`. This function eventually calls onto the function `dp_init_send()` in the *dp.c* file which contains most of the functions related to back-end configuration and management.

In this function, `dp_init_send()`, the back-ends fetch all the needed data for the bus' set up until finally calling `sbus_server_create_and_connect_send()`. The `sbus` library takes care of the rest and, after it is done, returns all the bus data through an asynchronous `tevent recv` function.

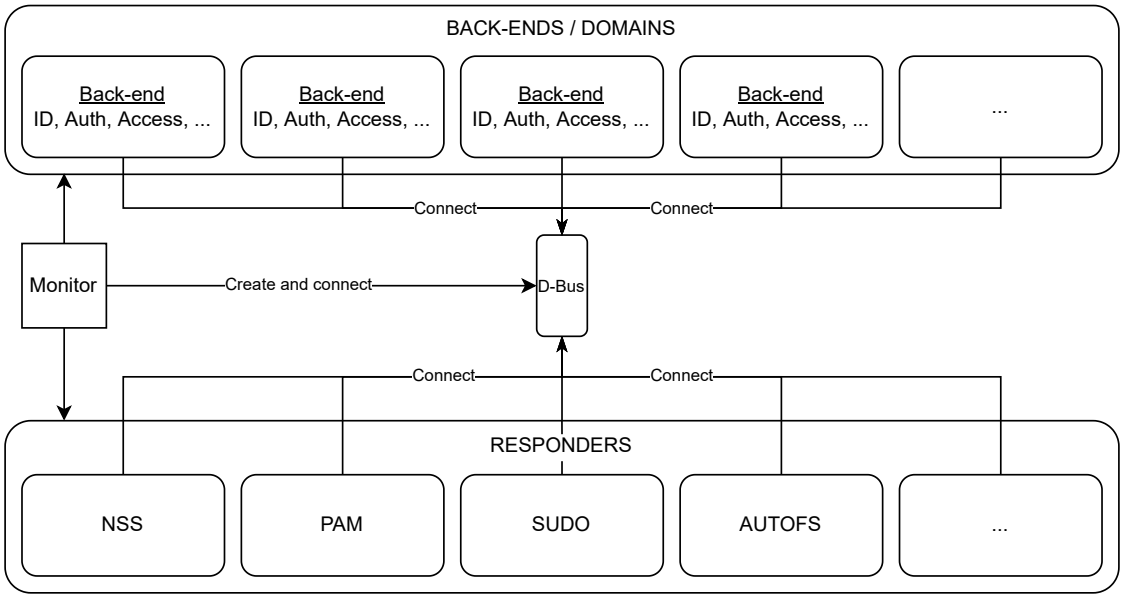

Figure 4.1: A schematic illustration of the new proposed architecture based on a single star topology.

To alter the process in a way that is required by the new topology, the back-ends needed to connect to monitor's already existing bus instead of creating their own. This closely resembles the way responders behave, as can be seen in their *.*srv.c* configuration files (e.g., *nsssrv.c*, *ifpsrv.c*). There I have found that their process of connecting to the monitor is managed through the `sss_monitor_service_init()` function.

Through this function, responders get access to `sbus_conn` filled with all relevant data about the monitor's bus and connect to it. The exact point where the responders acquire the data is the function `sbus_connect_private()`, which had almost the same functionality as was required in the back-ends. This function connects to a running bus specified by an address and a well-known name but only in a synchronous parts of the code. Since the back-end bus' set up is an asynchronous process that meant the found function could not be used for their purposes. However, the `sbus_connect_private()` function also has `tevent`-based asynchronous alternatives `sbus_connect_private_*()` with the same functionality. This set of functions was also used inside `sbus_server_create_and_connect_send()` as a way to connect to the bus that it creates, which supported the indication that it could be used as a replacement.

After the original functions for creating the buses in `dp_init_send()` were replaced with the `sbus_connect_private_*()` functions, back-ends no longer could spawn their own buses. Nevertheless, one more feature was still missing in the monitor's central bus, which was crucial for the back-ends to work. They needed a way to add their D-Bus interfaces and methods. Formerly, this was done in a handler function `dp_initialized()`, executed after all back-end set up was finished. In this function, two things needed to be adjusted:

1. The original `recv` function for `dp_init_send()` was not meant to return any value other than the back-end context itself. Through `talloc` and `tevent`, I was able to modify it so that it could pass along the earlier created connection to the monitor in

the form of `sbus_conn`. This data was essential to acquire before starting with the next step's task.

2. Furthermore, the function `dp_initialized()` was originally the place where back-ends could connect to the monitor's bus and did so in a similar fashion to responders. It did so through the function `sss_monitor_service_init()`, but for it to be used in the new implementation some parts of its functionality had to be changed. Thus I have created the function `sss_monitor_provider_init()` which removed all redundant steps of the original function and simply registered to monitor through the `RegisterSerice` D-Bus method.

To reiterate, in the current state, no back-end can create their own bus anymore; they can connect to the central one spawned by the monitor and register their interfaces to it. This means that the task related to the back-ends was finished, and it was time to focus on the responders and the routing logic.

To change the routing logic, I had to rework the way responders keep track of the available bus connections, addresses, and other values related to the message buses. Previously, the responders kept track of all of the relevant data in their context structure `resp_ctx`'s member `be_conn`. As I noted before, `be_conn` is a doubly linked list whose purpose was to match accessible bus names and addresses to existing D-Bus connections. This meant that after taking away all other buses, its functionality would be meaningless and should be replaced by something else entirely.

Moreover, since there were no longer multiple buses, SSSD only needed to keep track of a single address. Since that was the case, responders could now differentiate between the buses only by their well-known names.

The `conn_name` format change, described in 4.1.1, was necessary so I could use the `sbus`-generated functions (e.g., `sbus_call_dp_autofs_GetMap_send()`) properly. These are the functions responsible for sending and receiving D-Bus message requests. Coupled with the `sbus_conn` structure, `conn_name`'s value was vital to enable communication that was previously accessed from the aforementioned `be_conn` list.

Since `conn_name` could be accessed by responders through the `sss_domain_info` structure at any time, it could be used as a `be_conn->bus_name` replacement. Therefore, the next step was to also find a viable replacement for `be_conn->conn`.

The first point of interest was responders' context structure `resp_ctx`, I found out that other than back-ends' buses[2], it also contained the connection to the monitor's bus. If this value was available to responders at the right time during their start up, it could be the last missing part of the solution.

When looking at the entire start up process of responders located in the beforementioned *.*srv.c* files, the function `sss_monitor_service_init()` appears in all of them. To reiterate, this function is used for connecting components to monitor as services and does so by connecting to its message bus among other things. When called, it returns the `sbus_conn` structure full of data related to the monitor's own bus. This meant that the value was already available to responders by the time any request could be received and that it could be used in routing during the actual run of SSSD without any problems.

