Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Diplomová práce

Simulátor distribuovaného souborového systému

Plzeň 2017 Martin Kučera

Místo této strany bude zadání práce.

Prohlášení

Prohlašuji, že jsem diplomovou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 17. května 2017

Martin Kučera

Poděkování

Rád bych poděkoval vedoucímu diplomové práce Ing. Luboši Matějkovi za odborné vedení, připomínky a konzultace poskytnuté při vypracování diplomové práce.

Abstract

The purpose of this thesis is to research the principles of distributed file systems with hierarchical storage models and design and implement a simulator that will allow us to simulate these systems. The theoretical section proposes models of distributed file systems with hierarchical models for data storage in order to increase the file system throughput. In the implementation section a distributed file system simulator is implemented in order to validate and compare proposed file system models.

Abstrakt

Cílem této práce je prostudovat principy fungování distribuovaného souborového systému a hierarchického modelu uložení dat a následně navrhnou a implementovat simulátor, který umožní simulaci těchto systémů. V teoretické části jsou navrženy modely pro uložení dat v distribuovaném souborovém systému s využitím hierarchického modelu pro ukládání dat za účelem zvýšení propustnosti souborového systému. Dále je vytvořen simulátor, pomocí kterého jsou navržené modely ověřeny a porovnány.

Obsah

1 Úvod

Cílem této práce je prozkoumat a porovnat různé modely pro uchování dat v distribuovaném souborovém systému s využitím hierarchického uložení dat a možnosti přístupu k těmto datům za účelem zvýšení propustnosti a následně navrhnout a implementovat simulátor, který umožní simulaci libovolných modelů distribuovaných souborových systémů.

V době s velkým množstvím uživatelů vlastnících mobilní zařízení přináší distribuované souborové systémy mnoho výhod oproti klasickým lokálním souborovým systémům díky zpřístupnění dat pomocí počítačové sítě. Spolu s výhodami ovšem přichází i určité nevýhody, mezi které se řadí např. zvýšená režie přenosu dat a nutnost zabezpečit data při jejich uchování či přenosu.

Jelikož mohou být v případě distribuovaného souborového systému data ukládána a zpřístupněna z několika různých umístění, je při návrhu a implementaci těchto systémů důležitým faktorem volba strategie pro práci s daty, a to konkrétně jejich ukládání, přenosy mezi jednotlivými servery a přenosy mezi servery a klienty.

Existuje mnoho různých distribuovaných souborových systémů, které se liší svými vlastnostmi. Jedním z těchto systémů je KIVFS [45], na jehož vývoji se podílí studenti a vyučující Katedry informatiky a výpočetní techniky na Západočeské univerzitě v Plzni. KIVFS má některé unikátní vlastnosti jako je např. dynamické routování při přenosu dat přes sousedící servery a aktuálně zkoumané a vyvíjené možnosti využití hierarchického modelu uložení dat, jejichž skutečné přínosy v reálném systému nelze dopředu jednoduše stanovit.

Testování a měření výkonu na reálném systému může být obtížné a to jak z důvodu časově náročně implementace tak z důvodu působení rušivých faktorů během měření. Faktory jako saturovaná linka nebo vytížení disku cizím procesem mohou nepříznivě ovlivnit naměřené výsledky.

Simulační nástroje umožňují ověřit a porovnat klíčové vlastnosti distribuovaných souborových systémů bez nutnosti funkčnost nejdříve implementovat a poté testovat na reálném systému. Simulace probíhá nad izolovaným

systémem, který je zproštěn rušivých faktorů. Jelikož je simulace deterministická, je možné měření v budoucnosti opakovat a naměřené hodnoty porovnat. Díky simulaci jsme tedy schopni dopředu určit, jaké bude chování systému v reálném provozu a zda jsou vybrané vlastnosti přínosné či nikoliv.

Požadovaným výsledkem této práce je simulátor, který bude napomáhat při výzkumu a vývoji distribuovaných souborových systémů.

2 Způsoby uložení dat

V počítači jsou nejčastěji data uložena pomocí souborového systému na lokálním úložišti. Souborový systém organizuje ukládání a čtení dat z fyzického média, kterým bývá pevný disk či nověji SSD a uložená data jsou přístupná pouze z daného počítače.

S velkým rozšířením mobilních zařízení, mezi které se řadí např. mobilní telefony a notebooky, se požadavky na ukládání a práci s daty mění. Lidé často vlastní více než jedno zařízení, na kterých pracují se svými daty. Ruční synchronizace těchto dat je nepraktická a zdlouhavá. Dalším omezením bývá limitovaný úložný prostor, který se u mobilních telefonů pohybuje typicky v jednotkách až desítkách GB. Tyto problémy se snaží řešit distribuované souborové systémy.

2.1 Lokální úložiště

Lokální úložiště představuje nejčastěji využívaný typ úložiště. Jedná se o paměťové médium fyzicky připojené k danému zařízení. Velikosti disků v osobních počítačích se dnes pohybují mezi 1 a 4 TB. Tato velikost typicky klesá spolu s velikostí zařízení, kde například u mobilních telefonů se dostáváme na jednotky až desítky GB.

V tabulce 2.1 jsou vidět některé typy úložišť a jejich orientační přenosová rychlost, typická velikost a cena za GB. Úložiště jsou v tabulce vzestupně

Tabulka 2.1: Vlastnosti vybraných typů úložišť

Typ úložiště	**Rychlost**	**Velikost**	**Cena za GB**
Magnetická páska	10-300 MB/s	jednotky TB	<1 Kč
HDD	200 MB/s	jednotky TB	1 Kč
SSD	500 MB/s	stovky GB	5 Kč
RAM	1-10 GB/s	jednotky GB	>100 Kč

Uvedené hodnoty jsou pouze orientační.

uspořádána podle ceny za GB. Je patrné, že přenosová rychlost roste spolu s cenou, zatímco velikost klesá až k jednotkám GB.

Nejrychlejším typem úložiště bývá úložiště založené na RAM paměti, zato nejpomalejším magnetická páska. Úložiště založené na RAM paměti sice nabídne největší rychlost ale data na nich nejsou trvale uložena, jedná se totiž o volatilní paměť, která vyžaduje konstantní napájení pro udržení informace a tak se většinou využívají pouze jako cachovací úložiště.

Magnetická páska je v tabulce uvedena jako nejpomalejší i přes to, že okamžitá přenosová rychlost dat může dosahovat vyšších hodnot než v případě HDD. Při přenosu dat je ovšem nutné pásku fyzicky přetočit na požadované místo, což způsobuje veliké zpomalení a tak je skutečná rychlost mnohem menší.

Lokální úložiště se vyznačují nízkou latencí přístupu k datům a obecně bývají nejlevnějším úložným prostorem. Mezi hlavní nevýhody se řadí chybějící redundance, u mobilních zařízení menší dostupný prostor a nutnost ruční synchronizace mezi více zařízeními.

2.2 RAID

U lokálních úložišť byla jako jedna z nevýhod zmíněna chybějící redundance, kterou je možné řešit technologií RAID. RAID je kombinace více fyzických disků do jedné logické jednotky za účelem zvýšení rychlosti nebo zajištění redundance dat či obojího. Existují dvě odlišné metody vytvoření RAIDu a to SW a HW.

Technologie RAID se typicky používá spíše na síťových úložištích a serverech, než na domácích stanicích. Podle použitého typu RAID může mít výsledný disk menší rychlost a menší výslednou kapacitu než při použití disků samostatně.

Některé populární typy RAID jsou uvedeny v tabulce 2.2 spolu s teoretickými zrychleními při čtení a zápisu, celkovou kapacitou a maximálním počtem tolerovaných výpadků disku.

HW RAID je tvořen pomocí řadiče, který umožňuje přístup operačnímu systému pouze k vytvořenému raid disku. Výpočet parity a logika spojená s ukládáním dat je přesunuta na HW řadič, čímž je ulehčeno procesoru. Další

Tabulka 2.2: Vlastnosti typů RAID

RAID	**Min. disků**	**Zr. čtení**	**Zr. zápis**	**Kapacita**	**Max. výpadků**
0 (prokl)	2	až n krát	až n krát	$n * c$	0
1 (zrc.)	2	až n krát	0	c	n- 1
5	3	až n - 1 krát	0	(n - 1) * c	1
6	4	až n - 2 krát	0	(n - 2) * c	2
10 (0+1)	4	až n krát	až n / 2 krát	(n / 2) * c	min. 1

Hodnoty zrychlení čtení a zápisu jsou teoretickými maximálními hodnotami a v praxi často nižší. Hodnota c představuje minimální velikost disku ze všech použitých disků v zapojení a n celkový počet použitých disků.

výhodou může být přítomnost nevolatilní cache paměti na řadiči, která slouží k minimalizaci ztracených dat při nečekaném výpadku napětí a pro urychlení čtení a zápisu.

SW RAID je vytvořen pomocí operačního systému. Jedná se o levnější variantu, která je dostupná ve většině známých operačních systémech. K vytvoření RAIDu lze použít různé typy úložišť a nejsme tak omezeni rozhraním HW řadiče. Obnova degradovaného SW RAIDu je zároveň více transparentní než u HW řešení díky možnosti detailně diagnostikovat stav nástroji pro správu RAIDu přítomnými přímo v OS nebo programy stavějícími nad funkcionalitou dostupnou v jádru OS. U HW RAIDu se může stát, že nástroje pro správu vůbec neexistují nebo mají v porovnání s nástroji pro SW raid pouze omezenou funkcionalitu.

2.3 Síťový souborový systém

Lokální úložiště (včetně RAIDu) neumožní přístup více klientům najednou, jelikož se jedná o vlastní úložiště každého klienta. Pro zpřístupnění dat více klientům lze využít síťového souborového systému.

Jedná se o souborový systém, který je přístupný pomocí počítačové sítě. Umožňuje uživateli pracovat se vzdálenými daty, jako kdyby byla uložena na lokálním disku. Data jsou fyzicky uložena na jiném počítači (serveru) a práce s nimi je umožněna pomocí speciálního protokolu.

Síťové souborové systémy jsou závislé na propustnosti sítě mezi klientem

a serverem, efektivitou protokolu a vyžadují větší výpočetní výkon procesoru než lokální úložiště. Samy o sobě nezaručují redundanci dat. Existuje více druhů síťových souborových systémů a nejpoužívanější budou popsány v následujících kapitolách.

2.3.1 Server Message Block (SMB)

SMB [8] je víceúčelový síťový protokol, který umožňuje mimo jiné přístup ke sdíleným datům a tiskárnám. Prvotní verze protokolu SMB 1 měla při přenosu dat velkou režii a tak se rychlost přenosu dat může propadat až k 1/5 maximální dostupné rychlosti. Ve verzi SMB 3 došlo k mnoha výkonnostním optimalizacím a tak se skutečná rychlost přenosu dat více blíží k té maximální. Hlavní výkonnostní optimalizací verze SMB 3 je technologie SMB Multichannel [6], která umožňuje využít více síťových spojení pro paralelní přenos dat.

Šifrování přenosu dat na úrovni protokolu u verzí SMB 1 a SMB 2 nebylo možné a šifrována byla pouze hesla pro autentizaci. Od verze SMB 3 lze šifrovat i samotný přenos dat šifrovacím algoritmem AES-CCM[23], pomocí kterého lze ověřit i integritu přenesených dat.

Nejčastěji používán na operačním systému MS Windows, ale existují i otevření klienti, mezi které patří např. *Samba* [5].

2.3.2 Network File System (NFS)

NFS [40] je implementací síťového souborového systému využívající technologie RPC. První verze protokolu, NFSv1, nebyla vydána na veřejnost a tak první veřejně dostupnou verzí byla až verze NFSv2 a poté její nástupce NFSv3 v roce 1995. V aktuální verzi NFSv4.1 umožňuje ukládat data do clusterů a využívat tak paralelního přístupu k datům [24]. Protokol lze od verze NFSv3 provozovat jak nad UDP tak nad TCP.

NFS je primárně používán na operačních systémech vycházejících z UNIXu, ale existují i implementace pro MS Windows. Sdílení dat je řízeno pomocí exportů, kde každý záznam specifikuje sdílenou složku, IP adresu klienta (či klientů) a další možnosti, mezi které patří např. specifikace, zda lze složku

připojit k zápisu nebo pouze ke čtení. Přístup k souborům je ošetřen pouze pomocí přístupových práv závisejících na uživatelském ID.

2.4 Paralelní souborový systém

Síťové souborové systémy jsou tvořeny jedním serverem uchovávajícím metadata (dodatečné informace o souborech jako např. přístupová práva) a vlastní data, se kterými pracují klienti. Při velkém počtu klientů může počet požadavků významně snížit přenosovou rychlost nebo přesáhnout hranici maximálního počtu požadavků, které dokáže server odbavit. Pro účely, které vyžadují vysokou propustnost, byl navržen paralelní souborový systém.

Tento souborový systém uložená data distribuuje na několik datových serverů, čímž umožňuje klientům paralelní přístup a lze tak dosáhnout vyšších přenosových rychlostí, než když by se soubor přenášel pouze z jednoho umístění a lze se vyhnout přetížení serverů rozložením zátěže. Paralelní souborové systémy často využívají řídícího serveru, který uchovává metadata souborů, řídí paralelní přístup a distribuování dat mezi datovými servery.

Hlavní využití nachází tento typ souborového systému při vědeckých výpočtech, které kladou vysoké požadavky na přenosové rychlosti u rozsáhlých datasetů. Zástupci paralelních souborových systému jsou například Lustre [29] a PVFS [48].

2.4.1 Lustre

Lustre je souborový systém navržený pro rozsáhlé distribuované výpočty. Nabídne maximální velikosti svazku v řádech stovek PB, velikosti souboru v jednotkách PB, počet souborů v řádech jednotek až desítek miliard a propustnost systému v řádu jednotek TB/s. Architektura systému má tři hlavní části a to servery s metadaty, servery s daty a klienty.

2.4.2 PVFS

PVFS, aktuálně ve verzi PVFS 2, je ve svém zaměření a architektuře podobný systému *Lustre*. Klientské knihovny podporují rozhraní MPI pro vysokorychlostní přístup k datům. Metadata jsou jako systému *Lustre* oddělena

od dat a to hlavně z výkonnostních důvodů. Metadata mohou být klienty při prvotním požadavku nacachovány a další požadavky už jsou směřovány přímo na datové servery. Pro operace s metadata s metadaty existuje speciální daemon, který má za úkol např. kontrolu práv pro klientské požadavky ale neúčastní se samotného čtení a zápisu dat. PVFS je vyvíjen pro operační systém Linux a pod novým jménem *OrangeFS* je součástí linuxového jádra od verze 4.6.

2.5 Distribuovaný souborový systém

Distribuovaný souborový systém vychází ze síťového souborového systému, tedy jedná se také o souborový systém přístupný pomocí počítačové sítě. Od síťového souborového systému se odlišuje především tím, že data v něm mohou být rozmístěna (replikována) na více míst najednou za účelem zvýšení odolnosti systému.

2.5.1 Požadované vlastnosti

Při návrhu nebo výběru distribuovaného souborového systému se zaměřujeme na následující vlastnosti, které by měl systém splňovat.

Transparentnost

Distribuovaný souborový systém by měl před uživatelem skrývat detaily architektury a komunikace mezi servery, aby se jevil jako jeden centrální systém. Transparentnost se řeší v následujících oblastech:

- **Transparentnost přístupu** Klient by si neměl být vědom, že se data nacházejí na vzdálených serverech. Přístup k datům by měl probíhat jednotným způsobem.
- **Transparentnost umístění** Klient by neměl muset znát na kterých fyzických serverech se data nacházejí.
- **Transparentnost migrací** Soubory mohou být přesouvány bez vědomí klientů (i pokud je klient používá během přenosu).

- **Transparentnost replikací** Klient by si neměl být vědom existence replikací. Systém se zároveň stará o konzistentnost replik.
- **Transparentnost souběžných přístupů** Souběžný přístup a sdílení objektů by nemělo být uživateli patrné.
- **Transparentnost chyb** Klient by měl schopen správně fungovat i po výskytu chyby v systému.

Bezpečnost přenosu dat

Jelikož mohou data být přenášena po nezabezpečené síti, distribuovaný souborový systém musí obsahovat mechanismy pro zabezpečení přenášených informací. Je třeba bránit se proti dvěma základním typům útoků a to:

- **Pasivní útok** Při tomto typu útoku útočník odposlouchává přenášené informace nebo analyzuje kdo posílá komu jaká data.
- **Aktivní útok** Útočník může modifikovat přenášená data, blokovat přenášená data, zpožďovat přenášená data nebo vytvářet falešná spojení.

Pro zajištění bezpečné komunikace se zaměřujeme na šifrování vnitřních přenášených dat (mezi datovými servery) a při komunikaci s vnějšími entitami (komunikace s klientem). Šifrování může probíhat dvěma metodami a to buď symetrickými nebo asymetrickými.

Symetrické metody využívají k zašifrování i dešifrování stejný klíč, tedy sdílené tajemství mezi více entitami. Symetrické šifrování je typicky rychlejší než asymetrické šifrování ovšem vyžaduje, aby si komunikující entity bezpečně vyměnily tajemství. Často používanými algoritmy jsou AES [46], DES [36] a Blowfish [42].

Asymetrické metody, nebo také metody s veřejným klíčem, využívají dvou typů klíčů: veřejného a soukromého. Data jsou zašifrována pomocí veřejného (šifrovacího) klíče a odeslána. Dešifrování lze provést pouze soukromým (dešifrovacím) klíčem. Velkou výhodou je fakt, že veřejný klíč lze přenést (publikovat) i nezabezpečeným kanálem. Příklady asymetrického šifrování jsou RSA [38], DSA [1, Kapitola 4] a Diffie-Hellman [16].

Autentizace uživatelů

O autentizace uživatele se stará autentifikační vrstva systému, která může používat některou z následujících metod pro ověření uživatele:

- **Jednoduché ověřování** probíhá pomocí jména a hesla, které jsou přeneseny v otevřené podobě.
- **Elementární metody ověřování** používají k ověřování symetrické a asymetrické klíče.
- **Ověřovací servery** uchovávají databázi klíčů. Komunikace klientů s ověřovacím serverem probíhá šifrovaně.

Příklady mechanismů pro autentizaci uživatelů jsou Kerberos [47] a LDAP [43].

Oprávnění uživatelů k přístupu k datům

Ochrana uložených dat před neoprávněným přístupem je jedním z důležitých bezpečnostních opatření v distribuovaných souborových systémech. Přístup k datům je typicky řízen pomocí přístupových seznamů nebo kapabilit. Oprávněními jsou typicky čtení, zápis a vykonání.

Přístupové seznamy, anglicky access control list (ACL), specifikují subjekt a jeho povolená oprávnění. Subjektem může být například vlastník, skupina nebo ostatní uživatelé.

Kapability, anglicky capabilities, je odlišný přístup k řízení oprávnění. Zatímco každá položka ACL je spjata s objektem (souborem či složkou), kapability jsou přiřazovány k uživatelům. Každý uživatel má tedy striktně definovány které operace může provádět nad jakými soubory.

Škálovatelnost

Škálovatelností distribuovaného systému myslíme jeho schopnost reagovat na rostoucí zátěž. Existují dva typy škálování a to vertikální a horizontální. Vertikální škálování znamená zvýšení výkonu stávajících prvků systému a to například softwarovými optimalizacemi nebo vylepšením stávajícího hardwaru

(příkladem může být nahrazení rychlejším diskem). Horizontálním škálováním rozumíme přidání nových zdrojů (uzlů).

Distribuovaný souborový systém lze škálovat oběma způsoby, ale ve většině případů je škálován horizontálně. Horizontální škálování mimo zvýšení výkonu zvyšuje i odolnost distribuovaného souborového systému.

Odolnost proti ztrátě dat

Data v distribuovaném souborovém systému mohou být replikována, tedy umístěna na více serverech současně. Zajišťuje se tak redundance dat, díky které se zvyšuje odolnost systému. V případě výpadku jednoho ze serverů s replikou požadovaného souboru budou data zpřístupněna z jiného, dostupného, serveru.

Současně s vyšší spolehlivostí může díky replikaci být zvýšena i propustnost systému. Jelikož jsou identická data na více umístěních, je možné použít paralelního přenosu. Soubor se rozdělí na několik částí a tyto části jsou přenášeny sítí paralelně z dostupných serverů.

Replikace dat zároveň umožní výběr takového zdroje dat, který nabídne nejoptimálnější cestu pro přenos dat vzhledem k přenosové rychlosti a latenci cesty (viz kapitola 2.5.3). Pokud jsou servery systému geograficky rozmístěny, replikace dat zvyšuje šanci, že se budou požadovaná data nacházet blízko klienta.

Replikace nepřináší pouze výhody, ale také některá omezení. Jedním z omezením je dodatečná komplexita řízení replikace a zajištění konzistence dat v systému. Výkonnostní výhody replikace závisí na konkrétním systému, a to především poměru čtení a zápisů. V systému, kde se data často zapisují nebo mění, dochází ke zpomalení z důvodu replikace na další úložiště. Replikace také zvyšují nároky na úložný prostor. Strategie replikace dat a konzistence dat bude dále popsána v kapitole 2.5.2.

2.5.2 Strategie replikace a jejich konzistence

Replikace souborů může probíhat dvěma způsoby a to buď optimisticky nebo pesimisticky. V systému mohou být repliky pouze pro čtení RO a nebo i pro zápis RW.

Pesimistická replikační strategie

Systémy s tradiční pesimistickou replikační strategií se snaží zajistit, aby všechny existující repliky byly identické (konzistentní). Konzistentnosti dosahují pomocí dvou typů replik a to RW (master) a RO. Změny souborů probíhají pomocí RW replik, z kterých jsou následně aktualizovány RO repliky. Tento typ replikační strategie je výhodný svou jednoduchostí, ovšem ne příliš vhodný pro systémy, kde se soubory často mění.

Optimistická replikační strategie

U optimistických strategií jsou všechny repliky RW. Aktualizace replik může probíhat současným zápisem klienta do všech existujících replik nebo zápisem klienta do jedné repliky a poté jsou změny automaticky propagovány uvnitř systému do ostatních replik.

Modely konzistence

Replikace souborů a současný přístup více uživatelů ke sdíleným datům představuje problém při zajištění konzistence výsledných dat. Nejběžnějšími modely konzistence jsou:

- **Striktní** - čtení vždy vrátí aktuální data.
- **Sekvenční** - aktuální data nemusí být ihned dostupná, ovšem musí být zachováno pořadí všech operací.
- **Kauzální** - podobně jako sekvenční, ovšem pořadí operací musí být zachováno pouze u těch, které mohou být kauzálně vázané.
- **FIFO** - pořadí operací jednoho uzlu je vždy zachováno, ovšem absolutní pořadí operací mezi uzly není zajištěno.
- **Slabá** - pořadí je zajištěno pouze u skupin operací (transakcí). Řízení probíhá pomocí synchronizačních proměnných.
- **Uvolňovací** - pro přístup k datům je využíváno kritické sekce. Před výstupem z kritické sekce musí být dokončeny všechny aktualizace dat.

- **Přístupová** - obdobně jako u uvolňovací, ovšem všechny aktualizace dat musí proběhnout před vstupem do kritické sekce.

Zajištění konzistence

K zajištění konzistence replik jsou využívány primárně dva mechanismy a to *uzamykání* a *uspořádání podle časových značek*.

Cílem *uzamykání* je vyloučit souběh požadavků, které by spolu kolidovaly. Požadavky, které by způsobily kolizi při přístupu musí být vyřízeny postupně a může tak docházet k čekání na zámek. Řízení přidělování zámků lze rozdělit na následující metody:

- **Centralizované uzamykání** - Jeden, primární, uzel řídí přidělování zámků k souborům. Všechny požadavky na zámek jsou směřovány na tento uzel a tak může být snížena průchodnost systému a jeho spolehlivost.

- **Uzamykání primární kopie** - V systému může existovat více replik souboru, ale pouze jedna primární. Pokud chce některý z uzlů systému provést změnu nad daným souborem, musí nejdříve od uzlu s primární kopií získat zámek. Před samotným uvolněním zámku jsou změny propagovány na ostatní repliky.

- **Decentralizované uzamykání** - Řízení zámků je rozloženo mezi uzly systému. Přidělení zámku probíhá pomocí distribuovaných algoritmů, např. *Redlock* [4]. Zvyšuje odolnost systému jelikož výpadek jednoho uzlu neovlivní fungování přidělování zámků.

Mechanismus *uspořádání podle časových značek* slouží k zajištění správného pořadí operací prováděných nad soubory, které jsou umístěny na různých uzlech. Každý uzel má své logické hodiny tvořené monotónně rostoucí funkcí. Hodnota hodin se řídí následujícími pravidly:

- Pokud přijme uzel od jiného uzlu zprávu s časovou značkou, nová hodnota hodin uzlu je maximum z těchto dvou hodnot + 1.

- Odesílá-li uzel zprávu, zvýší hodnotu hodin o 1.

Používanými koncepty logických hodin v distribuovaných systémech jsou *Lamportovy hodiny* [27] a *vektorové hodiny* [17].

2.5.3 Výběr cesty

Při přístupu klienta k datům je třeba vybrat cestu, přes kterou budou data přenášena. K dostupným serverům mohou být různé propustnosti linek, různě rychlé disky a různě veliká latence.

V klasické počítačové síti se o plnění routovacích tabulek, podle kterých jsou následně routovány pakety, starají routovací protokoly jako OSPF [32], EIGRP [12] a RIP [22]. Výběr vhodné cesty pro přenos dat ovšem nelze provést pomocí routovacích protokolů na síťové vrstvě.

Překryvná síť

Pro routovací potřeby jsou distribuované souborové systémy stavěny nad překryvnou sítí. Překryvná síť je logická síť fungující nad podkladovou sítí, kterou je typicky IP síť. Překryvné sítě jsou používány pro řešení potřeb konkrétních systémů, které nelze vhodně řešit na úrovni podkladové sítě jako je kvalita spojení (zaměřená na stabilitu, propustnost a odezvu), zajištění kvality služeb či poskytování dat.

