VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
DEPARTMENT OF COMPUTER SYSTEMS

PLUGINY PRO EFEKTIVNÍ ÚLOŽIŠTĚ DAT V KNIHOVNĚ SYSREPO

PLUGINS FOR EFFICIENT DATASTORE IN THE SYSREPO LIBRARY

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **ONDREJ KUŠNÍRIK**
AUTHOR

VEDOUCÍ PRÁCE **Ing. JIŘÍ MATOUŠEK, Ph.D.**
SUPERVISOR

BRNO 2024

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Zadání bakalářské práce

154310

Ústav: Ústav počítačových systémů (UPSY)
Student: **Kušnírik Ondrej**
Program: Informační technologie
Název: **Pluginy pro efektivní úložiště dat v knihovně sysrepo**
Kategorie: Databáze
Akademický rok: 2023/24

Zadání:

1. Seznamte se s knihovnou sysrepo a jejím mechanismem pluginů pro úložiště dat.
2. Porovnejte různá potenciální úložiště dat a pro knihovnu sysrepo vyberte dvě nejvhodnější varianty z pohledu efektivity práce s daty.
3. Navrhněte způsob použití zvolených úložišť dat v knihovně sysrepo ve formě pluginů.
4. Implementujte navržené pluginy pro knihovnu sysrepo.
5. Proveďte měření rychlosti běžných operací se stávajícím a dvěma novými pluginy.
6. Porovnejte a zhodnoťte naměřené hodnoty a diskutujte možnosti dalších optimalizací.

Literatura:

- Dle pokynů vedoucího a konzultanta práce.

Při obhajobě semestrální části projektu je požadováno:
Splnění bodů 1 až 3 zadání.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Matoušek Jiří, Ing., Ph.D.**
Konzultant: Mgr. Michal Vaško
Vedoucí ústavu: Sekanina Lukáš, prof. Ing., Ph.D.
Datum zadání: 1.11.2023
Termín pro odevzdání: 9.5.2024
Datum schválení: 30.10.2023

Abstrakt

Tato práce se zabývá seznámením čtenáře s knihovnou **sysrepo** a jejími pluginy pro úložiště, problémy s ukládáním dat do souborů a řešením těchto problémů napojením **databáze** na knihovnu sysrepo. Ve zkratce se tato knihovna využívá jako komplexní úložiště konfiguračních dat **YANG** v unixových/linuxových systémech. Data, která výchozí plugin knihovny ukládá do souborů, nemohou být dostatečně rychle a efektivně spravována, což způsobuje celkové zpomalení práce s daty. Jako řešení se nabízí použití databáze, která bude na knihovnu sysrepo napojena za pomoci implementace pluginu pro úložiště. Porovnáním jednotlivých databází byly zvoleny dvě (**MongoDB** a **Redis**) z hlediska efektivity práce s daty YANG a dalších kvalitativních vlastností. Pro tyto databáze byly následně implementovány **pluginy pro úložiště**, které byly nakonec podrobeny optimalizacím a výkonnostním testům. V porovnání s původním pluginem založeným na souborech vynikají implementované pluginy založené na databázích především v operacích s malým množstvím dat, kdy například při načítání jednoho prvku ze statisíc dochází ke zrychlení až o tři řády.

Abstract

This work concerns an introduction to the **sysrepo** library and its datastore plugins, problems with storing data to files and solving these problems by connecting a database to the sysrepo library. In short, this library serves as a complex repository for configuration **YANG** data on Unix/Linux systems. Data, which the default plugin stores in files, cannot be managed fast and efficiently enough causing overall slowdown of data management. A **database** connected to the sysrepo library via a datastore plugin could however solve this issue. After comparing different databases, two were selected (**MongoDB** and **Redis**) based on work efficiency with YANG data and other qualities. The **datastore plugins** were then implemented for these databases, optimized and tested for performance at the end. In comparison to the original plugin based on files, the plugins based on databases primarily excel at management of low amounts of data, where for instance the loading of an element from one hundred thousand is faster by up to three orders of magnitude.

Klíčová slova

sysrepo, YANG, databáze, plugin pro úložiště, MongoDB, Redis

Keywords

sysrepo, YANG, database, datastore plugin, MongoDB, Redis

Citace

KUŠNÍRIK, Ondrej. *Pluginy pro efektivní úložiště dat v knihovně sysrepo*. Brno, 2024. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Jiří Matoušek, Ph.D.

Pluginy pro efektivní úložiště dat v knihovně sysrepo

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Jiřího Matouška, Ph.D. Další informace mi poskytl pan Mgr. Michal Vaško ze sdružení CESNET, který byl odborným konzultantem práce. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....
Ondrej Kušnírik
5. května 2024

Poděkování

Rád bych poděkoval svému konzultantovi Mgr. Michalu Vaškovi za seznámení s knihovnou sysrepo, za trpělivý přístup při řešení praktických problémů a za užitečné rady a nápady při vypracovávání této práce. Dále bych také rád poděkoval vedoucímu Ing. Jiřímu Matouškovi, Ph.D. za prospěšné rady a poznámky a za vzorné vedení bakalářské práce.

Obsah

1 Úvod **2**

2 Seznámení s knihovnou sysrepo a pluginy pro úložiště **3**
2.1 Seznámení s modely jazyka YANG 3
2.2 Jak knihovna sysrepo funguje 5
2.3 Druhy úložišť knihovny sysrepo 5
2.4 Druhy odběrů v knihovně sysrepo 6
2.5 Pluginy pro úložiště knihovny sysrepo 8
2.6 Notifikační pluginy knihovny sysrepo 9
2.7 Problémy výchozího úložiště 9

3 Seznámení s databázemi a volba vhodných kandidátů **11**
3.1 Srovnávací testy 12
3.2 Grafové databáze 13
3.3 Sloupcové databáze 14
3.4 Dokumentové databáze 14
3.5 Klíč-hodnota databáze 17

4 Návrh a implementace pluginů pro databáze **20**
4.1 Napojení na databázi 20
4.2 Nové funkce pro pluginy 20
4.3 Způsob ukládání dat YANG v databázi 21
4.4 Načítání dat 21
4.5 Ukládání dat 22
4.6 Přístupová práva 22
4.7 Poslední modifikace 23
4.8 Kandidátní úložiště 24
4.9 Uživatelem řazené seznamy 24
4.10 Značka výchozího stavu 27
4.11 Úložiště operačních dat 27

5 Optimalizace a zhodnocení výsledných rychlostí pluginů **28**
5.1 Náměty na optimalizace a optimalizace operací s daty 28
5.2 Zhodnocení výsledných rychlostí pluginů 30

6 Závěr **37**

Literatura **38**

Kapitola 1

Úvod

Sysrepo (z angl. System Repository) je knihovna napsaná v jazyce C pro ukládání konfiguračních a operačních dat YANG [21]. **YANG** je modelovací jazyk pro vytváření obecné struktury (modelů) konfiguračních a stavových dat, speciálně vytvořený zejména pro data, se kterými pracují protokoly pro správu sítě, např. NETCONF (z angl. Network Configuration Protocol) [14]. **Protokoly pro správu sítě** využívají zejména některé firmy pro rychlou a snadnou vzdálenou konfiguraci síťových zařízení (např. směrovačů). Knihovnu sysrepo lze tedy využít jakožto centrální úložiště pro tato konfigurační data. Knihovna ovšem není nutně vázána pouze na správu sítí a je možné do ní uložit data jakéhokoli modelu YANG.

Řešený problém knihovny sysrepo spočívá v tom, jak jsou data ukládána. U dosavadní implementace pro data každého modelu YANG existuje právě jeden **soubor**, který je schraňuje. Toto řešení se vyplatí z důvodu jednoduchosti a nevelké časové složitosti implementace. Zde ale výhody tohoto řešení končí. Načítání dat z takového souboru musí být vždy prováděno celistvě. To znamená, že obsah celého souboru musí být načtený do paměti a až pak lze provádět změny. Prosté vyhledání nebo úprava pouze jedné informace (například 32bitového čísla) má za následek zdlouhavé načtení dat celého modelu.

Možným řešením tohoto problému je právě **databáze**. Ta by zabránila zdlouhavému načítání dat celého modelu, jelikož k částem modelu by se přistupovalo jednotlivě. Rozdíly mezi rychlostmi databáze a dosavadní implementace pomocí souborů budou nejvíce viditelné při malých změnách na datech (např. přepsání jednoho čísla v celých datech). Naopak nejméně viditelné budou při změně všech dat modelu, protože jak soubor, tak databáze budou muset načíst všechna data modelu. Tudíž nejdůležitějším měřítkem při volbě vhodné databáze bude zejména rychlost, s jakou je schopna měnit všechna data modelů YANG. Nyní totiž právě tato operace trvá potenciálně nejkratší dobu, a proto je kladen zřetel na to, aby se rychlost této operace blížila rychlosti dosavadní implementace.

V první kapitole se čtenář seznámí s knihovnou sysrepo a jejím mechanismem pluginů pro úložiště. V druhé kapitole jsou na základě požadavků pro ukládání dat a na základě dalších kvalitativních vlastností vybrány dvě nejvhodnější databáze (**MongoDB** a **Redis**). Ve třetí kapitole jsou popsána úskalí samotné implementace a ve čtvrté kapitole jsou závěrem navrženy a provedeny různé optimalizace obou pluginů.

Kapitola 2

Seznámení s knihovnou sysrepo a pluginy pro úložiště

Knihovna sysrepo ukládá data modelu YANG, který určuje, jaká data se smí ukládat a v jaké podobě. Na základě modelu lze následně data validovat, což je hlavní výhodou tvorby datových modelů. Jelikož jednotlivé prvky jazyka YANG budou do značné míry rozebírány v následujících kapitolách, je nutno uvést alespoň základní přehled o tomto jazyce a jeho modelech.

2.1 Seznámení s modely jazyka YANG

Data modelu YANG, stejně jako model samotný, jsou reprezentovány pomocí stromové struktury. Model obsahuje listové a nelistové uzly, přičemž pouze listové uzly uchovávají hodnotu (např. číslo nebo text). Příkladem nelistového uzlu je „container“, příkladem listového je „leaf“.

Container je uzel modelu bez hodnoty. Může obsahovat strukturu uzlů, které pod něj patří, a tím pádem logicky oddělit různé části modelu.

Leaf je uzel modelu s hodnotou, který ovšem nemůže obsahovat žádné další uzly. Slouží tedy jako hlavní nosič dat. U tohoto typu uzlu je možné nastavit v modelu i tzv. **výchozí hodnotu**. V takovém případě pokud data modelu neobsahují uzel s explicitně danou hodnotou, bude obsažen implicitně s výchozí hodnotou. V modelu musí být u uzlu vždy uveden daný typ hodnoty, která se následně bude vyskytovat v datech modelu.

Leaf-list označuje množinu uzlů „leaf“, které mají stejný typ. Tuto množinu je možné seřadit podle systému, nebo podle uživatele (viz níže).

List označuje strukturu libovolných uzlů, kterým mohou patřit další uzly. Jednotlivé instance uzlu „list“ jsou rozlišeny podle klíče, kterým je jeden nebo více uzlů typu „leaf“. Seznamy typu „list“ lze řadit podle nastavení systému, nebo podle uživatele. **Systémové řazení** není nijak omezeno a systém smí zvolit vlastní styl řazení. U **uživatelsky řazených** seznamů je pořadí ovlivněno pořadím dat na vstupu. Při zpětném získávání dat musí být toto pořadí stejné.

Klíčové slovo **when** se využívá pro podmíněnou existenci uzlů v datech modelu. Existence uzlů je podmíněna výrazem XPath (viz dále). Pokud je tento výraz vyhodnocen jako pravdivý, uzel se smí vyskytovat v datech.

Data modelu YANG (viz Výpis 2.1) mohou být následně reprezentována formáty XML (viz Výpis 2.2) nebo JSON (viz Výpis 2.3) a jsou adresována výrazy XPath [15], přičemž **XPath** je jazyk pro adresaci jednotlivých částí dokumentů ve formátu XML [16].

```
module modulename {
    namespace s;
    prefix s;
    container containername {
        leaf leafname {
            type boolean;
            default true;
        }
        leaf-list leaflistname {
            type string;
            when "../leafname = 'true'";
        }
        list listname {
            key keyname;
            leaf keyname {
                type uint8;
            }
        }
    }
}
```

Výpis 2.1: Příklad modelu YANG

```
<containername xmlns="s">
    <leafname>true</leafname>
    <leaflistname>first</leaflistname>
    <leaflistname>second</leaflistname>
    <listname>
        <keyname>42</keyname>
    </listname>
</containername>
```

Výpis 2.2: Příklad dat modelu YANG (viz Výpis 2.1) ve formátu XML

```
{ "containername": {
    "leafname": true,
    "leaflistname": [
      "first",
      "second"
    ],
    "listname": {
      "keyname": 42
    }
  }
}
```

Výpis 2.3: Příklad dat modelu YANG (viz Výpis 2.1) ve formátu JSON

2.2 Jak knihovna sysrepo funguje

Ještě před manipulací s daty je zapotřebí navázat **spojení**. To uchovává informace o sdíleném paměťovém prostoru a kontextu knihovny libyang. **Libyang** je knihovna napsaná v jazyce C pro správu modelů YANG a k nim příslušných dat a pro manipulaci s nimi [20]. Knihovna sysrepo je úzce provázána s knihovnou libyang, jelikož využívá její funkce právě pro manipulaci s daty YANG. Všeobecně platí, že jeden uživatelský proces navazuje právě jedno připojení (až na výjimky) [22].