Therefore, I have renamed `mon_conn` in `resp_ctx` to `sbus_conn` to represent that it would no longer serve only as a means to connect to the monitor and removed the `be_conn` structure from it entirely. Now I could replace all instances of the original `be_conn->conn`

[2]This data was stored inside `resp_ctx->be_conns`.

with the `resp_ctx->sbus_conn` value. At this point, the last step was to remove the `be_conn` structure from SSSD altogether and test if everything worked as intended.

4.1.4 A bug found during the reimplementation

As it turned out, all these changes uncovered a bug[3] inside SSSD. While SSSD starts up, providers send out messages to responders' interface methods `SetInvalidate` and `SetActive`, which disable and enable domains for responders. During this stage, the monitor is not yet aware which responders are up and running, which are starting up, and which are not configured at all; therefore, it attempts to send method calls to all of them. This was not a problem before since the old format of *request keys* (as shown in Table 4.1) did not include the destination's name. That meant the requests to each responder interface had the same key and thus were being chained, thus the desired action got triggered only once. After extending the keys with the destination string, SSSD could distinguish between these requests, and these actions started to get triggered simultaneously from multiple points, which created problems. The whole reason is beyond the scope of this thesis, and the solution is still being figured out at this time. For now, to enable SSSD to run with a singular central D-Bus server, I had to use a workaround that included simply commenting out the PAM responder from the list of desired recipients.

4.2 Transform viable interface methods to signals

As mentioned at the beginning of Chapter 4, now that the topology included only a single D-Bus message bus, it enabled SSSD to take advantage of the D-Bus signal's broadcast ability. Because there was no longer only a single recipient on the bus' other end, and the callers' messages were no longer *routed* before going to the bus itself.

Therefore, it was now possible to transform a few of the interface's methods into signals, lowering the complexity of SSSD's communication through D-Bus even more. The candidate methods had to meet two requirements:

1. **No *out* arguments in the method's definition.** As mentioned in Section 2.2, signals can not be implemented with an *in* argument. Because of that, this requirement may seem contradictory, but after the transformation of a method into a signal, the original concept reverses. While a method is a function call with an expected return value, a signal is like receiving the same return value without ever calling a function. Therefore, what previously was an *in* argument in a signal's definition is assumed to be an *out* argument and vice versa.

2. **No part of the code on the caller's side should require a confirmation about a successfully finished operation.** In some cases, even if the method does not specify that it requests a reply, the functionality of SSSD's components may depend on a confirmation of a successful operation.

SSSD uses a combination of XML definitions and a set of `C` macros defined in the `sbus` wrapper to implement interfaces. In SSSD, four XML files serve as templates used to generate the actual code for the interfaces:

- *dbus.xml* – defines a general D-Bus interface.

[3] https://github.com/SSSD/sssd/issues/6286

- *external_iface.xml* – defines interfaces for products that are not part of SSSD internally: `systemd`[4] and `Fleet Commander`[5].
- *ifp_iface.xml* – defines interfaces for the *InfoPipe* responder.
- *sss_iface.xml* – a general file for defining interfaces of responders and data providers.

The files use the official *Document Type Definition*[6] created specifically for D-Bus. All files have a root `<node>` element in which they implement their interfaces and methods using the `interface`, `method`, and `arg` elements, as shown in Listing 4.1. In SSSD's case, the `annotation` element is used to guide the code generator, i.e., what interface name to use in the generated function's name or whether the method is synchronous or asynchronous.

```
<node>
    ...
    <interface name="sssd.nss.MemoryCache">
        <annotation name="codegen.Name" value="nss_memcache" />
        <annotation name="codegen.SyncCaller" value="false" />
        <method name="UpdateInitgroups">
            <arg name="user" type="s" direction="in" />
            <arg name="domain" type="s" direction="in" />
            <arg name="groups" type="au" direction="in" />
        </method>
        <method name="InvalidateAllUsers" key="True" />
        <method name="InvalidateAllGroups" key="True" />
        <method name="InvalidateAllInitgroups" key="True" />
        <method name="InvalidateGroupById" key="True">
            <arg name="gid" type="u" direction="in" key="1" />
        </method>
    </interface>
</node>
```

Listing 4.1: XML definition of the original NSS interface from the *sss_iface.xml* file.

A large set of macros is used to make use of the `sbus`-generated functions more straightforward. The central macro `SBUS_INTERFACE()` is used to define all other methods, signals, and properties by passing the macros `SBUS_METHODS()`, `SBUS_SIGNAL()`, etc.

These two macros both accept a variable number of arguments, and their count depends solely on the number of methods or signals that a given interface implements. The viable arguments for `SBUS_METHODS()` are `SBUS_SYNC()`, `SBUS_ASYNC()`, and `SBUS_NO_METHODS`. The last one is self-explanatory, as it discloses that the interface does not implement any methods. To describe the key difference between the first two, let's inspect the arguments of `SBUS_SYNC()`:

- `type` – describes the type of a method. One of three values can be assigned to it: `METHOD`, `GETTER`, or `SETTER`.

[4] https://www.freedesktop.org/wiki/Software/systemd/
[5] https://wiki.gnome.org/Projects/FleetCommander
[6] https://specifications.freedesktop.org/

- `iface` – the interface's name with underscores as a replacement to the dots in the actual name. They are used because the macro takes in the value as a token rather than as a string.
- `method` – the name of the D-Bus method.
- `handler` – the method's handler function in SSSD.
- `data` – any private data required by the handler function (context structures or `NULL`).

Similar to the way synchronous and asynchronous functions are differentiated in SSSD, the asynchronous macro `SBUS_ASYNC()` differs by requiring two `send` and `recv` handlers functions instead of a single one. During the reimplementation, the way methods were defined using these macros provided guidance while defining signals later on.

Moving onto the `SBUS_SIGNALS()`, this macro takes in a variable number of arguments of two types: `SBUS_NO_SIGNALS` and `SBUS_EMITS()`. In contrast to the method's macros, the `SBUS_EMITS()` only takes in two arguments: the name of the interface and the name of the signal. That is because the rest of the signal logic is implemented through signal listeners.

To add the listeners, `sbus` implements the `SBUS_LISTENERS()` macro, which follows the same structure as the previously mentioned `SBUS_METHODS()` and `SBUS_SIGNALS()`. It accepts two arguments, one for synchronous listeners: `SBUS_LISTEN_SYNC()`, and one for asynchronous ones: `SBUS_LISTEN_ASYNC()`. These are defined similarly to the method macros: the `method` argument is replaced by `signal`, they require the bus' `path`, and do not accept the `type` argument. To fully enable the listeners, it takes two more steps compared to the method or signal definitions:

1. The listeners must be stored in an array of `sbus_listener` structures. The macros that can be passed to `SBUS_LISTENERS()` expand into functions whose return value is the `sbus_listener` structures. Furthermore, the `SBUS_LISTENERS()` macro itself expands to an array, so their declaration in itself creates the required format.

2. This array then needs to be passed to the `sbus_router_listen_map()` function, which maps the defined listeners to the D-Bus, thus activating them.