Jednou z často používaných překryvných sítí je *Resilient Overlay Network (RON)* [7]. Tato síť si klade za cíl vytvořit robustní síť, která bude minimalizovat následky způsobené náhlými změnami sítě, mezi které se řadí zahlcení nebo úplný výpadek některých spojů.

Pro směrování uvnitř překryvné sítě lze definovat libovolnou metriku, kterou jsou ohodnoceny spojení (hrany) mezi uzly. Nejčastějšími používanými metrikami je propustnost spojení a jeho zpoždění [44]. Vlastnosti spojení jsou periodicky testovány a podle naměřených hodnot je výsledná metrika upravována.

Nejkratší cesta

Existující distribuované systémy jako *OpenAFS*[34], *Coda*[41], *Google File System*[19] nebo *Ceph* [50] před zahájením přenosu vyberou cestu podle aktuálního stavu a nastavení vah jednotlivých spojení. Může se ovšem stát,

že je vybraná linka zatížena několika různými klienty a tak by bylo vhodné dynamicky reagovat na zatížení linky během přenosu dat.

Dynamické routování

Dynamické routování požadavků je jedna z metod výběru cesty, která je unikátní distribuovanému souborovému systému KIVFS. Tato metoda reaguje na zatížení linek a jejich latenci periodickým testováním spojení, podle kterého může výslednou cestu upravovat [31]. Zároveň dovoluje využít sousedící server jako datový proxy server, který od cílového serveru přenese data ke klientovi.

Klient je při požadavku spojen se serverem s linkou s nejvyšší propustností, který ovšem nemusí mít požadovaná data. Požadovaná data jsou následně přenesena z jiného serveru s využitím rychlého vnitřního spojení propojujícího uzly systému. Během přenosu jsou periodicky testovány vlastnosti spojení (propustnost, latence) a existuje-li výhodnější cesta, je přenos přes původní cestu přerušen a pokračuje přes cestu nově vybranou.

2.5.4 Zástupci

Zde je uveden přehled nejpoužívanějších distribuovaných souborových systémů.

AFS

AFS je distribuovaný souborový systém původně navržený pro použití v prostředí univerzity. Klienti pracují lokálně s kopií souboru. Po uložení souboru jsou změněné části odeslány zpět na server. Autentizace probíhá pomocí protokolu Kerberos a přístup k složkám je řízen pomocí ACL.

Coda

Coda je systém vycházející z *AFS*. Cílem toho souborového systému je poskytnout uživatelům nepřetržitou dostupnost dat bez ohledu na výpadky částí systému. Zároveň při částečné nebo úplné nedostupnosti systému dovoluje uživateli pracovat se svojí lokální kopií dat. Změny lokálních dat jsou

propagovány do systému jak jen je to možné.

Google File System

Jedná se o distribuovaný souborový systém založený na hlavním (master) serveru a několika tzv. chunk serverech. Soubory jsou rozděleny na fragmenty o pevné velikosti, které jsou replikovány na chunk servery. Klient při požadavku na data odesílá masteru název souboru a index fragmentu. Master zpátky posílá informace o dostupných replikách daného fragmentu. Klient následně vybírá jednu z replik, ve většině případech tu nejbližší.

Ceph

Ceph je open source platforma zaměřená na uchování dat. Neomezuje se na ukládání pouze souborů, ale umožňuje ukládání objektů, kdy jsou se samotnými daty uložena i metadata, nebo bloků, kdy jsou soubory rozděleny na bloky dat. Cíle Cephu jsou snadná škálovatelnost do řádu petabytů, vysoká propustnost a spolehlivost. Celý systém se skládá ze 4 hlavních komponent a to klientská část, cluster s metadaty, cluster s objekty (daty) a monitorovací cluster.

KIVFS

KIVFS je distribuovaný souborový systém vyvíjený od roku 2009 na Katedře informatiky a výpočetní techniky Západočeské univerzity v Plzni. Hlavním podnětem k vývoji byla snaha eliminovat nedostatky existujících distribuovaných souborových systémů a obsahuje proto několik unikátních funkcionalit, z kterých vyčnívají dynamické routování (viz sekce 2.5.3) a multi master online repliky [30].

2.5.5 Shrnutí

Přestože distribuované souborové systémy přinášejí výhody oproti síťovým souborovým systémům, a to zejména zvýšenou odolnost, větší dostupnost a možnost snadného horizontálního škálování, některé nedostatky zůstávají. Jedním z nedostatků je zpomalení přenosu dat kvůli vyšší režii spojené s

dodatečnou komunikací mezi uzly distribuovaného systému. Další nedostatek se může projevit v případě použití několika různých úložných zařízení napříč systémem. V systému může docházet k přetížení pomalejších disků, přestože existují rychlejší disky, které by zátěž zvládly odbavit.

2.6 Hierarchická správa úložiště

Jedním ze zmíněných nedostatků u distribuovaných souborových systémů s různě rychlými úložišti bylo nevyužití rychlejších úložišť k zajištění rychlejšího vyřízení požadavků. Hierarchická správa úložiště, někdy také vrstvené úložiště, je přístup k uložení dat v případě, kdy máme úložiště s výrazně různými parametry, typicky rychlostí a kapacitou a data mohou být mezi těmito úložišti (vrstvami) přesouvána (migrována).

Hlavním důvodem použití hierarchické správy úložiště je zvýšení propustnosti úložiště a to přesouváním často používaných dat na rychlejší úložiště, princip obdobný z cachovacích pamětí. V ideálním případě by se všechna data nacházela na rychlých úložištích, ale s objemem dat rostou i ekonomické nároky, které jsou v mnoha případech příliš veliké.

Většina dat je uložena na pomalejších discích a data jsou v případě potřeby přesunuta na rychlejší disk. Kdy a která data budou migrována určuje použitá strategie migrace.

2.6.1 Strategie migrace

Existuje několik strategií, které určují za jakých podmínek se spustí migrace a která data budou migrována.

- **Prahová strategie** [28] definuje horní a spodní práh zaplnění rychlejšího disku. Při překročení horního prahu je spuštěn proces migrace, který přesouvá soubory z rychlejšího disku na pomalejší, dokud se nedostaneme pod horní práh zaplnění nebo pokud nedojdou data, která by šla migrovat. Při překročení dolního práhu jsou migrována data z pomalejšího disku na rychlejší, dokud není zaplnění disku větší než dolní práh nebo pokud nedojdou data, která by šla migrovat. Tento

přístup je především zaměřen na zaplnění disku, ale nezohledňuje využití souborů či další parametry jako velikost a typ souboru.

- **Cachovací strategie** je strategie, která k uvolnění místa na rychlejším úložišti využívá algoritmů obdobných z cachovacích pamětí. Tyto algoritmy se snaží o to, aby se z rychlejšího disku (cache paměti) odstranila ta data, která již nebudou v budoucnosti potřeba. K jakým datům se bude přistupovat ovšem většinou není dopředu známo, a tak se snaží o co nejlepší předpověď podle metrik jako je např. LRU, nejmenší počet celkových přístupů, FIFO nebo náhodně.

- **Hashovací strategie** umísťuje soubory náhodně na vrstvy úložiště dle specifikovaného hashovacího algoritmu.

- **F4** je algoritmus založený na hierarchickém úložišti Facebooku [33]. Data jsou přesouvána v ohledu na jejich stáří. Nová data jsou umístěna do rychlejší vrstvy a po uplynutí specifikované doby jsou odmigrovány do nižší vrstvy.

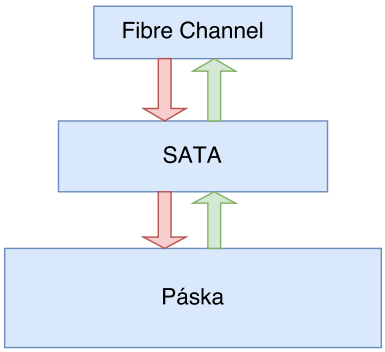
2.6.2 Využití

Hierarchický model uložení dat nachází největší využití v archivačních úložištích. Data se postupně propadají až na nejpomalejší úložiště, kterým je typicky magnetická páska. Další oblastí, ve které jsou hierarchické modely používány mohou být platformy s mediálním obsahem (např. streamování videa). Takovéto platformy typicky uchovávají obrovské množství dat a zároveň mají vysoké nároky na přenosovou rychlost obsahu.

Jedním příkladem použití může být archivační úložiště CESNET HSM Plzeň na ZČU, jehož jednotlivé vrstvy jsou vidět na obrázku 2.1. Úložiště využívá 3 typy úložišť a to fibre channel, SATA a páska [14].

2.6.3 Zástupci

V této podkapitole jsou uvedeni dva zástupci pro hierarchickou správu úložiště.



Obrázek 2.1: Hierarchické archivační úložiště CESNET HSM Plzeň

ZFS

ZFS [35] je jak souborovým systémem tak správcem logických disků[1]. Díky této kombinaci má kompletní přehled nejen o souborovém systému ale i o fyzických zařízeních, kterými je tvořen. Tato znalost umožňuje ZFS migrovat data na rychlejší úložiště podle potřeby. Migrace souborů probíhá automaticky a rozhoduje na základě četnosti přístupů k datům. Další z předností ZFS je možnost vytvářet snapshoty zachycující aktuální verzi souboru, ke kterým se následně můžeme vrátit bez ovlivnění chodu zbytku systému.

HPSS

HPSS [2] je implementací hierarchického úložiště od firmy IBM. HPSS automaticky spravuje životní cyklus souborů a nepoužívaná data automaticky odsouvá na nejnižší vrstvy. Úložiště je navrženo pro clusterové využití, lze ho tedy jednoduše škálovat přidáním dalších uzlů. Pro práci s úložištěm lze využít speciální API nebo VFS.

2.6.4 Shrnutí

Hierarchický model uložení dat může zvýšit propustnost systému využitím rychlejší vrstvy úložiště, na které jsou umístěna často používaná data se zachováním velké kapacity úložiště použitím pomalejších, vysokokapacitních úložišť. Hierarchický model uložení dat ovšem postrádá některé výhody distribuovaného souborového systému, a to především replikaci dat.

[1]Možnost spojit více fyzických úložišť či diskových oddílů do logického celku

2.7 Distribuovaný souborový systém s hierarchickým modelem uložení dat

V sekci 2.5 byly popsány distribuované souborové systémy a jejich výhody a nevýhody. Jednou z uvedených nevýhod je snížená propustnost dat. Snížení propustnosti může být způsobeno vyšší režií distribuovaného souborového systému, ale také současným přístupem několika klientů.

Propustnost a latence souborového systému jsou kritické pro účely jako je streamování videí nebo shromažďování velkého množství naměřených dat, u kterých jsou přenášená obrovská množství dat často s vysokými časovými nároky na přenos. Při použití pomalejších mechanických disků (HDD) v systému s velkým počtem požadavků může docházet ke snížení propustnosti dat a zvýšení latence. Dražší úložiště založená na flash paměti (např. SSD) nebo úložiště na bázi paměti RAM dosahují značně vyšších propustností (přenosových rychlostí) a nižších latencí [25], ovšem za mnohem vyšší cenu GB (viz 2.1). Při velkém množství dat proto ve většině případů není možné využít pouze tyto výkonnější úložiště, zejména kvůli vysokým finančním nárokům.

Jedním z možných způsobů, jak zvýšit propustnost u distribuovaného souborového systému a zároveň zachovat realistické pořizovací náklady na HW, je kombinace spolu s hierarchickým modelem uložení dat. Tento systém by si zachoval výhodné vlastnosti distribuovaných systémů a zároveň by nabídl vysokou propustnost díky rychlým vrstvám úložišť. Cenou za zvýšení propustnosti přenosu dat je vyšší složitost systému a režie spojená s migrací souborů.

Distribuovaný souborový systém s hierarchickým modelem dat je předmětem výzkumu na katedře KIV, vedený p. Ing. Pešičkou. Prováděný výzkum je podnětem pro vznik simulátoru, na kterém bude možné odzkoušet vlastnosti systému a zároveň poskytne referenční model pro měření, který bude porovnán s několika dalšími modely distribuovaného souborového systému.

2.7.1 Princip fungování

Hierarchický model uložení dat může být použit bez zásadního ovlivnění principu fungování distribuovaného souborového systému. Hierarchické úložiště lze použít transparentně nebo jako součást distribuovaného souborového systému.

V případě transparentního použití si hierarchické úložiště interně řeší migrování souborů a distribuovanému souborovému systému se jeví jako klasické úložiště. Tímto řešením bychom mohli dosáhnout zvýšení propustnosti při zachování nezávislosti mezi hierarchickým úložištěm a distribuovaným souborovým systémem.

Při použití hierarchického modelu uložení dat jako součásti distribuovaného souborového systému je výhodou možnost dělat rozhodnutí na základě stavu hierarchického úložiště. Systém bude moct během plánování migrace zohledňovat klientské požadavky a pokusit se podle nich optimalizovat rozmístění souborů na dostupné vrstvy. Jelikož má informace o probíhajících migracích souborů, může se během výběru cesty pro přenos dat pokusit vyhnout úložištím, která jsou aktuálně zpomalena migrací, volbou alternativní cesty. Na základě uvedených přínosů bylo vybráno zapojení hierarchického modelu dat jako součásti distribuovaného souborového systému.

2.7.2 Vrstvy úložišť

Při použití hierarchického modelu uložení dat budou úložiště každého serveru rozdělena podle výkonnostních parametrů (přenosové rychlosti) na několik vrstev, aby bylo možné využít rychlejších úložišť pro data, ke kterým je často přistupováno. K využití výhod hierarchického úložiště musí mít server alespoň 2 vrstvy úložišť, ze kterých budou data přístupná. Pokud má server pouze 1 vrstvu, bude fungovat jako klasický server bez hierarchického úložiště. Jelikož je počet vrstev individuální pro každý server, systém bude možné provozovat i v heterogenním prostředí.

2.7.3 Migrace souborů

Výkonnostní výhody hierarchického modelu uložení dat plynou z možnosti migrovat soubory mezi jednotlivými vrstvami úložišť. Migrace probíhá na základě použité migrační strategie (popsány v kapitole 2.6.1). Jako migrační strategie byla zvolena cachovací strategie, konkrétně poté algoritmus LFU, který nahrazuje data s nejmenším počtem přístupů. Zvolení konkrétní strategie závisí na předpokládaném použití systému a chování uživatelů. Pro naše potřeby předpokládáme, že uživatelé přistupují k rozsáhlému počtu různých dat, ale zároveň existuje množina dat, ke které přistupují častěji než k ostatním (oblíbená data).

Pro potřeby zvolené migrační strategie je nutné u každého souboru zaznamenávat počet přístupů. Při každém přístupu k souboru se zvýší čítač, na základě kterého bude vyhodnoceno, zda má být soubor odmigrován na vyšší vrstvu úložiště. Vyhodnocení, zda provést migraci souboru, může probíhat např. při každém přístupu k souboru nebo periodicky ve zvoleném intervalu. V případě použití úložných zařízení s rozdílnými rychlostmi čtení a zápisu lze navíc rozlišovat přístup typu čtení a přístup typu zápis.

Během migrace je soubor kopírován ze zdrojového úložiště na cílové. Je-li vyžádán soubor ze serveru, na kterém aktuálně probíhá migrace, bude soubor čten ze zdrojového úložiště, dokud se celý neodmigruje na cílové úložiště. Po dokončení migrace je soubor ze zdrojového úložiště odstraněn. Alternativou je ponechat soubor na zdrojovém úložišti a využít tak vyšší vrstvu pouze jako cache paměť. Tento přístup ovšem značně zvyšuje režii systému kvůli nutnosti zajisit konzistenci souboru mezi více vrstvami a snižuje celkové použitelné místo v systému.

Při zaplnění vyšší vrstvy úložiště musí být migrační strategie schopna určit množinu souborů, která bude muset být odmigrována na nižší úložiště, za účelem uvolnění místa. Jiným řešením při nedostatku místa může být migrace souboru na rychlou vrstvu úložiště jiného serveru. Toto řešení ovšem zvyšuje komplexitu systému a přináší dodatečnou režii při přenosu souborů mezi servery.

3 Existující simulátory

Jak je zřejmé z kapitoly 2, existujících způsobů uložení je mnoho. Při návrhu a nasazení nového systému je tak komplikované odhadnout skutečné chování systému při očekávané zátěži. Stejně tak při vývoji nových funkcionalit dokážou simulátory dopředu napovědět, zda se implementace vyplatí či nikoliv. Z těchto důvodu existují simulátory, které se snaží věrohodně simulovat chování systému.

Mezi zmíněnými simulátory jsou i simulátory, které nejsou specializovány na simulaci úložišť, ovšem i přes to souvisí s touto prací.

3.1 DiskSim

DiskSim [10] je efektivní, přesný a vysoce konfigurovatelný simulátor úložných zařízení. Umožňuje simulaci disku, řadičů, sběrnic, ovladačů, plánovačů a dalších. Tento simulátor se specializuje na přesné simulování disků a jejich spolupráci s operačním systémem. Umožňuje konfiguraci plánování požadavků, cachování, organizaci dat na několika úrovních, jakými jsou například operační systém, řadič a samotný disk.

Pro běh simulace lze použít existující logy I/O požadavků nebo simulátorem vygenerované syntetické požadavky. DiskSim je vhodné použít pro ověření výkonnosti úložného systému, vlivu výkonnosti úložiště na celkový výkon systému a simulaci nových architektur úložného subsystému.

Simulátor je zaměřen především na hardwarovou část úložiště. Simulátor ovšem nenabídne možnost simulace komplexnějších systémů pro uložení dat jako jsou například hierarchické či distribuované souborové systémy.

3.2 OGSSim

OGSSim [20] je simulátor inspirovaný simulátorem *DiskSim* a snaží se řešit některé nedostatky tohoto simulátoru. Hlavním rozdílem je možnost simulovat komplexnější systémy s větším počtem úložišť a možnost kombinovat

více technologií úložišť (např. HDD a SSD).

Konfigurace disků je možná v následujících formátech: JBOD, RAID1, RAID-01 a RAID (n+p), kde n představuje počet disků s daty a p počet disků s paritou. Dále umožní výběr strategie pro umísťování parity na disky v raidu. Jednotlivé disky nebo vytvořené svazky lze přiřadit do vrstev a tak simulovat hierarchický souborový systém.

Simulátor se skládá ze 7 hlavních modulů:

- **Workload** Zpracovává požadavky ze vstupního souboru.
- **Hardware Configuration** Načítá konfiguraci simulovaného systému z XML souboru.
- **Simulation Configuration** Načítá konfiguraci simulátoru z XML souboru.
- **Pre-processing** Předzpracování požadavků a spuštění simulace.
- **Volume Driver** Modul řídící úložiště a jejich konfigurace.
- **Device Driver** Simulace ovladače zařízení, implementuje hardwarově specifické vlastnosti.
- **Execution** Vykonává požadavky směřované na jednotlivá úložiště.

Pro spuštění simulace jsou potřeba minimálně 3 vstupní soubory. Tento přístup umožní uživateli detailní popis simulovaného systému ovšem vytváření a orientace v konfiguračních souborech může být poměrně náročná. Hlavními výsledky jsou výkonnostní ukazatele, ale nabídne například i ukazatele spolehlivosti systému.

Simulátor je určený na simulaci rozsáhlých úložišť s detailní konfigurací hardwaru a použitých technologií spolu se simulací hierarchického úložiště. Součástí simulátoru není grafické prostředí a tak vyžaduje konfiguraci pomocí vstupních souborů. Stejně tak výsledky jsou prezentovány v textovém formátu. Simulátor ovšem neumožňuje simulaci distribuovaných souborových systémů.

3.3 StorageSim

StorageSim [39] je simulátor zaměřený na hierarchická úložiště. Umožňuje měření propustnosti hierarchického modelu úložiště vs. klasické úložiště a ideální poměr velikostí rychlejší a pomalejší vrstvy pro výkon úložiště.

Simulátor má ve výchozím stavu 3 strategie migrace a to hashovací, cachovací a f4. Další strategie je možné doplnit díky modulárnímu návrhu. Simulace tedy umožňuje otestování navrhované strategie migrace před nasazením do reálného systému.

Simulované požadavky jsou čteny ze vstupního souboru ve formátu dostupného od asociace *Storage Networking Industry Association (SNIA)*. Simulátor byl autory testován při simulaci reálného provozu vyhledávače Yahoo, který představoval téměř 55 miliónů diskových operací během 1 hodiny. Při testování byl StorageSim schopný zpracovat zhruba 8000 operací za sekundu.

Simulátor je více abstraktní než např. *DiskSim* nebo *OGSSim* za účelem zvýšení rychlosti simulace. Při simulaci jsou zanedbány věci jako jsou např. režie spojená s vytvářením či mazáním souborů, cachovací paměť a plánovač požadavků. Simulaci distribuovaných souborových systémů ovšem simulátor neumožňuje.

3.4 Google File System Simulator

Google File System Simulator [26] je simulátor zaměřen na distribuovaný souborový systém Google File System (GFS). Primárním cílem simulátoru je ověřit chování a výkon GFS při různorodých zátěžích a podmínkách. Součástí simulace je simulace replikace souborů, vyvažování zátěže a simulace výpadku disku.

Simulace je založena na diskrétních událostech, kde každý požadavek je reprezentován událostí. Události jsou vybírány z fronty a postupně zpracovávány.

Simulátor umožňuje volbu propustnosti spojení klienta se systémem od 0 do 1000 Mbps. Spojení mezi servery má neomezenou propustnost, což může být limitující faktor při modelování reálného systému. Simulátor je ome-

zen pouze na jeden distribuovaný souborový systém bez možnosti simulace hierarchického modelu uložení dat.

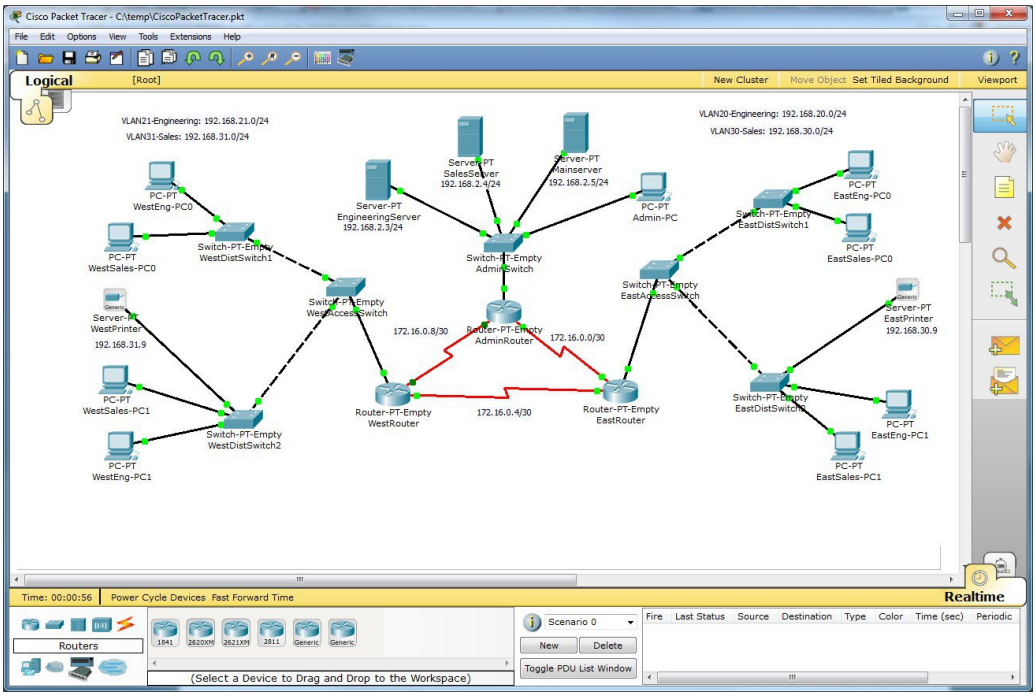
3.5 GridSim

GridSim [11] je simulátor zaměřený na paralelní a distribuované výpočty. Umožňuje modelovat síťové spojení mezi uzly, různé typy zdrojů, uživatele, aplikace a plánovače.

Dále slouží k porovnávání plánovacích algoritmy při simulované zátěži nebo k analýze zatížení různých zdrojů (procesor, disk). Parametry zdrojů mohou být individuálně nastaveny pro každý zdroj, čímž lze simulovat i heterogenní systémy. Simulátor je využíván jako podklad mnoha akademických výzkumů, také díky jeho modulární architektuře, která umožňuje uživatelům rozšíření o dodatečnou funkcionalitu. Výstupní formát výsledků simulace je řízen programátorem pomocí dat, která jsou mu zpřístupněna veřejným rozhraním simulovaných objektů. Simulátor neumožňuje simulovat distribuované souborové systémy s hierarchickým modelem uložení dat.

3.6 Packet Tracer

Packet Tracer [37] je nástroj pro simulaci počítačové sítě určený pro výuku. Umožňuje ruční modelování sítí, detailní analýzu datového toku, vizualizaci toku a volbu protokolů na jednotlivých vrstvách TCP/IP modelu. Simulace může probíhat v reálném čase nebo zrychleně.



Obrázek 3.1: Ukázka modelování v Packet Tracer

Modelování simulovaného prostředí probíhá tažením myši prvků z knihovny (např. routery, switche) a následným propojením, viz obr. 3.1. Tento simulátor se nezaměřuje na simulaci úložiště a byl zkoumán a zmíněn hlavně kvůli inspiraci při návrhu vlastního simulátoru.

3.7 Existujících simulátory a vlastní řešení

Byly otestovány některé existující simulátory lokálních úložišť, hierarchických úložišť a simulátor konkrétního distribuovaného souborového systému. Simulace lokálních úložišť umožnila detailní hardwarové modelování a spolupráci hardwaru s operačním systémem. S rostoucí složitostí simulovaného systému se zvyšovala i abstrakce simulace hlavně z důvodů komplexnosti a rychlosti simulátoru. Pro distribuovaný souborový systém byl otestován simulátor simulující *Google File System*, který je zaměřen pouze na tento souborový systém a tak jsou možnosti konfigurace poměrně omezené.