Poté následuje vytvoření libovolného počtu **sezení**. Zde je nutné pro udržení správné funkcionality pro každé vlákno vyhradit separátní sezení kvůli správné synchronizaci vláken, která je plně automatická a uživatel se jí již nemusí zabývat. V každém sezení je posléze možné vybrat si úložiště dat, se kterým chce uživatel pracovat. Na něm jsou pak prováděny veškeré změny. Typ úložiště může být v rámci jednoho sezení kdykoli změněn. Při používání úložiště Running (viz Sekci 2.3) lze využít i mezipaměť, takže není vždy potřebné při požadavku načítat data z operační paměti, ale je možné efektivně získat data právě z mezipaměti [22].

Nakonec je ještě nezbytné do knihovny sysrepo nainstalovat všechny modely YANG, které udávají datový kontext, tedy konkrétně s jakými daty bude knihovna sysrepo pracovat. Tyto modely mohou být za běhu mazány či aktualizovány, přičemž při aktualizaci modelu již uložená data zůstanou nedotčená. Jedná se ovšem o drahou operaci, která může trvat delší dobu. Jedinou podmínkou pro funkční aktualizaci modelu je obsažení pozdější revize v kódu daného modelu YANG [23]. Zisk dat z knihovny sysrepo se jednoduše provede pomocí výrazu XPath. Tato operace obvykle vrací strukturu z knihovny libyang, která může obsahovat větší množství dat, např. celý podstrom daného modelu [24]. Knihovna sysrepo podporuje i správu **událostí**, což znamená, že změny na datech mohou být snímány a následně je při takové změně vyvolána událost. Události jsou explicitně snímány pomocí tzv. **odběrů**.

2.3 Druhy úložišť knihovny sysrepo

Knihovna sysrepo poskytuje celkem pět druhů úložišť dat (Startup, Running, Candidate, Operational a Factory Default) [21].

Startup slouží k počátečnímu nastavení konfigurace v čase spuštění zařízení. Při navázání připojení je obsah tohoto úložiště zkopírován do úložiště Running.

Running obsahuje stávající konfiguraci zařízení. Za běhu zařízení při změně konfigurace je tedy toto úložiště vždy aktualizováno. Jedná se tedy o zdaleka nejvytíženější druh úložiště. Zároveň po vypnutí zařízení je obsah úložiště smazán.

Candidate se využívá jako místo pro přípravu konfigurace bez přímého ovlivnění stávající konfigurace. Následně lze zkopírovat obsah úložiště Candidate do úložiště Running, čímž se připravená konfigurace aplikuje.

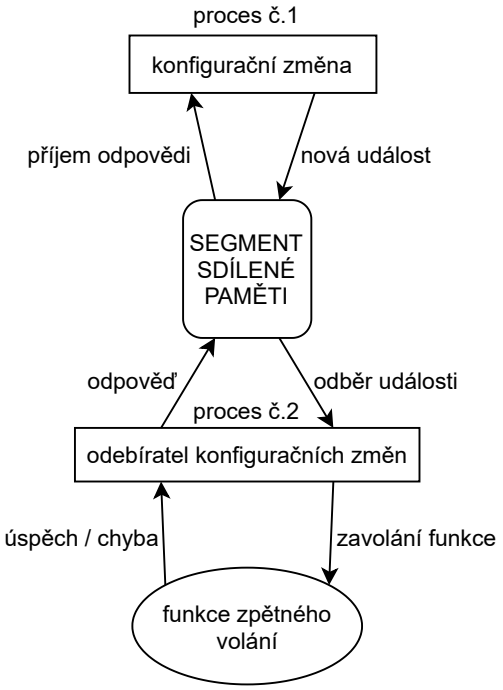
Operational často schraňuje převážně stejná data jako úložiště Running s některými výjimkami (např. pouze toto úložiště může obsahovat stavová data, tedy data nastavená systémem, nikoli uživatelem). Toto úložiště odráží momentální stav systému a přímo s tímto úložištěm systém pracuje. Důvodem pro oddělení úložiště Running (tedy okamžité konfigurace systému nastavené uživatelem) a úložiště Operational (tedy reálné konfigurace, se kterou pracuje systém) je především možnost nekonzistence mezi nimi. Pokud je úložiště modifikováno a pokud systém podporuje modifikaci úložiště z vícero zdrojů, chvíli může trvat, než se změna propaguje do celého systému.

Factory Default odráží výchozí tovární konfiguraci dat. Toto úložiště je inicializováno při instalaci modelu a dále jej nelze nijak modifikovat, slouží pouze pro čtení. Z úložiště Factory Default lze data zkopírovat do úložiště Startup a obnovit tak výchozí konfiguraci systému.

2.4 Druhy odběrů v knihovně sysrepo

Odběr umožňuje snímat události a pokaždé, kdy nastane určitá událost, je spuštěn kód daného odběru, přičemž jich existuje více druhů. Odběr je zpracováván pomocí tzv. funkce zpětného volání (angl. callback — uživatelem vytvořená funkce, kterou knihovna sysrepo volá při vyvolání události).

Odběr konfiguračních změn je nejobvyklejší druh odběru. Jedna a ta samá data mohou být snímána několika odběry. Odběr je vlastní tomu úložišti dat, které bylo aktivní při vytvoření daného odběru. Aktivní úložiště dat je takové, které je využíváno v daný okamžik. Aktivita přitom může být libovolně přepínána mezi úložišti a smí být aktivní právě jedno úložiště v rámci jednoho sezení. Změna aktivity úložiště dat nemá žádný vliv na předchozí vytvořené odběry. Změny v úložišti na odebíraných datech generují dvě události. První událost „změna“ (z angl. “change”) je vygenerována při požadavku na změnu dat. Pokud jsou tato data odebírána, funkce zpětného volání odběru je zavolána, přičemž je schopna tento požadavek na změnu dat odmítnout (viz Obrázek 2.1) [25].

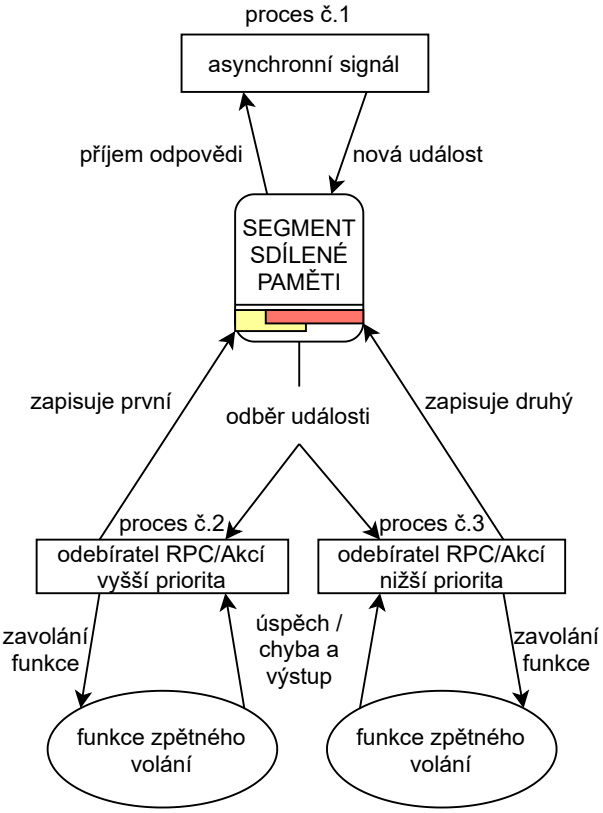


Obrázek 2.1: Odběr konfiguračních změn

Jestliže data přijme, následuje druhá událost „hotovo“ (z angl. “done”), která pouze oznamuje, že data byla úspěšně zapsaná do úložiště. Po události „hotovo“ a zavolání funkce zpětného volání odběru již není možné danou událost zvrátit. Existuje ještě další typ událostí „aktualizace“ (z angl. “update”), které ve výchozím režimu odběru konfiguračních změn nelze očekávat. Při vytváření odběru je třeba explicitně nastavit vyvolávání událostí

„aktualizace“. Tato událost je generována při požadavku na změnu dat ještě před událostí „změna“ kvůli možnosti dalších úprav datových změn. Další typ události „povoleno“ (z angl. “enabled”) se nastavuje podobně jako „aktualizace“. Tato událost je generována při vytvoření odběru. Je tak díky ní možné zjistit počáteční konfiguraci úložiště [25].

Odběr RPC/Akcí spravuje vstupní asynchronní signály (např. od uživatele). Po zpracování takovéto události může vygenerovat výstup (např. návratovou hodnotu funkce) jako odezvu uživateli (viz Obrázek 2.2).



Obrázek 2.2: Odběr RPC/Akcí

RPC (z angl. Remote Procedure Call) je protokol, který se všeobecně používá pro komunikaci mezi procesy na různých strojích (lze jej využít i pro meziprocesovou komunikaci na jednom stroji) [9]. Události související se vstupem jsou generovány, pouze pokud existuje odběr, který by je přijal. Při více odběrech na jednu událost RPC má hlavní slovo odběr s nejmenší prioritou. Tedy ten, jehož funkce zpětného volání se spustí poslední [25].

Notifikační odběr zpracovává události vyvolané serverem. Tyto události jsou pro případ zpětného přístupu ukládány. Pokud uživatel vytvoří notifikační odběr s požadavkem na zpětné přehrání minulých událostí, všechny události od uživatelem daného času dále jsou opět zaslány odběru. Díky této funkcionalitě lze přistupovat i ke starším událostem, které vznikly ještě před jakýmkoli odběrem [25].

Operační odběr poskytuje data pomocí vygenerování operačního úložiště. Získává data dvěma způsoby – „vytažení“ (z angl. “pull”) a „vytlačení“ (z angl. “push”). Metoda „vytažení“ volá funkce zpětného volání, které jsou schopny vrátit momentální stav operačního úložiště. Tento přístup je vhodný zejména pro data, která se mění často (např. čítače). Metoda „vytlačení“ na druhé straně spočívá v tom, že uživatel přímo pomocí výrazu XPath

vytvoří data reprezentující jeho stav. Tato data jsou uložena a jsou využita až při generování operačního úložiště. Tento přístup je vhodný zejména pro data, která se často nemění (např. základní informace o zařízení). Pokud by došlo ke kolizi, vždy jsou upřednostněna data získaná metodou „vytažení“, jelikož ta se získávají přímo při požadavku na data [25].

2.5 Pluginy pro úložiště knihovny sysrepo

Manipulace s úložišti probíhá přes plugin pro úložiště. Sysrepo přistupuje k datům úložiště přes rozhraní pluginu, jež je nezávislé na konkrétní implementaci úložiště. Sysrepo obsahuje jeden výchozí plugin, který organizuje data pomocí funkcí knihovny libyang do **souborů ve formátu JSON**. V tomto výchozím pluginu jsou data každého modelu YANG ukládána do separátního souboru. Zároveň parsování dat YANG do formátu JSON probíhá nativně v knihovně libyang.

Pluginy pro úložiště knihovny sysrepo umožňují uživateli vytvořit si vlastní úložiště dat a napojit tak již existující aplikaci na knihovnu sysrepo. To se provádí implementací již předpřipravených hlaviček funkcí z rozhraní pluginu, které realizují základní operace s úložištěm. Mezi ně se řadí například funkce zpracovávající získání, ověření a změnu přístupových práv uživatele k datům jednotlivých modelů. Přístupová práva se shodují s modelem unixových systémů a vzhledem k tomu, že výchozí plugin ukládá data modelů YANG do souborů, je velmi jednoduché tato práva vyčíst právě z metadat souborů uložených v adresáři disku. Pak zde lze nalézt hlavičky specifických funkcí pro správu úložiště Candidate. Další funkce zahrnují standardní operace kopírování jednoho úložiště do dalšího úložiště, smazání dat konkrétního modelu v úložišti, inicializace dat nově přidaného modelu a obnovení modelových dat v případě selhání systému. Ve výchozím pluginu je operace obnovení možná díky tomu, že data se do úložiště ukládají robustně, tedy nejprve se vytvoří záloha, která se postupně kopíruje do úložiště. Pokud se operace provede správně, záloha je smazána, pokud ne, záloha se využije k obnovení dat.

Jak je již zmíněno výše, veškerá data jsou výchozím pluginem ukládána do souborů ve formátu JSON, tudíž tyto operace převážně pracují s vytvářením, mazáním a změnou těchto souborů. Dále sem patří volitelné funkce zajišťující načítání, aktualizaci a mazání dat z úložiště Running v mezipaměti. Nakonec zdaleka nejdůležitějšími funkcemi zůstavají funkce načtení a uložení dat, bez nichž se žádné úložiště neobejde.

Data lze z knihovny sysrepo načíst pomocí výrazu XPath. Byť byl jazyk XPath vyvinut zejména pro adresaci částí dokumentů ve formátu XML, slouží velmi dobře i pro adresaci jednotlivých částí dat modelů YANG (toto zajišťuje právě knihovna libyang). Výrazy XPath poskytují velmi obšírnou funkcionalitu. Nejenomže je s nimi uživatel schopen určit konkrétní uzel v datech modelu YANG, ale také například celý podstrom dat pouze s použitím jednoho výrazu. XPath se podobá regulárním výrazům, pro větší volnost při psaní XPath se dají uplatnit i tzv. „divoké karty“ (angl. “wildcards”). Díky nim dokáže uživatel tvořit složité požadavky na načtení dat (viz Výpis 2.4).

```
/module_prefix:container1/container2/*/node
```

Výpis 2.4: Ukázka výrazu XPath. Viz divokou kartu „*“, která umožňuje odkazovat na více různých instancí uzlů „node“ současně.

Uložení dat se již provádí pomocí struktur z knihovny libyang. Sysrepo dodá funkci konkrétní strukturu z knihovny libyang reprezentující celý strom, který je potřeba uložit (což se považuje při obsáhlých datech modelu YANG za velmi náročnou operaci) a zároveň dodá pouze uzly, které je potřeba změnit, a to pomocí další speciální struktury **diff**

z knihovny libyang. Díky tomu, že není vždy vyžadováno ukládat znovu všechna data modelu YANG, se operace vkládání/mazání stává efektivnější (viz Výpis 2.5). Výchozí plugin ovšem strukturu diff nevyužívá a pracuje vždy s celým modelem.

```
<module_name xmlns="namespace"
             xmlns:yang="urn:ietf:params:xml:ns:yang:1"
             yang:operation="none">
    <list>
        <key>a</key>
        <list2 yang:operation="delete">
            <key2>a</key2>
        </list2>
    </list>
</module_name>
```

Výpis 2.5: Na seznamu list2 se provádí operace mazání, ostatní data zůstávají.