After a thorough evaluation, the interface I have chosen for reimplementation is from the NSS responder: `sssd.nss.MemoryCache`. At first glance, its template, shown in Listing 4.1, suggested that it could be a candidate for transformation since none of its methods contained an *out* argument.

All the bus' `Invalidate.*` methods defined for this interface had handler functions defined in *nss_iface.c*. Their handler functions all had similar structures; they added a string to the logs notifying that they got triggered, called another function, and returned `EOK`[7]. This value was always returned, regardless of the outcome of the called function.

This behavior confirmed that this interface could be rewritten since it satisfied both of the requirements mentioned at the beginning of this section.

In comparison, the method `resInit` in `sssd.services` has arguments that suggest that there might be a possibility to reimplement it too. However, its handler function `data_provider_res_init()` returns a value depending on the outcome of a function it calls itself.

[7] This macro represents successful operation in SSSD and is equal to 0.

To start the process, the method definitions written in *sss_iface.xml* had to be replaced by signal definitions. The three methods: `InvalidateAllUsers`, `InvalidateAllGroups`, and `InvalidateAllInitgroups`, required simply changing the element's name from `method` to `signal`. As an additional modification, since the `InvalidateGroupById` accepts a single argument, I removed the `direction` attribute from its `arg` element. This slight modification was unnecessary, but since the `direction`'s value in signals is set to *out* by default, it would become redundant if not removed.

The last modification to its XML representation of the interface was removing the `key` attribute from the `signal` elements. This attribute indicated that `sbus` should add the generated request keys, described in Section 4.1.2, to the generated functions as arguments. As discussed earlier, this would enable the signals to be chained when SSSD would notice repeated requests. I have removed it because chaining coupled with signals resulted in unexpected behavior, whereas without the key present, everything worked without a problem. The resulting XML representation of the interface is shown in Listing 4.2.

```
<node>
    ...
    <interface name="sssd.nss.MemoryCache">
        <annotation name="codegen.Name" value="nss_memcache" />
        <annotation name="codegen.SyncCaller" value="false" />
        <method name="UpdateInitgroups">
            <arg name="user" type="s" direction="in" />
            <arg name="domain" type="s" direction="in" />
            <arg name="groups" type="au" direction="in" />
        </method>
        <signal name="InvalidateAllUsers" />
        <signal name="InvalidateAllGroups" />
        <signal name="InvalidateAllInitgroups" />
        <signal name="InvalidateGroupById">
            <arg name="gid" type="u" key="1" />
        </signal>
    </interface>
</node>
```

Listing 4.2: XML definition of the modified NSS interface from the *sss_iface.xml* file with the four `Invalidate.*` methods changed to signals.

Whenever any of the four XML files is modified, SSSD requires to be rebuilt so that `sbus` has the chance to regenerate the files in the *sss_iface* directory. Now that the `sssd.nss.MemoryCache`'s `sbus_call_.*` functions were replaced by the `sbus_emit_.*` variants, the code which used the former had to be rewritten.

Since the original methods were created for back-ends to interact with responders, the file where these functions were utilized was *dp_responder_client.c*. Previously the functions used the `tevent`'s callback-based way of dealing with subsequent sub-requests, shown in Figure 2.4. This way back-ends could receive the wanted, although unnecessary, reply from responders. I have removed both the original `sbus_call_.*` methods which assigned value to the `subreq` variable and removed the `tevent's` callback function. To finish this part, the `sbus_emit_.*` functions were added in the previous functions' places to enable the

back-ends to emit the respective signals. Two last things needed to be modified, or created, to finish up the reimplementation of the interface:

1. **Change the definition of the interface in *nss_iface.c*.** This required removing the `SBUS_NO_SIGNALS` macro from `SBUS_SIGNALS()` and using the `SBUS_EMITS()` macros in its place. I have added the values required (interface's name and signal's name) by copying the same values from the `SBUS_SYNC()` macros before removing them.

2. **Add signal listeners to the `sssd.nss.MemoryCache` interface.** As previously mentioned, the values which were previously passed to the method macros are used for defining listeners as well. Hence, I have utilized those as well, and all that was left was to call the function `sbus_router_listen_map()`.

This concluded the methods' transformation into signals and thus wrapped up SSSD's reimplementation.

Chapter 5

Performance evaluation

With the required changes to SSSD's topology and interface finished, it was now needed to find out whether these changes could be used in production. To answer that, a thorough performance analysis had to take place to see whether the product's performance regressed.

This chapter will describe the process of developing the performance test suite used for the mentioned analysis. There are two types of tests in it: response time measurements implemented with `SystemTap`, and benchmarks done through `hyperfine`.

Furthermore, the environment used to run the tests and the conditions that must be adhered to will also be discussed. With the test runs finished it will be also possible to come to conclusions about the proposed question of SSSD's performance.

5.1 Test suite development

To evaluate SSSD's performance, I have decided to use two tools discussed in the Chapter 3: `SystemTap` and `hyperfine`. The former one, `SystemTap`, is an excellent choice for measuring the response time of responder requests. Since there is now a single message bus responsible for managing all SSSD communication between back-ends and responders, it was required to check whether this substantial change affected SSSD's performance. As described in Section 3.1, `SystemTap` can examine code at explicitly specified points inside SSSD's code. In this thesis, this and other features were utilized to assess the exact times it takes to deal with a user-triggered request. Measurements were made by running the tests on both topologies and the captured times were compared to systematically evaluate whether SSSD's performance has changed.

To expand on the results acquired by `SystemTap`'s measurements, I also wanted to measure the entire process between the user request and received reply. Therefore, I have used the tool `hyperfine` to run benchmarks on the `id` command, which requests data from SSSD through its NSS responder.

The root directory of the test suite contains the following files and directories:

- *runner.py* – a Python script responsible for running everything in the test suite (apart from the figure plotting). User supplies data to this script through command-line parameters to modify the runs of both `SystemTap` and `hyperfine` tests.
- *requirements.txt* – a file containing all required libraries for the test suite. It is used to setup Python's virtual environment: `venv`.

- *util/* – a directory containing modules responsible for parameter parsing and data provider management.
- *systemtap/* – a directory with `SystemTap` probe script *sbus_stap.stp*, a Python module implementing functions to run the `SystemTap` tests. Additionally, it contains the file *evaluator.py* which plots the captured data.
- *hyperfine/* – similarly to the *systemtap* directory, it has both a Python module and a script for plotting data.

Before discussing the test runs, the two following subsections will discuss the process required to create the tests.

5.1.1 Measuring back-end response time with `SystemTap`

To add even a single `SystemTap`'s probe points, prep-work needs to be done to ensure that it works as expected in a project this large. Fortunately, as with the macros used for SSSD's interfaces, the groundwork for `SystemTap` has been laid out as well already.