3.7.1 Vlastní řešení

Z otestovaných simulátorů se nejeví ani jeden jako vhodný kandidát na simulování obecných distribuovaných souborových systémů, proto bude v praktické části implementováno vlastní řešení. Simulátor by měl umožnit modelovat spojení mezi klientem a vnitřní propojení mezi servery, aby bylo možné simulovat reálný systém a jeho provozní vlastnosti.

Vlastní řešení by mělo být schopné modelovat libovolnou strukturu distribuovaného souborového systému spolu s propojením serverů mezi sebou a mezi klientem a servery. Pro jednoduchost návrhu by mělo být součástí grafické rozhraní, které většině ze zmíněných simulátoru chybělo, pomocí něhož si bude moct uživatel systém nakonfigurovat a následně otestovat.

Při návrhu vlastního řešení se můžeme inspirovat vhodnými vlastnostmi otestovaných simulátorů a to například detailními možnostmi konfigurace systému, možným porovnáním výsledků simulace s různými parametry simulovaného systému (např. strategie volby cesty, strategie migrace souborů), modulárním návrhem, který umožní jednoduché rozšíření simulátoru a grafickým rozhraním, které zajistí uživatelsky přívětivé modelování systému.

Shrnutí požadavků

Vlastní simulátor by měl splňovat následující požadavky:

- umožnit modelovat spojení mezi servery a mezi servery a klientem,
- umožnit modelovat libovolnou strukturu systému,
- umožnit modelovat systém pomocí grafického rozhraní,
- umožnit konfiguraci prvků systému a to serveru, klienta a spojení,
- umožnit replikaci souborů mezi servery,
- umožnit zobrazení výsledků simulace (průměrná rychlost, počet přenesených dat, graf průběhu rychlosti v závislosti na čase) pro jednotlivé testované metody,
- umožnit simulovat hierarchický model uložení dat,
- umožnit snadné rozšíření simulátoru modulárním návrhem.

4 Návrh simulátoru

Cílem simulace bude porovnat různé modely zapojení distribuovaných souborových systémů. Navržený simulátor se bude řídit požadavky uvedenými v kapitole 3.7.1.

4.1 Struktura

Simulátor se bude skládat z aplikační části a grafické části. Aplikační část bude zajišťovat běh simulace, sbírání výsledků a měření požadovaných veličin. Aplikační část by neměla být nijak závislá na grafické části. Díky oddělení těchto dvou částí bude v budoucnosti možné nahrazení grafického rozhraní bez nutnosti provádět změny v existujícím kódu simulátoru.

Grafická část umožní uživateli modelování distribuovaného systému z grafického rozhraní, vytváření plánu simulace, spuštění simulace a detailní zobrazení průběhu simulace a naměřených výsledků. Dále bude umožňovat exportování výsledků do formátu, aby s nimi bylo možné dále pracovat.

4.2 Technologie

Na základě požadavků na simulátor byly vybrány technologie pro implementaci. Aby se mohl simulátor dostat k většímu počtu uživatelů, neměl by být svázán s konkrétním operačním systémem. Cílem je tedy podpora alespoň všech hlavních operačních systémech a to *MS Windows*[1], *Linux*[2] a *Mac OS*[3].

Stav simulátoru a modelovaný distribuovaný souborový systém by mělo být možné exportovat, aby se měření mohla opakovat. Modelování rozsáhlého systému může být časově velice náročné a tak by měl být formát uloženého stavu simulátoru strojově zpracovatelný, díky čemuž bude možné strojově generovat modely.

[1] `https://www.microsoft.com/en-us/windows/`

[2] `https://www.kernel.org/`

[3] `https://www.apple.com/cz/macos/`

Jako programovací jazyk jsem zvolil jazyk *Java*, konkrétně *Java 8*[4] spolu s platformou *JavaFX* [15] pro vývoj grafického rozhraní. Java umožní běh na požadovaných operačních systémech a platforma *JavaFX* zajistí jednotný vzhled napříč těmito systémy. Základní funkčnost simulátoru bude otestována pomocí Unit testů frameworkem *JUnit* [21].

Popis stavu simulátoru a modelovaného distribuovaného souborového systému bude řešen pomocí jazyka značkovacího *XML*[5]. Jazyk *XML* je jednoduše vytvářen a zpracován standardními knihovnami Javy.

4.3 Modelování systému

Model systému bude reprezentován grafem. Uzly grafu budou dvojího typu a to buď klient nebo server. Hrany mezi uzly představují propojení pomocí počítačové sítě. Pro modelování distribuovaného souborového systému budou použity následující komponenty:

- **Klient** Představuje klienta, který komunikuje se servery distribuovaného souborového systému.
- **Server** Představující jeden server distribuovaného souborového systému.
- **Úložiště** Každý server může mít libovolný počet úložišť. Úložiště bude mít parametry rychlost (čtení i zápis) a velikost.
- **Složka** Složka bude sloužit pro organizaci struktury dat v systému. Složku bude mít jméno a bude jí možné připojit na libovolné existující úložiště serveru.
- **Soubor** Soubor má jméno a velikost. Musí mít rodiče typu složka. Jeden soubor může být umístěn (replikován) na více serverů.
- **Linka** Linka představuje spojení mezi serverem a klientem, její parametry jsou rychlost a odezva.

[4] `http://www.oracle.com/technetwork/java/javase/overview/java8-2100321.html`

[5] `https://www.w3.org/XML/`

- **Charakteristika linky** Slouží k simulaci zatížení linky v čase. Bude určovat skutečnou propustnost.

4.4 Modularita

Simulátor by měl mít modulární strukturu, aby byla možná jednoduchá implementace dalších metod výběru cesty, strategií migrace souborů v hierarchickém modelu uložení dat a grafického klienta (např. specializovaný klient pro rozsáhlé systémy).

4.5 Formát výstupu

Výstup simulace by měl být ve strojově zpracovatelném formátu ale zároveň by mělo být uživateli umožněno přímo v prostředí simulátoru zobrazit výsledky pomocí grafu.

4.6 Rozsah simulovaného systému

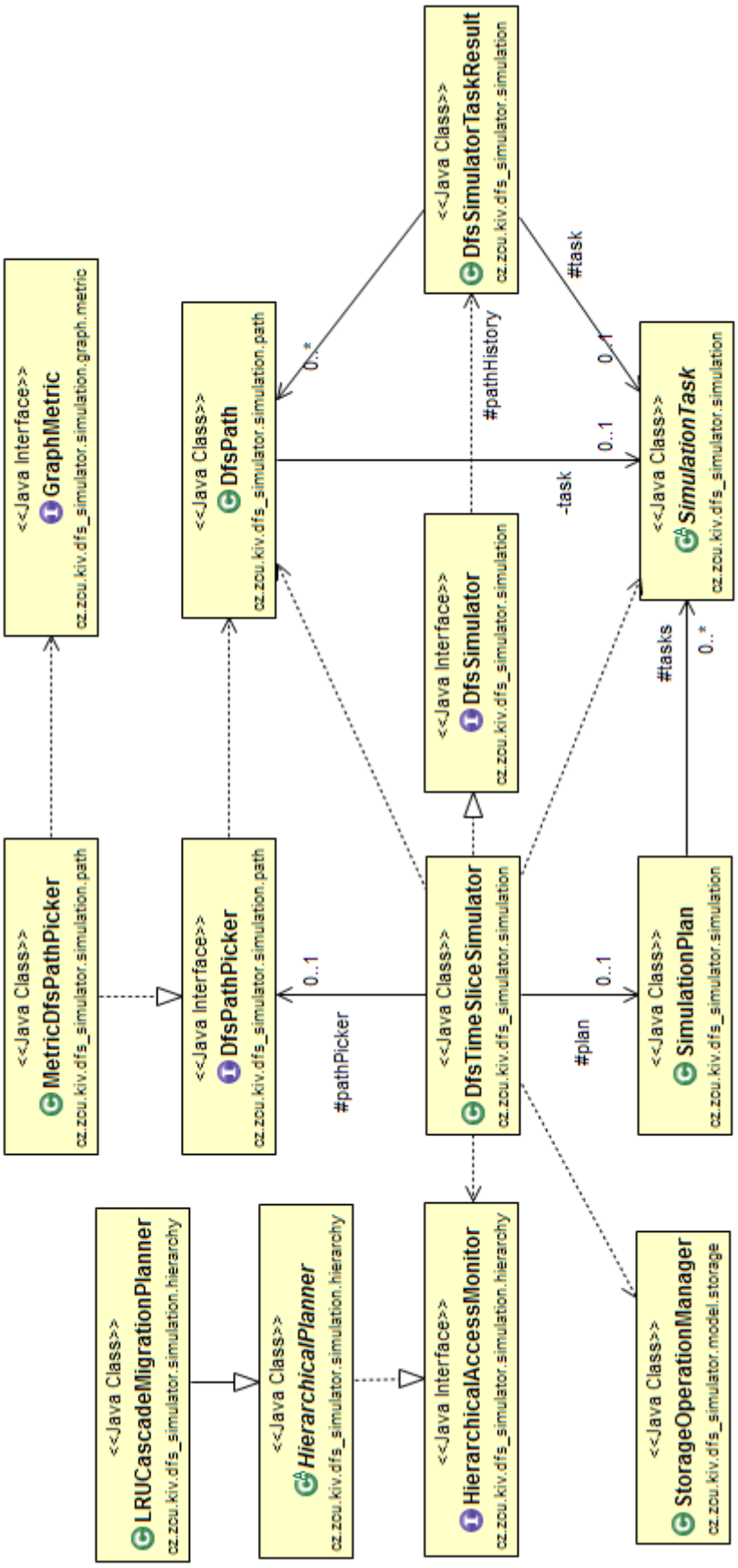
Simulátor by měl být schopný simulovat jak miniaturní systémy tak systémy o velkém počtu serverů a velkém počtu požadavků na přenos dat. Během simulace by měl být uživatel informován o průběhu v grafickém prostředí a po dokončení o výsledcích. Jelikož bude simulátor dimenzován na velký počet serverů a požadavků, měla by vizualizace průběhu doběhnout v rozumném čase nebo uživateli nabídnout možnost ji přeskočit.

5 Implementace aplikační části

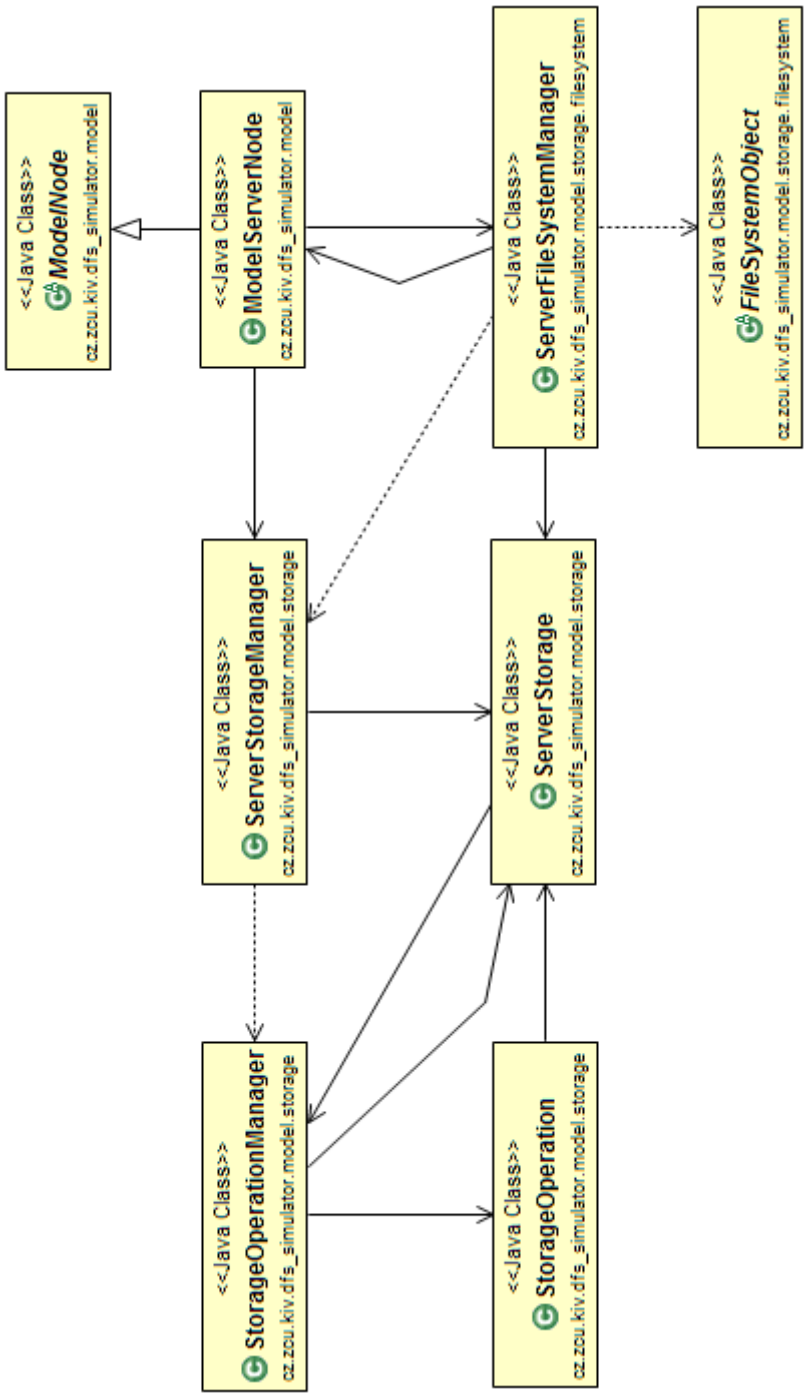
V následující části je popsána implementace a detaily aplikační části, tedy části programu, která řídí modelování systémů a běh simulace. Všechny cesty k balíkům, které jsou v této části odkazovány, jsou pouze relativní cesty k rodičovskému balíku *cz.zcu.kiv.dfs_simulator*.

5.1 UML diagram

Na následujících obrázcích 5.1 a 5.2 jsou znázorněny pomocí UML diagramu nejdůležitější části aplikační části simulátoru, samotný simulátor respektive model serveru a úložiště.



Obrázek 5.1: UML diagram simulační části



Obrázek 5.2: UML diagram serveru a úložiště

5.2 Model

Modelovací část poskytuje prvky k modelování distribuovaného souborového systému. Modelovaný systém je interně reprezentován jako orientovaný graf (důvod popsán v 5.2.3). Hrany grafu mají ohodnocení kombinované z propustnosti a latence spojení. Zdrojové soubory modelu jsou obsaženy v balíku *model*.

5.2.1 Jednotky rychlosti a velikosti

Přenosová rychlost je reprezentována objekty třídy *ByteSpeed*. Výchozí jednotkou je 1 byte/s, ale součástí třídy jsou metody pro získání hodnoty v násobcích kilobyte/s, megabyte/s a gigabyte/s. Velikost je obdobně reprezentována třídou *ByteSize*. Výchozí jednotkou velikosti je 1 byte a součástí třídy jsou opět metody pro získání násobků kilobyte, megabyte a gigabyte. Násobky předpon jsou uvedeny v tabulce 5.1.

Tabulka 5.1: Předpony jednotek

Předpona	**Násobek**	10^n
B	1	10^0
kB	1000	10^3
mB	1 000 000	10^6
gB	1 000 000 000	10^9

5.2.2 Uzly

Uzly grafu dědí od rodičovské třídy *ModelNode* a mohou být dvou typů a to klient nebo server. Uzly grafu, které mezi sebou mají hranu, představují sousedy a mohou spolu komunikovat. Uzly jsou registrovány do globálního registru *ModelNodeRegistry*, aby byly viditelné ostatním částem simulátoru. Registrace uzlu probíhá pomocí unikátního identifikátoru typu *StringProperty*.

Server

Server je reprezentován třídou *ModelServerNode* a v distribuovaném souborovém systému zastupuje datový server, který může mít neomezené množství různých úložišť a na nich uložena data. Každý server má vlastní kořenovou složku, která je kořenem adresářové struktury serveru, správce úložišť a správce adresářové struktury.

Klient

Klient je uživatel, který může vytvářet požadavky na čtení nebo zápis souborů z některého ze serverů. Seznam požadavků klienta tvoří simulační plán a je uložen v objektu třídy *SimulationPlan*.

5.2.3 Propojení uzlů

Logika pro propojení uzlů je v samostatném balíku *connection*. Jednotlivé uzly mohou být propojeny instancemi třídy *ModelNodeConnection*. Propoj je orientován z výchozího do cílového uzlu, díky čemuž lze vytvářet mezi uzly nesymetrické propoje. Nesymetrické propoje mohou představovat například připojení typu ADSL, které mají často několikrát větší rychlost stahování než nahrávání. Dalšími vlastnostmi propoje je maximální propustnost dána instancí třídy *ByteSpeed*, latence v milisekundách a zátěžová charakteristika (viz 5.2.4). Latence propoje udává za jak dlouhou dobu obdržíme odpověď od protějšku po vyslání požadavku (např. ICMP ping).

5.2.4 Zátěžová charakteristika propojení uzlů

Zátěžová charakteristika propoje slouží k simulaci zátěže v závislosti na běžícím čase simulace. Implementace charakteristiky musí implementovat rozhraní *ConnectionCharacteristic* a jeho metody. V závislosti na konkrétní implementaci může být charakteristika jednorázová nebo periodická. Výchozí implementací je periodická lineární charakteristika definována pomocí několika diskrétních bodů.

Rozhraní ConnectionCharacteristic

Rozhraní *ConnectionCharacteristic* má následující metody:

```
double getAverageBandwidthModifier(long sTime,
      long intervalLength);

void setPeriodInterval(long time);
LongProperty periodIntervalProperty();

double getYLowerBound();
double getYUpperBound();

double getXLowerBound();
double getXUpperBound();
```

Metoda *getAverageBandwidthModifier* vrací průměrný modifikátor propustnosti v časovém intervalu $sTime, sTime+intervalLength$, *setPeriodInterval* nastavuje délku časového intervalu charakteristiky (u periodických nastavuje délku periody). Metody *getYLowerBound*, *getYUpperBound* a *getXLowerBound*, *getXUpperBound* vrací rozmezí osy X (čas), resp. Y (modifikátor).

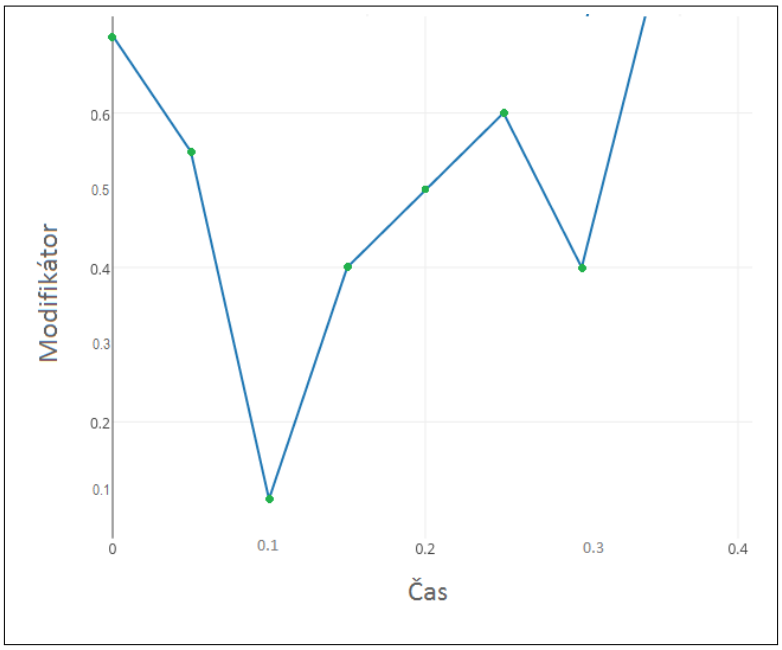
LineConnectionCharacteristic

LineConnectionCharacteristic je implementace zátěžové charakteristiky, konkrétně lineární periodická charakteristika, využívající několik diskrétních bodů propojených úsečkami (po částech lineární funkce). Na ose x je průběh času a na ose y je modifikátor propustnosti v intervalu $< 0.01, 1 >$.

Na obrázku 5.3 je zobrazena část charakteristiky. Pokud bychom tedy měli linku s maximální propustností 10 mB/s a zajímala by nás skutečná propustnost v čase 0.2, vynásobíme maximální propustnost hodnotou 0.5 a získáme 5 mB/s.

Pokud bychom chtěli zjistit propustnost v čase, ve kterém přímo neleží některý z definovaných bodů, je využito následujících rovnic:

$$mod = A_y(1 - t) + B_y t \tag{5.1}$$



Obrázek 5.3: Lineární zátěžová charakteristika

$$t = (B_x - A_x)/(d_x - A_x) \tag{5.2}$$

kde d_x je hledaný čas, A a B jsou body ležící na časovém intervalu, do kterého spadá námi hledaný čas a t je normalizovaná vzdálenost hledaného času od výchozího bodu A.

Pokud bychom hledali modifikátor v čase $d_x = 0.225$ (obr. 5.3) musíme nalézt mezilehlé body, které jsou v našem případě $A = (0.2, 0.5)$ a $B = (0.25, 0.6)$.
Výpočet vzdálenosti t:

$$t = (0.25 - 0.2)/0.25 = 0.5 \tag{5.3}$$

Výpočet modifikátoru v čase d_x:

$$mod = 0.5(1 - 0.5) + 0.6 * 0.5 = 0.55 \tag{5.4}$$

5.2.5 Adresářová struktura

Pomocí adresářové struktury jsou na serverech uspořádána data. Adresářová struktura je stromem, jehož kořenem je kořenová složka / příslušného serveru, tedy obdobně jako na UNIXových systémech. Implementace adresářové struktury a věcí s ní společných je obsažena v balíku *storage.filesystem*.

5.2.6 Správce adresářové struktury

Adresářová složka je spravována objektem třídy *ServerFileSystemManager*, který spravuje všechny datové objekty (soubory či složky) uložené na serveru. Interně je mapování datových objektů na úložiště realizováno pomocí mapy *HashMap*, jejímž klíčem je instance datového objektu a hodnotou úložiště. Součástí jsou zároveň pomocné metody na přidávání nových potomků do adresářové struktury serveru, které hlídají dostupné místo na patřičném úložišti a v případě nedostatku je vyvolána výjimka typu *NotEnoughSpaceLeftException*.

Cesta

Cesta souboru nebo složky je tvořena rekurzivně přes rodiče až ke kořeni stromu a jednotlivé vrcholy jsou odděleny pomocí oddělovače /. Pro soubor $f1$ ve složce $d1$ by výsledná cesta byla $/d1/f1$ a pro složku $s1$ ve složce $d1$ poté $/d1/s1/$. Jak je vidět, tak cesty složek končí oddělovačem /.

Implementace datových objektů

Datové objekty jsou reprezentovány abstraktní třídou *FileSystemObject*, konkrétně potom *FsFile* a *FsDirectory* pro soubor respektive složku. Každý objekt má jméno, které je unikátní na dané úrovni ve stromové struktuře a rodiče typu složka. Soubory mají navíc ještě velikost a čítač přístupů. Jediný objekt, který nemá rodiče, je kořenová složka.

Vyhledávání

Vyhledávání souborů v adresářové struktuře serveru lze provést metodou objektu třídy FsDirectory

`FileSystemObject getChildObject(String path)`

která prohledává potomky dané složky. Hledáme-li od kořenové složky, cesta musí začínat /. Na nižších úrovních předáváme pouze relativní cestu bez počátečního /, tedy například pro soubor s absolutní cestou $/d1/s1/f1$ bychom při hledání od kořene použili celou absolutní cestu a při hledání ze složky $d1$ relativní cestu $s1/f1$.

Registr datových objektů

Pro jednoduchost jsou všechny datové objekty registrovány do globálního registru *FsGlobalObjectRegistry*, který mapuje cesty datového objektu na objekty instance *ObjectRegistryEntry*. V objektu třídy *ObjectRegistryEntry* jsou seznamy serverů a instancí třídy *FileSystemObject* pro každou cestu.

5.2.7 Replikace

Replikace souborů je implementována v balíku *storage.replication*. Pokud sdílí více souborů absolutní cestu na několika serverech, jsou jejich instance považovány za repliky jednoho souboru. Správu replik zajišťuje třída *FsGlobalReplicationManager*, která poskytuje metody pro získání všech instancí replik jednoho souboru, přejmenování replik, mazání a měnění velikosti replik.

Repliky se řídí optimální strategií - všechny repliky jsou RW, lze tedy zapisovat do libovolné instance. Po přepsání vybrané instance repliky dochází k propagaci změn ze serveru s vybranou replikou na zbylé instance (synchronizace). Při synchronizaci replik jsou vytvořeny diskové operace na čtení (zdrojový server) a zápis (cílové servery).

5.2.8 Úložiště

Úložiště a s ním spojené operace jsou implementovány v balíku *storage*. Samotné úložiště je implementováno třídou *ServerStorage*, která má danou velikost, maximální přenosovou rychlost sdílenou pro čtení a zápis a vlastní instanci správce I/O operací (viz 5.2.9).

Úložiště jako takové neví nic o datech, která jsou na něm uložena. Informace o datech jsou součástí adresářové struktury a ta je spravována pro každý server individuálně pomocí objektu *ServerFileSystemManager*.

Správce úložišť

Správce úložišť, implementován třídou *ServerStorageManager*, uchovává seznam všech úložišť daného serveru, ke kterému správce patří. Pomocí správce lze provádět hromadné operace nad úložišti jako jsou např. metody pro komunikaci se správcem I/O operací jednotlivých disků.