2.6 Notifikační pluginy knihovny sysrepo

Notifikační pluginy umožňují uživateli vytvořit si vlastní úložiště pro ukládání systémem vygenerovaných notifikačních událostí. Stejně jako u pluginů pro úložiště se i u notifikačních pluginů uplatňují funkce zpětného volání, které zajišťují základní operace s úložištěm. A stejně jako u pluginů pro úložiště je i u notifikačních pluginů k dispozici výchozí notifikační plugin. Nacházejí se zde funkce spravující přístupová práva, funkce pro inicializaci a smazání úložiště notifikací modelu YANG, funkce pro přehrání další notifikace a pro uložení notifikace do úložiště.

U výchozího pluginu soubory notifikací nesou název sestávající z časové značky první a poslední notifikace, které do nich byly vloženy. Mimo samotné notifikace je tedy ukládán i čas vyvolání notifikace. Při ukládání nové notifikace je vybrán poslední soubor, do kterého bylo zapisováno, a je zjištěno, zda se nová notifikace vejde do daného souboru. Pokud v souboru existuje dostatečný prostor, je nová notifikace uložena do souboru. Přičemž je ještě pozměněna druhá část názvu souboru, tedy časová značka poslední vložené notifikace. Avšak jestliže v souboru neexistuje dostatečný prostor, je vytvořen nový soubor, do kterého se notifikace zapíše, přičemž název souboru nese časové značky nové notifikace (nově zapsaná notifikace je první i poslední). Výchozí maximální velikost souboru je 1MB. Starší notifikace mohou být i archivovány do komprimovaných souborů, které nezabírají tolik místa na disku.

2.7 Problémy výchozího úložiště

Jak již bylo uvedeno, knihovna sysrepo schraňuje data jednotlivých modelů YANG v souborech ve formátu JSON. Co soubor, to data jednoho konkrétního modelu. Tento jednoduchý systém zaručuje velmi rychlé vkládání dat nových modelů, protože v tomto případě stačí, když je vytvořen nový soubor a do něj jsou vložena všechna potřebná data. Ovšem při úpravě již uložených dat nastává problém. Pokud totiž chceme upravovat již vytvořený soubor, musí se všechna data z tohoto souboru načíst do paměti, tam je potřeba je upravit a vytvořit nový soubor, kam jsou všechna tato upravená data následně zapsána. Tudíž i malá změna vyžaduje načtení dat celého modelu YANG. Dalším problémem je neefektivní vyhledávání dat v takovém souboru. Veškerá data se opět musí načíst do paměti, kde pak

vyhledávání dále probíhá již efektivně za pomoci struktur z knihovny libyang (avšak načtení celého souboru do paměti trvá dlouho). A v neposlední řadě výrazy XPath se nedají využít při načítání konkrétní části dat modelu YANG, poněvadž je vždy načten celý soubor (nelze načíst pouze část souboru). Celý tento problém spočívá v ukládání dat do souborů. Jako řešení se nabízí použití **databáze**.

Kapitola 3

Seznámení s databázemi a volba vhodných kandidátů

Databáze je organizovaná kolekce strukturovaných informací, která je obvykle řízena databázovým správním systémem (DBMS – Database Management System) [2]. Nejprve zde uvedu dva nejpoužívanější druhy databází. **Relační databáze** ukládají data ve strukturované podobě do tabulek. Obvykle se využívá pro implementaci databáze jazyk SQL (Structured Query Language). Naproti tomu **nerelační databáze** ukládají data v nestrukturované podobě do seznamů, což umožňuje větší flexibilitu, adaptabilitu a škálovatelnost databáze. Pro sysrepo se jeví jako nejvhodnější volba nerelační databáze, a to zejména z důvodu lepší škálovatelnosti. V tabulkovém formátu by každý uzel musel být reprezentován jedním řádkem a v případě seznamů YANG by řádky v databázi nabyly velmi rychle.

Dokumentová (document-based) databáze schraňuje data v dokumentech, což jsou soubory ve formátu např. JSON, BSON nebo XML. Skupiny podobných dokumentů jsou pak skladovány v kolekcích. Dokumenty v kolekcích nemusí mít stejné datové schéma, čímž poskytují mnohem větší flexibilitu než tabulkový formát relačních databází [3].

Klíč-hodnota (key-value) databáze ukládá pouze dvojice dat (klíč a hodnotu), přičemž hodnota může být libovolný druh dat, který je potřeba z databáze získat pomocí klíče. Hlavními přednostmi tohoto druhu databáze jsou jednoduchost, škálovatelnost a rychlost. Nevýhodou je, že tento druh databází kvůli své jednoduchosti často nepodporuje vrácení většího množství dat na základě jednoho dotazu (všechna data jsou reprezentována pouze v párech klíč — hodnota, tedy vyhledávání je často obtížnější) [3].

Sloupcová (column-oriented) databáze nabízí ukládání dat ve sloupcích místo řádků, díky čemuž je čtení a zisk dat rychlejší [3].

U **grafové** (graph-based) databáze jsou data uložená v uzlech (angl. “nodes”), které jsou propojovány vztahy (angl. “links”). Ty umožňují logické propojení mezi uzly [3].

Data YANG lze reprezentovat pomocí všech výše uvedených druhů databází, ovšem grafové databáze jsou schopny narozdíl od ostatních simulovat stromovou strukturu dat YANG právě pomocí uzlů a vztahů. Jedním z problémů, které se objevily při volbě databáze, byl i fakt, že valná většina bezplatných grafových databází byla implementována zejména pro webové aplikace. Mnoho databází je také napsáno v jiném jazyce než knihovna sysrepo a nenabízí žádné oficiální rozhraní pro jazyk C.

3.1 Srovnávací testy

Nejdůležitějším měřítkem, podle kterého bude vybírána budoucí databáze, jsou především operace pracující s velkým množstvím dat. Sysrepo v současné době pracuje velmi neefektivně tak, že ukládá data do prostých souborů. Operace pracující s velkým množstvím dat (resp. s daty celého modelu) proto trvají minimální dobu, ale ostatní operace pracující s malým množstvím dat mohou být zrychleny. Ideálním předpokladem by pro budoucí databázi bylo, aby např. operace vkládání velkého množství dat (počáteční naplnění databáze) trvala pokud možno podobně dlouho jako nyní, přičemž by se zkrátily doby pro operace s malým množstvím dat.

Pro tyto potřeby byly sestaveny jednoduché skripty v jazyce Bash, které na ukázkovém modelu YANG (viz Výpis 3.1) zjistí rychlosti vkládání dat u jednotlivých databází. Tyto skripty posílají na databázový server příkazy za pomoci klientského shellu dané databáze, přičemž čas je měřen pomocí linuxového příkazu `date`. Vždy je změřen počáteční čas a konečný čas příkazu a rozdíl těchto časů je pak výslednou délkou trvání. Naproti tomu pro aproximaci manipulace s daty výchozího pluginu pro úložiště knihovny sysrepo byl implementován program v jazyce C a měření jsou také prováděna v jazyce C za pomoci funkce `clock_gettime()` ze standardní knihovny jazyka C. Tyto časy jsou ovšem pouze referenční a jsou využity pouze pro porovnání jednotlivých databází. Zároveň byly ještě pro databáze, které měly nejlepší časy a které poskytují oficiální rozhraní pro jazyk C, implementovány programy v jazyce C, ve kterých je kromě ukládání dat, měřena i další manipulace s daty.

Jednotlivé časy jsou měřeny podle toho, kolik prvků je ukládáno do seznamu YANG (angl. “list”), jelikož právě ten často způsobuje největší zátěž úložiště při nasazení. Jednotlivá měření byla provedena 10krát kvůli minimalizaci nepřesností a lepší aproximaci výsledného chování databáze, uvedeny jsou pouze aritmetické průměry těchto měření.

```
1   module test-yang {
2       namespace s;
3       prefix s;
4
5       container simple-cont {
6           container simple-cont2 {
7               container ac1 {
8                   leaf acd1 {
9                       type string;
10                  }
11                  list acl1 {
12                      key acs1;
13                      leaf acs1 {
14                          type string;
15                      }
16                  }
17              }
18          }
19      }
20  }
```

Výpis 3.1: Testovací model YANG

3.2 Grafové databáze

Neo4j je open-source databáze, která podporuje vlastní snadno pochopitelný dotazovací jazyk CQL (Cypher Query Language). Velkou výhodou této databáze je velmi dobrá škálovatelnost, kvalitní přehledná dokumentace a stálá podpora. Mezi nevýhody patří především komerčnost databáze. V základní bezplatné verzi tato databáze nepodporuje přístupová práva, která jsou ovšem pro knihovnu sysrepo nezbytná. [8]

ArangoDB je multimodelová open-source databáze napsaná v jazyce C++, která kromě grafů nativně podporuje i jiné datové modely (např. klíč-hodnota, dokumentové, atd.). [5]

Blazegraph patří mezi vysoce výkonné grafové databáze napsané v Javě s podporou více než 50 miliard grafových hran (vztahů) na jednom stroji. Hlavní nevýhodou je neexistence aplikačního rozhraní pro jazyk C. [13]

BrightstarDB je webová databáze, která se řadí mezi RDF (Resource Description Framework) databáze, jež spadají pod grafové databáze. Tento model se opírá o ukládání dat sémanticky ve trojicích (subjekt-predikát-objekt). Výhodou je jednoduchost použití a možnost výběru z vícero nabízených datových modelů. Naopak tato databáze je používána především u webových aplikací a postrádá detailní dokumentaci. [18]

Bitsy je malá rychlá grafová databáze, která udržuje celou kopii databáze v rámci operační paměti. Jejími základními principy jsou "No seek" (zrychlení vyhledávání na disku pro co nejrychlejší zápis), což by bylo výhodné zejména pro úložiště Running, které nepožaduje odolnost vzhledem k výpadkům systému, "No socket" (databáze je přímo vložena do aplikace), "No SQL" (zrychlení průchodů grafem). Nevýhodou Bitsy je, že je napsána v Javě a neexistuje žádné oficiální aplikační rozhraní pro jazyk C, tudíž efektivní zapojení do knihovny sysrepo může být komplikovanější. Dalším možným problémem může být horší škálovatelnost. [6]

Bitsy se jeví jako dobrá volba databáze pro knihovnu sysrepo. Podle dokumentace je rychlá, plně využívá operační paměť (což by bylo výhodné zejména pro nejvytíženější úložiště dat Running) a navíc se jedná o malou databázi, která není náročná na místo na disku. Bitsy se tedy jeví jako nejnadějnější z grafových databází, a proto budou provedena měření pouze s ní.

Insert list [počet elementů]	sysrepo [s]	bitsy [s]
1000	0,001366	46,376508

Tabulka 3.1: Porovnání databáze Bitsy a knihovny sysrepo při ukládání dat ukázkového modelu YANG

Z porovnání databáze Bitsy s knihovnou sysrepo (viz Tabulku 3.1) vyplývá, že Bitsy je zhruba 34000krát pomalejší než dosavadní implementace, což je z praktického hlediska naprosto nepřípustné. Proto s touto databází již nebyla prováděna žádná další měření a byla okamžitě vyškrtnuta ze seznamu potenciálních databází pro nasazení.

Po průzkumu grafových databází a provedených měřeních bylo zjištěno, že grafové databáze nemusí nutně být jedinou volbou při výběru vhodné databáze. Důvodem je způsob ukládání dat YANG. Seznamy mohou obsahovat velký počet prvků a z hlediska výkonu by pro tento případ užití byl následně na databázi vyvíjen největší nátlak. Path, tedy jedinečná cesta k uzlu v rámci datového modelu, byl využit jako primární klíč. Uložení celé

cesty k uzlu jakožto primárního klíče je však zanedbatelné vůči době ukládání, protože doba ukládání n prvků je závislá zejména na počtu prvků spíše než na délce primárního klíče. Ukládání dat však nepřineslo uspokojivé výsledky, a proto byla databáze zavržena. Mnohé grafové databáze jsou využívány zejména pro vývoj webových aplikací a nebyly by tedy vhodnými kandidáty pro knihovnu sysrepo. Zároveň grafové databáze nejsou velmi efektivní při zpracovávání velkého množství transakcí a mezi nejhorší případy užití patří právě dotazy, které pracují s daty celé databáze [19].

3.3 Sloupcové databáze

Apache Cassandra je široko-sloupcová (wide-columnar, což je podkategorie sloupcových databází) distribuovaná (na několika discích) databáze, která je dle dokumentace velmi dobře škálovatelná, nabízí rychlou organizaci velkého množství dat a díky distribuovanosti databáze je možnost ještě zvýšit rychlost databáze zapojením většího množství strojů, na kterých databáze běží. Dále poskytuje celkem obsáhlou dokumentaci a díky tomu, že se jedná o open-source, také značnou zákaznickou podporu. [4]

Jednou z nevýhod Apache Cassandra může být až přílišná obsáhlost, která dokáže znesnadnit případné zapojení do knihovny sysrepo. Apache Cassandra také nepodporuje oficiální aplikační rozhraní pro jazyk C.

Apache Cassandra nemusí být nutně špatná volba pro knihovnu sysrepo. Je dobře škálovatelná a při případné nedostatečné rychlosti lze v budoucnu zapojit větší množství strojů pro ještě lepší výsledky, avšak její nasazení stále zůstává zejména pro aplikace s distribuovanými systémy.

