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

Diplomová práce

KIVFS - Synchronizace a trasování požadavků

Plzeň, 2012 Jindřich Skupa

ZDE VLOŽIT LIST ZADÁNÍ

Z důvodu správného číslování stránek

Prohlášení

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

V Plzni dne 17. května 2012 , Jindřich Skupa

Abstract

Goals of this thesis are description and implementation synchronization and routing mechanism, which will be able to use in KIVFS distributed file system. Thesis contents description of common algorithms of distributed systems and theirs practical application, which is copared with actual version and judge results which ware obtained during her creation. The result of thesis is implementation synchronization layer with support of route searching and delivering between nodes.

OBSAH

1 ÚVOD

Cílem této práce je popsat problematiku synchronizace v distribuovaných systémech a směrování požadavků. Čtenář bude seznámen s definicí, charakteristikou a obecnými principy distribuovaných systémů, jejich dělením a požadavky, které jsou na tyto systémy kladené. Bude rozebrána problematika týkající se globálního stavu, komunikace, času v distribuovaných systémech, synchronizace požadavků a zpracování distribuovaných transakcí. Poznatky načerpané z těchto kapitol distribuovaných systémů budou pak aplikovány při implementaci v KIVFS jako samostatné komponenty systému.

KIVFS je distribuovaný souborový systém (dále také DFS) vyvíjený na katedře informatiky a výpočetní techniky. Projekt si klade za cíl vytvořit novou implementaci distribuovaného souborového systému, vhodného i pro mobilní zařízení. Pro tento segment trhu nejsou v současné době dostupné žádné implementace distribuovaných souborových systémů, nabízí se pouze množství tzv. cloudových úložišť, které jsou svojí povahou DFS blízké.

Cílem práce je primárně synchronizace požadavků, konzistenční modely a transakční zpracování, které je důležité pro zachování konzistentního stavu metadat a dat distribuovaného souborového systému.

V praktické části bude popsána implementace vybraných funkčností na základě teoretických poznatků jako samostatná serverová aplikace. Realizační část popíše fungování nově implementovaných funkcí a ověří jejich správnou funkčnost.

Ve výsledcích pak bude diskutován vliv nově zavedených vlastností na KIVFS.

2 DISTRIBUOVANÝ SYSTÉM

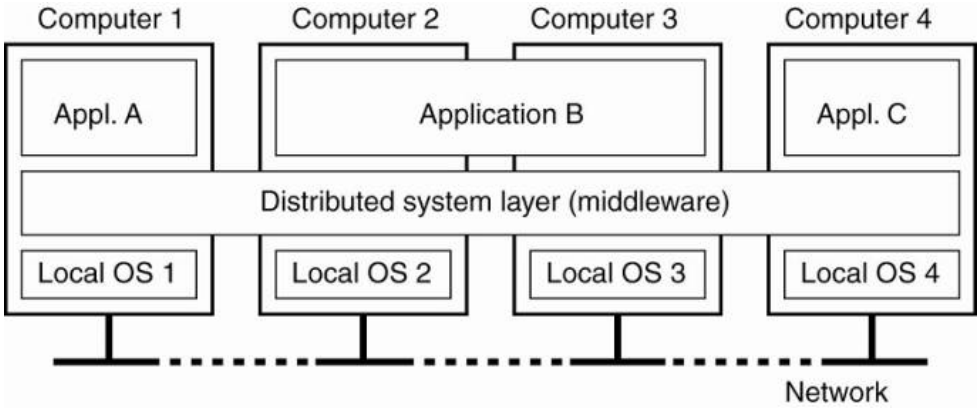
Odborná literatura nabízí několik definicí distribuovaných systémů (dále také DS). Jedna z definic říká: „Distribuovaný systém je kolekce nezávislých entit, která spolupracuje k řešení problémů, které nemohou být vyřešeny samostatně“ [1]. Jinými slovy lze distribuovaný systém charakterizovat jako skupinu autonomních výpočetních prostředků propojených počítačovou sítí a pracujících za pomoci specializovaného software (anglicky označován middleware, viz obr. 2.1) jako jednotný integrovaný systém. Distribuované systémy mají následující vlastnosti:

- **neobsahují sdílené fyzické hodiny** - vzhledem k tomu, že DS je distribuovaný přes množství autonomních systémů, nelze využívat sdílené fyzické hodiny, je třeba zajistit synchronizaci lokálních hodin každého uzlu (tématu času v DS se bude práce věnovat v kapitole 5)
- **neobsahují sdílenou paměť** - nelze předpokládat přítomnost sdílené paměti, jednotlivé entity komunikují například pomocí zpráv, kterými lze distribuovanou sdílenou paměť implementovat, není to ovšem sdílená paměť v klasickém chápání, jak ji známe z prostředí multiprocesorových architektur
- **geografická oddělenost** - je třeba si uvědomit, že distribuovaný systém nemusí být umístěn v jediné lokalitě, může být na různých síťových linkách v různých místech (nelze proto předpokládat jejich kvalitu, spolehlivost ani dostupnost), při komunikaci může docházet ke zpoždění
- **autonomní a heterogenní prostředí** - distribuovaný systém může být (a často bývá) složen z počítačů různých architektur, různých výkonnostních parametrů, odlišných operačních systémů, různých vlastností, které jsou spravovány různými administrátory (nelze předpokládat homogenní prostředí), proto musí být DS navržen tak, aby tyto vlastnosti reflektoval a nabízel dostatečnou míru abstrakce. Jednotlivé prvky DS jsou zároveň autonomní jednotky, mohou DS opouštět, v rámci systému se přeskupovat i přibývat

Distribuované systémy jsou konstruovány za účelem sdílení prostředků, transparentního přístupu k těmto prostředkům, řešení konkurenčního přístupu a škálování. V reálných implementacích se obyčejně z některých vlastností kladených na distribuovaný systém rezignuje, protože se systém navrhuje, tak aby nabízel služby daných parametrů a priorit, kdy některé z požadavků mohou mít protichůdný charakter a mohou se vzájemně vylučovat, nebo negativně ovlivňovat.

2.1 Sdílení prostředků

I přes Mooreův[3] zákon nedostačuje výkon a disková kapacita jediného počítače požadavkům na výkon v prostředí moderního internetu a počítačových sítí, kdy



Obr. 2.1: Distribuovaný systém jako middleware poskytující nutnou infrastrukturu pro aplikace

k jednomu prostředku, nebo službě mohou přistupovat paralelně tisíce až miliony uživatelů. Dostupnost těchto prostředků a služeb nelze realizovat jinak. Servery, které tuto službu realizují, tak sdílejí svůj výpočetní výkon, operační paměť, diskovou kapacitu, síťové linky a další prostředky, aby navenek pracovaly jako jediný systém.

2.2 Transparentní přístup

Transparentní přístup k distribuovanému systému znamená, že uživatel, nebo další služba přistupují k distribuovanému systému jako k běžnému jednoduchému systému, tzn. služba vnější svět odstiňuje od vnitřní architektury distribuovaného systému a poskytuje služby jako centralizovaný systém. Oblasti, které je třeba při návrhu distribuovaného systému řešit, popisuje klasifikace Advanced Networked Systems Architecture (ANSA) [4][5]. Ideální distribuovaný systém musí být transparentní vůči uživateli v následujících oblastech:

- **přístupová transparentnost** - skrývá rozdíl mezi přístupem k lokálním a vzdáleným prostředkům, k službě se přistupuje stejně jako by byla lokální
- **místní transparentnost** - skrývá lokalitu vzdálených prostředků, prostředek je dostupný stejně bez ohledu na lokalitu uživatele nebo poskytovatele
- **migrační nebo relokační transparentnosti** - skrývá uživateli změnu struktury uvnitř DS (výměnu nebo stěhování počítačů podílejících se na poskytování služby), přidání dalších výpočetních prostředků (počítačů), jejich redukci nebo přesun
- **transparentnost selhání a perzistence** - systém je odolný vůči parciálním selháním (selhání jednotlivých prvků DS) a data jsou v něm trvalá, po výpadku