5.2.9 I/O operace

I/O operace *StorageOperation* jsou operace čtení a zápisu na konkrétním úložišti. Operace jsou interně rozlišeny pomocí výčtového typu *StorageOperationType* a aktuálně mohou být 4 druhy operací: READ, WRITE, CLIENT_READ a CLIENT_WRITE. Operace s předponou *CLIENT* jsou operace, které nejsou automaticky spravovány (jak bude popsáno dále v „Správce operací“). Ke každé operaci je přiřazen seznam přenášených souborů, maximální rychlost operace, přidružená operace na jiném disku, seznam navazujících operací a callback.

Správce operací

I/O operace jsou spravovány třídou *StorageOperationManager*. Hlavním účelem správce operací je výpočet dostupné rychlosti a počtu přenesených dat u každé operace. Správce je unikátní pro každé úložiště a spravuje dva seznamy operací - běžící operace a čekající operace. Čekací operace jsou operace, které čekají na dokončení některé rodičovské operace, která může aktuálně čekat či běžet.

Řízení operací je rozlišeno spravovanými a nespravovanými operacemi. Oba typy operací jsou v příslušných frontách, ovšem u nespravovaných operací není správcem řízen jejich průběh (přenesená data). Nespravované operace jsou využívány v situacích, kdy se operace nemusí dokončit celá, tedy např. při přenosu ke klientovi a zapnutém dynamickém routováním.

Výpočet rychlosti operací

Výpočet rychlosti operací je explicitně spouštěn uvnitř třídy *StorageOperationManager* pro aktuálně běžící operace (i nespravovaného typu), konkrétně metodou

`void updateAvailableThroughput(long sTime).`

Rychlost je operacím přidělována spravedlivě - maximální rychlost disku je dělena počtem všech operací. V případě, že má operace nastavenou omezující maximální rychlost, je nevyužitý díl rychlosti rozdělen mezi zbylé operace.

Výpočet přenesených dat

Výpočet přenesených dat probíhá explicitně uvnitř třídy *StorageOperationManager* pro aktuálně běžící spravované operace, konkrétně metodou

`void updateTransferedSize(long timeInterval, long sTime),`

která aktualizuje pro každou běžící operaci počet přenesených dat v intervalu daném proměnnou *timeInterval*. Jako přenosová rychlost je použita vypočtená rychlost v metodě *updateAvailableThroughput*. Po dokončení je operace odebrána ze seznamu běžících operací a je vyvolán její callback, pokud má nějaký nastaven.

Přidružené operace

Přidružené operace slouží k propojení dvou diskových operací a jsou využívány v případě, kdy se data přesouvají mezi úložišti, tedy např. při hierarchické migraci dat. Jelikož se může stát, že jedna z operací má větší dostupnou rychlost než druhá, musí být operace propojeny a jejich společná maximální rychlost je minimem z maximálních rychlostí samotných operací.

Diskové operace hierarchické migrace lze vytvořit metodou *addMigrationOperation* z instance správce operací, která zajistí vytvoření příslušných operací a jejich přidružení.

Navazující operace

Pomocí navazujících operací lze vytvářet řetězec operací, které musí být vykonány popořadě. Navazující operace jsou využívány např. u hierarchické migrace dat v případě, kdy je nejdříve třeba z vyšší vrstvy data odmigrovat na vrstvu nižší, aby se uvolnil dostatek místa.

Každá disková operace může mít neomezený počet navazujících operací. Navazující operaci lze přidat k instanci *StorageOperation* metodou

```
void addFollowingOperation(StorageOperation followingOperation).
```

Po dokončení rodičovské operace jsou navazující operace automaticky správcem operací zařazeny do běžících operací.

Callback

Callback je funkce, která je předána do instance diskové operace a očekává se její zavolání při určité události. Ve skutečnosti je implementována přes abstraktní třídu *StorageOperationCallback*, která má dvě metody

```
abstract void onOperationStarted(long sTime)
abstract void onOperationFinished(long sTime).
```

Metoda *onOperationStarted* je vyvolána správcem operací před převedením diskové operace do běžícího stavu a metoda *onOperationFinished* je vyvolána po dokončení běhu operace. Obě metody mají jako parametr aktuální čas simulátoru.

Callback funkce umožňují automatické provedení kódu, který si přejeme vykonat až při specifikované události. Při vytváření operace nejsme schopni předem určit v který čas operace začne (navazující operace) a kdy doběhne (závisí na dostupné rychlosti). Callback funkce jsou využívány např. u hierarchické migrace, kde je vytvořeno několik navazujících operací, které vyžadují po dokončení provést aktualizaci mapování souboru na úložiště.

5.3 Simulace

Logika řídící simulaci je obsažena v balíku *simulation*. Hlavní součástí simulace je simulátor implementující rozhraní *DfsSimulator*. Spolu se simulací

probíhá logování všech důležitých akcí, díky kterému lze detailněji prozkoumat výsledek simulace a vzorkování rychlosti, které umožní náhled na průběh přenosové rychlosti v čase.

5.3.1 Diskrétní událostní simulátor

Implementovaným diskrétním událostním simulátorem je *DfsTimeSliceSimulator*. Simulace probíhá jako sled událostí, kde události představují požadavky klienta na čtení nebo zápis souboru. Parametry simulátoru jsou simulační plán *SimulationPlan*, instance *DfsPathBuilder* zprostředkovávající výběr cesty od klienta k serveru a výčtový typ *SimulationType* detailněji specifikující typ simulace (5.3.4).

Vstupní simulační plán obsahuje klientské požadavky, které při simulaci představují události a jsou vykonávány postupně jedna po druhé. Zpracovávané požadavky jsou simulátorem krokovány, jelikož nelze dopředu jednoduše určit, jaký bude mít požadavek průběh kvůli proměnlivé propustnosti spojení a měnící se přenosové rychlosti disků v závislosti na existujících diskových operacích.

Krok simulátoru je určen konstantou *TIME_RESOLUTION_MS*, která má ve výchozím nastavení hodnotu 500 ms. Výběr velikosti je kompromisem mezi rychlostí běhu simulace a přesností simulace. Kratší simulační krok způsobí větší počet výpočtů pro každou událost a větší paměťové nároky pro uchování všech vzorků rychlosti. Delší simulační krok může mít negativní vliv na přesnost simulátoru, a to na navzorkované hodnoty rychlosti a na přesnost průběhu diskových operací, které jsou aktualizovány při každém kroku.

5.3.2 Stahování a nahrávání souborů

Klientské požadavky jsou reprezentovány abstraktní třídou *SimulationTask*, konkrétně *GetSimulationTask* a *PutSimulationTask* pro stahování resp. nahrávání. Parametrem každého požadavku je objekt *FsFile*, který se stahuje nebo nahrává.

V případě stahování se cílové úložiště určuje podle toho, na kterém úložišti je soubor uložen. Při nahrávání souboru je cílové úložiště zděděné od

rodičovské složky, do které je soubor nahráván.

5.3.3 Výsledky simulovaných požadavků

Výsledkem běhu simulátoru je seznam *DfsSimulatorTaskResult*, kde každá položka seznamu je výsledek jednoho simulovaného požadavku. Výsledek obsahuje referenci na požadavek *SimulationTask*, výčtový typ *DfsSimulatorTaskResultState* určující stav výsledku, celkový čas pro vykonání požadavku, historii cest, přes které byl požadavek (soubor) přenášen a vzorky rychlosti. Simulace požadavku může skončit některým z následujících stavů:

- **SUCCESS** úspěšné dokončení požadavku
- **NO_NEIGHBOURS_AVAILABLE** klient nemá žádné spojení se servery
- **NO_PATH_AVAILABLE** k serveru nebo serverům, na kterých se nachází požadovaný soubor, neexistuje cesta
- **NOT_ENOUGH_SPACE_ON_DEVICE** na nahrání souboru není na cílovém úložišti dostatek volného místa
- **OBJECT_NOT_FOUND** pokud v cílové cestě nebylo nic nalezeno.

5.3.4 Metoda simulace

Metoda simulace určuje, jaká metrika bude použita pro nalezení cesty, zda je zapnuto dynamické routování, interval přepočtu cest u dynamického routování a zda je během simulace povoleno hierarchické migrování souborů. Metodu simulace lze specifikovat výčtovým typem *SimulationType*, který je použit jako vstupní parametr do simulátoru.

5.3.5 Výběr cesty

Během každého simulovaného požadavku se simulátor snaží nalézt cestu od klienta k jednomu z datových serverů, na kterém se nachází cílová cesta. K nalezení cesty u simulátoru *DfsTimeSliceSimulator* je využíván parametr

konstruktoru typu *DfsPathPicker*. Při nalezení cesty je vždy na začátku započtena celková latence cesty k výslednému času simulace požadavku.

DfsPathPicker

DfsPathPicker popisuje rozhraní, které musí implementovat třída umožňující výběr cesty. Samotný DfsPathPicker se nestará o nalézání všech možných cest, pouze o výběr nejlepší cesty ze všech dostupných. Rozhraní nabízí následující metody:

```
DfsPath selectPath(SimulationTask task, long sTime,
     DfsPath previousPath, SimulationType simType)

long getObjectRegistryQueryTime().
```

Metoda *selectPath* slouží k výběru nejlepší cesty a jejími parametry jsou simulovaný požadavek, již používaná cesta (je-li nastavena, metoda musí vrátit cestu vedoucí do stejného cílového serveru) a typ simulace. Pokud *selectPath* vrátí hodnotu *null*, znamená to, že nebyla nalezena žádná cesta a simulátor připočte k času simulace hodnotu vrácenou metodou *getObjectRegistryQueryTime*, která vrací čas potřebný k dotázání na existenci souboru či složky jednoho ze serverů.

MetricDfsPathPicker

Výchozí implementací *DfsPathPicker* pro výběr cesty je *MetricDfsPathPicker*, který využívá objektu implementující rozhraní *GraphSearcher* pro vyhledávání cest. Při výběru cesty nejdříve zjistí všechny cílové servery, na kterých se nalézá požadovaná cesta a poté pro každý server nechá *GraphSearcher* nalézt optimální cestu. Ze všech cest poté vybírá tu nejoptimálnější porovnáním jejich ohodnocení.

Dynamické routování

Má-li typ běžící simulace zapnuté dynamické routování, simulátor musí po určité době provést opětovně výběr nové cesty. V případě nahrávání souborů

musí mít nová cesta stejný cílový server (tzn. nahrávání se musí vždy dokončit na tom serveru, na kterém začalo). Interval přepočtu cest lze získat z *SimulationType* metodou

```
int getDynamicPathingRecalcInterval(),
```

která vrací hodnotu v milisekundách. Simulátor tedy v každém kroku simulace požadavku kontroluje, zda je třeba provést přepočet cest. Pokud je vybrána nová cesta, je k výslednému času opět připočtena latence cesty a nová cesta je uložena do seznamu, který uchovává historii cest daného požadavku.

5.3.6 Vyhledávání v grafu

Vyhledání cesty od klienta k cílovému serveru představuje problém nalezení optimální cesty v orientovaném ohodnoceném grafu pomocí zvolené metriky. Prohledávání v grafu zajišťuje objekt implementující rozhraní *GraphSearcher*, které má následující metody:

```
Long findPath(ModelNode origin, ModelServerNode target,
     SimulationTask task, long sTime,
     List<ModelNodeConnection> path, SimulationType simType)

GraphMetric getMetric().
```

Metoda *findPath* slouží k nalezení cesty z výchozího uzlu *origin* do cílového serverového uzlu *target* a metoda *getMetric* vrací metriku, která byla použita při hledání cesty.

DijkstraGraphSearcher

Třída *DijkstraGraphSearcher* je implementací *GraphSearcher* sloužící k nalezení optimální cesty pomocí algoritmu vycházejícího z Dijkstrova algoritmu [13, Sekce 24.3].

Dijkstrův algoritmus byl zvolen na základě potřeby nalézt optimální cestu z jediného výchozího uzlu v grafu s kladně ohodnocenými hranami. Složitost Dijkstrova algoritmu je $O(|V^2|)$, kde $|V|$ je celkový počet uzlů.

Alternativami mohou být například algoritmy Bellman-Fordův algoritmus ($O(|V||E|)$, kde $|E|$ je celkový počet hran) [9] nebo Floyd-Warshallův algoritmus [18] ($O(|V^3|)$).

Hrany mezi uzly jsou hodnoceny metrikou *GraphMetric*, která zároveň poskytuje komparátor pro výběr nejlepší hrany. Lze tedy hledat cesty podle více kritérií a nejsme omezeni pouze výběrem nejkratší, jak je tomu u základní verze algoritmu.

5.3.7 Metrika grafu

Metrika je využívána při prohledávání grafu k nalezení optimální cesty a je reprezentována objektem implementující rozhraní *GraphMetric*.

Dostupné metriky

V základní verzi simulátoru je implementováno následujících 6 metrik pro nalezení cesty:

- **DistanceMetric** Metrika sloužící k nalezení nejkratší cesty.
- **LinkBwMetric** Metrika sloužící k nalezení cesty s největší propustností (nejširší cesta).
- **LinkBwLatencyMetric** Metrika kombinující nalezení cesty s největší propustností a zároveň nejmenší latencí.
- **PathThroughputMetric** Metrika kombinující nalezení cesty s největší propustností a zároveň s nejrychlejším diskem.
- **PathThroughputLatencyMetric** Podobně jako *PathThroughputMetric*, ovšem minimalizuje zároveň latenci.
- **HierarchicalThroughputMetric** Podobně jako *PathThroughputLatencyMetric*, ale vybírá cestu s ohledem na maximální možnou propustnost disku po provedení hierarchické migrace, tedy např. pokud je stahovaný soubor *f1* na serverech *S1* a *S2*, kde na S1 má disk rychlost 100 MB/s a na S2 také 100 MB/s s tím, že na S2 existuje rychlejší disk 150 MB/s, který má volné místo, bude vybrána cesta k S2.

Popis rozhraní

Rozhraní má následující metody:

- **getBestMetricValue** vrací nejlepší ohodnocení hrany
- **getWorstMetricValue** vrací nejhorší ohodnocení hrany
- **getEdgeWeight** vrací ohodnocení orientované hrany mezi dvěma uzly
- **getCombinedEdgeWeight** vrací kombinované ohodnocení existující cesty a nové hrany
- **getComparator** vrací objekt sloužící k porovnání dvou metrik
- **getPossibleDiskBandwidth** vrací rychlost úložiště na nejvyšší vrstvě, kam by bylo možné soubor odmigrovat

Implementace metriky

Zde je uvedena ukázka implementace spolu s vysvětlujícími komentáři pro metriku, která nalezne nejkratší cestu:

```
// nejkratší možná cesta
long getBestMetricValue() {
     return 1L;
}
// nejdelší možná cesta
long getWorstMetricValue() {
     return Long.MAX_VALUE;
}
// libovolná hrana má vzdálenost 1
long getEdgeWeight(...) {
     return 1L;
}
// kombinace délky existující cesty a nové hrany je součtem
long getCombinedEdgeWeight(...) {
     return (weightCurrent + weightEdge);
```

```
}
// vrací hodnotu menší než 0 pokud je o1 kratší než o2
// hodnotu 0 pokud jsou si rovny a hodnotu větší než 0
// pokud je o1 delší než o2
Comparator<Long> getComparator() {
     return (Long o1, Long o2) -> (o1.compareTo(o2));
}
// podporuje-li hierarchické migrace, vrací nejrychlejší
// dostupné úložiště, jinak vrátí rychlost aktuálního disku
public ByteSpeed getPossibleDiskBandwidth(...) {
     ...
}
```

5.3.8 Replikace nahraných souborů

Po nahrání souboru, který je replikovaný na více serverech, je třeba změny z cílového serveru replikovat na ostatní servery. Replikace je spuštěna po dokončení nahrání souboru metodou *propagateReplicaResize* v třídě *FsGlobalReplicationManager*. Pro migraci jsou vytvořeny patřičné operace čtení a zápisu a cesta mezi servery je nalezena podle grafové metriky běžící simulace. Rychlost migrace je limitována propustností disků obou serverů a nalezenou cestou mezi servery.

5.3.9 Hierarchické migrování

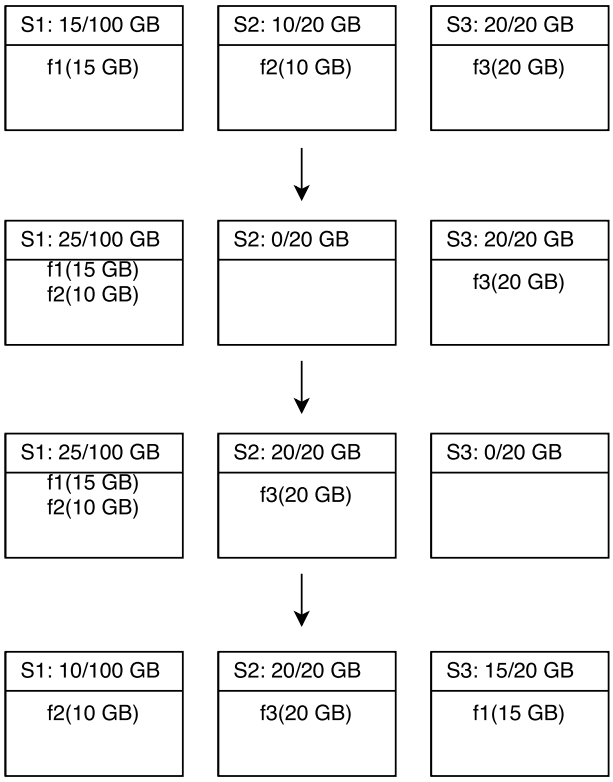
Pokud běží simulace, jejíž typ má povolené hierarchické migrování souborů, je před a po každém přístupu k souboru volána metoda objektu implementujícího rozhraní *HierarchicalPlanner*. Tento objekt monitorující změny čítačů přístupu a implementuje migrační strategii, podle které se soubory přesouvají mezi úložišti. Okamžik spuštění migrace záleží na konkrétní implementaci.

5.3.10 Plánovač migrace souborů LFU

Výchozí implementací je *LFUCascadeMigrationPlanner* využívající cachovací strategii LFU pro uvolnění místa na rychlejších úložištích. K plánování migrace je využita metoda *onBeforeAccess*, která je zavolána před začátkem přenosu souboru.

Nejprve je u vstupního souboru inkrementován čítač přístupů. Následně se hledá nejrychlejší úložiště, které má dostatek místa pro umístění souboru. V případě, že žádné vyšší úložiště nemá dostatek místa, je vybráno takové úložiště, z kterého lze určitou množinu souborů odmigrovat na nižší úroveň, aby se uvolnilo dostatek místa. Množina souboru k migraci na nižší úložiště je nejmenší množinou s čítačem přístupů menším, než má vstupní soubor. Není-li na nižším úložišti pro tuto množinu dostatek místa, algoritmus se rekurzivně snaží odmigrovat data na nižší vrstvy, aby se místo uvolnilo (popsáno na příkladu).

Příklad migrace



Obrázek 5.4: Příklad rozmístění souborů v hierarchickém systému

Na obr. 5.4 máme 3 disky *S1*, *S2* a *S3* seřazeny podle jejich rychlosti a chceme umístit soubor *f1* o velikosti 15 GB na disk *S3*. Výsledný proces migrace má celkem 3 kroky. Disk *S3* má 0 GB volného prostoru, ale za předpokladu, že soubor *f3* má nižší čítač přístupů než *f1*, můžeme se pokusit odsunout *f3* na disk *S2*. Disk *S2* nemá dostatek místa pro soubor *f3*, ale po odsunutí souboru *f2* na úložiště *S1* již ano, které má zároveň dostatek volného místa. V 1. kroku tedy umístíme soubor *f2* na úložiště *S1*. V kroku 2 můžeme z disku *S3* přesunout soubor na *S2*. Posledním krokem je přesunutí souboru *f1* na nyní již prázdný disk *S3*.

5.3.11 Logování

Pro poskytnutí detailnějšího průběhu simulace jsou zaznamenávány detaily vybraných událostí, jakými jsou např. výběr cesty pro přenos dat, migrace

dat a dokončení simulace požadavku. Logovací objekt implementující rozhraní *DfsSimulatorLogger* je předán simulátoru při startu simulace.

Výchozí implementací je *DfsStringSimulatorLogger*, který události ukládá jako řetězec do interního seznamu. Seznam všech logovaných událostí lze získat metodou *getMessages*.

5.3.12 Vzorkování rychlosti

Aby bylo možné po doběhnutí simulace analyzovat průběh přenosu dat, je při každém kroku simulátoru vzorkována aktuální rychlost přenosu dat daného požadavku. Vzorkování probíhá pomocí objektu, který implementuje rozhraní *SimulationThroughputSampler*.

Filtrování vzorků

Výchozí implementací vzorkovače je *FilteringThroughputSampler*, který se snaží minimalizovat počet záznamů dodatečným filtrováním hodnot. Filtrování probíhá na základě porovnání předchozí vzorkované hodnoty s novou a pokud se rovnají, tak je nová hodnota zanedbána. Pokud by bylo žádoucí ještě více zmenšit počet vzorků, lze využít např. hranice minimálního rozdílu, o kterou se musí nový vzorek lišit či jiné pokročilé metody filtrování.

5.4 Import a export prostředí

Modul perzistence v balíku *persistence* slouží k uložení modelovaného systému a simulačního plánu vybraného klienta. Třída, která chce být exportovatelná a následně obnovitelná musí implementovat rozhraní *StatePersistable*. Export probíhá od kořenového prvku přes všechny jeho následníky.

5.4.1 Elementy a atributy

Exportují se dva typy objektů a to elementy *StatePersistableElement* a atributy *StatePersistableAttribute*. Atributem je jednoduchý objekt, který uchovává klíč (název atributu) a hodnotu. Element může mít atributy a další

elementy jako následníky. Každý exportovaný objekt *StatePersistable* je exportován jako element a své potomky může exportovat manuálně vytvořením objektů typu *StatePersistableElement* a nebo vrácením seznamu potomků implementujících rozhraní *StatePersistable* v metodě *getPersistableChildren*.

5.4.2 Uložení stavu

Uložení stavu simulátoru umožňuje objekt implementující rozhraní *StatePersistor* metodou *persist*, které se předá kořenový element. Každá třída implementující rozhraní *StatePersistable* si řídí, jaké atributy a elementy budou součástí exportu.

V metodě *export*, která je rekurzivně volána od kořene, vrací *StatePersistableElement*, kterému může nastavit atributy metodou *addAttribute* a následníky metodou *addElement*. Následníci, které třída vrací v metodě *getPersistableChildren* budou rekurzivně exportováni stejným způsobem, dokud se nenarazí na následníka, který žádné další následníky nemá.

Uložení do XML

Výchozí implementací je *FileXmlStatePersistor*, který nejprve exportuje stav do formátu XML a následně ho uloží do souboru. XML soubor je vytvořen pomocí knihoven z balíku *javax.xml*, které jsou součástí výchozí instalace Javy.

5.4.3 Načtení stavu

Načtení stavu probíhá obdobně jako uložení. Řízení obnovení stavu má na starosti objekt implementující rozhraní *StateRestorer* metodou *restore*. Každá třída implementující rozhraní *StatePersistable* si řídí obnovení svých atributů a potomků.

Obnovení objektu probíhá zavoláním metody *restoreState*, u které je jedním z parametrů aktuální element *StatePersistableElement*, ze kterého lze získat všechny atributy elementu a jeho následníky. Element při obnově svých následníků volá opět metodu *restoreState*.

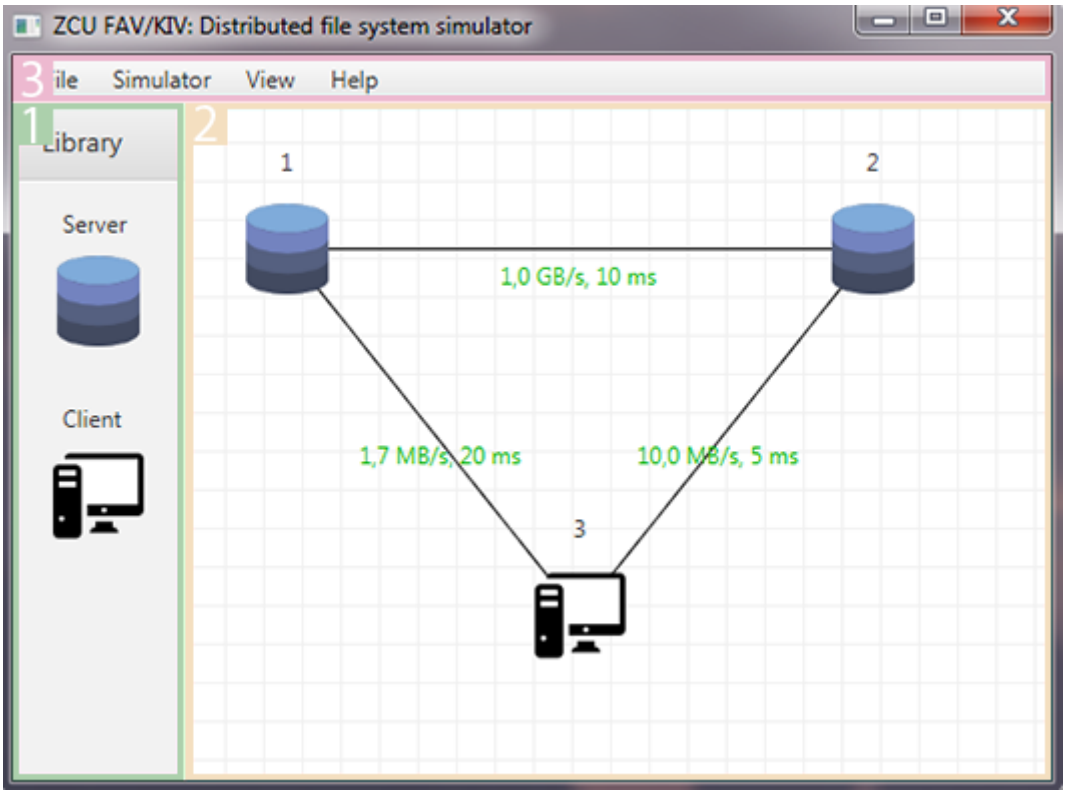
Načtení z XML

K třídě, která ukládá stav do XML je implementována třída *FileXmlStateRestorer*, která tento uložený stav umožňuje obnovit. Obnova probíhá zavoláním metody *restore*, která dostává jako parametr soubor s uloženým stavem.