Insert list [počet elementů]	sysrepo [s]	cassandra [s]
1000	0,001366	2,085668
10000	0,013201	15,976230
100000	0,145756	152,175911

Tabulka 3.2: Porovnání databáze Apache Cassandra a knihovny sysrepo při ukládání dat ukázkového modelu YANG

Z porovnání databáze Cassandra s knihovnou sysrepo (viz Tabulku 3.2) vyplývá, že Cassandra je zhruba 1000krát pomalejší než dosavadní implementace. Oproti databázi Bitsy je databáze Cassandra sice rychlejší, ovšem časy pořád neuspokojují potřeby knihovny. Seznam (angl. "list") o 100000 elementech je běžný případ užití a doba ukládání 2,5 minuty je nepřiměřeně dlouhá.

3.4 Dokumentové databáze

Marklogic je multimodelová databáze, která podporuje ukládání dat v dokumentech (ve formátu XML nebo JSON). Vyniká škálovatelností a bezpečností dat. Hlavní nevýhodou Marklogic zůstává fakt, že se plně nezaměřuje na rychlost a jednoduchost, ale spíše na všestrannost v ukládání rozličných dat a bezpečnost při manipulaci s daty. Ani jedné z těchto vlastností však nelze nijak přímo využít v knihovně sysrepo. [1]

MongoDB je dokumentová databáze, která dle dokumentace vyniká svou jednoduchostí, výkonem, flexibilitou a škálovatelností. Dokumentace je přehledná. Díky širokému využití a popularitě se řadí mezi databáze s velmi dobrou zákaznickou podporou. Nabízí také oficiální aplikační rozhraní pro jazyk C. [7]

Výborná škálovatelnost, přehledná dokumentace a existence aplikačního rozhraní pro jazyk C působí jako ideální kombinace pro knihovnu sysrepo. Nevýhodou může být neexistence bezplatné verze pro WiredTiger Storage Engine, který si jakožto součást MongoDB Enterprise edice klade za cíl vykonávat veškeré transakce přímo v operační paměti. Tento engine by se dal s výhodou využít právě u úložiště Running.

Insert list [počet elementů]	sysrepo [s]	mongoDB [s]
1000	0,001366	0,626842
10000	0,013201	1,610558
100000	0,145756	11,197769

Tabulka 3.3: Porovnání databáze MongoDB a knihovny sysrepo při ukládání dat ukázkového modelu YANG

Z porovnání databáze MongoDB s knihovnou sysrepo (viz Tabulku 3.3) vyplývá, že databáze MongoDB je v závislosti na počtu ukládaných dat zhruba 80-460krát pomalejší než dosavadní implementace. Jelikož pro databázi MongoDB existuje rozhraní v jazyce C, byla provedena další měření v jazyce C pro získání lepších a případně přesnějších výsledků s ohledem na finální implementaci (viz Tabulky 3.4, 3.5, 3.6 a 3.7) týkající se ukládání, vyhledávání, aktualizace a mazání dat.

Insert list		
počet elementů	sysrepo [s]	mongoDB [s]
1000	0,001366	0,004331
10000	0,013201	0,037371
100000	0,145756	0,363239

Tabulka 3.4: Porovnání databáze MongoDB a knihovny sysrepo při ukládání dat ukázkového modelu YANG (implementace v jazyce C)

Find in list		
počet elementů	sysrepo [s]	mongoDB [s]
1000	0,004889	0,000302
10000	0,055173	0,000401
100000	0,673450	0,000447

Tabulka 3.5: Porovnání databáze MongoDB a knihovny sysrepo při hledání jednoho prvku v datech ukázkového modelu YANG (implementace v jazyce C)

Jak je vidět, časy se při ukládání mnohonásobně zlepšily a při zadávání příkazů v jazyce C podává databáze MongoDB přibližně 2-3krát horší výkon než dosavadní implementace. Všechny ostatní časy u databáze MongoDB vyšly řádově rychlejší než u dosavadní implementace a co je důležité, rychlost databáze zůstává konstantní bez ohledu na zvětšení množiny prohledáváných dat. Bohužel ukázkový model YANG, na kterém byla databáze testována, kvůli své jednoduchosti plně neodráží standardní případ užití. Proto byl navržen

Modify in list		
počet elementů	sysrepo [s]	mongoDB [s]
1000	0,004776	0,000248
10000	0,055207	0,000294
100000	0,693380	0,000300

Tabulka 3.6: Porovnání databáze MongoDB a knihovny sysrepo při změně jednoho prvku v datech ukázkového modelu YANG (implementace v jazyce C)

Delete in list		
počet elementů	sysrepo [s]	mongoDB [s]
1000	0,004775	0,000216
10000	0,055412	0,000236
100000	0,694423	0,000235

Tabulka 3.7: Porovnání databáze MongoDB a knihovny sysrepo při mazání jednoho prvku v datech ukázkového modelu YANG (implementace v jazyce C)

nový model (viz Výpis 3.2) a měření byla provedena ještě jednou (viz Tabulky 3.8, 3.9, 3.10 a 3.11) pro lepší aproximaci reálného užití.

```
1   module test-yang {
2       namespace s;
3       prefix s;
4
5       container simple-cont {
6           container simple-cont2 {
7               container ac1 {
8                   list acl1 {
9                       key acs1;
10                      leaf acs1 {
11                          type string;
12                      }
13                      leaf acs2 {
14                          type string;
15                      }
16                      leaf acs3 {
17                          type string;
18                      }
19                      container inner {
20                          leaf inner-leaf {
21                              type string;
22                          }
23                      }
24                  }
25              }
26          }
27      }
28  }
```

Výpis 3.2: Testovací model YANG, verze 2

Insert list		
počet elementů	sysrepo [s]	mongoDB [s]
1000	0,004120	0,018245
10000	0,043456	0,158653
100000	0,440622	1,448361

Tabulka 3.8: Porovnání databáze MongoDB a knihovny sysrepo při ukládání dat ukázkového modelu YANG, verze 2 (implementace v jazyce C)

Find in list		
počet elementů	sysrepo [s]	mongoDB [s]
1000	0,014587	0,000383
10000	0,165009	0,000484
100000	1,704355	0,000671

Tabulka 3.9: Porovnání databáze MongoDB a knihovny sysrepo při hledání prvku v datech ukázkového modelu YANG, verze 2 (implementace v jazyce C)

Modify in list		
počet elementů	sysrepo [s]	mongoDB [s]
1000	0,014504	0,000289
10000	0,167201	0,000331
100000	1,745945	0,000328

Tabulka 3.10: Porovnání databáze MongoDB a knihovny sysrepo při změně jednoho prvku v datech ukázkového modelu YANG, verze 2 (implementace v jazyce C)

Delete in list		
počet elementů	sysrepo [s]	mongoDB [s]
1000	0,014422	0,000222
10000	0,166516	0,000252
100000	1,734410	0,000620

Tabulka 3.11: Porovnání databáze MongoDB a knihovny sysrepo při mazání prvku v datech ukázkového modelu YANG, verze 2 (implementace v jazyce C)

Vzhledem k tomu, že jsou ukládány všechny uzly modelu a vzhledem k tomu, že nový model obsahuje seznam se čtyřmi uzly (předchozí obsahoval seznam pouze s jedním), se zvýšil čas ukládání dat přibližně čtyřnásobně. Při ukládání všech dat podává nyní databáze MongoDB zhruba 3-5krát horší výkon než dosavadní implementace. Ostatní časy jsou jinak pro databázi MongoDB srovnatelné s předchozím modelem. Z toho vyplývá, že nový model v této implementaci není užitečný a pro následující měření bude využit pouze první model.

3.5 Klíč-hodnota databáze

Memcached je open-source vysoce výkonná distribuovaná databáze napsaná v jazyce C. Poskytuje účelné a jednoduché API a přehlednou dokumentaci. Memcached se řadí především mezi databáze pro dynamické webové stránky, kde slouží zejména jako mezipaměť urychlující předávání odpovědí klientům [17].

Riak KV je distribuovaná odolná databáze vynikající především dobrým zvládáním chybových stavů při výpadcích systému. Poskytuje komerční i open-source verzi, ovšem pro účely jednoduchého úložiště postačí pouze open-source verze. Nemá oficiální rozhraní pro jazyk C [12].

Redis je open-source víceúčelová databáze, která provádí transakce výhradně uvnitř operační paměti. Nabízí zdarma i různé moduly, které doplňují funkcionalitu podobnou

dotazům v jazyce SQL jako např. vyhledávání většího množství dat pomocí jednoho dotazu. Oficiální rozhraní pro jazyk C je dostupné v bezplatné verzi a dokumentace k tomuto rozhraní je velmi detailní a přehledná [10].

Jako nejlepší kandidát pro implementaci se zde jeví Redis. Veškeré transakce provádí uvnitř operační paměti, čehož by šlo s výhodou využít u úložiště Running. Také je zde možnost pravidelného ukládání dat na disk a pokud by nastal výpadek systému, databáze Redis je schopna obnovit původní stav (do stavu řádově v minutách před tím, než nastal výpadek) [11].

Databáze Redis podporuje různé přístupy k ukládání dat. Nejobvyklejším přístupem je příkaz „SET“. Pomocí něj lze nastavit klíč a k němu přidělenou hodnotu. Klíčem v případě modelu YANG je „Path“, tedy jedinečná cesta k ukládanému uzlu, a hodnotou pak hodnota daného uzlu. Pomocí příkazu „MSET“ je možné nastavit vícero takových klíčů najednou. Příkaz „SADD“ přidává nové prvky do množiny jednoznačně charakterizované zvoleným klíčem. Tento příkaz jde s výhodou využít pouze u systémem řazených seznamů YANG.

Insert list [počet elementů]	sysrepo [s]	redis [s]
1000	0,001366	0,006778
10000	0,013201	0,032091
100000	0,145756	0,293742

Tabulka 3.12: Porovnání databáze Redis a knihovny sysrepo při ukládání dat ukázkového modelu YANG

Z Tabulky 3.12 je zřejmé, že databáze Redis podává při ukládání dat pomocí příkazu „SET“ zhruba 2-5krát horší výkon než dosavadní implementace. Pro databázi Redis existuje rozhraní v jazyce C, proto byla provedena další měření.

Insert list [počet elementů]	sysrepo [s]	redis SET [s]	redis MSET [s]	redis SADD [s]
1000	0,001366	0,061376	0,005416	0,001714
10000	0,013201	0,597554	0,040761	0,010716
100000	0,145756	6,008112	0,409712	0,107713

Tabulka 3.13: Porovnání různých přístupů databáze Redis a knihovny sysrepo při ukládání dat ukázkového modelu YANG (implementace v jazyce C)

Z Tabulky 3.13 vyplývá, že nejlepším způsobem ukládání systémem řazených seznamů je příkaz „SADD“. Další data, jako např. samostatné uzly nebo uživatelem řazené seznamy, ovšem takto ukládat nelze, proto bude zpočátku implementace záviset zejména na příkazech „SET“ a „MSET“ a přidání dalších příkazů pro ukládání dat bude následně otázkou optimalizací.

Z Tabulek 3.14, 3.15 a 3.16 je zřejmé, že všechny ostatní operace zůstávají při exponenciálním zvětšení konstantní.

Find in list		
počet elementů	sysrepo [s]	redis GET [s]
1000	0,004889	0,000150
10000	0,055173	0,000109
100000	0,673450	0,000180

Tabulka 3.14: Porovnání databáze Redis a knihovny sysrepo při hledání prvku v datech ukázkového modelu YANG (implementace v jazyce C)

Modify in list		
počet elementů	sysrepo [s]	redis SET [s]
1000	0,004776	0,000063
10000	0,055207	0,000101
100000	0,693380	0,000098

Tabulka 3.15: Porovnání databáze Redis a knihovny sysrepo při změně jednoho prvku v datech ukázkového modelu YANG (implementace v jazyce C)

Delete in list [počet elementů]	sysrepo [s]	redis DEL [s]
1000	0,004775	0,000069
10000	0,055412	0,000086
100000	0,694423	0,000085

Tabulka 3.16: Porovnání databáze Redis a knihovny sysrepo při mazání prvku v datech ukázkového modelu YANG (implementace v jazyce C)

Kapitola 4

Návrh a implementace pluginů pro databáze

Implementace pluginu pro úložiště spočívá v implementaci jednotlivých funkcí zpětného volání, které knihovna sysrepo volá při požadavcích na manipulaci s daty. Tyto funkce poskytují správu přístupových práv k jednotlivým datovým modelům, načítání, ukládání a další funkce pro správu dat. Při snaze o napsání pluginu pro úložiště pro databáze ovšem vyvstalo několik problémů.

4.1 Napojení na databázi

Databáze **MongoDB** poskytuje knihovnu funkcí API v jazyce C pro zjednodušení zadávání příkazů do databáze. Prvním problémem vůbec je správné napojení na databázi za všech okolností. Sysrepo totiž nepodporuje funkci zpětného volání, která by byla spuštěna při znovuspuštění systému po výpadku nebo cíleném vypnutí. Ovšem pro umožnění správného volání funkcí API databáze MongoDB je nejprve potřeba spustit inicializační funkci a ustanovit klienta, který se připojuje k databázi. Toto bylo vyřešeno funkcí, která je volána na začátku každé funkce zpětného volání pluginu a která kontroluje, zda existuje klient připojený k databázi. Pokud ne, ustanoví se klient. Inicializační funkce se smí volat pouze jednou a pouze z jednoho připojujícího se procesu při prvotním připojení k databázi.

S databází **Redis** je možno komunikovat přes knihovnu Hiredis napsanou v jazyce C. Stav připojení k databázi je uložen přímo ve struktuře knihovny, přičemž připojení je potřeba ustanovit pro každé nově se připojující vlákno zvlášť. Toto je opět kontrolováno funkcí, která je volána na začátku každé funkce zpětného volání pluginu.