As mentioned in Section 3.1, to enable `SystemTap`'s abilities in a project, configuration through *autotools* and `dtrace` is necessary. Before each build, users can decide whether they want to have SSSD with `SystemTap` functionality by using (or not using) the `--enable-systemtap` while using `configure` to prepare the project. This, coupled with the SSSD's implementation of macros for defining probes in the *probes.h* and more, has been highly inspired by the `SystemTap` documentation.

However, SSSD expands on the concept of macro definitions of probes. Thus, the `PROBE(NAME, ...)` macro (shown in Listing 5.1) is used when adding a probe. This macro takes in a variable number of arguments and, during build-time expands to the format required by SSSD. The format is defined by *sbus_generated_probes.h*, an auto-generated file, whose macro formats are declared in the *sssd_probes.d* file and generated by `dtrace`.

```
#define PROBE_EXPAND(NAME, ...) NAME(__VA_ARGS__)

#define PROBE(NAME, ...) do { \
    if (SSSD_ ## NAME ## _ENABLED()) { \
        PROBE_EXPAND(SSSD_ ## NAME, \
                     __VA_ARGS__); \
    } \
} while(0);
```

Listing 5.1: Definition of macros with a variable number of arguments used for `SystemTap`'s user-space probes. Taken from `SSSD`'s codebase.

As previously mentioned, since SSSD can be compiled with or without `SystemTap`'s functionality enabled, the file *probes.h* defines an alternative macro function. This function expands the macros inside SSSD's functions into an empty string, and thus leaves them out of the binary altogether.

The following list introduces the places where the response time measurements could take place and the tests' possible scenarios:

- *SSH responder* – use `ssh` to connect to one of the remote data providers.

  - Measure the time between the SSH responder requests access for the currently logged in user and the back-end's reply.

- *PAM responder* – login into one of the existing users with the `su` command.

  - Measure the time between the PAM responder requests of authentication of the specified user and the back-end's reply.

- *NSS responder* – request user data through the `getent` command from the command-line and expect a reply.

  - Measure the time between the NSS responder request data about the specified user and the back-end's reply.

The first two scenarios, while seemingly viable, lead too too much overhead between the responder receiving the user's request, forwarding it to the back-end, and receiving a reply.

The NSS scenario, on the other hand, is straightforward; the request from the user gets forwarded to the requested back-end almost without any *outside* influences. When the responder receives the request, SSSD only uses the bare minimum of its capabilities, as shown in Figure 5.2.

While evaluating the request flow, I noticed that the best place to insert the probes was in the *responder_dp.c* file. The two functions, `sss_dp_get_account_send()` and `sss_dp_get_account_done()`, are the last place before the responder starts *building* the request structure as well as the first place that processes the reply from the back-end, respectively. Therefore, I have inserted markers by adding the `PROBE()` macro to those two functions and started the work required to measure the actual time.

Even though I could now measure the start and stop time whenever these two probes were triggered, `SystemTap` had no way of knowing whether the request which stopped the timer was the same one that started it. This could lead to problems since the requests originating from the test suite will not necessarily be the only ones. After observing the whole process that the request goes through, I came up with a solution to differentiate between the requests that required four more probes to be added to SSSD.

```
NSS_SEND: Got request for: admin@ipa.test
  SSS_SEND: Sending request: admin@ipa.test to ipa.test
    BUS_SEND: Method: getAccountInfo
    BUS_DONE: Received reply from: ipa.test
  SSS_DONE: Outcome: Success
NSS_DONE: Received reply for: admin@ipa.test
```

Listing 5.2: A look at the process through which a `getent admin@ipa.test` request goes through. Each line states the name of its marker and the variable that the probe acquires from the marker.

The following list describes those four along with the original two probes:

- `nss_getby_name_send` (*`nss_getby_name()`*) – the first place which gets triggered after a successful *getent passwd* command. The purpose of probing this function is to acquire the initial format of the requested username. This value is used throughout the script to make sure the probes do not get triggered by any other request during the test's run.

- `sss_dp_send` (*`sss_dp_get_account_send()`*) – here the script stores the request's start time (in microseconds). Furthermore, it also stores the name of the target data provider needed by the `sbus_req_call_done` probe.

- `sbus_req_call_send` (*`sbus_call_method_send()`*) – this probe serves as a way to check whether the request was successfully built. Furthermore, when starting the script in a verbose mode (used for debugging), it deals with all the random nested D-Bus methods that tend to get triggered.

- `sbus_req_call_done` (*`sbus_call_method_done()`*) – this is the first probe which gets called after the back-end replies to the request. The script checks if the request's target corresponds to the script's saved `curr_conn_name` (acquired in the `sss_dp_send` probe) and sets the `can_stop` variable. It also checks the variable `curr_bus`, containing bus' name, since in the previous topology, the `sss_domain_info->conn_name` is not set to the newly required value. This variable signals to the `sss_dp_done` probe that the request is the right one, and it can stop the timer once it returns to the place of origin.

- `sss_dp_done` (*`sss_dp_get_account_done()`*) – the script saves the time that it took for the responder to receive a reply when this probe is triggered. The time is calculated by subtracting the time acquired in `sss_dp_send` from the current time.

- `nss_getby_name_done` (*`nss_getby_done()`*) – this probe is chronologically the last probe of the whole process. Therefore, it is responsible for storing the time calculated in `sss_dp_done` to both the associative array `results[nss_getby_rawname]` and the `C` array. After the values are stored, it resets the variables used to measure the time and thus enabling another request to be measured.

Once the script could reliably measure the request's response time, the next step in the `SystemTap`'s implementation was to figure out how to generate output in a suitable format. Whereas many tools have built-in functionalities to generate output in multiple file formats, SSSD can only use `stdout`. Either by printing to it directly or redirecting its output to a file when the `-f` parameter is specified when running the `stap` command. Once I had finished the work on the probe functions and acquired the required data, it was time to format it for output.

I have decided that the most elegant solution would be to format the measured values so that the `SystemTap`'s output would produce a makeshift CSV file, as shown in Listing 5.3. That is why, as mentioned previously, the times were stored in two different arrays:

```
ipa.test,123,13213,3123,31231,312313
ldap.test,123,13213,3123,31231,312313
samba.test,123,13213,3123,31231,312313
wrong@ipa.test,123,13213,3123,31231,312313
```

Listing 5.3: An example of a file created by the `SystemTap` script. The first value represents the name of the data provider with the requested user; the rest are the actual request times.

- *`C` array* – the time of requests get stored and sorted into arrays depending on the provider they are targeting. Thus, the script has four different arrays: `ipa_results`, `ldap_results`, `samba_results`, and one used for failed requests, `wrong_results`.

Once the `end` probe gets triggered, the values inside these arrays are printed out separated by commas.

- *`SystemTap`'s associative array* – although this array stores all values as well, it is only used as a support for the C arrays. The times get stored with the instruction: `results[provider] <<< time`, which ensures that the script can, in its final stages, use this array as a way to check which providers were present during a test run. As mentioned in Subsection 3.1.2, the script could not use this array directly to create an output because the operations which are possible with associative arrays are limited.