6 Implementace grafického rozhraní

Soubory grafického rozhraní jsou rozděleny na dvě části, jednou je balík *cz.zcu.kiv.dfs_simulator.view*, který obsahuje veškerou logiku spojenou s GUI a druhou je adresář *src/main/resources*, ve kterém jsou CSS[1] soubory, obrázky a FXML soubory popisující rozmístění grafických komponent.

6.1 Rozložení GUI



Obrázek 6.1: Rozložení grafických částí simulátoru

Na obrázku 6.1 je vidět základní rozložení simulátoru, které se skládá ze tří částí a to: postranní panel s komponentami (1), plochou pro modelování

[1]https://www.w3.org/Style/CSS/Overview.en.html

systému (2) a horní menu (3). Všechny tyto části jsou součástí *SimulatorLayoutPane*, které řeší jejich inicializaci a zprostředkovává jejich komunikaci pomocí událostí.

6.2 Grafické komponenty

Pro modelování systému jsou součástí GUI grafické komponenty, které jsou nadstavbou komponent modelu. Všechny grafické komponenty dědí od některého potomka *javafx.Scene.Node*.

6.2.1 Grafické uzly

Hlavní grafickou komponentou je abstraktní třída *FxModelNode*, která představuje uzel grafu a společnou grafickou logiku pro klient a server. Nadstavbou klienta je *FxModelClientNode* a serveru *FxModelServerNode*. Součástí grafického uzlu je i speciální textové pole *FxModelNodeLabel*, které zobrazuje identifikátor uzlu.

6.2.2 Grafická hrana

Grafické propojení mezi uzly je řešeno pomocí třídy *FxNodeConnectionWrapper*, která obaluje grafický prvek zobrazující propoj mezi uzly, dvě instance spojení *ModelNodeConnection* a jednu společnou charakteristiku spoje *LineConnectionCharacteristic*.

Tato obalová třída vyžaduje v konstruktoru mimo jiné zdrojový a cílový uzel a propustnost spoje. Přestože model umožňuje nesymetrické propoje, tedy různé vlastnosti spoje závislé na směru, grafická nadstavba aktuálně pracuje pouze se symetrickými spoji.

Viditelná hrana je na plátnu reprezentována třídou dědící od abstraktní třídy *FxNodeLink*. Hrana může mít vlastní popisek a kontextové menu pro nastavení vlastností spoje. V základní verzi simulátoru jsou dvě třídy umožňující vykreslení hrany a to propojení úsečkou *LineFxNodeLink* nebo Bézierovo křivkou *BezierFxNodeLink*.

6.3 Plátno pro modelování

Modelování systému probíhá na plátně realizovaném pomocí *ModelGraphLayoutPane*, které dědí od rodičovské třídy *ScrollPane*. Rodičovská třída *ScrollPane* umožní automatické rozšiřování plátna vytvořením nové komponenty či přesunutím stávající mimo pravý nebo dolní okraj plátna.

Plátno akceptuje události vyvolané upuštěním komponenty z knihovny a reaguje na ně vytvořením odpovídající komponenty na plátně se souřadnicemi získanými z kurzoru myši. Mimo tyto události ještě slouží pro odchytávání událostí vyvolaných komponentami na plátně, a to události jako je vytvoření hrany mezi uzly či otevření konfiguračního dialogu komponenty. Reakce na události jsou registrovány v metodě *setActionEventHandlers*.

6.4 Přesun komponent myší

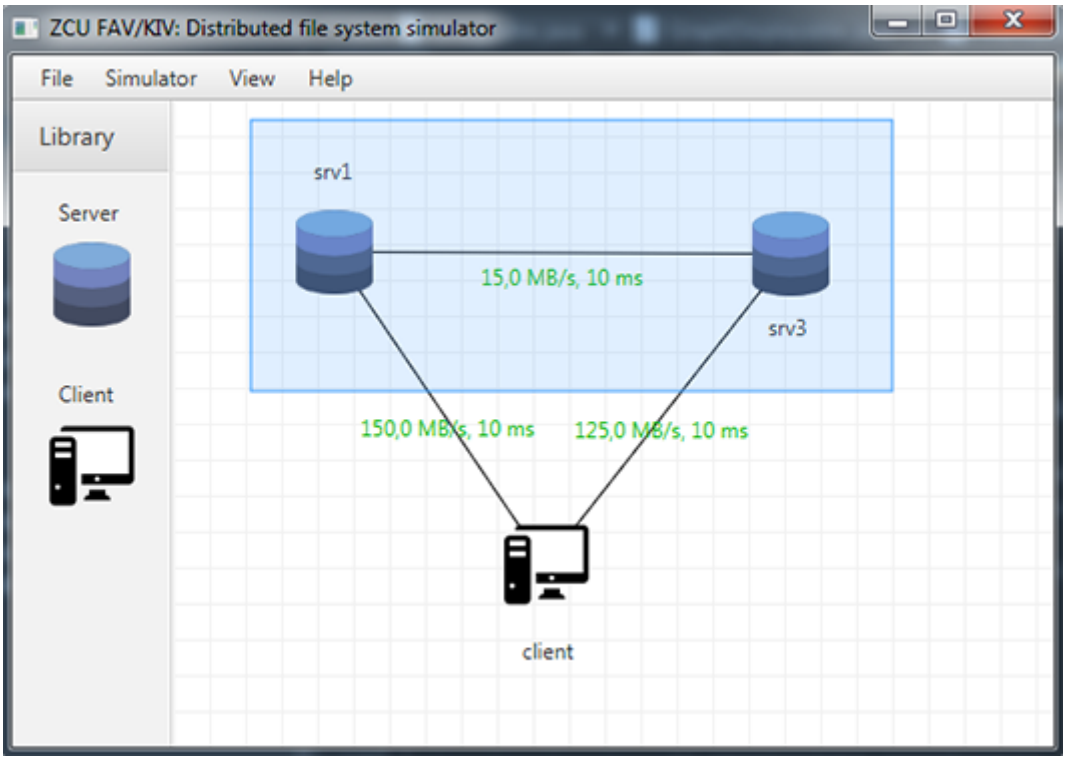
Některé komponenty ležící plátně reagují na události vyvolané tažením myši aktualizací své pozice podle pozice kurzoru myši. Logika přesunu komponenty je předána metodě *setOnMouseDragged* dostupné z rodičovské třídy.

Ještě před tažením myši je zaregistrována událost kliknutí myší na komponentu, během které si komponenta interně uloží rozdíl pozice kurzoru od svého levého horního rohu. Rozdíl pozice kurzoru je uložen aby bylo možné správně vypočítat nové souřadnice komponenty na základě aktualizované pozice kurzoru.

V případě grafického uzlu je ještě před aktualizací pozice v metodě *handleNodeDragged* ověřeno, zda nové souřadnice neleží mimo plátno. Pokud je vše v pořádku, komponenta aktualizuje svojí pozici metodami *setLayoutX* a *setLayoutY*.

6.4.1 Výběr více komponent

Aby bylo možné komponentu vybrat spolu s jinými komponentami, musí implementovat rozhraní *MultiSelectableDraggable*. Vybrání více komponent probíhá tažením obdélníku z některého bodu plátna a všechny komponenty nacházející se vně tohoto obdélníku jsou označeny najednou (viz obr. 6.2).



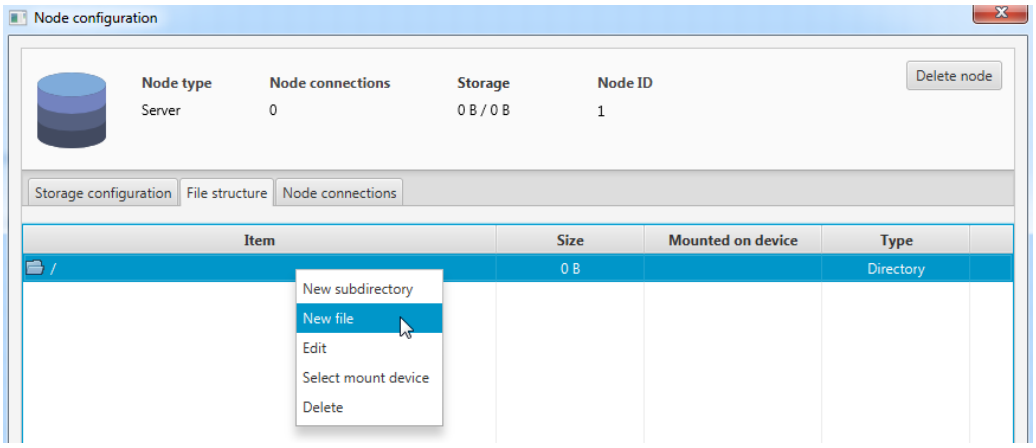
Obrázek 6.2: Výběr více komponent najednou

Výběr více komponent zařizuje třída *RubberBandSelection*, která v konstruktoru dostane instanci plátna pro modelování *ModelGraphLayoutPane*. Do předané instance vloží své handlery událostí vyvolané myší, kterými se výběr vytváří. Po dokončení výběru projde všechny potomky uvnitř výběru, kteří implementují rozhraní *MultiSelectableDraggable* a přidá je do výběru.

Každá komponenta je notifikována metodou *setMultiSelectableState*, zda se nachází ve vícenásobném výběru komponent a tento stav si interně uloží. Komponenta poté v handleru *setOnMouseDragged* na základě stavu může notifikovat ostatní komponenty ve výběru vyvoláním události *MultiNodeSelectionEvent*, konkrétně typu *DRAG_SELECTION*.

6.5 Konfigurace komponenty

Komponenty typu uzel a linka mohou být dodatečně konfigurovány. Konfigurace probíhá přes kontextové menu vyvoláním konfiguračního dialogu pro danou komponentu.



Obrázek 6.3: Dialog konfigurace serveru - adresářová struktura

6.5.1 Konfigurace serveru

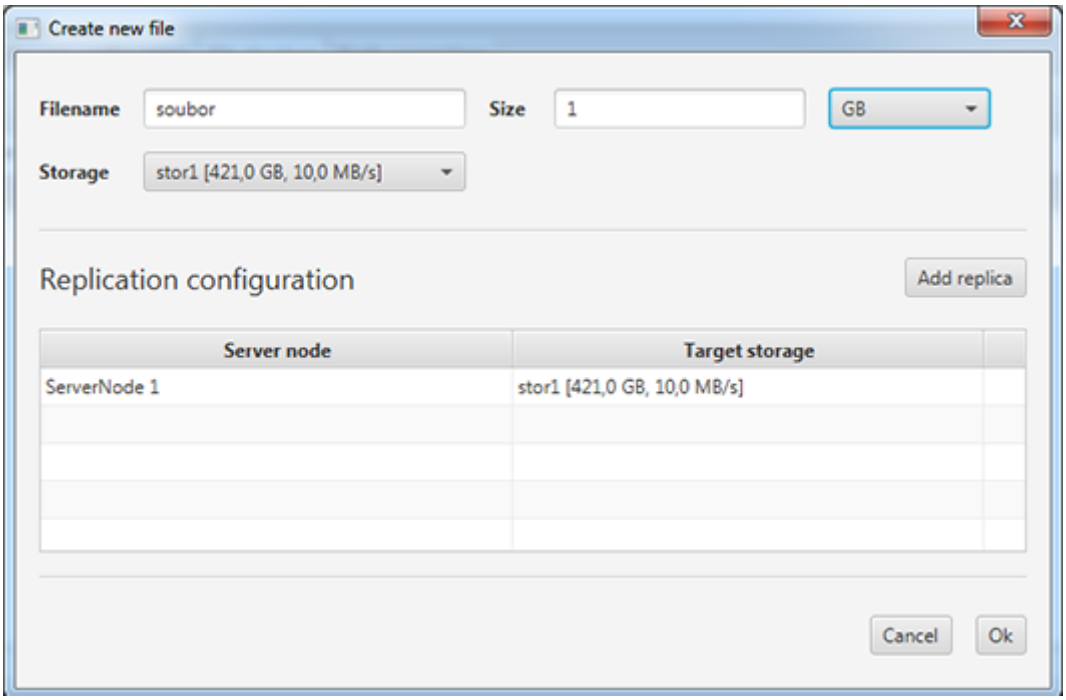
Konfigurace serveru probíhá pomocí dialogu *FxServerNodeContextDialog*. Dialog je rozdělen na záložky a umožňuje konfigurovat úložiště, adresářovou strukturu a spojení s ostatními uzly. Konfigurace úložiště probíhá pomocí tabulky *FxStorageTable*, která zobrazuje aktuální úložiště serveru a pomocí kontextového menu s ním lze dále pracovat (viz obr. 6.3).

Vytváření adresářové struktury

Adresářová struktura serveru je přístupná pomocí tabulky se stromovou strukturou *FxFsTable*. Tabulka přes kontextové menu umožňuje vytváření adresářů, souborů a nastavení úložiště, na kterém se složka nebo soubor nachází.

Vytvoření adresáře probíhá přes jednoduchý dialog *FxFsDirectoryDialog*, ve kterém stačí specifikovat název nové složky. Rodičem složky bude složka z odpovídající úrovně stromové struktury, ze které bylo vyvoláno kontextové menu.

Vytváření a úpravu souborů umožňuje dialog *FxFsFileDialog* (obr. 6.4), který vyžaduje jméno souboru, velikost a úložiště. Při vytváření lze specifikovat i repliky souboru, které budou automaticky vytvořeny na odpovídajících serverech a vybraných úložištích. Při replikaci souboru je nejdřív zkontrolováno, zda je na všech úložištích dostatek místa pro soubor ve třídě *FsGlobalReplicationManager* metodou *updateReplicaTargets*. Není-li dostatek místa,



Obrázek 6.4: Dialog pro vytvoření a úpravu souboru

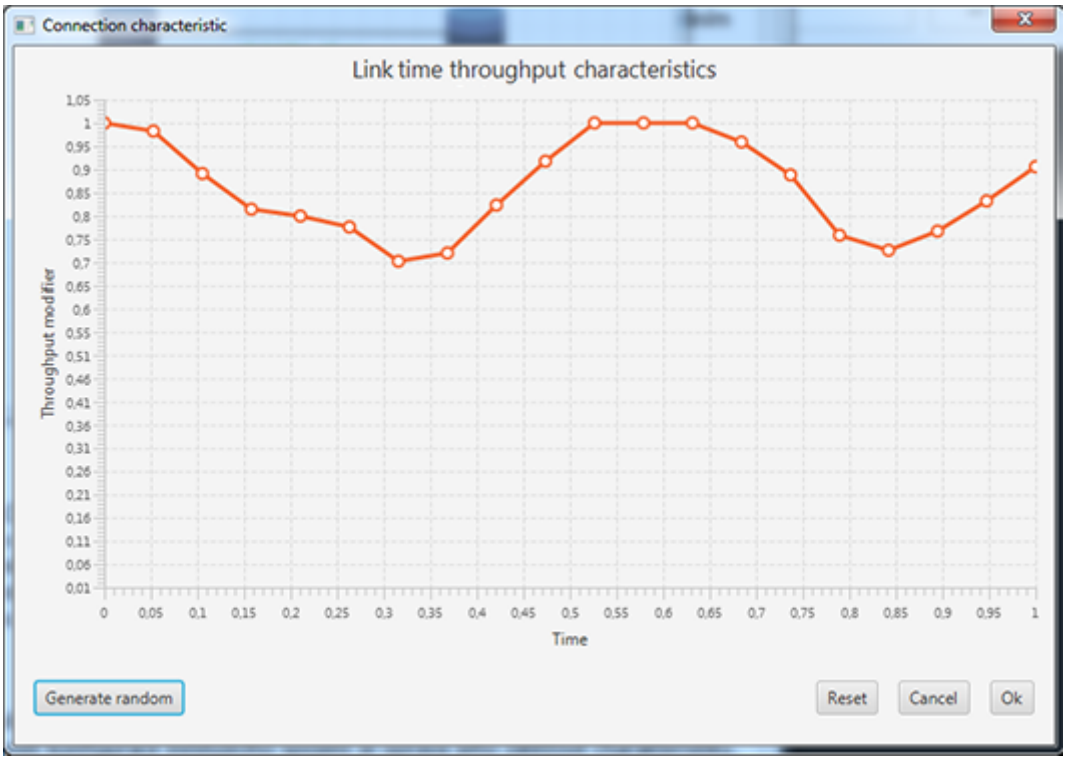
je vyvolána vyjímka *NotEnoughSpaceLeftException*, která je zachycena a pomocí dialogu je uživateli umožněno automaticky místo rozšířit.

6.5.2 Konfigurace klienta

Konfigurace klienta probíhá dialogem *FxClientNodeContextDialog* na podobném principu jako při konfiguraci serveru. Dialog je rozdělen na dvě záložky kde v jedné je konfigurace simulačního plánu a v druhé tabulka s spojení s ostatními uzly. V záložce simulační plán lze rovněž vybrat jeden nebo více typů simulace, odstartovat simulaci podle simulačního plánu a následně spustit její vizualizaci vyvoláním události *FxNodeSimulationEvent* typu *SIMULATION_VISUALISATION_ON_REQUESTED*, kterou zachytí *SimulatorLayoutPane*.

6.5.3 Konfigurace spojení

Dialog *FxNodeLinkDialog* slouží k vytvoření nového spojení a konfiguraci existujícího. U spojení umožňuje konfigurovat jeho rychlost a latenci. Další



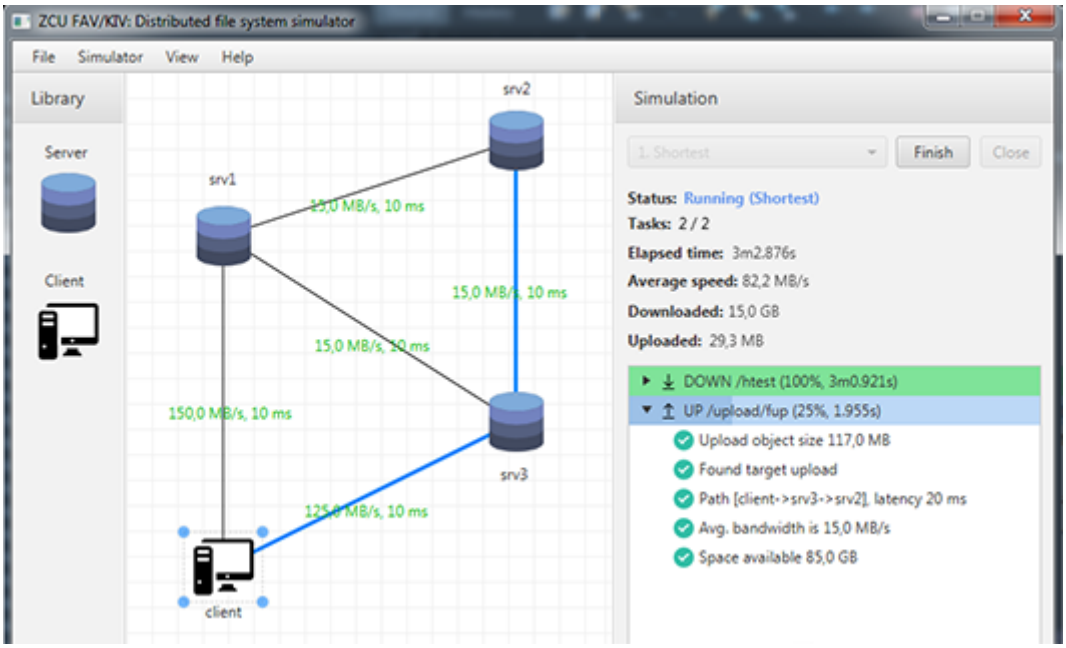
Obrázek 6.5: Zátěžová charakteristika spojení

možností konfigurace existujícího spojení je úprava jeho zátěžové charakteristiky, která probíhá vyvoláním dialogu *FxNodeLinkCharacteristicDialog* přes kontextové menu spojení.

Konfigurace lineární zátěžové charakteristiky probíhá pomocí spojnicového grafu, který vykresluje diskrétní body charakteristiky a spojnice mezi nimi (obr. 6.5). Body grafu, jejichž chování je řízeno třídou *MarkerNodeMoveHandler*, lze posouvat myší podél osy y udávající modifikátor propustnosti v daném čase.

6.6 Průběh simulace a její vizualizace

Spuštění simulace probíhá nastavení konkrétního klienta. Klient může vybrat jeden nebo více typů simulace (*SimulationType*), které budou simulovány a vytvořit seznam simulovaných požadavků. Při vybrání více typů simulace proběhne simulace opakovaně pro každý typ zvlášť se stejnými simulovanými požadavky. Simulované požadavky jsou provedeny opakovaně



Obrázek 6.6: Vizualizace simulace

pro každý typ simulace. Po spuštění simulace je zobrazen dialog s progress barem, který indikuje kolik typů simulací ještě zbývá provést. Během simulace jsou výsledky simulace akumulovány v seznamu, který je následně předán do *SimulatorLayoutPane* pomocí události *FxNodeSimulationEvent*.

Průběh simulace je rekonstruován z výsledků a zrychleně přehrán uživateli objektem třídy *FxSimulationPlayer*. Podoba vizualizace je zobrazena na obrázku 6.6. Na plátně pro modelování lze vidět zvýrazněnou cestu modrou barvou od klienta k cílovému serveru a v pravé části průběh simulace požadavků.

6.6.1 Zvýraznění cesty

Při vizualizaci jsou pro každý požadavek zvýrazněny všechny cesty, kudy tekla data od klienta k serveru (replikace není zohledněna). Během simulace je ukládána historie cest u každého požadavku (viz 5.3.3). Všechny unikátní hrany z historie jsou sjednoceny do množiny *FxNodeLink*, pro kterou je nad každým prvkem zavolána metoda

```
void setLinkHighlighted(boolean highlighted),
```

kde parametr *highlighted* udává, zda má být zvýraznění zapnuto či vypnuto.

6.6.2 Průběh požadavků

Rozhraní *SimulationLogDisplayable* popisuje metody, které musí být implementovány pro simulaci průběhu požadavků. Základní implementací je *FxSimulationBar*, který tvoří postranní panel se seznamem proběhlých typů simulací a aktuálním pohledem na simulované požadavky.

Seznam požadavků je zobrazen pomocí objektu třídy *TreeView*. Každý požadavek tvoří vrchol stromu a může mít libovolné následníky, poskytující dodatečné stavové informace. Přidání vrcholu do stromu požadavků probíhá metodou

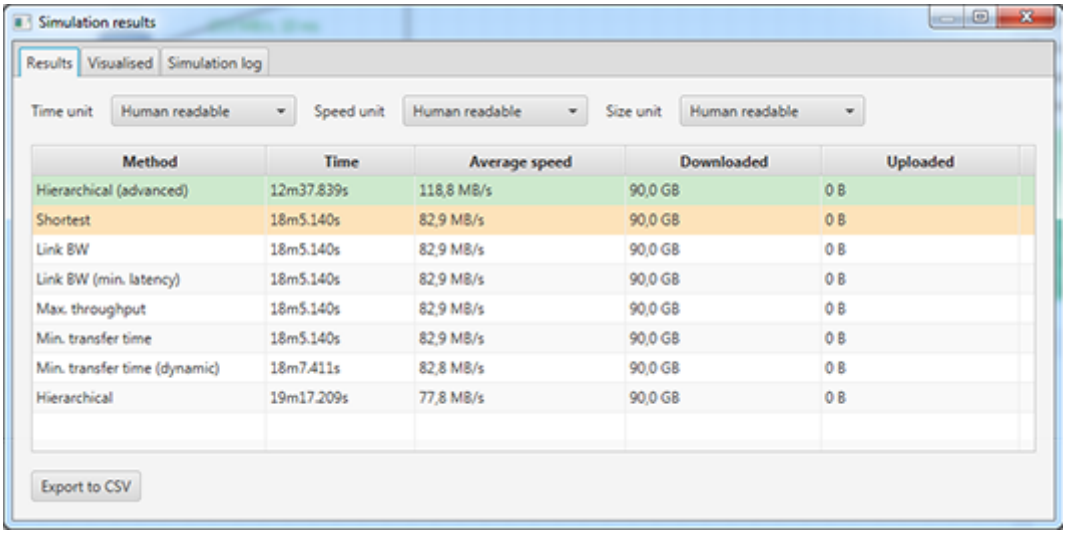
```
void logSimulationEvent(FxSimulationLogEvent event),
```

která může pomocí parametru *event* specifikovat svého předchůdce. Vizualizace umožňuje vykreslit maximálně dvě úrovně, kde první úrovní jsou požadavky a druhou úrovní jsou stavové informace přiřazené k požadavkům. Stávající vrcholy lze aktualizovat metodou *updateSimulationEvent*.

Animace průběhu

Během vizualizace průběhu požadavku je periodicky měněna CSS třída pozadí vrcholu požadavku. Pozadí požadavku vizualizuje procentuální průběh a je implementováno pomocí třídy *PseudoClass*, která umožňuje dynamickou změnu aktivních CSS tříd elementu. Změna pozadí probíhá v metodě *updateCellPseudoClass* po každé aktualizaci vrcholu.

6.7 Výsledky simulace



Obrázek 6.7: Okno s výsledky simulace

Pro zobrazení výsledků slouží okno *SimulationResultsWindow*. Ve výsledcích jsou tři záložky a to souhrnné výsledky, graf průběhu rychlosti a detailní log průběhu simulace (viz obr. 6.7).