4.2 Nové funkce pro pluginy

V průběhu implementace byly do knihovny sysrepo, jakožto reakce na vývoj nového pluginu, přidány nové funkce zpětného volání pro pluginy. Funkce `conn_init` je volána knihovnou sysrepo pokaždé, když je vytvořeno nové připojení ke knihovně (jinými slovy když je vytvořen nový proces, který knihovnu používá). Funkce `conn_destroy` je naopak volána vždy při ukončení připojení ke knihovně. Výchozí plugin tyto funkce nevyužívá, jelikož pracuje pouze se soubory. S výhodou jsou ale využité u pluginů, které používají databázi pro ukládání dat. Tato nová funkcionalita byla přidána správci knihovny sysrepo.

Např. konkrétně u databáze MongoDB funguje tzv. **per-process connection**. To znamená, že různé procesy se musí k databázi připojovat každý nezávisle na sobě. Zároveň ještě před ukončením je proces, jakožto samostatný klient, vždy nejprve od databáze odpojen.

Dále je ještě dodatečně každé funkci předáván nový parametr, do kterého lze zapisovat privátní data. Díky tomu může knihovna sysrepo předávat informace o připojení k databázi všem funkcím, aniž by musela interně implementovat konkrétní manipulaci s databází MongoDB.

4.3 Způsob ukládání dat YANG v databázi

Databáze **MongoDB** se dělí na kolekce, které obsahují záznamy. Při manipulaci s daty je tedy nutné nejprve specifikovat kolekci, ve které se hledaná data nachází, a až poté je proveden dotaz. Zde je možné vidět analogii s modelem YANG. Ten obsahuje uzly (podobně jako kolekce obsahuje záznamy) a jednotlivé modely YANG jsou logicky i fyzicky oddělené (podobně jako kolekce). Modely jsou tedy reprezentované v databázi jako kolekce a jeho uzly jako záznamy. Při prvotní implementaci byla prozatimně zvolena možnost užití „Path“ (tedy unikátní cesty k uzlu) jako primárního klíče. Tato možnost nemusí být nutně nejefektivnější, ovšem z časového hlediska je nejméně náročná na implementaci. Efektivnější způsoby ukládání uzlu v databázi by následně byly otázkou optimalizací a následných úprav výsledného pluginu.

Pokud je uzel uzlem datovým, do záznamu jsou kromě primárního klíče uložena i data tohoto uzlu. Zároveň jsou v databázi MongoDB jednotlivá úložiště Startup, Running, Candidate, Operational a Factory Default oddělena jako separátní databáze obsahující vlastní kolekce, přičemž každá databáze má prefix „sr_“ pro oddělení databází využívaných knihovnou sysrepo od ostatních. Název kolekce je ještě doplněn o prefix prostoru sdílené paměti pro odlišení různých aplikací přistupujících ke stejnému modelu, ovšem majících jiný datový obsah. V rámci kolekcí je ještě potřeba rozlišovat různé druhy záznamů. Datové uzly mají primární klíč začínající znakem „/“ (tedy standardní začátek výrazu XPath). Ostatní data, jako např. přístupová práva, časová značka poslední modifikace, atd., začínají číslicí (viz níže).

Naopak v databázi **Redis** neexistuje žádné hierarchické dělení dat do databází a kolekcí. Všechna data jsou zde pohromadě, proto je nutné odlišit je prefixem primárního klíče (viz Výpis 4.1). Ten obsahuje prefix „sr_“ pro odlišení dat knihovny sysrepo od dat jiných aplikací. Pak následuje název úložiště, prefix prostoru sdílené paměti, název modelu, druh ukládaných dat a unikátní cesta k datovému uzlu. Mezi druhy ukládaných dat patří „glob“, „perm“, „meta“ a „data“ (viz níže).

```
sr_<datastore>:<shm_prefix>:<module>:<type>:<path>
```

Výpis 4.1: Obecný zápis struktury primárního klíče v databázi Redis.

4.4 Načítání dat

Při načítání dat u původní implementace pluginu pro knihovnu sysrepo bylo nutno celý datový model (soubor) načíst do operační paměti, kde byl prohledáván již efektivně pomocí hashovací funkce. Tomuto přístupu je potřebné se vyhnout z důvodu pomalého získávání dat z paměti. Naštěstí funkce zpětného volání pluginu je k tomuto účelu vybavena a při každém volání poskytuje seznam výrazů XPath všech uzlů, které potřebuje, nebo prázdný seznam,

pokud potřebuje právě všechny uzly modelu. Uzly jsou ovšem v prozatimní implementaci v rámci daného modelu vždy načítány všechny, opět z důvodu jednoduchosti implementace.

U databáze **MongoDB** je nejprve podle názvu modelu specifikována kolekce, ve které se má hledat. Poté je pomocí databázového kurzoru procházeno přes vrácená data a ta jsou postupně zapisována do uzlové struktury knihovny libyang. Tato uzlová struktura je následně vrácena zpět volajícímu. Data jsou v rámci databáze zapsána ve formátu BSON, což je binární podoba souborů ve formátu JSON. MongoDB takto ukládá veškerá data a veškeré transakce provádí právě v tomto formátu. Tedy při načítání z databáze je potřeba data z tohoto formátu převést do textových řetězců, které je pak možné uložit do uzlové struktury knihovny libyang. K tomuto účelu bylo použito vestavěných funkcí aplikačního rozhraní databáze MongoDB pro jazyk C.

U databáze **Redis** jsou data každého modelu indexována. Pomocí databázového kurzoru je procházeno přes vrácená data modelu. Posléze jsou všechna data uložena do struktury knihovny libyang podobně jako u databáze MongoDB.

4.5 Ukládání dat

Při ukládání dat u původní implementace pluginu pro knihovnu sysrepo byl ukládán celý datový model při libovolné změně na datech. To v implementaci odpovídalo přepisu celého souboru, ve kterém byla data modelu uložena. Při malé změně na datech by ovšem bylo výhodnější přepsat pouze onu malou změnu a ostatní data zbytečně nepřepisovat. Funkce zpětného volání pro ukládání naštěstí poskytuje při svém spuštění strukturu libyang, která obsahuje datový model provedených změn (tzv. **diff**). V principu je tento strom následně rekurzivně prohledáván a dané změny jsou pak zapisovány jako příkazy pro databázi.

U databáze **MongoDB** jsou tyto příkazy nejprve ukládány ve formátu BSON do tří vnitřních dynamicky alokovaných polí. Po rekurzivním průchodu všemi změnami se ukazatele na tato pole předávají funkcím API databáze MongoDB pro jazyk C jako celek a ty pak mohou nová data uložit najednou v rámci jedné transakce. Mazání a modifikaci dat je nutno provádět postupně v rámci vícero volání na API databáze. Tato pole jsou rozdělena podle typu změny a dělí se na vytvářecí (ukládání nových dat), odstraňovací (mazání dat) a modifikační (úprava uložených dat). Po provedení všech změn v databázi jsou všechna alokovaná pole uvolněna.

U databáze **Redis** také dochází k rekurzivnímu prohledávání stromu změn, ovšem tyto změny jsou aplikovány hned. Knihovna Hiredis totiž nepodporuje funkce API, které by umožňovaly hromadné uložení mnoha prvků najednou. Velké změny je možno aplikovat zasláním celého souboru příkazů do databáze, avšak takový přístup je již otázkou optimalizací.

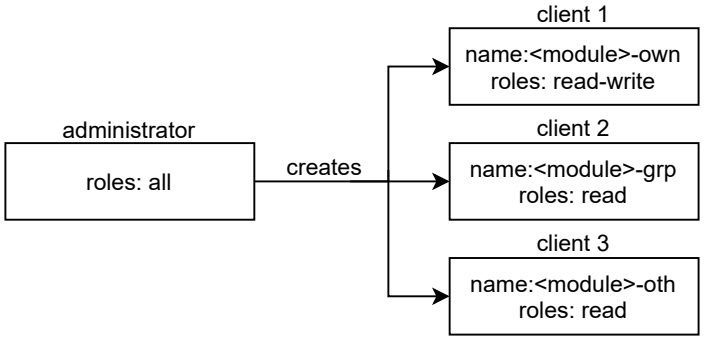
4.6 Přístupová práva

V původní implementaci byla přístupová práva uložena v metadatech jednotlivých souborů obsahujících datové modely. To zjednodušeně znamená, že každý model má svou vlastní standardní trojici tradičních unixových oprávnění pro vlastníka, skupinu a ostatní uživatele. Tato práva musí být nastavitelná a získatelná zavoláním patřičných funkcí zpětného volání.

Zásadním problémem je fakt, že **MongoDB** v základní bezplatné verzi nepodporuje tradiční unixová oprávnění a namísto nich poskytuje vlastní systém přístupových práv. V základu se jedná o systém klientů, ve kterém má každý klient svůj vlastní seznam rolí.

Klienti se po zadání uživatelského jména a hesla mohou k databázi připojit a na základě svých rolí mohou manipulovat s daty dané databáze. Role specifikují, jaké operace mohou klienti provádět s daty celé databáze, ovšem nikoliv pouze s daty jedné kolekce. Role dále specifikují, jaké konkrétní příkazy může daný klient databázovému systému zadat.

Nastalý problém byl řešen tak, že pro každou kolekci byli vytvořeni tři klienti. Tito klienti reprezentují trojici tradičních unixových oprávnění (tedy vlastníka, skupinu a ostatní). Každému z těchto klientů byly přiděleny role jemu náležející (role pro čtení a zápis). Názvy klientů jsou kombinací názvu modelu, pro který byli vytvořeni, a krátké sekvence tří znaků reprezentujících vlastníka, skupinu anebo ostatní (own, grp, oth). K databázi je vždy přihlášen pouze klient se všemi oprávněními (administrátor), zatímco ostatní klienti slouží pouze pro uložení potřebných přístupových práv k jednotlivým kolekcím (viz Obrázek 4.1).



Obrázek 4.1: Příklad vytváření klientů v databázi pro nově nainstalovaný model YANG.

Tento systém je sice funkční, ale značně neefektivní. Samotná manipulace s klienty je velmi náročná a instalace nových datových modelů, při které jsou ukládána i přístupová práva, trvá nepřiměřeně dlouho. Přístupová práva je možné ovšem ukládat i jako obyčejná data v kolekci daného modelu. Tato data mají jako primární klíč znak „4“, čímž je lze jednoznačně odlišit od dat daného modelu („Path“ začíná vždy s „/“). Tohoto je pak využito při kopírování dat modelu z jednoho úložiště do druhého, kdy data s primárním klíčem „4“ kopírována nejsou. To samé platí při načítání dat, kdy rovněž není žadoucí načítat přístupová práva. Ukládání přístupových práv jako dat je rychlejší než manipulace s klienty a vede na kratší a přehlednější kód, proto byla vybrána tato možnost.

Databáze **Redis** také podporuje systém uživatelů, kteří se mohou připojit k databázi. Ovšem stejně jako databáze MongoDB, databáze Redis nepodporuje tradiční unixová oprávnění. Veškerá oprávnění jsou tedy ukládána jako data (viz Výpis 4.2). Stejně jako u databáze MongoDB, k databázi přistupuje vždy jeden uživatel, který má potřebná práva k manipulaci se všemi daty knihovny sysrepo.

```
sr_<datastore>:<shm_prefix>:<module>:perm
```

Výpis 4.2: Forma primárního klíče pro přístupová práva v databázi Redis.

4.7 Poslední modifikace

U každého modelu je zapotřebí uchovávat informaci o čase poslední změny na datech daného modelu. V knihovně sysrepo byla tato informace uchovávána v metadatech souborů, které schraňovaly příslušné modely.

Databáze **MongoDB** ovšem neposkytuje žádný speciální prostor vyhrazený pro metadata jednotlivých kolekcí. Do každé kolekce je proto přidáván záznam, který obsahuje čas, kdy bylo naposledy s kolekcí manipulováno. Při změně na datech konkrétního modelu je zároveň měněn i tento záznam pro aktualizaci času poslední změny. Primárním klíčem záznamu poslední modifikace je znak „0“, čímž je možné ho jednoznačně odlišit od dat modelu („Path“ začíná vždy s „/“).

V databázi **Redis** je také ukládán čas poslední změny společně s daty modelu (viz Výpis 4.3).

```
sr_<datastore>:<shm_prefix>:<module>:glob:last-modified-sec
sr_<datastore>:<shm_prefix>:<module>:glob:last-modified-nsec
```

Výpis 4.3: Forma primárních klíčů v databázi Redis pro čas poslední modifikace dat modelu.

4.8 Kandidátní úložiště

Kandidátní úložiště slouží k přípravě dat, která budou následně zkopírována do úložiště Running. V původní implementaci je toto úložiště prázdné, pokud se shoduje s úložištěm Running. To, zda je kandidátní úložiště prázdné, či nikoli, značí přítomnost alespoň jednoho souboru náležejícího právě kandidátnímu úložišti.

V databázi **MongoDB**, která reprezentuje kandidátní úložiště, je navíc uložen záznam, který funguje jako proměnná určující stav tohoto úložiště. Díky ní lze pak snadno zkontrolovat, zda úložiště Running a Candidate mají uložena stejná data, či nikoli. Primárním klíčem záznamu o stavu kandidátního úložiště je znak „1“, čímž je možné ho jednoznačně odlišit od dat modelu („Path“ začíná vždy s „/“).

V databázi **Redis** je stav kandidátního úložiště také ukládán společně s daty modelu (viz Výpis 4.4).

```
sr_<datastore>:<shm_prefix>:<module>:glob:candidate-modified
```

Výpis 4.4: Forma primárního klíče v databázi Redis pro stav kandidátního úložiště.