Once the output formatted and saved in a file, the CSV can be easily parsed by Python's data-processing `pandas` library which can then in-turn be used by another Python's library `seaborn` to visualize the stored data in many different ways.

5.1.2 Benchmarking with hyperfine

Creating tests with `hyperfine` was less complex than the process described in previous Subsection 5.1.1. That is the case because `hyperfine` in contrast with `SystemTap` takes SSSD as a black-box. Thus it measures SSSD's response times from outside only by repeatedly using a command which generates requests to be processed.

The *runner.py* script takes four parameters regarding the test's configuration:

- `-run-hyperfine` – the switch which tells the script whether to run the tests or not,
- `-hf-parameters` – the list of usernames to use in the request,
- `-hf-runs` – the number of runs which will `hyperfine` do per each request,
- and `-hf-output` – the name of the JSON file to which `hyperfine` outputs the captured results.

After their values; validation, all of them together are passed to the `Popen`[1] function to start the test run. For example, by running the script with:

```
$ python runner.py --run-hyperfine \
--hf-parameters admin@ipa.test admin@ldap.test --hf-runs 100
```

The final `hyperfine` command looks like this:

```
$ sudo hyperfine -i -r 100 \
--parameter-list user admin@ipa.test,admin@ldap.test 'id {user}'
```

After `hyperfine` has finished its run, the measured data is exported to a specified file, later used in test evaluation.

`Hyperfine` measurements work better when running commands which take longer than 5 milliseconds on average (as it points out during its run). That is the reason why commands for testing response times and benchmarking differ. While `getent` was a great candidate for `SystemTap` thanks to its minimal overhead during the request's lifetime, it was less than ideal for `hyperfine`. The average time of the `getent` request was 0.5 milliseconds, so the `id`'s average of 50 milliseconds was more suitable for this use-case.

[1] https://docs.Python.org/dev/library/subprocess.html?#subprocess.Popen

5.2 Test result evaluation

After the implementation of the test suite was finished, what followed was to use it to evaluate the two topologies of SSSD. The tests were run on a machine with an *Intel i7-10610U* processor and 32 GB of RAM with data providers running in local containers. As previously mentioned, both SSSD and `SystemTap` are developed by Red Hat and thus work best in Fedora and RHEL so the machine was running a *Fedora release 35* as well. This, coupled with the FreeIPA's installation commands created for the same environment, made Fedora the best choice.

There are two rules regarding the data provider setup which must be adhered to so that the tests can be run successfully:

- **There can be only one instance of each type of data provider** (FreeIPA, LDAP, AD DS). The expected setup is: one to three connected providers to which the tests send requests. The test suite has the ability to disconnect from providers to comply with the ones chosen on the command-line. It can test SSSD's performance with each provider type, so having more than one instance of the same one would not produce any additional interesting results. Since the test suite can either test all providers at once, one at a time, or any other combination, it is enough to evaluate SSSD's performance.
- **Each provider instance must be named either `ipa.test`, `ldap.test`, or `samba.test`**. To comply with `SystemTap`'s algorithm to keep track of the different providers, these names are pre-configured in the test suite. It is not a limitation since a provider's name can be set in the *sssd.conf* by specifying it in the `[domain/<provider name>]` statement without any other modifications. There is an additional step of configuration regarding the LDAP provider. Since there is no command which can automatically setup SSSD to connect to it, when using any LDAP server with a different domain name a configuration snippet has to be supplied through command-line to the test suite.

The mentioned providers can be remote servers, local virtual boxes, or containers since the test suite manages them through SSSD and `ipa-client-install` and `realm` commands. The containers[2] used during the development of the test suite were implemented by the SSSD team and provided a faster set up.

5.2.1 `SystemTap` results

As mentioned in Section 5.1.1, the data used for the graphs, shown in Figure 5.1 (run **A**) and Figure 5.2 (run **B**), come from values in the CSV file. This file was read with the `pandas` library and plotted with `matplotlib` and `seaborn`.

To run the tests, the following parameters may be passed from command-line to the central script *runner.py*:

- `--run-systemtap` – the switch used to start the `SystemTap` test execution,
- `--stap-output` – the name of the output CSV file,
- `--stap-request-count` – the number of requests which the script should send out to SSSD while `SystemTap` is running,

[2] https://github.com/SSSD/sssd-ci-containers

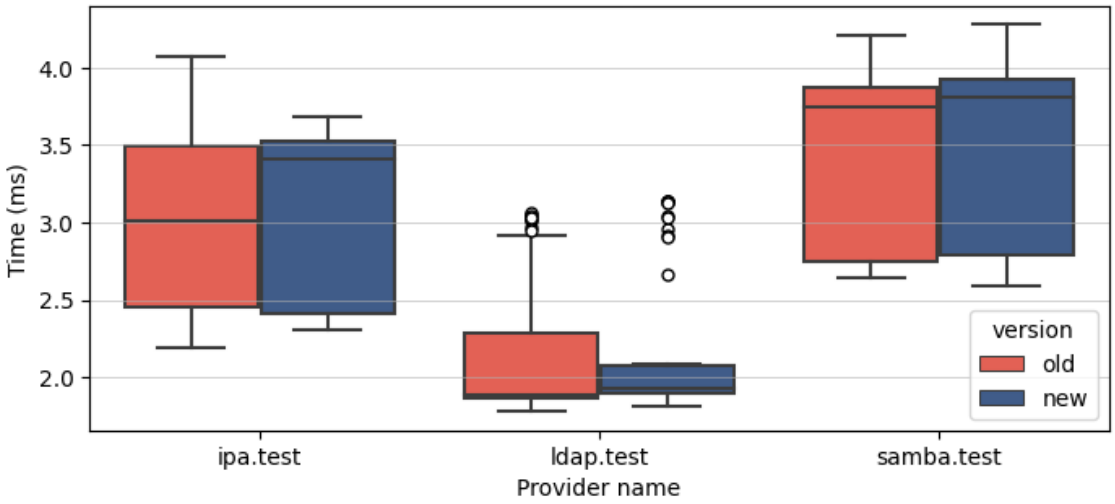

Figure 5.1: Run A: The graph shows `SystemTap`'s test results in a run for all three providers with 50 requests each, measuring both the new and the old topology.

- and `--stap-verbosity` – the switch for verbosity of the `SystemTap` script (used for debugging purposes only).

When started, the central script triggers a warm-up run with 5 requests to each backend to reduce the number of outlier results. After the warm-up rounds are finished, the script starts `SystemTap` and sending out requests to each of the selected data responders.

The data is depicted using boxplots since it has multiple features, which are ideal for comparing performance data in this scenario. With a few shapes, they describe multiple statistical values:

- *Median* – the horizontal black line going through each of the rectangles.
- *First* (Q1) and *third* (Q3) *quantile* – the bottom and the top sides of rectangles.
- *Interquartile range* (IQR) – the range of each rectangle on the y axis.
- *Minimum*[3] – the bottom whisker.
- *Maximum*[4] – the top whisker.
- *Statistical outliers* – the points which are spread along the y axis beyond the range of the whiskers.