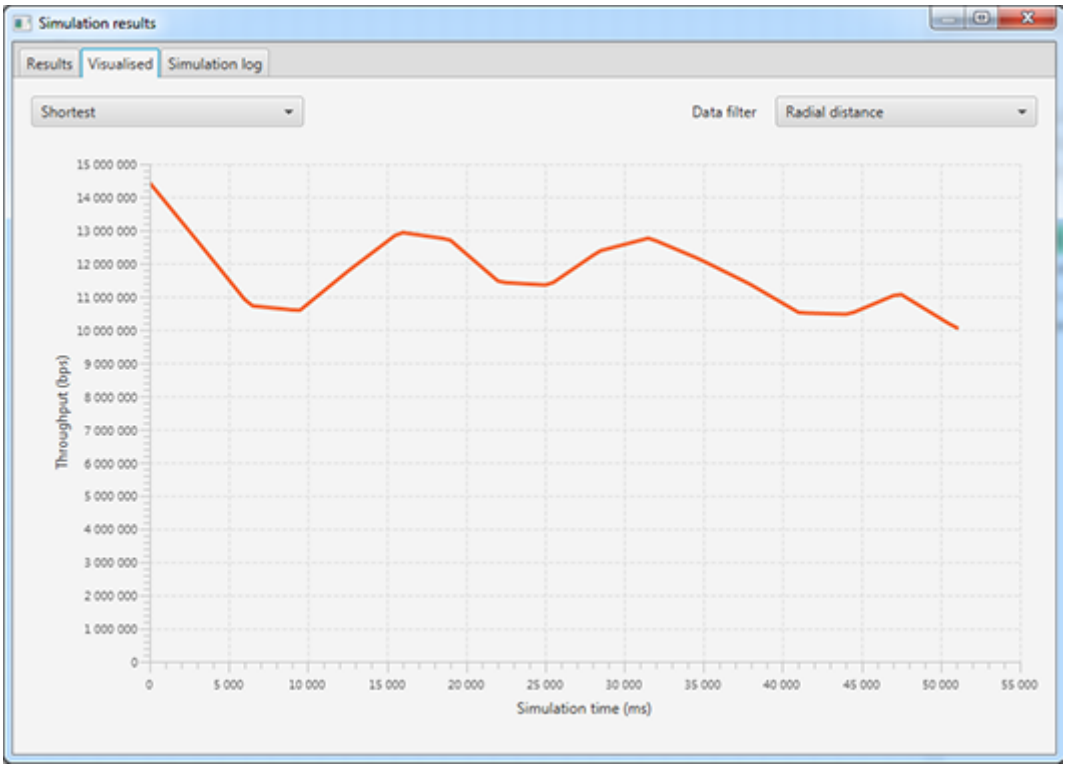
V tabulce se souhrnnými výsledky jsou pomocí změny CSS třídy zvýrazněny dva nejlepší výsledky. Změnou volby v jednom ze seznamů *ChoiceBox* pro výběr násobků je vyvolána aktualizace, která pomocí metod *getConvertedResultTime*, *getConvertedResultSize* a *getConvertedResultSpeed* získá přepočtené hodnoty měření a následně je vykreslí.

6.7.1 Graf průběhu rychlosti

Na obrázku 6.8 je vidět graf *LineChart* zobrazující průběh rychlosti simulace v závislosti na čase. Průběh rychlosti v grafu je složen ze všech simulovaných požadavků.

6.7.2 Filtrace vzorků rychlosti

Navzorkovaných hodnot rychlosti během simulace bývá mnoho a jejich vykreslení je poměrně náročné. Z tohoto důvodu jsou vytvořeny filtry hodnot grafu, které se snaží odstranit vzorky, jejichž odebrání nebude mít zásadní



Obrázek 6.8: Graf průběhu rychlosti

vliv na podobu grafu. Filtr musí implementovat rozhraní *SimulationThroughputHistoryReducer*, kde metodou

```
List<Pair<Long, Long>> reduce(
    List<Pair<Long, Long>> original,
    long totalElapsedTime,
    ByteSpeed totalAverageSpeed)
```

vrací redukovaný seznam původních hodnot v seznamu *original*. Implementovanými filtry jsou

- **AllInclusiveThroughputHistoryReducer** vrací všechny vzorky, náročný, přesný ale ne příliš vhodný pokud máme hodně vzorků.
- **DouglasPeuckerThroughputHistoryReducer** využívá Douglas-Peucker algoritmu [49].
- **RadialDistanceThroughputHistoryReducer** funguje na bázi eukleidovské vzdálenosti bodů.

- **MovingAverageReducer** výchozí filtr - dělí hodnoty do časových intervalů a pro každý časový interval počítá průměr hodnot. Filtrace probíhá rychle i pro větší množství vzorků.

7 Testy

V této kapitole jsou popsány provedené testy pro ověření programové funkčnosti simulátoru a měření navrhovaného distribuovaného souborového systému s hierarchickým modelem uložení dat. Model a plán simulace vycházejí z výzkumu Ing. Pešičky a slouží pro ověření vlastností navrhovaného modelu.

7.1 Unit testy

Pro ověření základní funkčnosti simulátoru byly vytvořeny unit testy, které pokrývají nejdůležitější funkčnost aplikační části simulátoru. Testy jsou umístěny ve složce *src/test*. V následujících kapitolách budou zmíněny testy nejdůležitější funkcionality. Uváděné názvy balíku jsou vždy relativní cestou k výchozímu balíku *cz.zcu.kiv.dfs_simulator*.

7.1.1 Testy spojení

Spojení mezi uzly je testována uvnitř balíku *model.connection* a to třemi třídami *ModelNodeConnectionTest*, *ModelNodeConnectionManagerTest* a *LineConnectionCharacteristicTest*.

ModelNodeConnectionTest

Uvnitř testovací třídy *ModelNodeConnectionTest* je testována základní funkčnost třídy *ModelNodeConnection*. Test ověřuje vytvoření spojení mezi dvěma uzly, správný počáteční a cílový uzel (test orientace spojení), hodnotu maximální propustnosti spojení a výpočet průměrné přenosové rychlosti během specifikovaného času.

ModelNodeConnectionManagerTest

Tato testovací třída testuje funkcionalitu *ModelNodeConnectionManager* a to konkrétně vytvoření obousměrného spojení mezi uzly, získání přímých sousedů uzlů a získání všech dostupných uzlů z výchozího uzlu.

LineConnectionCharacteristicTest

Cílem této testovací třídy je ověřit funkcionalitu lineární periodické charakteristiky *LineConnectionCharacteristic* a jejího vlivu na propustnost spojení mezi uzly. Během testu je nejdříve otestována propustnost linky v čase, ve kterém je přímo definována charakteristika, následně je otestována propustnost v čase ležící mimo definované body a nakonec průměrná propustnost během celého intervalu. Hodnoty charakteristiky jsou porovnávány s hodnotami vypočtenými podle vzorce uvedeného u lineární zátěžové charakteristiky v kapitole 5.2.4.

7.1.2 Testy souborového systému

Následující testy v balíku *model.storage.filesystem* slouží pro ověření funkčnosti spojené se souborovým systémem.

FsDirectoryTest

Testovací třída *FsDirectoryTest* ověřuje základní funkcionalitu spojenou s třídou *FsDirectory*. Hlavním cílem je testování správného vytváření cesty přes všechny potomky až ke kořeni, vyhledávání přímo v dané složce a vyhledávání v některém z potomků složky (vnořená složka) a vyhledání potomka podle cesty a typu (soubor či složka).

FsFileTest

Tato třída testuje funkčnost třídy *FsFile* a to získání absolutní cesty k souboru přes všechny jeho předchůdce až ke kořenové složce, získání velikosti souboru a nastavení nové velikosti souboru a funkčnost čítače přístupů k souboru.

FsObjectChildContainerTest

Uvnitř této třídy je testována třída *FsObjectChildContainer*, která slouží k výpočtu velikosti všech potomků a výpočtu velikosti všech potomků, kteří nejsou explicitně umístěni na některém úložišti (tedy budou sdílet úložiště rodiče).

ServerFsManagerTest

Cílem této třídy je otestovat funkcionalitu třídy *ServerFsManager*, která slouží ke správě souborového systému serveru. Uvnitř testovací třídy je testováno zjištění velikosti obsazeného místa na daném úložišti, úspěšné přidání potomka do složky a neúspěch při přidání potomka do složky, které by způsobilo nedostatek místa na úložišti. Dále je testováno umístění souboru na úložiště, zjištění, na kterém úložišti je soubor uložen a zda se může soubor zvětšit na danou velikost aniž by přesáhl limit dostupného místa na úložišti.

7.1.3 Testy diskových operací

Testy v balíku *model.storage* testují diskové operace probíhající na úložišti.

StorageOperationManagerTest

Třída *StorageOperationManagerTest* testuje správce diskových operací *StorageOperationManager*. V této testovací třídě je ověřováno správné vytvoření diskových operací, dělení rychlosti mezi běžící operace a výpočet přenesených dat pro každou operaci za danou dobu.

7.1.4 Testy vyhledávání cesty a metrik

Testy, které ověřují výpočet metriky cesty, jsou v balíku *model.simulation.path* a testy vyhledávání podle vypočtených metrik v *model.simulation.graph.metric*.

Testování výpočtu metriky cesty

Testy metriky testují správný výpočet metriky pro danou cestu. V následujícím seznamu je uveden seznam testovacích tříd a metrik, které testují:

- **DistanceMetricTest** testuje třídu *DistanceMetrik* a kontroluje ohodnocení nejkratší cesty,
- **LinkBwMetricTest** testuje třídu *LinkBwMetric* a kontroluje ohodnocení cesty s maximální propustností spojení,

- **LinkBwLatencyMetricTest** testuje třídu *LinkBwLatencyMetric* a kontroluje ohodnocení cesty s maximální propustností spojení a minimální latencí,
- **PathThroughputMetricTest** testuje třídu *PathThroughputMetric* a kontroluje ohodnocení cesty s maximální propustností spojení a maximální rychlostí disku,
- **PathThroughputLatencyMetricTest** testuje třídu *PathThroughputLatencyMetric* a kontroluje ohodnocení cesty s maximální propustností spojení, maximální rychlostí disku a minimální celkové latenci,
- **HierarchicalThroughputMetricTest** testuje třídu *HierarchicalThroughputMetric* a kontroluje správné určení maximální rychlosti disku, pokud by byla provedena migrace.

MetricDfsPathPickerTest

Tato testovací třída testuje třídu *MetricDfsPathPicker*, konkrétně nalezení odpovídající cesty podle použité metriky.

7.1.5 Testy simulace

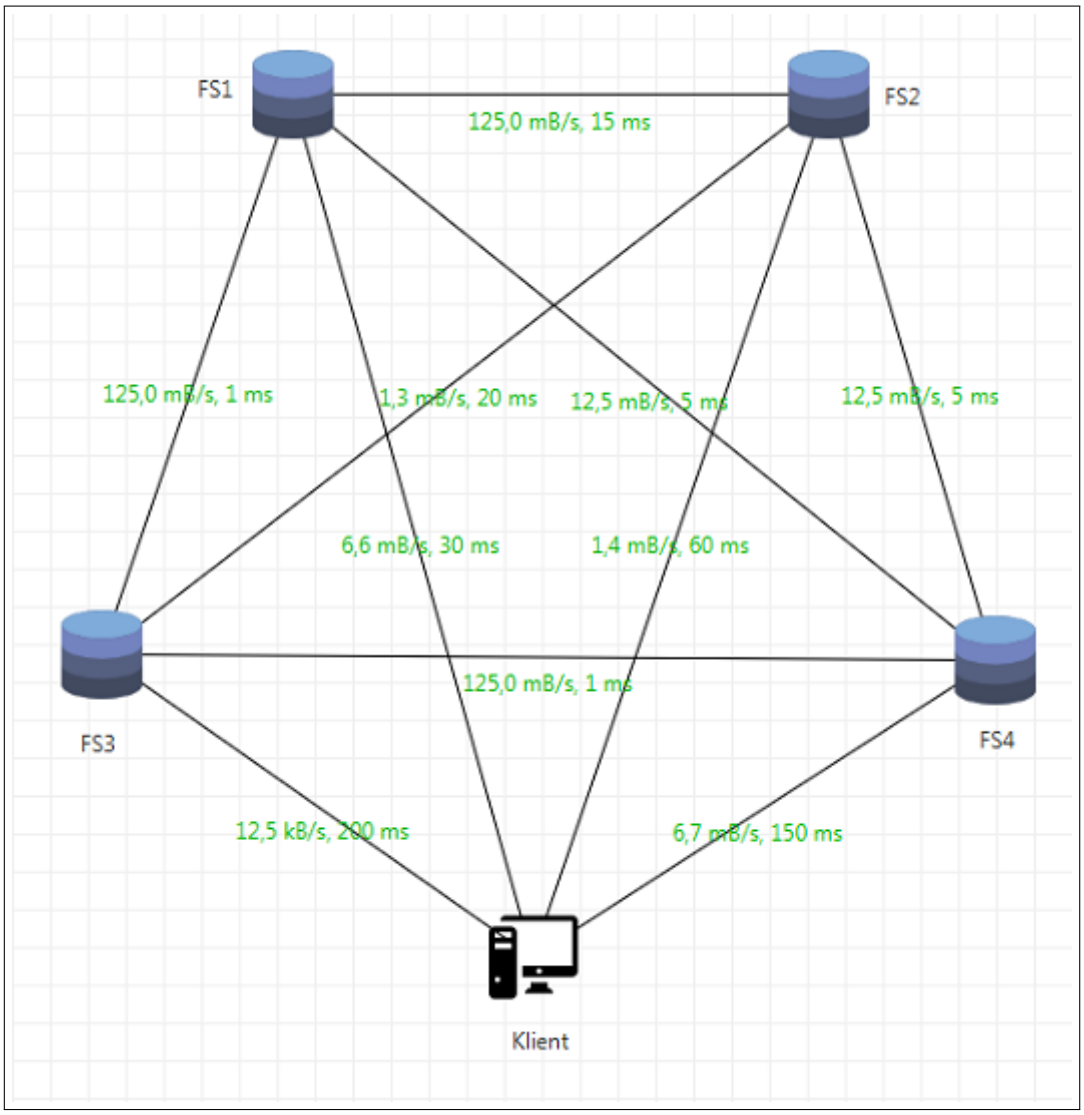
V balíku *model.simulation* je testována funkčnost simulace.

DfsTimeSliceSimulatorTest

V této testovací třídě je testován simulátor *DfsTimeSliceSimulator*. Během testu je ověřeno správné stažení souboru, nahrání souboru, ověření úspěšného zapsání souboru na cílový server a simulace přenosu dat přes zatíženou linku konstantním modifikátorem propustnosti.

DfsSimulatorSimulationResultTest

Tato třída testuje třídu *DfsSimulatorSimulationResult*, která reprezentuje výsledky běhu simulace. Na třídě je testováno zejména výpočet souhrnných ukazatelů simulace jako je celkový čas, průměrná rychlost a celkový počet přenesených dat.



Obrázek 7.1: Testovaný model distribuovaného souborového systému

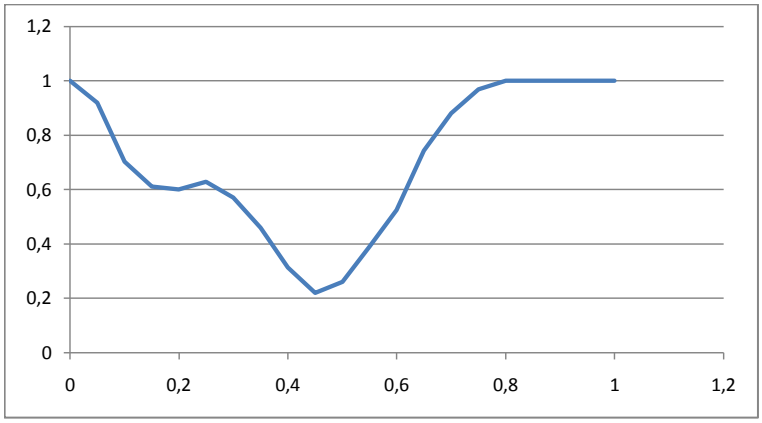
7.2 Model distribuovaného souborového systému

Na obrázku 7.1 lze vidět vybraný model distribuovaného souborového systému, který se skládá ze 4 datových serverů (FS1-FS4) a 1 klienta. Popisek spojení udávající propustnost a latenci je vždy zhruba uprostřed úsečky spojující uzly. Každý uzel modelu je propojen se všemi ostatními, jedná se tedy o úplný graf (viz překryvná síť v kapitole 2.5.3) a vlastnosti hran jsou zaznamenány v tabulce 7.1. Použitý model distribuovaného souborového systému

Tabulka 7.1: Spojení mezi uzly

	FS1	**FS2**	**FS3**	**FS4**	**Klient**
FS1	-	125 mB/s, 15ms	125 mB/s, 1ms	12,5 mB/s, 5ms	6,6 mB/s, 30ms
FS2	125 mB/s, 15ms	-	1,3 mB/s, 20ms	12,5 mB/s, 5ms	1,4 mB/s, 60ms
FS3	125 mB/s, 1ms	1,3 mB/s, 20ms	-	125 mB/s, 1ms	12,5 kB/s, 200 ms
FS4	12,5 mB/s, 5ms	12,5 mB/s, 5ms	125 mB/s, 1ms	-	6,7 mB/s, 150 ms*
Klient	6,6 mB/s, 30ms	1,4 mB/s, 60ms	12,5 kB/s, 200ms	6,7 mB/s, 150 ms*	-

* Spojení je zatíženo podle charakteristiky v 7.2.2.



Obrázek 7.2: Zatížení spojení FS4 -> klient

lze najít na přiloženém CD v souboru *prostredi/configuration.xml*.

7.2.1 Testovací prostředí

Simulace proběhla na počítači s procesorem *i5 4670k@4.3 GHz*, *16 GB RAM@2133 MHz*, úložištěm *Samsung SSD 840 EVO* a operačním systémem *MS Windows 7*. Jako virtuální stroj javy byl použit *Java HotSpot(TM) 64-Bit Server VM (build 25.111-b14, mixed mode)* s parametry *-Xmx4g -Xms4g*.

7.2.2 Zatížení spojení

Spojení mezi serverem *FS4* a klientem bylo zatíženo periodickou lineární charakteristikou s periodou 60 sekund. Konkrétní hodnoty charakteristiky byly náhodně vygenerovány. Průběh charakteristiky lze vidět na obrázku 7.2 a detailněji popsán v tabulce 7.2 hodnotami diskrétních bodů definujících zátěžovou charakteristiku.

Tabulka 7.2: Hodnoty bodů definujících lineární zátěžovou charakteristiku

Čas	**Modifikátor propustnosti**
0	1
0.05	0,918539325842696
0.1	0,702359551
0.15	0,611797753
0.2	0,60011236
0.25	0,629325843
0.3	0,570898876
0.35	0,45988764
0.4	0,313820225
0.45	0,220337079
0.5	0,261235955

Čas	**Modifikátor propustnosti**
0.55	0,389775281
0.6	0,524157303
0.65	0,743258427
0.7	0,880561798
0.75	0,968202247
0.8	1
0.85	1
0.9	1
0.95	1
1	1

Čas charakteristiky je definován na intervalu $< 0, 1 >$ a pro získání simulačního času z tabulky je hodnota času vynásobena délkou periody, tedy např. čas 0.1 po vynásobení délkou periody $0.1 * 60s$ odpovídá zátěži v simulačním čase $6s$, v další periodě zase $66s$ a takto se opakuje až do konce simulace. Modifikátor propustnosti ovlivňuje propustnost spojení v odpovídajícím čase a výsledná propustnost je výsledkem vynásobení maximální propustností spojení definovaným modifikátorem. Ostatní spoje mají konstantní propustnost rovnou jejich maximální propustnosti.

7.2.3 Simulovaná data a úložiště

V distribuovaném souboru jsou celkem 4 soubory: *f1*, *f2*, *f3* a *f4*. Číslování souborů odpovídá názvům serverů. Jediný soubor *f4* je replikován a to na server *FS1* a *FS4*. Dva servery (*FS1* a *FS4*) mají dvě různá úložiště a lze na nich vyzkoušet hierarchické migrace dat. Parametry úložišť a rozmístění souborů lze vidět v tabulce 7.3

Tabulka 7.3: Rozmístění souborů na jednotlivá úložiště

Server	**Úložiště**	**Soubory**
FS1	STOR10 [100 mB, 1 gB/s]	
	STOR11 [10 TB, 5 mB/s]	f1 [15 mB], f4 [450 mB]
FS2	STOR20 [250 gB, 500 mB/s]	f2 [100 mB]
FS3	STOR30 [5 TB, 5 mB/s]	f3 [4 mB]
FS4	STOR40 [5 TB, 5 mB/s]	f4 [450 mB]
	STOR41 [1 TB, 200 mB/s]	

7.3 Testované modely

Pro otestování simulátoru byly vybrány následující modely distribuovaného souborového systému a jejich strategie hledání cesty mezi klientem a serverem.

V tabulce 7.4 jsou vyjmenované testované metody a jejich vlastnosti. Význam vlastností je:

- **Nejkr.** Metoda hledá nejkratší cestu (nejméně skoků), používána většinou existujících distribuovaných souborových systémů (viz 2.5.3).
- **Max. prop.** Metoda hledá cestu s maximální propustností linek.
- **Min. lat.** Metoda hledá cestu s nejmenší latencí.
- **R. disk** Metoda hledá cestu s nejrychlejším diskem.
- **D. rout.** Metoda využívá dynamického routování - metoda výběru cesty v KIVFS (viz 2.5.3).
- **Hie.** Metoda využívá hierarchické migrace souborů na základě čítačů přístupů a k přesunu souborů používá strategii LFU pro uvolnění místa (5.3.10).
- **H. adv.** Jako *Hie.*, ovšem při výběru cesty bere za rychlost disku nejvyšší možnou vrstvu, na kterou by šel soubor z požadavku umístit. Experimentální metoda vycházející z výzkumu Ing. Pešičky.

Tabulka 7.4: Testované metody

Metoda	Nejkr.	Max. prop.	Min. lat.	R. disk	D. rout.	Hie.	H. adv.
Shortest	✓						
Link BW		✓					
Link BW (min. lat)		✓	✓				
Max. throughput		✓		✓			
Min. tx time		✓	✓	✓			
Min. tx time (dyn.)		✓	✓	✓	✓		
Hierarchical		✓	✓	✓	✓	✓	
Hierarchical (adv.)		✓	✓	✓	✓	✓	✓

Tabulka 7.5: Výsledky měření času běhu a rychlosti

Metoda	Čas přenosu	Průměrná rychlost
Hierarchical (advanc.)	26h 14m 9.899s	6.024 mB/s
Hierarchical	28h 37m 55.316s	5.088 mB/s
Min. transf. time (dyn.)	36h 36m 46.061s	5.054 mB/s
Max. throughput	43h 49m 37.367s	4.865 mB/s
Min. transfer time	32h 29m 17.836s	4.865 mB/s
Link BW (min. lat)	39h 46m 57.808s	3.972 mB/s
Link BW	40h 2m 14.388s	3.947 mB/s
Shortest	134h 39m 8s	1.173 mB/s

7.4 Plán simulace

Plán simulace je tvořen 1000 požadavky v náhodném pořadí pro každý soubor *f1*, *f2*, *f3* a *f4*. Celkem se tedy jedná o 4000 požadavků s celkovým přenosem 569 gB dat. Testovaný plán simulace je dostupný v příloze této práce v souboru *prostredi/simulation-plan.xml*.

7.5 Výsledky

V tabulce 7.5 jsou uvedeny výsledky simulace pro testované metody. Celkový čas simulace pro všech 8 testovaných metod byl *47.53s* (samotná simulace bez vykreslení výsledků) a ve špičce bylo využito *1.54 GB RAM* (měřeno profilerem *Netbeans profiler* [3]).

Nejpomalejší metodou testu byla metoda *Shortest*, která hledá pouze nejkratší cestu. Nejrychlejší metodou byla *Hierarchical (advanc.)*, která hledá cestu s nejvyšší maximální propustností, využívá dynamického routování a hierarchického modelu uložení dat.

V následující části jsou uvedeny tabulky s cestou pro jednotlivé soubory a zdůvodnění výsledků testovaných metod. V tabulce cest je uveden soubor a cesta přes všechny uzly, u které je zobrazena maximální přenosová rychlost (zohledněna maximální přenosová rychlost disku) a latence.

7.5.1 Shortest

Tabulka 7.6: Cesty metody Shortest

Soubor	**Cesta**
f1	Klient -> FS1 [5 mB/s, 30 ms]
f2	Klient -> FS2 [1,4 mB/s, 60 ms]
f3	Klient -> FS3 [12,5 kB/s, 200 ms]
f4	Klient -> FS1 [5 mB/s, 150 ms]

U metody *Shortest* bylo největší zpomalení způsobeno výběrem cesty k serveru *FS3*, který má soubor *f3* (viz tabulka 7.6). Klient je propojen přímo se serverem *FS3* s propustností 12,5 kB/s.

7.5.2 Link BW

Tabulka 7.7: Cesty metody Link BW

Soubor	**Cesta**
f1	Klient -> FS4 -> FS1 [5 mB/s, 155 ms]
f2	Klient -> FS4 -> FS2 [6,7 mB/s, 155 ms]
f3	Klient -> FS4 -> FS3 [5 mB/s, 151 ms]
f4	Klient -> FS4 -> FS1 [5 mB/s, 155 ms]

Metoda *Link BW* již byla schopná obejít pomalou linku *Klient -> FS3* a tak je její výsledná průměrná přenosová rychlost téměř 4x vyšší, než je tomu

u metody *Shortest*. I přes zrychlení oproti předchozí metodě je na předposledním místě. Jak je patrné z tabulky cest 7.7, hlavním důvodem zpomalení je výběr cesty *Klient -> FS4*, jejíž propustnost je ovlivněna zátěžovou charakteristikou.

7.5.3 Link BW (min. lat.)

Tabulka 7.8: Cesty metody Link BW (min. lat.)