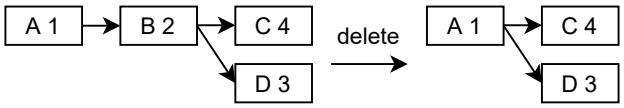
4.9 Uživatelem řazené seznamy

V modelech YANG nalezneme dva druhy seznamů. U systémem řazených seznamů není důležité, v jakém pořadí jsou data ukládána a v jakém pořadí jsou navrácena. Systém sám zvolí způsob, jakým bude se seznamy pracovat. V případě uživatelem řazených seznamů je důležité, aby systém dodržoval pořadí, v jakém jsou prvky seznamů do systému ukládány. V případě ukládání do souborů toto nebylo nutné řešit, jelikož byl celý soubor vždy přepsán novými daty, a tedy i novým pořadím prvků v seznamech. V základní verzi databáze MongoDB podporuje uživatelem řazené záznamy, tedy dokáže vracet jednotlivé záznamy ve stejném pořadí, v jakém byly uloženy. Ovšem systém musí umět podporovat i vložení prvku například doprostřed seznamu a na takový případ užití již databáze MongoDB připravena není.

Struktura **diff** obsahuje všechny provedené změny, které je potřeba aplikovat. V rámci uživatelem řazených seznamů se jedná o **vložení** nového prvku, **smazání** existujícího prvku a **přesun** prvku v rámci seznamu na novou pozici. Je nutno poznamenat, že všechny tyto změny v rámci seznamu jsou na sobě závislé a je zapotřebí je provést ve správném pořadí (není tedy možné vložit všechny nové prvky najednou v rámci jedné transakce).

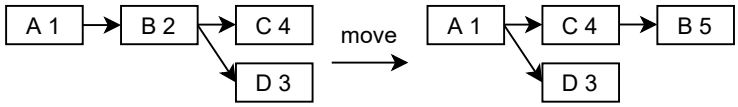
Jako první řešení tohoto problému se nabízí ukládat pro každý prvek **odkaz na předchozí prvek** (tj. unikátní hodnota předchozího prvku v rámci seznamu) a **relativní pořadové číslo prvku** pro rozlišení prvků, které se odkazují na stejný předchozí prvek. K tomuto je s výhodou možné využít knihovnu libyang, která nabízí funkci, jež je schopna postupně řadit prvky seznamu do uzlové struktury pouze na základě hodnoty řazeného prvku a hodnoty předchozího prvku. Předchozí prvek musí již být přítomen v seznamu, když je vkládán nový prvek. Je tedy nutno zajistit, aby předchozí prvek byl při načítání dat z databáze zpracován jako první.

Vytvoření nového prvku má v tomto případě konstantní složitost, jelikož při vkládání nový prvek dostane inkrementované pořadové číslo. **Čím vyšší je relativní pořadové číslo, tím blíže je daný prvek ke svému předchozímu prvku**. Prvek, za který má být vložen nový prvek, již může mít před sebou prvek, který se na něj odkazuje. Pokud by pořadí bylo vzestupné, pak by bylo potřeba při vložení nového prvku změnit relativní pořadové číslo všem prvkům, které se odkazují na stejný prvek jako nový prvek. Pokud ale uvažujeme sestupné pořadí, pak stačí pouze nastavit novému prvku pořadové číslo o jedna větší než má poslední vložený prvek, čímž je dosažena konstantní složitost. Odstranění má lineární složitost, přičemž je změněn odkaz na předchozí prvek u všech následujících prvků, aby se po smazání prvku následující prvky neodkazovaly na neexistující prvek (viz Obrázek 4.2).



Obrázek 4.2: Při vymazání prvku B je přiřazen prvku C a D nový předchůdce, a sice A. Pořadová čísla měněna nejsou. Čím vyšší je pořadové číslo, tím blíže je prvek svému předchozímu prvku (výsledné pořadí: ACD).

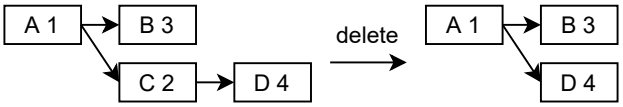
Přesunutí prvku má lineární složitost, přičemž stejně jako u odstranění je nejprve změněn odkaz na předchozí prvek u následujících prvků. Poté je u přesouvaného prvku změněn odkaz na předchozí prvek a je inkrementováno pořadové číslo (viz Obrázek 4.3). Problémem tohoto návrhu je, že není funkční za všech okolností (viz Obrázek 4.4).



Obrázek 4.3: Při přesunu prvku B je přiřazen prvku C a D nový předchůdce, a sice A. Prvek B dostane nového předchůdce a změní se mu pořadové číslo, aby se stal prvním prvkem za svým předchůdcem (výsledné pořadí: ACDB).

Druhým možným řešením je uložení absolutního pořadí, podle kterého budou data načtena. V rámci databáze budou tedy ukládány ke každému prvku **unikátní cesta** k prvku (jako primární klíč), **hodnota předchozího prvku**, který se v seznamu nachází před ukládaným prvkem, **absolutní pořadí** prvku v seznamu a nakonec **cesta k prvku bez predikátu** (tedy bez hodnoty daného prvku), viz Obrázek 4.5.

Absolutní pořadí nově přidaných prvků na konec seznamu je vždy o 1000 větší než aktuální největší pořadí existujícího prvku v seznamu. Toto je zavedeno z důvodu rychlejšího přidávání nových prvků do seznamu. Prvek, který bude vložen mezi prvky s pořadím 1000 a 2000 bude mít jednoduše pořadí 1500, tedy prostřední hodnotu pořadí mezi sousedními

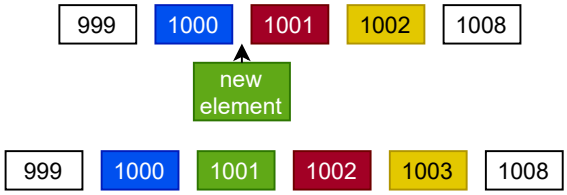


Obrázek 4.4: Původní pořadí prvků je ABCD (čím vyšší pořadové číslo, tím blíže se nachází ke svému předchůdci). Při vymazání prvku C je přiřazen prvku D nový předchůdce, a sice A. Správné pořadí má být ABD, ovšem prvek D má vyšší pořadové číslo než B, a tudíž bude načten jako první (výsledné pořadí: ADB).

path	prev	order	path_no_pred
/module:cont/list[key='value']	[key='prev']	2000	/module:cont/list

Obrázek 4.5: Příklad záznamu u prvku uživatelsky řazeného seznamu.

prvky. Pokud dojde k situaci, kdy by prvky mezi sebou neměly žádnou mezeru, tedy např. pořadí 1000 a 1001, je prvku přiřazeno pořadí 1001 a původní prvek s pořadím 1001 se pokusí získat pořadí 1002. Pokud pořadí 1002 již vlastní jiný prvek, bude tomuto původnímu prvku s pořadím 1002 přiřazeno pořadí 1003, atd. (viz Obrázek 4.6). Absolutní pořadí je uchováváno jako 64bitové celé číslo. Mezera o velikosti 1000 je dostatečně velká na to, aby do ní bylo možno vložit mnoho prvků bez nutnosti změny dalších a dostatečně malá na to, aby co nejméně zkrátila množství prvků, které lze uložit. Operace **vložení** nového prvku do seznamu bude mít tedy v nejhorším případě lineární složitost. Největší aktuální pořadí prvku v seznamu je ukládáno jako samostatný záznam.

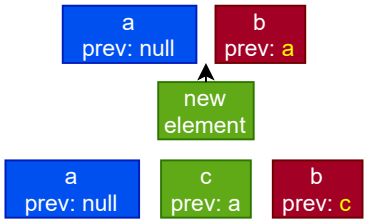


Obrázek 4.6: Příklad změny pořadí prvků v seznamu při vložení nového prvku (prvky jsou jednoznačně rozlišeny podle barev).

Hodnota předchozího prvku slouží k vyhledání prvků, mezi které bude nový prvek vložen. Pomocí hodnoty předchozího prvku lze získat předchozí prvek (pomocí **unikátní cesty**) a následující prvek (pomocí **hodnoty předchozího prvku**). Při vložení nového prvku do seznamu je zapotřebí změnit odkaz na předchozí prvek (**hodnotu předchozího prvku**) u následujícího prvku (viz Obrázek 4.7).

Cesta k prvku bez predikátu slouží pouze pro načítání dat. Při načítání dat z databáze se prvky nejprve řadí podle **absolutního pořadí** a následně podle **cesty k prvku bez predikátu** lexikograficky. Tímto způsobem je možno z databáze získat všechny uživatelské seznamy seřazené a k tomu ještě v pořadí od uzlů s nejkratší cestou od kořene po uzly s nejdelší cestou od kořene, což je podstatné při budování výstupního datového stromu.

Při mazání prvku ze seznamu je potřeba následujícímu prvku v seznamu nastavit nový odkaz na předchozí prvek, aby nedošlo k tomu, že by následující prvek obsahoval odkaz na předchozí prvek, který již neexistuje. Pořadí se při mazání neupravuje. Vymazání prvku má konstantní složitost. Při přesunutí prvku v seznamu je prvek nejprve smazán a poté vytvořen na jiném místě, čili v nejhorším případě bude mít lineární složitost.



Obrázek 4.7: Příklad změny hodnoty předchozího prvku (žlutě) u následujícího prvku při vložení nového prvku.

4.10 Značka výchozího stavu

Značka výchozího stavu (z angl. default flag) v modelu YANG označuje, že pokud uzlu není přiřazena žádná hodnota, má nastavenou výchozí hodnotu, která je dána modelem. Ve výchozím pluginu byla značka ukládána společně s daty modelu jako atribut uzlu. Značka buď je nastavená, nebo není. To znamená, že datový uzel buď nemá nastavenou žádnou hodnotu (tedy má výchozí hodnotu), anebo má nastavenou hodnotu explicitně. Ve stromové struktuře knihovny libyang má každá značka u uzlu nastavenou proměnnou, která tento stav uchovává. V databázi je uložena společně s daty uzlu. K těmto datovým uzlům pak již lze po načtení jednoduše připojit informaci o tom, zda obsahují výchozí hodnotu, či nikoli.

4.11 Úložiště operačních dat

S úložištěm Operational pracuje přímo systém a mohou se v něm vyskytovat nekonzistence jako např. duplicitní data. Operačními odběry je možno ovlivnit stav úložiště Operational, ale uživatel není schopen přímo přepsat data úložiště. Budování struktury „diff“ pro úložiště Operational není přesné a neodpovídá rozdílu mezi ukládanými daty a právě uloženými daty. Z tohoto důvodu je potřeba ukládat všechna data do úložiště znovu.

Mezi ukládanými daty se navíc vyskytují datové uzly typu **opaque**, které nejsou v modelu žádným způsobem zaznamenány. Tuto informaci je také nutné ukládat kvůli zpětné rekonstrukci datového stromu při načítání dat. Také sem patří **metadata** a **atributy** jednotlivých uzlů. Tato data jsou ukládána separátně.

U databáze **MongoDB** mají primární klíče metadat prefix „2“, za nimi následuje „Path“ (unikátní cesta k uzlu, kterého jsou součástí) a posléze ještě jejich název. Tato kombinace zajišťuje unikátnost primárního klíče. Atributy fungují stejně až na prefix, ten je „3“.

V databázi **Redis** se primární klíče metadat a atributů skládají z prefixů pro úložiště, paměťový prostor a model, z prefixu „meta“, z unikátní cesty k uzlu, z názvu modelu, který definuje daná metadata, nebo atributy, a z názvu metadat, nebo atributu (viz Výpis 4.5).

```
sr_<datastore>:<shm_prefix>:<module>:meta:<path>:<meta_module>:<meta>
```

Výpis 4.5: Forma primárního klíče v databázi Redis pro metadata a atributy.

Kapitola 5

Optimalizace a zhodnocení výsledných rychlostí pluginů

Ke změření výsledných rychlostí pluginů byly implementovány výkonnostní testy. Ty na běžném seznamu a uživatelsky řazeném seznamu o n prvcích měří běžné operace jako např. načtení celého datového stromu, načtení jedné položky, uložení více položek, atd. Část testů byla již implementována pro měření rychlostí knihovny sysrepo a knihovny libyang. Dále byly rozšířeny o 8 dalších pro změření rychlosti operací změny a odstranění prvků, operací s uživatelsky řazenými seznamy a operací s úložištěm Operational (**get user ordered tree**, **create user ordered items**, **remove all items**, **create an item**, **modify an item**, **remove an item**, **create all items oper** a **create an item oper**). Dosud nebyl důvod měřit tyto operace, protože výchozí plugin manipuluje se všemi daty najednou a při ukládání celého uživatelsky řazeného seznamu do souboru je implicitně dodrženo pořadí.

Podobně jako při výběru vhodné databáze se zde pracuje se seznamem o n prvcích. Ovšem při výběru databáze byly ke zpracování dat použity funkce knihovny libyang. Zde se již plně využívá funkcí knihovny sysrepo, které ke komunikaci s jednotlivými databázemi/soubory spoléhají na příslušné pluginy. Časy jsou měřeny v jazyce C opět za pomoci funkce `clock_gettime()` ze standardní knihovny jazyka C. Všechny testy byly pro různě velké seznamy provedeny 10krát a výsledky byly zprůměrovány. U všech testů je uvedena délka trvání a kolikrát je plugin v daném testu rychlejší či pomalejší než původní plugin (**JSON DS**).

K odhalení nejvytíženějších míst v kódu byl dále využit nástroj Callgrind. Jedná se o modul nástroje Valgrind, který slouží k profilování programů. Sleduje vykonané instrukce a tvoří na základě nich statistiky, pomocí nichž lze následně optimalizovat jednotlivé části programu[1].