In both runs, the IQR of results captured for the `ipa.test` is slightly broader for the new topology, suggesting a slightly higher number of unique times measured. Admittedly, that is where the similarities between the two runs end. The run **A** on the new topology has a more dense concentration of time values towards the top of the y axis, as can be seen from the location of its median. The values are set in a significantly narrower range than

[3] $Q1 - 1.5 * IQR$
[4] $Q1 + 1.5 * IQR$

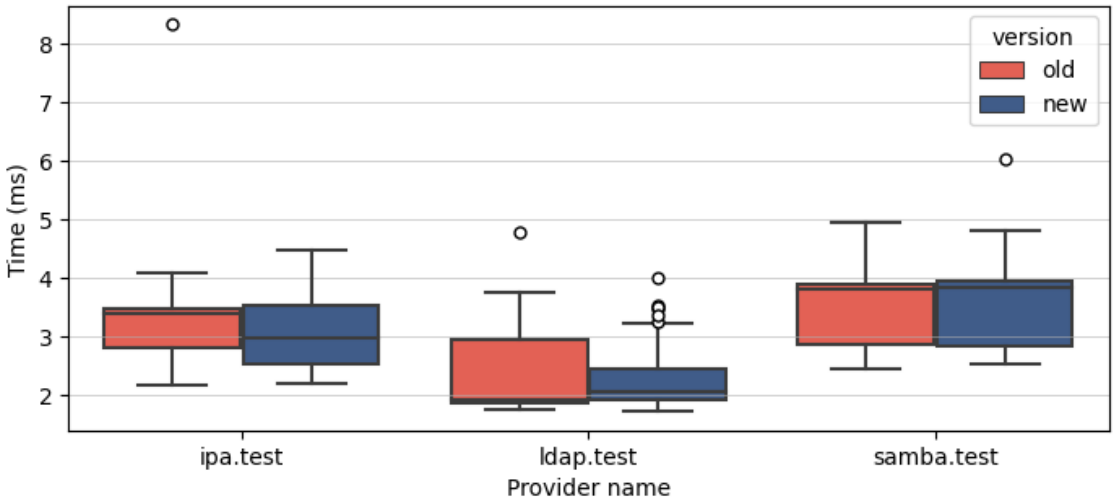

Figure 5.2: Run B: The graph shows `SystemTap`'s test results in a run for all three providers with 750 requests each, measuring both the new and the old topology.

the older topology. That changes for run **B**, as the results show a much broader range of values, and in this case, the median is lower in the new topology than in the old one.

Similar to the previous data provider, the results for `samba.test` do not show any radical change. In this case, the results are almost identical in both runs for both topologies, with the newer one being slightly slower than the old one. Through all of the test runs, it was clear that requests routed to AD DS tend to take longer than others, so the findings might suggest that the `samba.test` is a bottleneck.

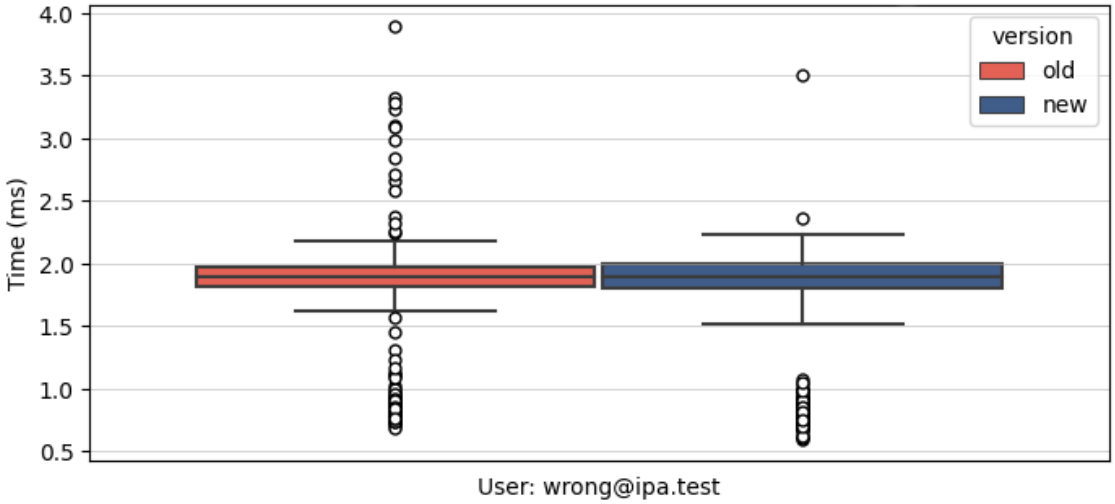

Figure 5.3: Graph shows `SystemTap`'s test results containing only runs where the test suite requested non-existent users.

Conversely, the `ldap.test` results show an apparent distinction between the topologies in both runs. The results indicate that the new topology might have helped stabilize the response times in the case of LDAP. This indication can be read clearly from the measured times' range on the new topology, which is much more narrow than the old one. Furthermore, the concentration of 99% of the results spans across such a narrow min-max range of the lower time values for this provider. This causes the results of the tests for LDAP to be the only one that shows multiple instances of statistical outliers in the higher values of the y axis.

To demonstrate the response times for a non-existing user, the test suite requests data for the user `wrong@ipa.test`. Its results demonstrate, as shown in Appendix 5.3, that while the IQR and median stayed the same for both topologies, the higher valued outliers are almost eliminated in the newer topology. This hints at the possibility that the change might have accelerated the process of dealing with error scenarios.

I used the `time` command to better grasp what possibilities `SystemTap`'s features unlocked for SSSD's evaluation. Over each of the runs of type **B**, the test suite has made 3000 requests (750 with a non-existent user in addition to 750 per responder), and the time it took to go through them all was six minutes on average. This shows us that with all of the SSSD's preparations, routing logic, and other types of overhead, the reply for each request took, on average, 0.12 seconds (calculated from the `time` measurements). That time is much longer than any of the times recorded through `SystemTap`. It shows that thanks to its ability, the tests were able to ignore any overhead in SSSD and focus only on the communication through the message bus.

5.2.2 `Hyperfine` results

Examining the results of both Figure 5.5 and Figure 5.4 is more straightforward than the previous measurements.