nebo selhání jsou stále konzistentní, bez ztrát

- **konkurenční a transakční transparentnost** - skrývá konkurenční přístup více uživatelů. Uživatel nezaznamená vliv ostatních uživatelů na svůj přístup k prostředkům. Konkurenční přístup je řešen pomocí transakcí splňujících požadavky na bezpečný transakční přístup, A - atomičnost, C - konzistence, I - izolace, D - trvanlivost
- **transparentnost replik** - skrývá skutečnost zda je prostředek originální nebo replikovaný a zda je replika synchronní s původním prostředkem
- **bezpečnostní transparentnost** - jednotná správa přístupů a zabezpečení dat

2.3 Konkurenční přístup

Distribuovaný systém je soubor dislokovaných paralelně běžících procesů, které pracují společně. V distribuovaném systému probíhá neustálé soupeření o sdílené zdroje, které musí být algoritmicky nebo autoritativně řízené, aby nedošlo k poškození dat. Přístupy ke společným prostředkům DS, zvláště pak ty, které manipulují s daty, musí být řízeny a synchronizovány, jinak může dojít ke ztrátě nebo poškození dat, případně i k výpadku celého systému.

2.4 Škálování

Distribuovaný systém musí být schopen přizpůsobit se požadavkům na něj kladených, to znamená uspokojit poptávku po rostoucích nárocích na dostupnost služeb, jejich odezvu nebo nárůst počtu uživatelů využívajících jeho prostředky. Přidání dalších zdrojů musí být možné a pro uživatele transparentní. Tento požadavek klade zvýšené nároky na design distribuovaného systému.

3 TYPY DISTRIBUOVANÝCH SYSTÉMŮ

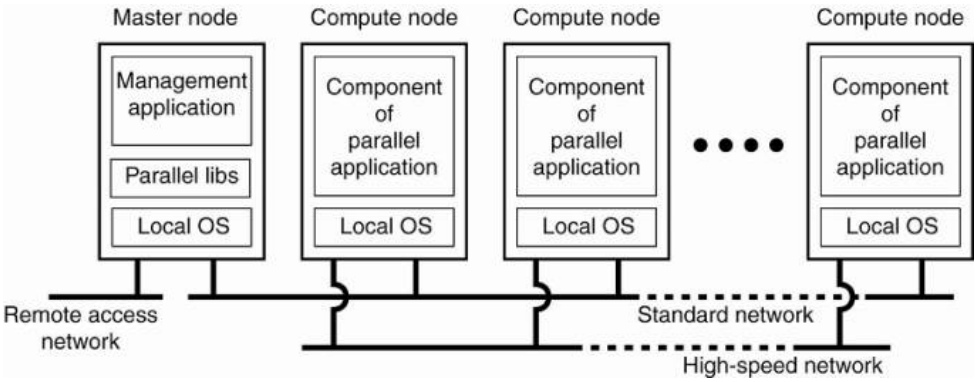
Distribuované systémy lze klasifikovat do skupin podle použití. Dle užití se může lišit důraz kladený na jednotlivé vlastnosti distribuovaných systémů, tak i jejich vnitřní a vnější architektura.

3.1 Distribuované výpočetní systémy

Tyto systémy jsou konstruovány hlavně za účelem řešení náročných výpočtů s důrazem na vysoký výpočetní výkon. Rozlišujeme zde dvě další podskupiny a to **klastrové výpočetní systémy** a **gridové výpočetní systémy**.

První, klastrové systémy (obr. 3.1), se skládají ze skupiny těsně vázaných systémů (vysoko rychlostní lokální síť) často se sdíleným hardware (úložiště), na kterých je provozován stejný operační systém (OS) se shodným softwarovým vybavením. Hardware jednotlivých výpočetních jednotek bývá také shodný nebo příbuzný.