Soubor	**Cesta**
f1	Klient -> FS1 [5 mB/s, 30 ms]
f2	Klient -> FS4 -> FS2 [6,7 mB/s, 155 ms]
f3	Klient -> FS1 -> FS3 [5 mB/s, 31 ms]
f4	Klient -> FS4 [5 mB/s, 150 ms]

Metoda *Link BW (min. lat.)* se oproti předchozí liší zohledněním výsledné latence cesty. Hlavním rozdílem oproti *Link BW* je výběr cesty pro soubor *f1* a *f4*, kde obě zvolené cesty měly menší nižší latenci.

7.5.4 Min. transfer time a Max. throughput

Tabulka 7.9: Cesty metod Min. transfer time a Max. throughput

Soubor	**Cesta**
f1	Klient -> FS1 [5 mB/s, 30 ms]
f2	Klient -> FS4 -> FS2 [6,7 mB/s, 155 ms]
f3	Klient -> FS1 -> FS3 [5 mB/s, 31 ms]
f4	Klient -> FS1 [5 mB/s, 30 ms]

Obě uvedené metody oproti předchozí zvolily pro soubor *f4* cestu *Klient -> FS1*, která má sice nižší propustnost propoje o 0.1 mB/s, ovšem v kombinaci s nižší latencí a faktem, že propustnost spoje *Klient -> FS1* je konstantní vyjde lépe, než cesta *Klient -> FS4*.

7.5.5 Min. transfer time (dynamic)

Tabulka 7.10: Cesty metody Min. transfer time (dynamic)

Soubor	**Cesta**
f1	Klient -> FS1 [5 mB/s, 30 ms]
f2	Klient -> FS4 -> FS2 [6,7 mB/s, 155 ms]
	Klient -> FS1 -> FS3 -> FS4 -> FS2 [6,6 mB/s, 37 ms]
f3	Klient -> FS1 -> FS3 [5 mB/s, 31 ms]
f4	Klient -> FS1 [5 mB/s, 30 ms]

Tato metoda byla díky dynamickému routování schopna obejít zatíženou linku *Klient -> FS4*. V případě poklesu propustnosti linky byla cesta po přepočtu změněna na linku *Klient -> FS1 -> FS3 -> FS4 -> FS2*, jejíž propustnost je o 0.1 mB/s nižší, ale konstantní.

7.5.6 Hierarchical

Tabulka 7.11: Cesty metody Hierarchical

Soubor	**Cesta**
f1	Klient -> FS1 [6.6 mB/s, 30 ms]
f2	Klient -> FS4 -> FS2 [6,7 mB/s, 155 ms]
	Klient -> FS1 -> FS3 -> FS4 -> FS2 [6,6 mB/s, 37 ms]
f3	Klient -> FS1 -> FS3 [5 mB/s, 31 ms]
f4	Klient -> FS1 [5 mB/s, 30 ms]

Metoda *Hierarchical* byla schopna zvětšit maximální přenosovou rychlost cesty *Klient -> FS1* z 5 mB/s na 6.6 mB/s migrací souboru *f1* na rychlejší úložiště *STOR11*. Během migrace je sice využitelná rychlost původního disku *STOR10* poloviční, tedy 2,5 mB/s, ale po dokončení migrace je již další požadavek vyřízen z rychlejšího disku *STOR11*.

7.5.7 Hierarchical (adv.)

Tabulka 7.12: Cesty metody Hierarchical (adv.)

Soubor	**Cesta**
f1	Klient -> FS1 [5 mB/s, 30 ms]
f2	Klient -> FS4 -> FS2 [6,7 mB/s, 155 ms]
	Klient -> FS1 -> FS3 -> FS4 -> FS2 [6,6 mB/s, 37 ms]
f3	Klient -> FS1 -> FS3 [5 mB/s, 31 ms]
f4	Klient -> FS1 [5 mB/s, 30 ms]
	Klient -> FS4 [6.6 mB/s, 32 ms]

Metoda *Hierarchical (adv.)*, oproti předchozí metodě *Hierarchical*, zohledňuje v maximální propustnosti cesty i nejrychlejší dostupné úložiště, na které lze soubor přesunout. Metoda při hledání cesty pro přenos souboru *f4* vypočítala, že při migraci na rychlejší úložiště *STOR41* serveru *FS4* se maximální přenosová rychlost cesty *Klient -> FS4* zvedne z 5 mB/s na 6.7 mB/s (stále je ovšem ovlivněna zátěžovou charakteristikou).

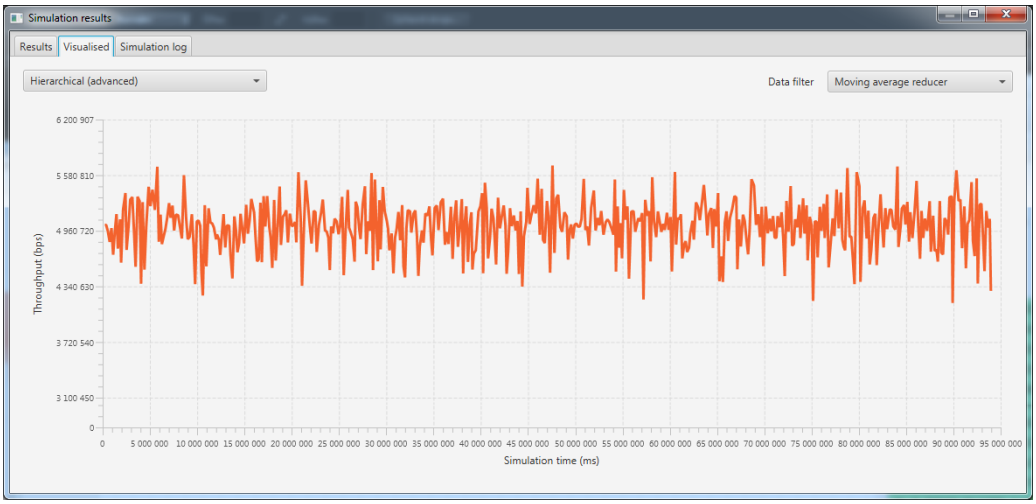
7.6 Shrnutí testů

V kapitole 7.1 byly zmíněny základní unit testy sloužící k ověření hlavní funkcionality simulátoru. Simulátor prošel všemi navrženými testy úspěšně.

Dále bylo testováno celkem 8 modelů distribuovaného souborového systému, na kterých byla ověřena funkčnost simulátoru. Výsledky testovaných metod byly detailněji zdůvodněny v odpovídajících podkapitolách. Navrhovaný distribuovaný souborový systém s hierarchickým modelem uložení dat vyšel pro tento konkrétní simulační plán nejlépe. Propustnost se při použití hierarchického modelu dat při testování zvedla téměř o 20% oproti hodnotě propustnosti identické metody *Min. transfer time (dynamic)* bez hierarchického modelu.

Naměřené hodnoty lze zároveň zkontrolovat pomocí detailního logu, který je dostupný po doběhnutí simulace pro každou měřenou metodu. Porovnat výsledky lze také pomocí grafů zobrazujících průběh rychlosti přenosu dat v

závislosti na čase pro všechny měřené metody. Naměřené výsledky lze exportovat do formátu CSV, který lze načíst ve většině tabulkových procesorech jako je např. MS Excel nebo Libreoffice Calc.



Obrázek 7.3: Graf průběhu rychlosti pro metodu Hierarchical (adv.)

method	time(ms)	avg.speed(B/s)	downloaded(B)
Hierarchical (advanced)	94010108	6052000	569000000000
Hierarchical	111646052	5096000	569000000000
Min. transfer time (dynamic)	112434211	5060000	569000000000
Max. throughput	116221391	4895000	569000000000
Min. transfer time	116221391	4895000	569000000000
Link BW (min. latency)	141125686	4031000	569000000000
Link BW	141809487	4012000	569000000000
Shortest	484748000	1173000	569000000000

Obrázek 7.4: Zobrazení CSV výsledků

Na obrázku 7.3 lze vidět ukázku grafu zobrazující rychlost přenosu dat s aplikovaným filtrem *MovingAverageReducer* (viz 6.7.2). Zobrazené exportované výsledky z formátu CSV jsou poté vidět na obrázku 7.4.

8 Závěr

Cílem této práce bylo prozkoumat principy fungování distribuovaného souborového systému, navrhnout možnosti využití hierarchického modelu uložení dat a následně implementovat simulátor, který umožní simulaci libovolných modelů distribuovaných souborových systémů. Na závěr měly být pomocí implementovaného simulátoru provedeny měření propustnosti základních modelů distribuovaného souborového systému bez hierarchického modelu uložení dat a s hierarchickým modelem uložení dat.

V praktické části byl navržen a úspěšně implementován simulátor, který umožňuje modelování, měření a porovnávání modelů distribuovaných souborových systémů. Součástí simulátoru je grafické prostředí umožňující modelování systému pomocí drag and drop operací a možnost importovat a exportovat prostředí do souboru. Simulátor podporuje simulaci zatížení linky podle definované charakteristiky propustnosti, která může odpovídat měřením propustnosti provedeným nad reálným systémem. Bylo implementováno celkem 8 modelů simulace, které byly následně otestovány pomocí společného scénáře a naměřené výsledky byly odůvodněny a zhodnoceny v kapitole 7. Součástí měření bylo měření vlivu navržené možnosti využití hierarchického modelu uložení dat na propustnost systému, u kterého se na testovaném scénáři výsledná propustnost zvedla téměř o 20% oproti identické metodě bez použití hierarchického modelu.

Architektura simulátoru spolu s UML diagramy tříd byla popsána v implementační části této práce. Zdrojové kódy jsou detailně okomentovány pomocí javadoc komentářů a doplněny unit testy, které byly průběžně během vývoje používány k ověření správné funkcionality programu. Díky poskytnuté dokumentaci a modulárnímu návrhu lze simulátor v budoucnu rozšířit o další funkcionalitu. Vytvořený simulátor lze provozovat na všech třech hlavních operačních systémech MS Windows, Linux a Mac OS.

Během testování a měření výkonnosti modelů jsem zjistil, že by dále bylo vhodné simulátor rozšířit například o nástroje umožňující automatické generování modelů zapojení distribuovaného souborového systému, generování

scénářů simulace z logů reálných systémů nebo přízpůsobit grafické rozhraní pro rozsáhlé systémy s velkým počtem uzlů.

A Použité zkratky

- **SSD** Solid State Drive
- **HDD** Hard Disk drive
- **JBOD** Just a Buch Of Disks
- **RAM** Random Access Memory
- **RW** Read Write
- **RO** Read Only
- **RPC** Remote Procedure Call
- **MPI** Message Passing Interface
- **LRU** Least Recently Used
- **LFU** Least Frequently Used
- **FIFO** First In, First Out
- **I/O** Input/Output
- **XML** eXtensible Markup Language
- **UML** Unified Modeling Language
- **ADSL** Asymmetric Digital Subscriber Line
- **ICMP** Internet Control Message Protocol
- **GUI** Graphic User Interface
- **VFS** Virtual File System
- **CSS** Cascading Style Sheets
- **API** Application Programming Interface
- **CSV** Comma Separated Values

B Přiložené CD

Součástí práce je přiložené CD s následujícím obsahem:

- **dist** spustitelný JAR soubor
- **doc** kopie této práce
  - **javadoc** javadoc dokumentace vygenerovaná ze zdrojových kódů
  - **dpsrc** zdrojové kódy práce v TEX
- **Poster** poster k této práci
- **prostredi** testované prostředí DFS
- **src** zdrojové kódy
  - **main** zdrojové kódy simulátoru
  - **test** zdrojové kódy JUnit testů

C Uživatelská příručka

Tato část slouží jako návod pro uživatele při práci se simulátorem.

C.1 Přeložení a spuštění

Pro běh simulátoru je vyžadováno *JRE* verze alespoň *1.8*[1] a přítomnost knihovny *JavaFX* verze *8.0.40* a vyšší na classpath. *JavaFX* je od verze *7u6* součástí *Javy* od *Oracle*.

C.1.1 Přeložení

Pro přeložení simulátoru je nutné mít *JDK* verze alespoň *1.8*[2] a nainstalovaný *Apache Ant*[3].

Pro přeložení projektu se musíme nejdřív přepnout do složky se zdrojovými soubory (na přiloženém CD složka *src*). Samotné přeložení poté spustíme příkazem

```
ant dist,
```

který vytvoří ve složce *dist/lib* JAR soubor *dfs_simulator.jar*.

C.1.2 Spuštění JUnit testů

JUnit testy ve složce *src/test* lze spustit ze složky *src* příkazem

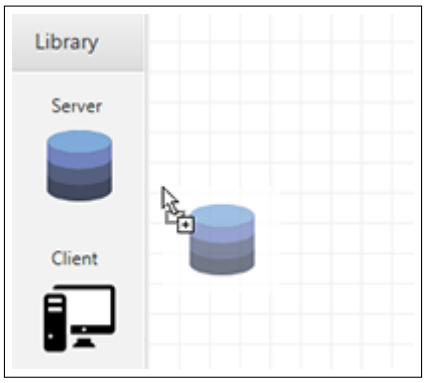
```
ant test,
```

který na obrazovce vypíše výstup všech testů.

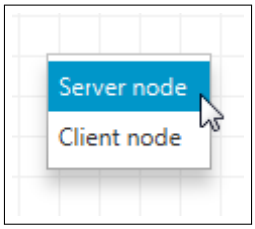
[1] `http://www.oracle.com/technetwork/java/javase/downloads/jre8-downloads-2133155.html`

[2] `http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html`

[3] `https://ant.apache.org/bindownload.cgi`



Obrázek C.1: Vytvoření uzlu tažením myší



Obrázek C.2: Vytvoření uzlu přes kontextové menu

C.1.3 Spuštění

Pro spuštění projektu slouží dodaný nebo námi vytvořený JAR soubor. Výchozím umístěním JAR souboru je *src/dist/lib/dfs_simulator.jar* a ze složky *src* ho lze spustit příkazem

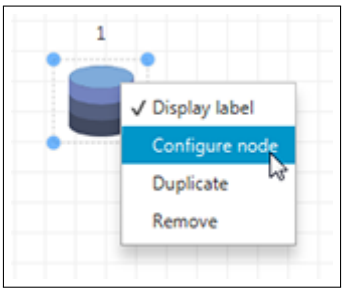
```
java -jar dist/lib/dfs_simulator.jar.
```

C.2 Vytvoření uzlu

Vytvoření uzlu typu klient nebo server probíhá tažením myši odpovídajícího prvku z panelu *Library* na modelovací plátno, viz obr. C.1. Nově vytvořený uzel je vytvořen na místě, na které byl tažen myší. Alternativně lze vytvořit uzel pomocí pravého tlačítka myši přes kontextové menu, viz obr. C.2.

C.2.1 Konfigurace uzlu

Vyvolání dialogu pro konfiguraci uzlu lze provést stiskem pravého tlačítka myši nad daným uzlem a vybráním možnosti „Configure node“ (obr. C.3)



Obrázek C.3: Konfigurace uzlu

nebo dvojklikem myší na uzel.

Seznam sousedů

V konfiguračním dialogu lze v záložce „Node connections“ zobrazit tabulku se seznamem všech sousedů. Spojení se sousedem lze upravit pomocí kontextového menu možností „Alter link“ nebo smazat možností „Delete link“.

C.2.2 Popisek uzlu

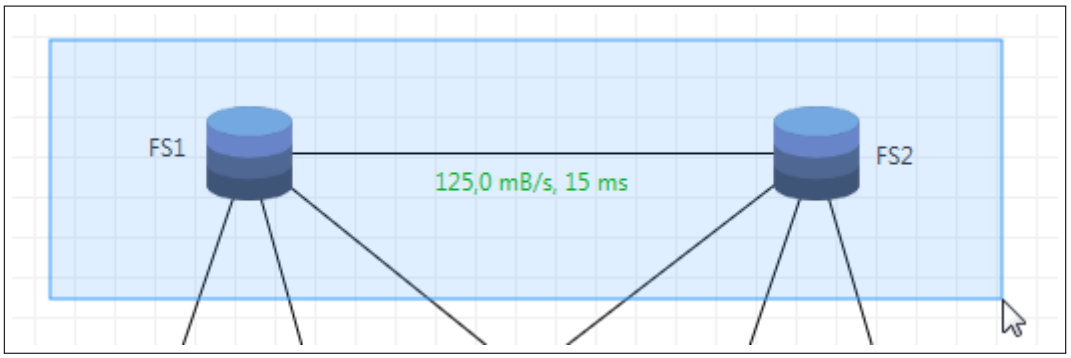
Všechny uzly na modelovacím plátně mohou mít zobrazitelný popisek (lze vidět na obr. C.3), který zároveň představuje unikátní identifikátor uzlu. Skrýt či zobrazit popisek lze přes kontextové menu uzlu vyvolané pravým tlačítkem a vybráním volby „Display label“. Popisek je ve výchozím stavu vždy zobrazen.

C.2.3 Přemístění uzlu na plátně

Všechny uzly, včetně popisků uzlů, lze na plátně libovolně přemístit tažením levým tlačítkem myši. Je-li tažený prvek přemístěn mimo viditelné hranice, je modelovací plátno automaticky rozšířeno.

Pro výběr více prvků najednou lze použít hromadného výběru, který lze vytvořit tažením levého tlačítka myši a obsažením požadovaných prvků ve výběrovém obdélníku, viz obr. C.4.

Pro přidání dalšího prvku do aktuálního výběru lze použít klávesu *Shift* a provést výběr nad požadovanými prvky. Pro přidání prvku ležícího mimo



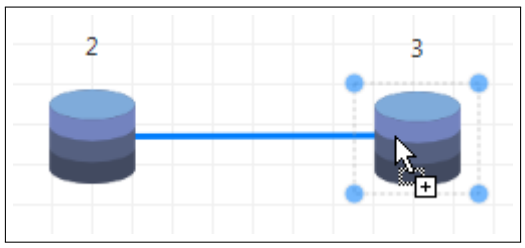
Obrázek C.4: Hromadný výběr uzlů

Obrázek C.5: Výchozí body pro vytvoření spojení

aktuální výběr nebo odebrání prvku z aktuálního výběru lze použít klávesu *Control* a provést výběr nad daným prvkem či prvky.

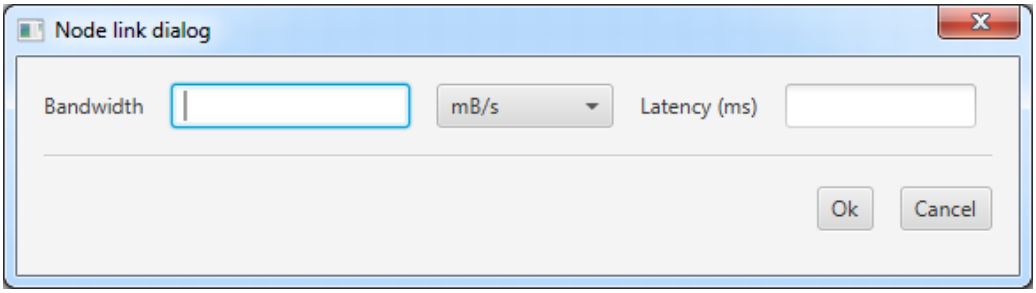
C.3 Vytvoření spojení

Spojení mezi uzly lze vytvořit tažením myší z jednoho ze čtyř modrých kruhů zobrazených při najetí myší nad uzel, viz obr. C.5. Pro úspěšné spojení musí být spojení přetaženo nad jiný uzel, který kolem sebe zobrazí ohraničení (obr. C.6), do kterého musí být spojení přetaženo, aby bylo úspěšně vytvořeno.



Obrázek C.6: Zobrazené ohraničení pro akceptování spojení

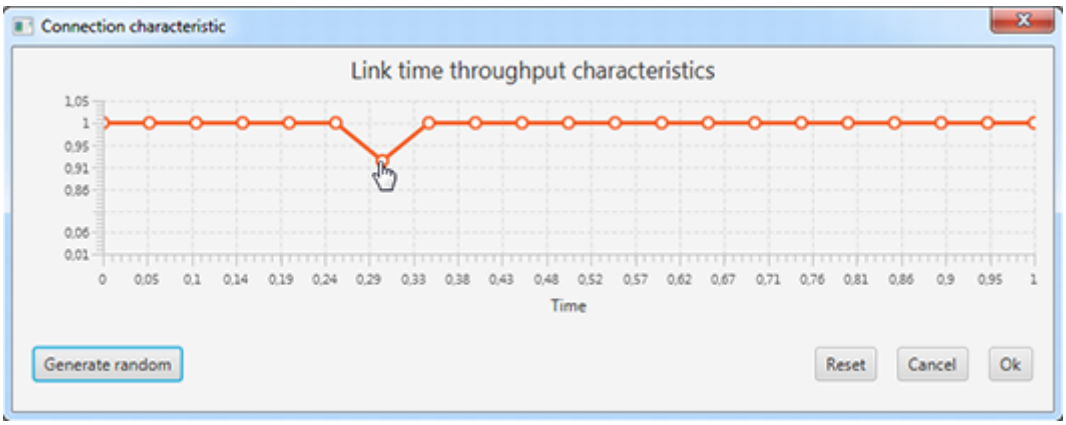
C.3.1 Konfigurace spojení



Obrázek C.7: Dialog konfigurace spojení

Po úspěšném vytvoření spojení mezi uzly je zobrazen dialog (obr. C.7), pomocí kterého určíme propustnost spoje a jeho latenci. Upravit vlastnosti spojení lze vyvoláním stejného dialogu pomocí volby „Alter link“ z kontextového menu spojení.

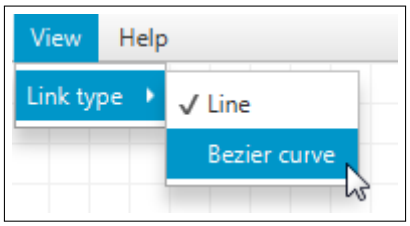
C.3.2 Zátěžová charakteristika



Obrázek C.8: Dialog zátěžové charakteristiky

Pro nastavení zátěžové charakteristiky vybraného spojení je třeba vyvolat dialog pomocí volby „Link characteristics“ z kontextového menu spojení. Dialog zátěžové charakteristiky lze vidět na obrázku C.8.

Charakteristika je definována celkem 21 body a perioda charakteristiky je 60 sekund. Pro upravení propustnosti ve vybraném časovém okamžiku lze



Obrázek C.9: Typ grafického zobrazení spojení

bod vertikálně táhnout myší, čímž je upravena hodnota propustnosti spoje. Pomocí tlačítka *Generate random* lze vygenerovat náhodné zatížení.

C.3.3 Grafická reprezentace spojení

Spojení mezi uzly lze graficky zobrazit dvěma způsoby a to buď přímkou nebo Bézierovo křivkou. Podobu zobrazeného spojení lze vybrat z hlavního menu aplikace, viz obr. C.9.

C.4 Úložiště a adresářová struktura

V konfiguračním dialogu serveru lze spravovat úložiště na daném serveru a jeho adresářovou strukturu.

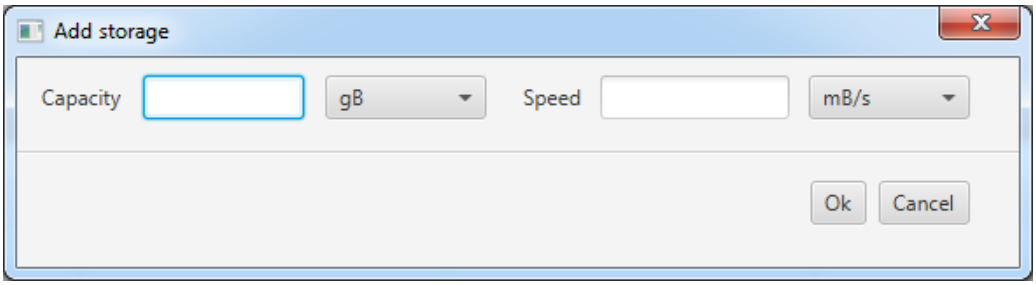
C.4.1 Seznam úložišť

V záložce „Storage configuration“ lze přidávat nová úložiště a upravovat existující. Tlačítkem „Add storage“ lze přidat nové úložiště. Úložiště jsou zobrazeny v tabulce a vyvoláním kontextového menu pravým tlačítkem lze existující úložiště volbou „Edit storage“ upravit nebo volbou „Delete“ smazat.

Vytvoření úložiště nebo úprava existujícího probíhá pomocí dialogu na obr. C.10, který umožňuje nastavení rychlosti úložiště a jeho kapacity.

C.4.2 Adresářová struktura

Záložka „File structure“ v konfiguračním dialogu slouží pro správu adresářové struktury serveru.Výchozí složkou je kořenová složka /, kterou nelze



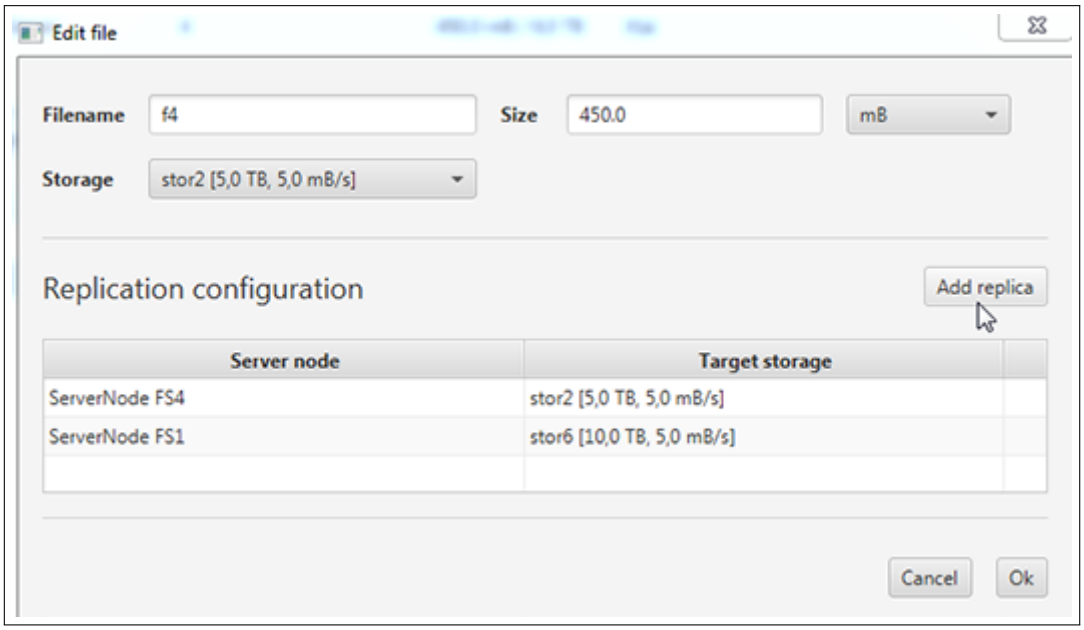
Obrázek C.10: Dialog pro vytvoření nebo úpravu úložiště

smazat ani upravovat. Přidání souboru či složky do kořenové složky probíhá přes kontextové menu volbou „New file“ a „New subdirectory“ pro přidání souboru, respektive složky. Upravit existující soubor či složku lze pomocí možnosti „Edit“, která vyvolá odpovídající dialog.