5.1 Náměty na optimalizace a optimalizace operací s daty

Prvním problémem u výkonnostních testů byla rychlost pluginu pro databázi MongoDB při ukládání dat uživatelsky řazeného seznamu. Při velikosti seznamu 10000 prvků trvala operace u původního pluginu 0,5 s a u databáze Redis přibližně 3,5 s, ovšem u databáze MongoDB 33 s, což značně zpomalovalo testy. Při bližším prozkoumání bylo zjištěno, že většinu času strávil test v databázi při načítání prvku, před který měl být daný prvek uložen. Pro představu při jednom měření bylo celkové trvání testu 36 s, přičemž z toho pouze

[1] `http://www.fit.vutbr.cz/~martinek/clang/profiling.html`

načítáním následujících prvků strávil test 31,5 s. Načítání následujících prvků je podmíněno hodnotami dvou polí záznamů prvků (hodnota předchozího prvku a cesta k prvku bez predikátu). Tato dvě pole ovšem nejsou indexována, proto trvalo načítání dlouho. Pro jedno pole (hodnota předchozího prvku) byl vytvořen index, který zrychlil načítání prvků. Druhé pole (cesta k prvku bez predikátu) se ovšem vyskytuje i u prvků systémem řazených seznamů a přítomnost indexu by mohla negativně ovlivnit rychlost ukládání prvků do těchto seznamů. Zároveň index u tohoto pole by mohl být užitečný jedině tehdy, kdyby uživatel definoval spoustu uživatelsky řazených seznamů, které by obsahovaly stejné hodnoty prvků. Tehdy by totiž vyhledávání prvku podle předchozí hodnoty nebylo jednoznačné a bylo by potřebné hledání často upřesňovat právě na základě cesty k prvku bez predikátu. Takováto situace je ovšem při běžném používání vysoce nepravděpodobná. Po přidání indexu na pole hodnoty předchozího prvku se rychlost databáze MongoDB při ukládání 10000 prvků uživatelsky řazeného seznamu dostala na 6,2 s. Přidáním druhého indexu se čas nezměnil, proto byl tento index odebrán. U databáze Redis byly již indexy přidány v rámci implementace, jelikož příkaz, který načítá data, je přímo vyžaduje.

Druhým problémem byly rychlosti při ukládání, modifikaci a mazání jednoho prvku v seznamu. Ty byly totiž oproti původnímu pluginu zanedbatelně lepší, nebo horší. Taková situace je však nepřípustná, jelikož nové pluginy by měly právě v tomto případě podávat prokazatelně lepší výkon. Při bližším prozkoumání bylo zjištěno, že při budování struktury změn **diff** jsou nejprve načítána všechna data pomocí pluginu, pak jsou na ně aplikovány změny v rámci interních funkcí knihovny sysrepo a výsledný strom změn je následně pomocí pluginu uložen. Načítání dat pomocí nových pluginů ovšem není v tomto okamžiku ideální, jelikož jako primární klíč figuruje unikátní cesta k uzlu v podobě textového řetězce. Ten musí být zdlouhavě zpracován funkcemi knihovny libyang, aby mohl být integrován jako uzel do výsledného datového stromu. Pokud je ovšem datový strom uložen v mezipaměti, pak ukládání nového prvku neobnáší zdlouhavé načítání všech dat a časy pak lépe odpovídají rychlosti samotného ukládání. Z tohoto důvodu byly do výkonnostních testů přidány další 4 testy, které využívají mezipaměť a jejich časy tedy odráží přímo ukládání bez načítání (**remove all items cached**, **create an item cached**, **modify an item cached** a **remove an item cached**). Lepší ukládání dat bez přímého použití výrazů XPath jako primárního klíče je otázkou budoucích optimalizací.

Třetím problémem bylo načítání malého množství dat. V takovémto případě totiž načítaly databáze MongoDB a Redis všechna data. Při načítání dat ovšem není nutno vždy načítat všechna data. Často stačí pouze jeden uzel či podstrom v daném modelu. Při načítání může knihovna sysrepo poskytnout pluginu seznam výrazů XPath, které označují, jaká data musí být vrácena. Pokud jsou tyto výrazy jednoznačnou cestou k uzlu, či podstromu (bez použití operátorů), pak je možné použít daný výraz jako součást regulárního výrazu při vyhledávání konkrétních uzlů v databázi. Výraz XPath, který používá operátory (např. and, or, atd.), není vhodný k tvorbě regulárního výrazu. To proto, že operátory nejsou součástí unikátní cesty k uzlu, která byla uložena do databáze (viz Výpis 5.1). Naproti tomu výraz XPath (viz Výpis 5.2) lze použít při tvorbě regulárního výrazu (viz Výpis 5.3).

```
/perf:cont/lst[k1='0' and k2='str0']/l
```

Výpis 5.1: Výraz XPath, který není přímo použitelný jako součást vyhledávání.

```
/perf:cont/lst[k1='0'][k2='str0']/l
```

Výpis 5.2: Výraz XPath, který lze přímo využít jako součást vyhledávání.

```
\/perf\:cont\/lst\[k1\=\'0\'\]\[k2\=\'str0\'\]\/l.*
\/perf\:cont\/lst\[k1\=\'0\'\]\[k2\=\'str0\'\]\/l*
```

Výpis 5.3: Konkrétní podoba regulárního výrazu pro databáze MongoDB a Redis. Před všechny speciální znaky je umístěno zpětné lomítko.

To, zda je výraz XPath unikátní cestou k uzlu či podstromu, lze zjistit pomocí funkce z knihovny libyang. Výrazy XPath, které používají operátory, by bylo potřeba nejprve upravit. Výsledný regulární výraz by pak byl komplikovanější. Podpora vyhledávání podle takových výrazů XPath je dobrým námětem pro další optimalizace.

Dalším problémem je hromadné ukládání prvků do databáze Redis. Při ukládání mnoha prvků totiž databáze MongoDB podává lepší výkon než původní plugin. Tohoto je docíleno pomocí specifické funkce API knihovny libmongoc, která umožňuje uložení více prvků najednou. V databázi Redis je momentálně k uložení prvků používán příkaz **HSET**, který nemá alternativu, jež by byla schopna ukládat více prvků najednou. Původně se počítalo s využitím příkazů **SET** a **MSET**. Jenomže při použití těchto příkazů není možné řadit prvky při vyhledávání. Do databáze Redis je nyní v implementaci pro každý nově vytvořený prvek posílán samostatný nový příkaz. Řazení prvků je možné oddělit od vyhledávání tak, že bude existovat seznam seřazených prvků podle cesty k prvku bez predikátu a popř. seřazených uživatelsky řazených seznamů. Podle těchto seznamů následně mohou být vrácena postupně všechna data. Vedlo by to však na daleko více příkazů poslaných databázovému serveru. Tím pádem není jisté, zda by se celkově nezhoršilo vyhledávání.

Dalším problémem je postupné vymazávání prvků u obou databází. Pro každý odstraňovaný prvek totiž existuje separátní příkaz, který ho maže, což je velmi pomalé. Databáze MongoDB však poskytuje funkci, která je schopna mazat prvky za použití regulárního výrazu. Tím pádem by např. vymazání celého podstromu bylo otázkou pouze jednoho příkazu. Samozřejmě problém nastává, pokud chceme vymazat pouze většinu podstromu. Pak je opět zapotřebí vymazat prvky postupně, anebo vytvořit složitější regulární výraz.

5.2 Zhodnocení výsledných rychlostí pluginů

Test **get tree** měří, jak dlouho trvá načítání všech prvků z uložených dat. K tomuto využívá výraz XPath s operátorem **and** pro načtení konkrétního prvku, tudíž oba nové pluginy načítají všechna data (výraz XPath s operátorem **and** totiž nemůže být využit při tvorbě regulárního výrazu pro vyhledání konkrétního uzlu v databázi, viz Sekci 5.1). Původní plugin načítá vždy všechna data bez ohledu na výrazy XPath. Při načítání všech dat podávají nové pluginy zhruba 3-4krát horší výkon než původní plugin.

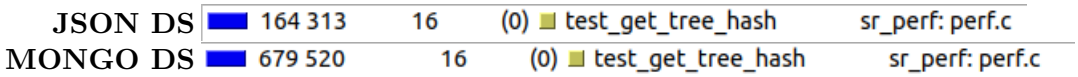
Test **get tree hash** měří, jak dlouho trvá načítání jednoho prvku ze všech uložených dat. K tomu využívá výraz XPath bez speciálních operátorů, který figuruje jako jednoznačná cesta k uzlu. V tomto případu využívá knihovna libyang hashovací funkci k vyhledání konkrétního uzlu z vrácených dat v konstantním čase. Tím pádem sice původní plugin musí načíst všechna data do paměti, ale vyhledání uzlu v paměti je již efektivní. Nové pluginy však načítají pouze jeden prvek, tím pádem u nich hashovací funkce zpracování dat neurychlí. Při načítání jednoho prvku podává databáze MongoDB 7krát lepší výkon a databáze Redis 12-13krát lepší výkon. Databáze Redis je rychlejší při načítání malého množství dat z databáze.

Test **get tree hash cached** měří, jak dlouho trvá vrácení dat, pokud je naplněna mezipaměť (mezipaměť může být využita pouze pro úložiště Running). Při tomto testu nejsou načítána data z databází, ovšem jak databáze MongoDB, tak databáze Redis podávají horší

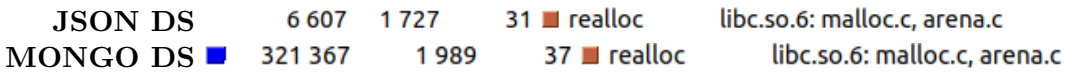
	JSON DS	MONGO DS	REDIS DS
test name	time[s]	time[s] comp[x]	time[s] comp[x]
get tree	0,008140	0,025304 3,108	0,029240 3,592
get tree hash	0,005421	0,000783 6,923	0,000441 12,292
get tree hash cached	0,000046	0,000331 7,195	0,000313 6,804
get user ordered tree	0,004605	0,020340 4,416	0,029282 6,358
create user ordered items	0,055345	0,586641 10,599	0,357596 6,461
create all items	0,077401	0,059971 1,290	0,208127 2,688
create all items oper	0,127199	0,077485 1,641	0,381277 2,997
remove all items	0,001183	0,041503 35,082	0,016935 14,315
remove all items cached	0,000879	0,038194 43,451	0,013763 15,657
create an item	0,044981	0,025368 1,773	0,029762 1,511
create an item cached	0,042372	0,004723 8,971	0,004064 10,426
create an item oper	0,085445	0,095992 1,123	0,508568 5,951
modify an item	0,009895	0,023984 2,423	0,028085 2,838
modify an item cached	0,006922	0,004684 1,477	0,004992 1,386
remove an item	0,009631	0,023550 2,445	0,028268 2,935
remove an item cached	0,006966	0,005073 1,373	0,005294 1,315

Tabulka 5.1: Výkonnostní testy pro 1000 prvků seznamu. Kolikrát jsou testy u pluginu pomalejší než u pluginu **JSON DS** je naznačeno červenou barvou a kolikrát rychlejší zelenou.

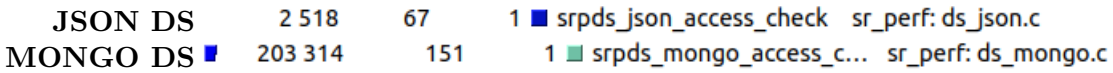
výkon. Z výpisu nástroje Callgrind vyplývá, že při používání pluginu pro databázi MongoDB je provedeno daleko více instrukcí než u původního pluginu (viz Obrázek 5.1). Databázový server totiž při své činnosti také alokuje značné zdroje, proto je následně činnost pluginu zpomalena delší alokací paměti (viz Obrázek 5.2). Při získávání přístupových práv k modelu stačí původnímu pluginu pouze přečíst metadata souboru, kdežto databázové pluginy musí načítat data přímo z databáze (viz Obrázek 5.3).



Obrázek 5.1: Výpis z nástroje Callgrind ukazuje přibližně čtyřnásobný počet instrukcí u pluginu pro databázi MongoDB.



Obrázek 5.2: Výpis z nástroje Callgrind ukazuje na problémy s dynamickou alokací paměti.



Obrázek 5.3: Výpis z nástroje Callgrind ukazuje na rozdíl mezi čtením přístupových práv z databáze a z metadat souboru.

Test **get user ordered tree** měří, jak dlouho trvá vrácení dat uživatelem řazeného seznamu. Při tomto testu musí obě databáze kromě vrácení všech dat ještě data správně seřadit podle absolutního pořadí. V tomto případu užití podává databáze MongoDB lepší výkon než databáze Redis.

Test **create user ordered items** měří, jak dlouho trvá uložení všech prvků uživatelem řazeného seznamu. Při tomto testu je potřeba postupně uložit všechny prvky jeden po druhém, protože operace pro uložení a přesun prvků jsou na sobě závislé a musí být provedeny ve správném pořadí. Zároveň je při uložení každého nového prvku nutno načíst z databáze pořadí předchozího prvku a pořadí následujícího prvku (viz Sekci 4.9). Jelikož je potřeba načítat data z databáze a k tomu nelze ukládat všechna data najednou, podává databáze MongoDB znatelně horší výkon než při ukládání prvků systémem řazeného seznamu. Databáze Redis v tomto případě vyniká díky lepší schopnosti načítat malé množství dat.

Test **create all items** měří, jak dlouho trvá uložení všech prvků systémem řazeného seznamu. Při tomto testu nezáleží na vstupním pořadí prvků. Díky tomu může databáze MongoDB uložit všechny prvky najednou. Databáze MongoDB v tomto případě podává lepší výkon než původní plugin, zatímco databáze Redis je nucena ukládat všechny prvky postupně po jednom, a tudíž je horší.

Test **create all items oper** měří, jak dlouho trvá uložení všech prvků systémem řazeného seznamu do úložiště Operational. Úložiště Operational je v tomto případě na začátku prázdné a probíhá pouze ukládání prvků, tudíž se časy přibližují testu **create all items**.