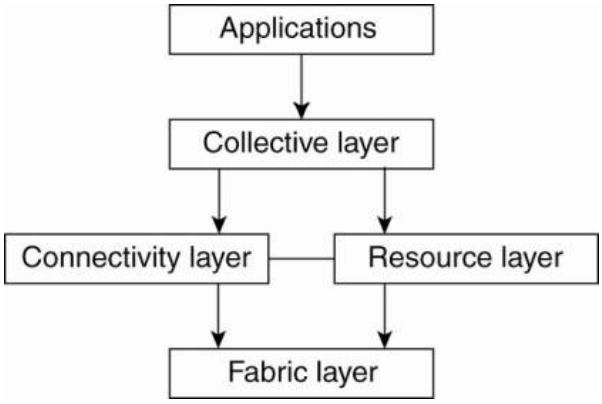
Oproti tomu v gridových výpočetních systémech je situace odlišná, gridové systémy jsou federativně organizované, kde každá výpočetní jednotka podléhá jiné správě, jsou geograficky vzdálené, síťové technologie a operační systémy mohou být zvoleny různě, ani hardwarové architektury nemusí být voleny jednotně.



Obr. 3.1: Příklad klastrového výpočetního systému

Klastrové výpočetní systémy: Stále nižší pořizovací ceny hardware vedly k závěru, že velké množství výpočetních jednotek může nabídnout výkon vyšší než tehdejší mainframe. To dalo vzniknout tomuto typu DS. Dnes na principu klastrů stojí architektury většiny superpočítačů. Jejich úkolem je vytvářet prostředí pro běh paralelních programů. Jedná se o velké množství zpravidla heterogenních výpočetních jednotek, propojených vysokorychlostní sítí. K tomuto prostředí se přistupuje přes

hlavní uzel. Ten má za úkol spravovat chod klastru, spravuje alokaci uzlů a jejich zdrojů, řídí frontu úloh (běžících, připravených a dokončených) a zprostředkovává administrační rozhraní pro správu klastru jako celku. Uzly klastru obsahují zpravidla pouze operační systém (v drtivé většině GNU/Linux nebo jiný UNIXový OS) a základní software pro zapojení uzlu do klastru a běh paralelních programů (midleware). Zbytek nástrojů, knihoven a dalšího programového vybavení si úlohy nesou sebou. Existují i prostředky, které skutečnou infrastrukturu klastru překrývají tak, že je pro uživatele skutečně transparentní. Jedním z takových prostředků je projekt MOSIX [6]. MOSIX je klastrový operační systém založený na Linuxovém jádře (MOSIX2 na verzích 2.6 a 3), který nad skupinou výpočetních uzlů vytváří infrastrukturu, jež simuluje klastr jako jeden jednoduchý uzel (není třeba speciálních úprav programů pro běh v tomto prostředí). Je to však vykoupeno ztrátou výkonu.



Obr. 3.2: Vrstvená architektura gridového výpočetního systému

Gridové výpočetní systémy: Základní vlastností gridů je, že se jedná o počítače nebo klastry různých organizací poskytujících svůj výkon ke společnému řešení strojově náročných výpočtů. Příkladem gridu může být MetaCentrum CESNETu [7] nebo EUROGRID [8]. Výpočetní prostředky jednotlivých organizací jsou poskytovány do tzv. virtuální organizace, kde mohou být sdíleny jejími členy. Architektura gridového systému vystavěná nad jednotlivými prostředky organizací je vrstvená, jak je zobrazeno na obr. 3.2. Dolní vrstva (fabric layer) zprostředkovává přístup k lokálním zdrojům ve specifické lokalitě, umožňuje dotazování na obsazení zdrojů a jejich možnosti pro řízení zdrojů. Spojová vrstva (connectivity layer) zajišťuje spojení mezi výpočetními body, přenos dat mezi uzly nebo sdílení vzdálených dat, dále zajišťuje autentizaci a autorizaci uživatelů (jejich úloh) a zdrojů. Vrstva zdrojů (resource layer) spravuje jednotlivé zdroje označené za dostupné, zajišťuje tvorbu procesů, čtení dat, jejich zápis a řídí k nim přístup. Společná vrstva (collective layer)

se stará o správu společných zdrojů, automaticky vyhledává volné zdroje, alokuje je a přiděluje jim plánované úlohy. Aplikační vrstva (application layer) pracuje nad zdroji virtuální organizace a provádí samotné výpočty.