Umístění objektu na úložiště

Libovolný objekt může být umístěn na libovolném úložišti. Výběr úložiště pro objekt lze provést nabídkou „Select mount device“ z kontextového menu objektu, která vyvolá dialog pro výběr úložiště. Pokud potomci nemají explicitně stanovené úložiště, na kterém jsou umístěni, potom mohou úložiště zdědit od prvního předka, který má vybrané úložiště.

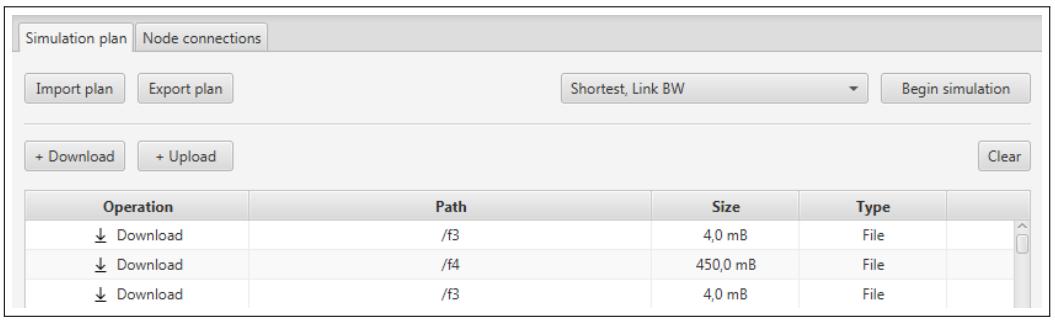
Replikace souborů



Obrázek C.11: Dialog pro vytváření a úpravu souborů

V dialogu pro vytváření souborů (obr. C.11 lze zároveň definovat, jak bude soubor replikován. Přidání nové repliky probíhá tlačítkem „**Add replica**“, které vyvolá dialog pro výběr cílového serveru a cílového úložiště, na které má být soubor replikován. V seznamu replik lze repliku odstranit pomocí kontextového menu a volby „Delete replica“.

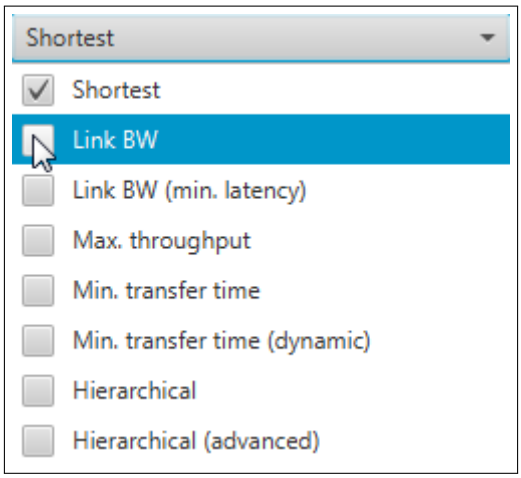
C.5 Plánování a spuštění simulace



Obrázek C.12: Simulační plán

Plánování a spuštění simulace probíhá v záložce „Simulation plan“ (obr. C.12 z konfiguračního dialogu konkrétního klienta. Tlačítkem „+ **Download**“ je vyvolán dialog, pomocí kterého můžeme vybrat 1 či více souborů, které mají být staženy. Tlačítkem „+ **Upload**“ je vyvolán dialog, ve kterém můžeme definovat název souboru, velikost a umístění, do kterého má být nahrán.

C.5.1 Výběr typu simulace



Obrázek C.13: Výběr typu simulace

Pomocí seznamu na obr. C.13 lze vybrat 1 či více typů simulace, které mají být provedeny nad vytvořeným plánem.

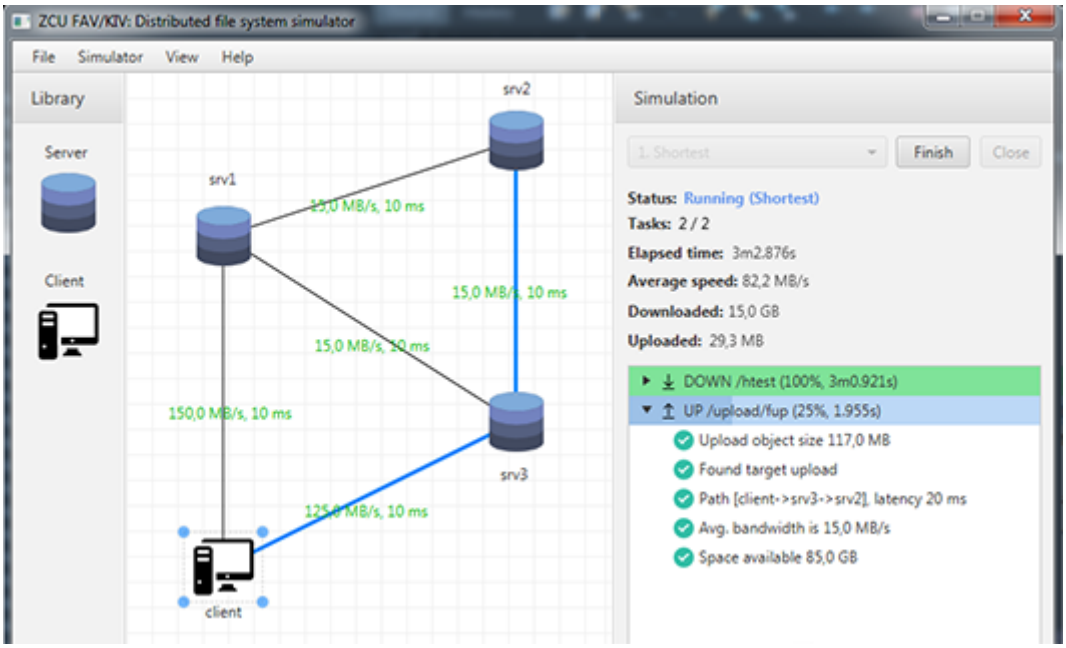
C.5.2 Spuštění simulace

Tlačítkem „**Begin simulation**“ spustíme simulaci vybraných typů podle společného plánu. Po spuštění simulace je zobrazen dialog, který informuje uživatele o průběhu simulace.

C.5.3 Vizualizace simulace

Vizualizace průběhu simulace je viditelná na obr. C.14. V postranním panelu na pravé straně lze vidět několik položek:

- **Status** - aktuální status průběhu simulace,



Obrázek C.14: Vizualizace simulace

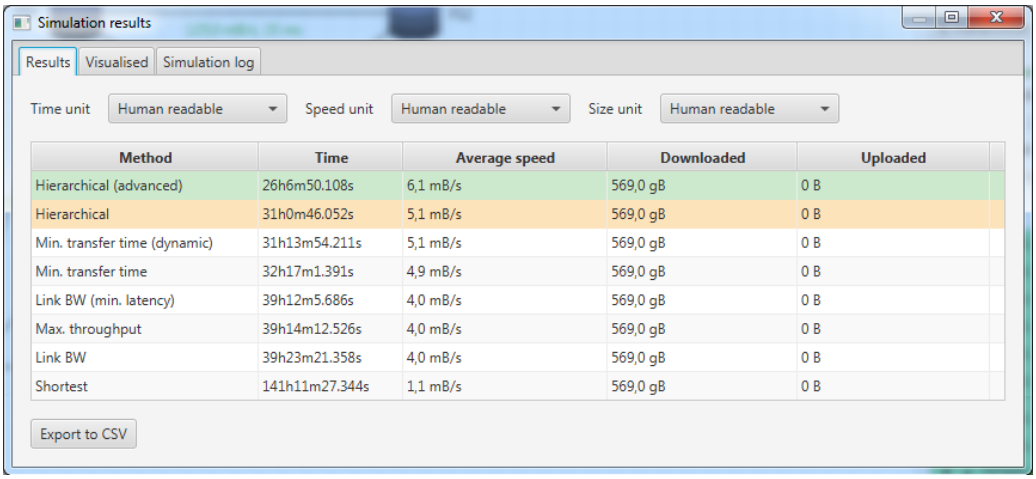
- **Tasks** - počet zpracovaných / počet celkových položek simulačního plánu,
- **Elapsed time** - čas, který byl potřebný k přenesení zpracovaných položek,
- **Average speed** - průměrná rychlost přenosu zpracovaných položek,
- **Downloaded** - množství stažených dat zpracovanými položkami,
- **Uploaded** - množství nahraných dat zpracovanými položkami.

U každé aktuálně zpracovávané položky je zároveň zvýrazněna cesta (na obr. C.14 modrou barvou), přes kterou jsou přenášena data.

Pokud si nepřejeme čekat na výsledky do doběhnutí vizualizace, lze použít tlačítko „**Finish**“, které urychlí vizualizaci a zobrazí výsledky.

C.6 Výsledky simulace

Po dokončení vizualizace simulace je uživateli zobrazeno dialogové okno s výsledky simulace.



Obrázek C.15: Výsledky simulace

C.6.1 Export do CSV

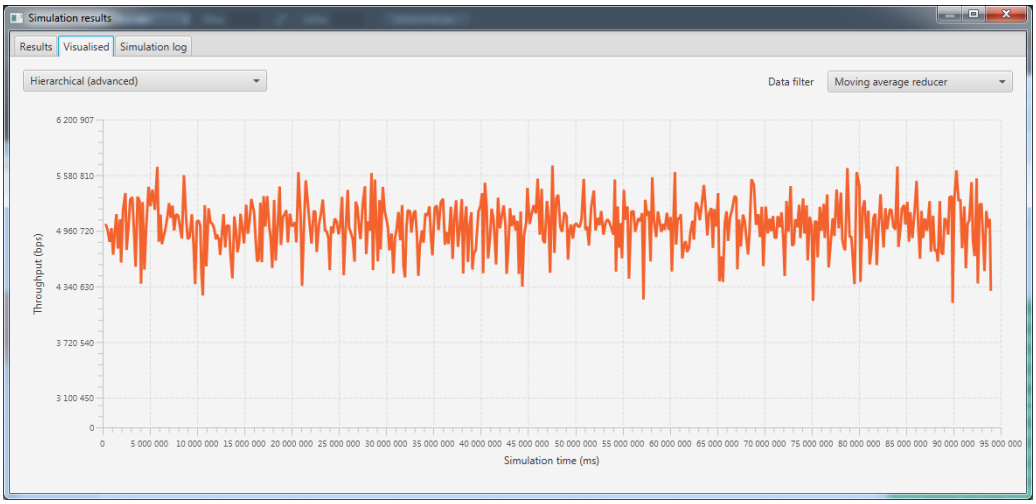
Naměřené výsledky lze prohlížet v tabulce v záložce „Results“ (obr. C.15). Tabulka má celkem 5 sloupců a to:

- **Method** - název testované metody,
- **Time** - celkový čas přenosu dat,
- **Average speed** - průměrná rychlost,
- **Downloaded** - objem stažených dat,
- **Uploaded** - objem nahraných dat.

Řádky tabulky jsou seřazeny od nejlepšího času po nejhorší a dva nejlepší výsledky jsou zvýrazněny zelenou respektive oranžovou barvou.

Násobky naměřených hodnot lze vybrat pomocí odpovídajících select boxů - „Time unit“ pro násobky času, „Speed unit“ pro násobky rychlosti a „Size unit“ pro násobky velikosti. Výchozí hodnotou všech 3 je „Human readable“, která převádí hodnoty na snadno čitelný formát, ovšem ne vždy úplně přesný (zaokrouhlování rychlosti a velikosti).

Tlačítkem „**Export to CSV**“ lze tabulku exportovat do vybraného souboru. Hodnoty budou exportovány ve stejném formátu, v jakém jsou aktuálně zobrazeny v tabulce (tzn. s vybranými násobky).



Obrázek C.16: Graf průběhu rychlosti

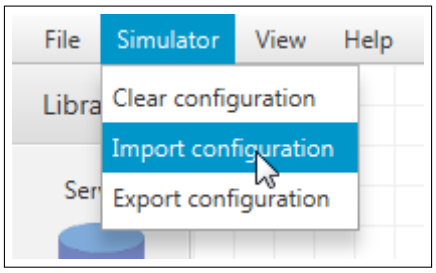
C.6.2 Graf průběhu rychlosti

V záložce „Visualised“ lze vidět spojnicový graf (obr. C.16), zobrazující průběh rychlosti přenosu v závislosti na uběhnutém času. Na ose X je čas v milisekundách a na ose Y je rychlost přenosu v B/s.

Filtr vzorků

Naměřené vzorky rychlosti jsou ve výchozím stavu dodatečně filtrovány, aby jejich vykreslení nebylo příliš náročné. Filtr lze zvolit select boxem s popiskem „Data filter“ a jsou na výběr následující filtry:

- **No filter (CPU Intensive)** vrací všechny vzorky, náročný, přesný ale ne příliš vhodný pokud máme hodně vzorků,
- **Douglas-Peucker** využívá Douglas-Peucker algoritmu [49],
- **Radial distance** funguje na bázi eukleidovské vzdálenosti bodů,
- **Moving average** výchozí filtr - dělí hodnoty do časových intervalů a pro každý časový interval počítá průměr hodnot. Filtrace probíhá rychle i pro větší množství vzorků.



Obrázek C.17: Import a export prostředí

C.6.3 Detaily simulace

V záložce „Simulation log“ je k zobrazení log průběhu simulace, který zaznamenává některé důležité události při provádění jednotlivých požadavků. Každý záznam má na začátku v hranaté závorce časové razítko.

C.7 Import a export prostředí

Modelovaný systém a simulační plán lze exportovat do XML souboru, z kterého lze později načíst.

C.7.1 Modelované prostředí

Modelované prostředí lze exportovat pomocí nabídky v hlavním menu (obr. C.17) *Simulator - Export configuration*. Následně je zobrazen dialog, pomocí kterého je vybrán cílový soubor. Import probíhá obdobným způsobem nabídkou *Simulator - Import configuration*, která vyzve uživatele k vybrání souboru s exportovaným stavem.

C.7.2 Simulační plán

Ve stejném dialogu, ve kterém se vytváří simulační plán (obr. C.12) je zároveň možnost plán uložit do externího souboru nebo ho z něj načíst. Tlačítkem „**Import plan**“ lze provést importování simulačního plánu z externího souboru a tlačítkem „**Export plan**“ lze uložit stávající plán do souboru.

Seznam obrázků

Seznam tabulek

Literatura

[1] FIPS PUB 186-3 - Digital Signature Standard. FEDERAL INFORMATION PROCESSING STANDARDS PUBLICATION, 2009.

[2] *HPSS On-line Documentation - HPSS Systems* [online]. 2017. [cit. 2017/04/29]. Dostupné z: `http://www.hpss-collaboration.org/online_doc.shtml`.

[3] *Netbeans profiler* [online]. 2017. [cit. 2017/04/30]. Dostupné z: `https://profiler.netbeans.org/`.

[4] *Distributed locks with Redis* [online]. [cit. 2017/04/27]. Dostupné z: `https://redis.io/topics/distlock`.

[5] *What is Samba?* [online]. [cit. 2017/04/24]. Dostupné z: `https://www.samba.org/samba/what_is_samba.html`.

[6] *Deploy SMB Multichannel* [online]. Microsoft, 2014. [cit. 2017/04/24]. Dostupné z: `https://technet.microsoft.com/en-us/library/dn610980(v=ws.11).aspx`.

[7] Andersen, D. et al. Resilient Overlay Networks. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, s. 131–145, New York, NY, USA, 2001. ACM. doi: 10.1145/502034.502048. Dostupné z: `http://doi.acm.org/10.1145/502034.502048`. ISBN 1-58113-389-8.

[8] Barreto, J. SMB remote file protocol (including SMB 3.0). Storage Networking Industry Association, 2012.

[9] BELLMAN, R. ON A ROUTING PROBLEM. *Quarterly of Applied Mathematics*. 1958, 16, 1, s. 87–90. ISSN 0033569X, 15524485. Dostupné z: `http://www.jstor.org/stable/43634538`.

[10] Bucy, J. S. et al. The DiskSim Simulation Environment Version 4.0 Reference Manual. 2008.

[11] Buyya, R. – Murshed, M. GridSim: A Toolkit for the Modeling and Simulation of Distributed Resource Management and Scheduling for Grid Computing. *CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE (CCPE)*. 2002, 14, 13, s. 1175–1220.

[12] Cisco. *Enhanced Interior Gateway Routing Protocol* [online]. [cit. 2017/04/27]. Dostupné z: `https://www.cisco.com/c/en/us/support/docs/ip/enhanced-interior-gateway-routing-protocol-eigrp/16406-eigrp-toc.html`.

[13] Cormen, T. H. et al. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009. ISBN 0262033844, 9780262033848.

[14] Daneček, M. Způsoby využití datových úložišť CESNET - Univerzita Karlova v Praze. May 2016.

[15] Dea, C. et al. *JavaFX 8: Introduction by Example*. Apress, 2nd edition, 2014. ISBN 1430264608, 9781430264606.

[16] Diffie, W. – Hellman, M. New Directions in Cryptography. *IEEE Trans. Inf. Theor.* September 2006, 22, 6, s. 644–654. ISSN 0018-9448. doi: 10.1109/TIT.1976.1055638. Dostupné z: `http://dx.doi.org/10.1109/TIT.1976.1055638`.

[17] Fidge, C. J. Timestamps in message-passing systems that preserve the partial ordering. *Proceedings of the 11th Australian Computer Science Conference*. 1988, 10, 1, s. 56–66. Dostupné z: `http://sky.scitech.qut.edu.au/~fidgec/Publications/fidge88a.pdf`.

[18] Floyd, R. W. Algorithm 97: Shortest Path. *Commun. ACM*. June 1962, 5, 6, s. 345–. ISSN 0001-0782. doi: 10.1145/367766.368168. Dostupné z: `http://doi.acm.org/10.1145/367766.368168`.

[19] Ghemawat, S. – Gobioff, H. – Leung, S.-T. The Google file system. *ACM SIGOPS Operating Systems Review*. December 2003, 37, 5, s. 29 – 43. ISSN 1520-9202. doi: 10.1145/1165389.945450. Dostupné z: `https://dl.acm.org/citation.cfm?id=945450`.

[20] Gougeaud, S. et al. A generic and open simulation tool for large multi-tiered hierarchical storage systems. *2016 International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS)*. July 2016. doi: 10.1109/SPECTS.2016.7570515. Dostupné z: `http://ieeexplore.ieee.org/document/7570515/`.

[21] Hamill, P. *Unit Test Frameworks*. O'Reilly, first edition, 2004. ISBN 0596006896.

[22] Hedrick, C. *Routing Information Protocol* [online]. 1988. [cit. 2017/04/28]. Dostupné z: `https://tools.ietf.org/html/rfc1058`.

[23] Housley, R. *Using AES-CCM and AES-GCM Authenticated Encryption in the Cryptographic Message Syntax (CMS)* [online]. Vigil Security, 2007. [cit. 2017/04/24]. Dostupné z: `https://tools.ietf.org/html/rfc5084`.

[24] IETF. *Network File System (NFS) version 4 Protocol - IETF* [online]. IETF, 2017. [cit. 2017/03/18]. Dostupné z: `https://www.ietf.org/rfc/rfc3530.txt`.

[25] Kim, E. SSD Performance – A Primer. Solid State Storage Initiative, 2013.

[26] Kolan, P. – Ramachandran, V. Google File System Simulator. December 2010.

[27] Lamport, L. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM*. July 1978, 21, 7, s. 558–565. ISSN 0001-0782. doi: 10.1145/359545.359563. Dostupné z: `http://doi.acm.org/10.1145/359545.359563`.

[28] Lugar, J. Hierarchical storage management: leveraging new capabilities. *IT Professional*. March 2001, 3, 2, s. 53 – 55. ISSN 1520-9202. doi: 10.1109/6294.918223. Dostupné z: `http://ieeexplore.ieee.org/document/918223/`.

[29] *Documentation / Lustre* [online]. 2017. [cit. 2017/03/23]. Dostupné z: `http://lustre.org/documentation/`.

[30] Matějka, L. – Šafařík, J. – Pešička, L. Distributed File System with Online Multi-master Replicas. *2011 Second Eastern European Regional Conference on the Engineering of Computer Based Systems*. 2011.

[31] Matějka, L. et al. Dynamic Routing in Distributed File System. *Journal of Networks*. October 2014, 9, 10, s. 2591 – 2597. ISSN 1796-2056.

[32] Moy, J. T. *OSPF: Anatomy of an Internet Routing Protocol*. Addison-Wesley Longman Publishing Co., Inc., 1998. ISBN 0201634724.

[33] Muralidhar, S. et al. f4: Facebook's warm BLOB storage system. *11th USENIX Symposium on Operating Systems Design and Implementation*. October 2014.

[34] *OpenAFS Wiki* [online]. OpenAFS, 2017. [cit. 2017/02/24]. Dostupné z: `https://wiki.openafs.org/`.

[35] Oracle. Architectural Overview of the Oracle ZFS Storage Appliance. September 2016.

[36] Paar, C. – Pelzl, J. *The Data Encryption Standard (DES) and Alternatives*, s. 55–86. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010. doi: 10.1007/978-3-642-04101-3_3. Dostupné z: `http://dx.doi.org/10.1007/978-3-642-04101-3_3`. ISBN 978-3-642-04101-3.

[37] *Cisco Packet Tracer* [online]. Cisco, 2017. [cit. 2017/03/24]. Dostupné z: `https://www.cisco.com/c/dam/en_us/training-events/netacad/course_catalog/docs/Cisco_PacketTracer_DS.pdf`.

[38] Rivest, R. L. – Shamir, A. – Adleman, L. A Method for Obtaining Digital Signatures and Public-key Cryptosystems. *Commun. ACM*. February 1978, 21, 2, s. 120–126. ISSN 0001-0782. doi: 10.1145/359340.359342. Dostupné z: `http://doi.acm.org/10.1145/359340.359342`.

[39] San-Lucas, C. – Abad, C. L. Towards a fast multi-tier storage system simulator. *Ecuador Technical Chapters Meeting (ETCM), IEEE*. October 2016. doi: 10.1109/ETCM.2016.7750836. Dostupné z: `http://ieeexplore.ieee.org/document/7750836/`.

[40] Sandberg, R. et al. Design and Implementation of the Sun Network File System. *Proceedings of the Usenix Conference*. June 1985, s. 119–130.

[41] Satyanarayanan, M. et al. Coda: A Highly Available File System for a Distributed Workstation Environment. *IEEE Transactions on Computers*. April 1990, 39, 4. ISSN 0018-9340.

[42] Schneier, B. Description of a New Variable-Length Key, 64-bit Block Cipher (Blowfish). In *Fast Software Encryption, Cambridge Security Workshop*, s. 191–204, London, UK, UK, 1994. Springer-Verlag. Dostupné z: http://dl.acm.org/citation.cfm?id=647930.740558. ISBN 3-540-58108-1.

[43] Sermersheim, J. *Lightweight Directory Access Protocol (LDAP): The Protocol* [online]. 2006. [cit. 2017/04/30]. Dostupné z: https://www.ietf.org/rfc/rfc4511.txt.

[44] Skupa, J. Dynamické směrování v překryvných sítích s využitím predikce: technical report no. DCSE/TR-2016-01 April, 2016. 2016. Dostupné z: http://hdl.handle.net/11025/21534.

[45] Skupa, J. et al. *KivFS: distribuovaný souborový systém*. Západočeská univerzita v Plzni, 2010. ISBN 978-80-7043-903-6.

[46] Stallings, W. The Advanced Encryption Standard. *Cryptologia*. July 2002, 26, 3, s. 165–188. ISSN 0161-1194. doi: 10.1080/0161-110291890876. Dostupné z: http://dx.doi.org/10.1080/0161-110291890876.

[47] Steiner, J. G. – Neuman, C. – Schiller, J. I. Kerberos: An Authentication Service for Open Network Systems. In *IN USENIX CONFERENCE PROCEEDINGS*, s. 191–202, 1988.

[48] Sterling, T. *PVFS: Parallel Virtual File System*. MIT Press, 2000. ISBN 9780262286770.

[49] Visvalingam, M. – Whyatt, J. D. The Douglas-Peucker Algorithm for Line Simplification: Re-evaluation Through Visualization. *Comput. Graph. Forum*. September 1990, 9, 3, s. 213–228. ISSN 0167-7055. doi: 10.1111/j.1467-8659.1990.tb00398.x. Dostupné z: http://dx.doi.org/10.1111/j.1467-8659.1990.tb00398.x.

[50] Weil, S. et al. Ceph: a scalable, high-performance distributed file system. *OSDI '06 Proceedings of the 7th symposium on Operating systems design and implementation*. 2006, s. 307–320.