Test **remove all items** měří, jak dlouho trvá vymazání všech prvků systémem řazeného seznamu. V tomto případě podávají obě databáze mnohem horší výkon než původní plugin, přičemž databáze Redis je zhruba 2-3krát efektivnější než databáze MongoDB.

Test **remove all items cached** měří, jak dlouho trvá vymazání všech prvků systémem řazeného seznamu za využití mezipaměti. Díky přítomnosti aktuálního datového stromu v mezipaměti pluginy nemusí načítat data z databáze pro vytvoření stromu datových změn **diff**. Tyto časy lépe odpovídají celkové rychlosti pluginů při vymazání všech prvků seznamu.

Test **create an item** měří, jak dlouho trvá vytvoření nového prvku v seznamu prvků. Obě databáze podávají lepší výkon než původní plugin.

Test **create an item cached** měří, jak dlouho trvá vytvoření nového prvku v seznamu prvků za využití mezipaměti. Toto je hlavní případ užití nových pluginů. Při ukládání malého množství dat vykazují mnohonásobné zrychlení oproti původnímu pluginu. Další optimalizace tohoto případu užití by již byly zanedbatelné, jelikož ve funkci, která data ukládá (operace store), se tráví pouze zlomek celého času. Z měření časů této funkce vychází, že při aplikování změn tráví původní plugin přibližně 90% času v této funkci. Kdežto plugin pro databázi MongoDB zde tráví 12% času a plugin pro databázi Redis pouze 10% času. Z toho vyplývá, že následné optimalizace by se zde vyplatily pouze v omezeném rozsahu.

Test **create an item oper** měří, jak dlouho trvá vytvoření nového prvku v seznamu prvků v úložišti Operational. Poněvadž není možné v tomto případě použít strom datových změn **diff**, je potřeba nejprve všechna data z databáze odstranit a poté všechna data (i s novým prvkem) znovu uložit. Z tohoto důvodu je test **create all items oper** rychlejší než tento, protože nemusí odstranit již uložená data. V tomto případě podávají oba nové pluginy horší výkon, přičemž databáze MongoDB je rychlejší díky hromadnému uložení nových prvků.

Test **modify an item** měří, jak dlouho trvá změna hodnoty jednoho prvku v seznamu prvků. V tomto případě jsou oba nové pluginy horší než původní plugin, protože je nejprve potřebné načíst data k vytvoření stromu změn (načítání všech dat je u nových pluginů pomalejší).

Test **modify an item cached** měří, jak dlouho trvá změna hodnoty jednoho prvku v seznamu prvků za využití mezipaměti. Pluginy podávají lepší výkon než původní plugin, protože nemusí načítat data. Zároveň zde není vidět takový rozdíl jako u testu **create an item cached**, protože původní plugin v tomto případu užití podává mnohem lepší výkon.

Test **remove an item** měří, jak dlouho trvá odstranění jednoho prvku ze seznamu prvků. Opět zde oba nové pluginy podávají horší výkon, protože musí načítat data (načítání všech dat je u nových pluginů pomalejší).

Test **remove an item cached** měří, jak dlouho trvá odstranění jednoho prvku ze seznamu prvků za využití mezipaměti. Zde oba nové pluginy podávají lepší výkon než původní plugin, ovšem opět ne tak signifikantně jako u testu **create an item cached**, protože i v tomto případu užití podává původní plugin mnohem lepší výkon.

	JSON DS	MONGO DS	REDIS DS
test name	time[s]	time[s] comp[x]	time[s] comp[x]
get tree	0,081018	0,260090 3,210	0,355047 4,382
get tree hash	0,057069	0,000761 74,992	0,000564 101,186
get tree hash cached	0,000110	0,000725 6,590	0,000616 5,599
get user ordered tree	0,044565	0,219780 4,931	0,390810 8,769
create user ordered items	0,568905	5,767643 10,138	3,640026 6,398
create all items	0,786388	0,604520 1,300	2,113323 2,687
create all items oper	1,278625	0,784385 1,630	3,945229 3,085
remove all items	0,011765	0,406011 34,510	0,183353 15,584
remove all items cached	0,009048	0,375506 41,501	0,146214 16,159
create an item	0,437442	0,261895 1,670	0,372802 1,173
create an item cached	0,408816	0,042750 9,562	0,040449 10,106
create an item oper	0,901205	0,958575 1,063	5,099239 5,658
modify an item	0,096530	0.259919 2,692	0,391533 4,056
modify an item cached	0,067587	0,049552 1,363	0,060481 1,117
remove an item	0,093633	0,246189 2,629	0,359989 3,844
remove an item cached	0,065235	0,049596 1,315	0,060089 1,085

Tabulka 5.2: Výkonnostní testy pro 10000 prvků seznamu. Kolikrát jsou testy u pluginu pomalejší než u pluginu **JSON DS** je naznačeno červenou barvou a kolikrát rychlejší zelenou.

Při manipulaci s desetinásobkem prvků (viz Tabulku 5.2) se rozdíl mezi původním pluginem a novými pluginy velmi nezměnil. U testu **get tree hash** jsou obě databáze schopny vyhledat konkrétní prvek prakticky v konstantním čase bez ohledu na velikost množiny prohledávaných dat, takže se zde projevil mnohonásobně větší rozdíl než u testů s tisícem prvků v seznamu.

Co je nejvíce zarážející, jsou časy při manipulaci pouze s jedním prvkem v seznamu. Tam se očekává mnohonásobné zrychlení oproti původnímu pluginu, avšak rozdíl zůstává stejný. Při průzkumu testů pomocí nástroje Callgrind bylo zjištěno, že při násobném zvětšení dat mají nové pluginy konstantní počet instrukcí. Tudíž v případě 10000 prvků již nové pluginy nezabírají 12% a 10% času, nýbrž přibližně pouhých 1,2% a 1% celkového času. Avšak ostatní instrukce knihovny sysrepo pro validaci, duplikaci dat, atd. se znásobily. Aby se tedy větší rozdíl mezi pluginy projevil, musela by být optimalizována samotná knihovna sysrepo.

	JSON DS	MONGO DS	REDIS DS
test name	time[s]	time[s] comp[x]	time[s] comp[x]
get tree	1,046213	2,875000 2,748	4,433513 4,237
get tree hash	0,721840	0,000663 1088,748	0,000504 1432,222
get tree hash cached	0,003479	0,003639 1,045	0,004588 1,318
get user ordered tree	0,630188	2,402602 3,812	4,518279 7,169
create user ordered items	6,052073	60,134133 9,936	35,340707 5,839
create all items	8,563663	6,437559 1,330	22,649828 2,644
create all items oper	13,713490	8,095914 1,693	40,470668 2,951
remove all items	0,147715	4,176438 28,273	2,061058 13,952
remove all items cached	0,128475	3,926555 30,562	1,481641 11,532
create an item	4,933221	2,988942 1,650	4,835745 1,020
create an item cached	4,488900	0,507115 8,851	0,488404 9,190
create an item oper	9,662270	9,904909 1,025	54,577531 5,648
modify an item	1,070803	2,897996 2,706	4,272252 3,989
modify an item cached	0,814250	0,596275 1,365	0,749451 1,086
remove an item	1,100931	2,682772 2,436	4,242243 3,853
remove an item cached	0,869094	0,581594 1,494	0,752640 1,154

Tabulka 5.3: Výkonnostní testy pro 100000 prvků seznamu. Kolikrát jsou testy u pluginu pomalejší než u pluginu **JSON DS** je naznačeno červenou barvou a kolikrát rychlejší zelenou.

Kapitola 6

Závěr

Cílem této práce bylo seznámit čtenáře s knihovnou sysrepo a jejím mechanismem pluginů pro úložiště, seznámit s problémem ukládání dat do souborů a navrhnout řešení v podobě ukládání dat do databáze. Dále byly porovnány vybrané databáze podle rychlosti ukládání dat na ukázkovém modelu YANG s dosavadní implementací a byly vybrány dvě nejlepší, MongoDB a Redis. Následně byly pluginy pro obě databáze implementovány a byla provedena měření rychlosti u všech pluginů při běžných operacích. Na základě výsledků těchto měření byly odhaleny nedostatky v implementaci, byly provedeny některé optimalizace, přičemž byla navržena řada dalších optimalizací, které byly naplánovány do budoucna. Hlavní motivací této práce bylo urychlení operací s malým množstvím dat v knihovně sysrepo za využití databáze. Tento cíl byl splněn, jak vyplývá z naměřených časů u testů pracujících s malým množstvím dat (**get tree hash**, **create an item cached**, **modify an item cached**, **remove an item cached**). Další urychlení ukládání, změny či odstranění malého množství dat je dále možné pouze optimalizací samotné knihovny sysrepo. Pokračováním této práce bude provést další optimalizace, zejména co se týče efektivnějšího načítání dat a dále tak zvýšit rychlost obou pluginů při běžném použití.

Literatura

[1] Overview of MarkLogic Server (Concepts Guide). *MarkLogic Server 11.0 Product Documentation* [online]. Květen 2019 [cit. 2023.06.01]. Dostupné z: https://docs.marklogic.com/guide/concepts/overview.

[2] What Is a Database. *Oracle* [online]. Září 2021 [cit. 2023.01.27]. Dostupné z: https://www.oracle.com/database/what-is-database/.

[3] Types of NoSQL Databases. *GeeksforGeeks* [online]. Červenec 2022 [cit. 2023.01.27]. Dostupné z: https://www.geeksforgeeks.org/types-of-nosql-databases/.

[4] Cassandra Basics. *Apache Cassandra* [online]. 2023 [cit. 2023.06.01]. Dostupné z: https://cassandra.apache.org/_/cassandra-basics.html.

[5] Introduction to ArangoDB's Technical Documentation and Ecosystem. *ArangoDB Documentation* [online]. 2023 [cit. 2023.06.01]. Dostupné z: https://www.arangodb.com/docs/3.11/index.html.

[6] Database of Databases. *Bitsy* [online]. 2023 [cit. 2023.06.01]. Dostupné z: https://dbdb.io/db/bitsy.

[7] Introduction to MongoDB. *MongoDB Manual* [online]. 2023 [cit. 2023.06.01]. Dostupné z: https://www.mongodb.com/docs/manual/introduction/.

[8] Getting Started. *Welcome to Neo4j* [online]. 2023 [cit. 2023.06.01]. Dostupné z: https://neo4j.com/docs/getting-started/.

[9] Remote procedure call. *Wikipedia* [online]. Leden 2023 [cit. 2023.01.27]. Dostupné z: https://en.wikipedia.org/wiki/Remote_procedure_call.

[10] Introduction to Redis. *Redis* [online]. 2024 [cit. 2024.01.23]. Dostupné z: https://redis.io/docs/about/.

[11] Redis persistence. *Redis* [online]. 2024 [cit. 2024.01.23]. Dostupné z: https://redis.io/docs/management/persistence/.

[12] Riak KV Features. *Riak* [online]. 2024 [cit. 2024.01.23]. Dostupné z: https://riak.com/products/riak-kv/index.html.

[13] Bebee, B. Home. *Blazegraph Wiki* [online]. Únor 2020 [cit. 2023.06.01]. Dostupné z: https://github.com/blazegraph/database/wiki.

[14] Bjorklund, M. The YANG 1.1 Data Modeling Language. *RFC 7950* [online]. Srpen 2016 [cit. 2023.01.27]. DOI: 10.17487/RFC7950. ISSN 2070-1721. Dostupné z: https://www.rfc-editor.org/rfc/rfc7950.

[15] Bjorklund, M. XPath Evaluations. *RFC 7950* [online]. Srpen 2016 [cit. 2024.01.29]. DOI: 10.17487/RFC7950. ISSN 2070-1721. Dostupné z: https://datatracker.ietf.org/doc/html/rfc7950#section-6.4.

[16] Clark, J. a DeRose, S. XML Path Language (XPath). *W3C* [online]. Říjen 2016 [cit. 2023.01.27]. Dostupné z: https://www.w3.org/TR/1999/REC-xpath-19991116/.

[17] Dormando. Overview. *Memcached* [online]. 15. listopadu 2015 [cit. 2024.01.23]. Dostupné z: https://github.com/memcached/memcached/wiki/Overview.

[18] Kal, A., Graham, M. et al. Concepts. *BrightstarDB 1.13 documentation* [online]. 2016 [cit. 2023.06.01]. Dostupné z: https://brightstardb.readthedocs.io/en/latest/Concepts/#concepts.

[19] Rund, B. The Good, The Bad, and the Hype about Graph Databases for MDM. *TDWI* [online]. 14. března 2017 [cit. 2024.01.23]. Dostupné z: https://tdwi.org/articles/2017/03/14/good-bad-and-hype-about-graph-databases-for-mdm.aspx.

[20] Vaško, M. About. *Libyang* [online]. 2023 [cit. 2023.01.27]. Dostupné z: https://netopeer.liberouter.org/doc/libyang/devel/html/index.html.

[21] Vaško, M. Introduction. *Sysrepo* [online]. 2023 [cit. 2023.01.27]. Dostupné z: https://netopeer.liberouter.org/doc/sysrepo/devel/html/index.html.

[22] Vaško, M. Connection and Session. *Sysrepo* [online]. 2024 [cit. 2024.01.23]. Dostupné z: https://netopeer.liberouter.org/doc/sysrepo/devel/html/conn_sess.html.

[23] Vaško, M. Schemas. *Sysrepo* [online]. 2024 [cit. 2024.01.23]. Dostupné z: https://netopeer.liberouter.org/doc/sysrepo/devel/html/schema.html.

[24] Vaško, M. Getting Data. *Sysrepo* [online]. 2024 [cit. 2024.01.23]. Dostupné z: https://netopeer.liberouter.org/doc/sysrepo/devel/html/get_data.html.

[25] Vaško, M. Subscriptions. *Sysrepo* [online]. 2024 [cit. 2024.01.23]. Dostupné z: https://netopeer.liberouter.org/doc/sysrepo/devel/html/subs.html.