3.2 Distribuované informační systémy

Další skupinou distribuovaných systémů jsou distribuované informační systémy. Tato skupina vznikla v důsledku nárůstu požadavků a množství dat v organizacích využívající informační systémy. Slouží také jako integrační nástroj různých dílčích systémů. Distribuované systémy jsou zde definovány jako middleware systémy nebo distribuované databázové systémy, podle toho na jaké úrovni jsou jednotlivé prostředky integrovány. Integrace na nižší úrovni jsou implementovány jako **distribuované transakce**, požadavky mohou probíhat na velkém množství serverů nebo služeb a uživatel chce mít jistotu, že operace proběhne v pořádku - buď všude nebo nikde. Použití v některých systémech si ovšem žádá sofistikovanější přístup. Podle toho, jak jsou jednotlivé komponenty rozdělené, jsou požadavky zpracovávány a prováděny selektivně pouze někde nebo za jistých podmínek.

3.2.1 Systémy transakčního zpracování

V praxi se distribuované transakční zpracování využívá hlavně v databázových systémech nebo nad nimi. Databázové systému často umožňují lokální transakce (konkurenční přístup v rámci lokálních prostředků), distribuované transakce jsou vyšší abstrakcí nad skupinou těchto systémů. Programy využívající transakce vyžadují speciální primitiva poskytovaná distribuovaným systémem nebo běhovým prostředím pro řízení transakcí (zahájení transakce, ukončení nebo zrušení a dalších). Sada primitiv a jejich sémantika pro práci s transakcemi se může lišit podle systému, knihovny nebo běhového prostředí. Jinou sémantiku a primitiva mohou mít účetní transakce a jiné mohou být použity v případě souborového systému.

Primitivum	Význam / Popis
`ZACATEK_TRANSAKCE`	označuje začátek transakce
`KONEC_TRANSAKCE`	označuje ukončení transakce
`ZRUSENI_TRANSAKCE`	ruší běžící transakci
`CTENI`	čtení dat
`ZAPIS`	zápis dat

Tab. 3.1: Přehled primitiv používaných u transakcí

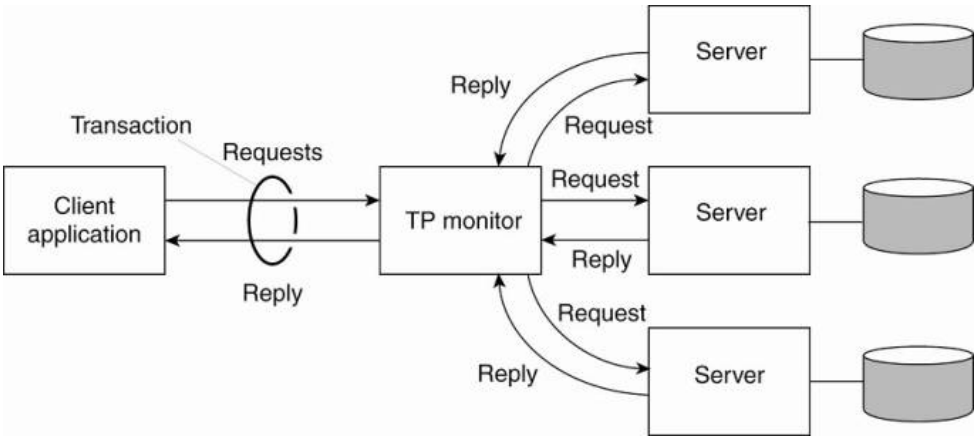
Tabulka 3.1 ukazuje příklad transakčních primitiv. Primitiva `ZACATEK_TRANSAKCE` a `KONEC_TRANSAKCE` označují platnost dané transakce, její začátek a konec. Operace mezi těmito značkami tvoří tělo transakce. Základní vlastností transakcí je že buď jsou provedeny všechny její operace nebo žádná z nich. V případě, že během transakce nastane chyba, je možné transakci předčasně zrušit pomocí `ZRUSENI_TRANSAKCE`. Možné operace, které jsou použity uvnitř transakce se nemusí omezovat pouze na operace `CTENI` a `ZAPIS`, jako je uvedeno v tab. 3.1, ale může se jednat o volání procedur a funkcí, systémová volání, větvení a dalších. Záleží na tom, jak a pro co je transakční zpracování implementováno. Ideální transakce musí splňovat čtyři požadavky, které jsou obyčejně označené jako ACID, (anglický akronym těchto požadavků). Transakce musí být :