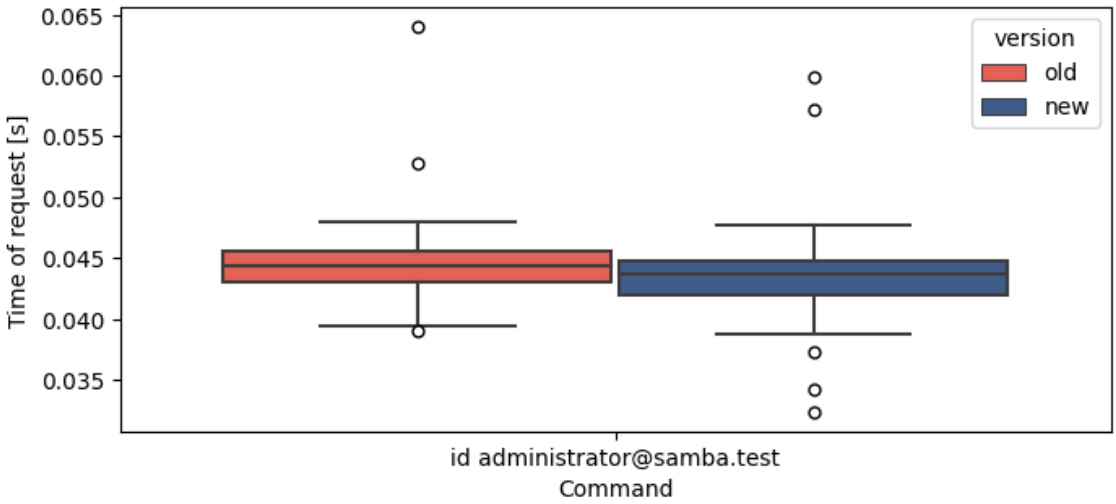

Figure 5.4: Graph shows `hyperfine`'s test results for the AD DS provider repeated 100 times. This request has been separated from the others since its measured times are almost quadruple the values measured on the other data providers or non-existent users.

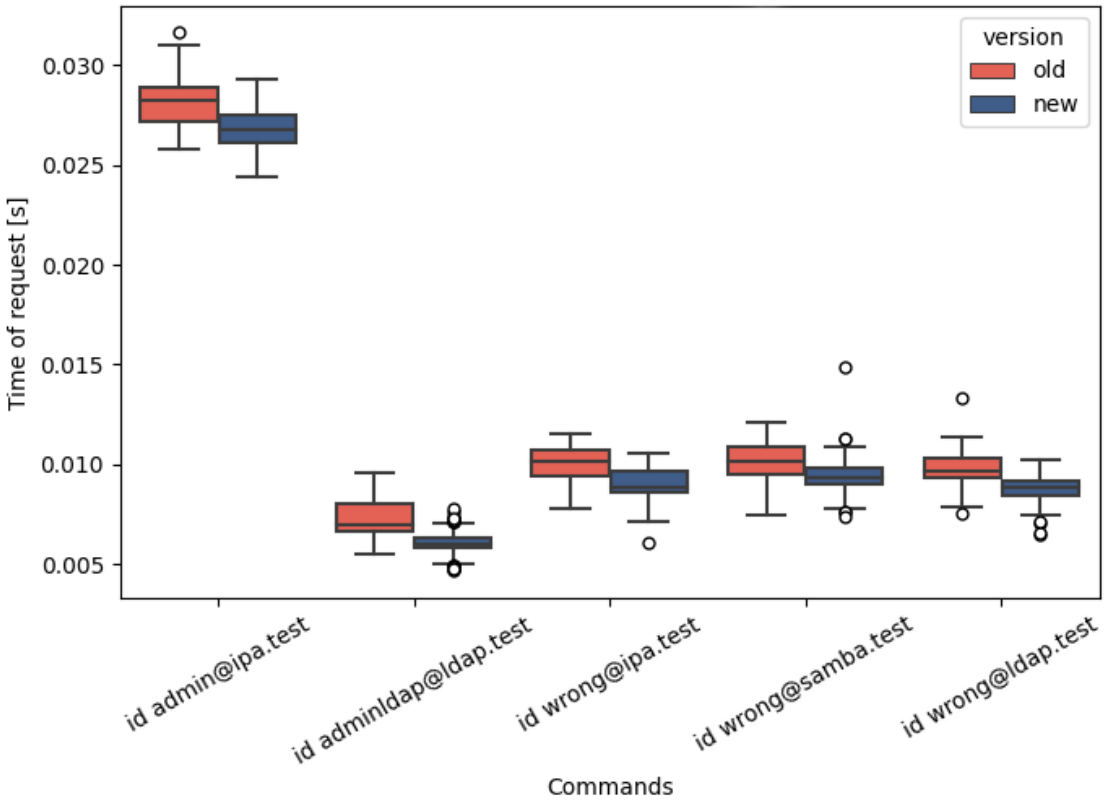

Figure 5.5: Graph shows `hyperfine`'s test results for 5 different commands each, repeated 100 times, for both the new and the old topology.

The ranges of the values measured by `hyperfine` while requesting data with the `id` command, stay almost identical in the case of IQR as well as between the minimum and maximum. As for the differences between the topologies, the new topology seems to contain a lot more statistical outliers in the lower and higher ranges of time values. Additionally, the new topology seems to have achieved overall speedup of the request service process. The measurements for all providers and non-existent users show a significant decrease in the durations.

The original prediction for the measurements of SSSD's performance was that after the changes made to its topology, it could service the requests faster or at least similarly fast as in the old topology. The prediction was based on the fact that with newer versions of the Fedora distribution, the `dbus-broker` replaced the original `dbus-daemon`. This new implementation of the D-Bus promised higher speeds[5] and upgraded reliability over the old one.

The results of `SystemTap` have shown us that in some cases, these higher speeds did not matter (FreeIPA, AD DS). However, in some cases, they accelerated and stabilized the process. As for the results of benchmarking with the `id` command, there is a clear distinction between the old and new topology.

[5] https://fedoraproject.org/wiki/Changes/DbusBrokerAsTheDefaultDbusImplementation

Looking at both the response time tests and benchmarks, it can be concluded that the performance of SSSD has not regressed, and in some cases, it is faster. In conclusion, the performance of the topology is almost identical if not better, than in the previous topology of the D-Bus communication. Hence, the goal of achieving a comparable, or better, performance has been accomplished.

Chapter 6

Conclusion

The goals of this thesis were to reimplement the SSSD's message bus topology, modify its interface to utilize the bus' new capabilities better, and measure its performance to ensure these changes would not negatively impact it.

In this thesis, I have described the inner workings of SSSD and its main data provider services: FreeIPA, AD DS, and LDAP. To illustrate the concept of SSSD's communication through D-Bus, I have also presented the concept of topologies and their applications in distributed systems. These building blocks were then used in redesigning and implementing the SSSD communication process based on a star topology. Next, I demonstrated the newly unlocked possibilities of the message bus by implementing signals to one of the interfaces. The inclusion of signals makes the complexity of the communication even lower since SSSD does not have to manage unnecessary method returns.

Moreover, the thesis presents the concepts of performance analysis and test development concerning performance. I have evaluated and tried many performance analysis tools and utilized two that were the best suited for the performance evaluation of SSSD. Through them, it was possible to assess whether the architectural changes made have caused SSSD's performance to regress. The results of this performance evaluation suggest that the performance of SSSD is comparable, if not better, than in its previous version.

Future evaluation of SSSD's interfaces is recommended to expand on the work I started. There is still room to upgrade other interfaces, and as this thesis shows, their reimplementation is now possible.

Regarding the test suite, load testing would enable an even better assessment of the SSSD's state. Unfortunately, none of the load testing tools that were considered and evaluated provided the ability to use them in this use case. They considered only remote requests over the HTTP protocol and in no way provided a way to use basic shell commands aimed at a local target. Therefore, as one of the possible ways to built upon this thesis, I suggest creating of a small load testing framework. This framework should be able to use any shell command and run it in several concurrent processes to measure the load the product can handle.

Bibliography

[1] Bokovoy, A. *Extending FreeIPA* [online]. November 2011 [cit. 19.1.2022]. Available at: https://abbra.fedorapeople.org/guide.html.

[2] Březina, P. *SSSD documentation*. November 2020 [cit. July 28, 2022]. Available at: https://sssd.io/index.html.

[3] Cohen, W., Domingo, D., Slávik, V., Kratky, R. and East, J. *Red Hat Enterprise Linux 7: SystemTap Beginners Guide* [online]. April 2019, Revised 29.9.2020 [cit. 25.3.2022]. Available at: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/pdf/systemtap_beginners_guide/red_hat_enterprise_linux-7-systemtap_beginners_guide-en-us.pdf.

[4] Dewata, E. S. *DogTag PKI* [online]. Red Hat, Inc., september 2020, Revised 11.10.2021 [cit. 17.1.2022]. Available at: https://github.com/dogtagpki/pki/wiki.

[5] Eigler, F. and Cohen, W. *Adding User Space Probing to an Application (heapsort example)* [online]. December 2009. Revised 15.7.2020 [cit. 19.3.2022]. Available at: https://sourceware.org/systemtap/wiki/AddingUserSpaceProbingToApps.

[6] Foulds, I. and Bahall, D. Active Directory Domain Services Overview. *Identity and Access documentation* [online]. 1st ed. may 2017, f052dfcd-dace-4485-8d0a-cc7df5cf3751, [cit. 29.1.2022]. Available at: https://docs.microsoft.com/en-us/windows-server/identity/ad-ds/get-started/virtual-dc/active-directory-domain-services-overview.

[7] *FreeIPA documentation* [online]. August 2009 [cit. 17.1.2022]. Available at: https://www.freeipa.org/.

[8] Gregg, B. *Systems Performance: Enterprise and the Cloud*. 2nd ed. Upper Saddle River, NJ: Pearson, january 2021. Addison-Wesley Professional Computing Series. ISBN 0-13-682015-8.

[9] Minar, N. *Distributed Systems Topologies: Part 2*. January 2002 [cit. 5.3.2022]. Available at: http://www.mba.intercol.edu/MBA731/DistSysTopologies.pdf.

[10] Muehlfeld, M., Bokoč, P., Čapek, T. and Ballard, E. D. *Red Hat Directory Server 11: Deployment Guide* [online]. Red Hat, Inc., november 2019, Revised 9.11.2021 [cit. 17.1.2022]. 184 p. Available at: https://access.redhat.com/documentation/en-us/red_hat_directory_server/11/pdf/deployment_guide/Red_Hat_Directory_Server-11-Deployment_Guide-en-US.pdf.

[11] Pennington, H., Carlsson, A., Larsson, A., Herzberg, S., McVittie, S. et al. *D-Bus Specification* [online]. 0.37. September 2003. Revised 17.12.2021 [cit. 26.12.2021]. Available at: https://dbus.freedesktop.org/doc/dbus-specification.html.

[12] Peter, D., Arrais, T., Gnatenko, I., Kudryavtsev, C., Schäfer, D. et al. *Hyperfine* [online]. January 2018 [cit. 31.3.2022]. Available at: https://github.com/sharkdp/hyperfine/blob/master/README.md.

[13] Software Freedom Conservancy, Inc. *Talloc documentation* [online]. May 2012. Available at: https://talloc.samba.org/talloc/doc/html/index.html.

[14] Software Freedom Conservancy, Inc. *Tevent documentation* [online]. 2012. Available at: https://tevent.samba.org/.

[15] Ts'o, T., Raeburn, K., Yu, T. and Hudson, G. *What is Kerberos?* [online]. December 1996. Revised 25.7.2021 [cit. 19.1.2022]. Available at: https://web.mit.edu/kerberos/.

Appendix A

Contents of the included storage media

The media storage includes:

- **sssd/** – The directory containing the source code of SSSD. It is a git project which includes the branches:
  - *original* – the original version with added `SystemTap` markers used by the performance test suite.
  - *centralized-sbus* – the version with the new topology, newly implemented signals, and `SystemTap` markers.
- **sssd-perf/** – The directory containing the performance test suite.
- **xuradn02-thesis.pdf** – This thesis.
- **README.md** – The file containing descriptions of files stored in the media storage.

Appendix B

Manual

Because of the nature of the products and programs this thesis is utilizing and modifying, the *Fedora* distribution is recommended for SSSD and required for the test suite.

B.1 SSSD installation

After opening SSSD's source directory, the following steps need to be taken to install it:

1. Source the preconfigured shell functions: `$ source contrib/fedora/bashrc_sssd`.
2. Configure and build it: `$ reconfig && chmake`.
3. Install SSSD: `$ sssinstall` (requires sudo privileges).
4. Update the service in `systemd`: `$ systemctl daemon-reload`.
5. Restart SSSD: `$ systemctl restart sssd.service`.

B.2 Test execution

The tests for `SystemTap` and `hyperfine` are both run through the *runner.py* script located in the root directory of *sssd-perf*. First, the environment needs to be set up. The preferred steps are written in the list below:

1. Clone the repository with the data provider containers:
   `$ git clone https://github.com/SSSD/sssd-ci-containers`.
2. Start the containers: `$ cd sssd-ci-containers; make up`
3. Connect to FreeIPA: `$ ipa-client-install --unattended --no-ntp --domain ipa.test --principal admin --password Secret123 --force-join`
4. Connect to AD DS: `$ echo Secret123 | realm join samba.test`
5. Connect to LDAP server by appending the contents of *sssd-perf/systemtap/conf/sssd-ldap.conf* to */etc/sssd/sssd.conf*.
6. Restart SSSD: `$ systemctl restart sssd.service`

To run the tests for both scenarios use the following commands:

- `python runner.py --run-systemtap` – This command is used for the default run of the `SystemTap`'s part of test suite. It will start the *sssd-perf/systemtap/sbus_tap.stp* script with verbosity turned off and send 5 request to each of the three providers. Furthermore, the results will be saved to the *sssd-perf/systemtap/csv/stap.csv* file.
- `python runner.py --run-hyperfine` – Runs the default configuration of the benchmarks. After `hyperfine`'s start, it will send out 10 `id` requests to ask for data regarding three users from three different back-ends and three non-existent users. It will save the measured data to *sssd-perf/hyperfine/json/hf.json* after the tests finish.

Further customizability is described in the project's *README.md*.