- **Atomická** (Atomicity) - transakce je provedena jako jedna operace
- **Konzistentní** (Consistency) - transakce zachovává systém konzistentní
- **Izolovaná** (Isolated) - jednotlivé transakce se nesmí vzájemně ovlivňovat
- **Trvalá** (Durable) - změny, které transakce provede jsou trvalé

Atomičnost transakcí zaručuje, že žádný její mezistav nebude viděn zvenčí. Transakce skončí jako celek úspěchem nebo ne, žádné dílčí operace se mimo ni neprojeví. Další požadavek- konzistence - znamená, že transakce zachovává stav věcí v konzistentním stavu. Požadavek izolovanosti transakcí znamená, že pokud transakce běží paralelně, jejich vliv na data musí být, jako by běžely sekvenčně v daném pořadí. Trvalost transakce pak požaduje, že pokud transakce jednou proběhne, pak změny které se během ní provedly jsou pro systém trvalé a nezvratné.

Takto jsou popsané a definované jednoduché transakce. Složitost do tohoto systému může vnést definice **podřízených transakcí**. Podřízené transakce musí splňovat stejné požadavky jako jednoduché transakce. Z požadavků na transakce plyne, že při jejím startu pracuje transakce ve vlastním pracovním prostoru, kde probíhají změny, které jsou na konci transakce promítnuté do původních dat. Vložené transakce si pak vytvářejí další vlastní pracovní prostor v těch nadřazených. Po jejich skončení vidí změny pouze nadřazená nebo následující podřazené transakce. Okolní svět změny nevidí. Dříve se zápis do různých podřízených systémů (v distribuovaném prostředí) implementoval právě pomocí podřízených transakcí.

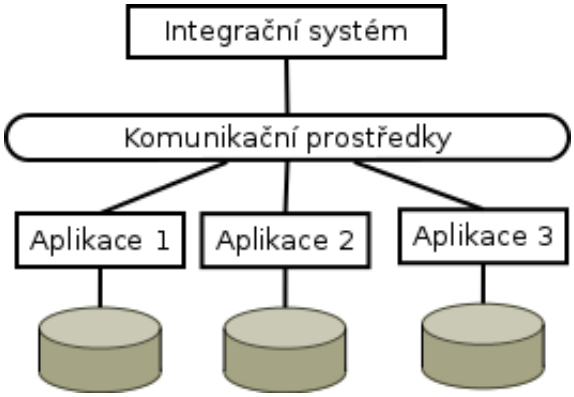
Transakční systémy mohou být implementovány tak, jak je zobrazeno na obr. 3.3, kde před jednotlivými servery systému stojí prvek, který transakce řídí (TP Monitor). Transakce mohou být řízeny centralizovaně, za pomocí middleware nebo distribuovaně, kdy si transakční zpracování vyjednávají servery mezi sebou.



Obr. 3.3: Model transakčního systému

3.2.2 Integrační systémy v organizacích

S vývojem aplikací, jejich množstvím a průběžnými změnami v nich, vznikly požadavky na integraci těchto aplikací. Integrace na úrovni databází přestávala dostačovat, začala se přesouvat na úroveň aplikací. Prvním krokem bylo vzdálené volání procedur RPC. Tato implementace umožňovala oddělit implementaci databáze a jejího schématu od implementace na úrovni DS. Později s rozšířením objektového programování se více uplatnil princip vzdáleného vyvolávání metod RMI.



Obr. 3.4: Model integračního systému

Tyto principy jsou použité a zobecněné jako API (definované mnoha dalšími standardy) - rozhraní pro komunikaci mezi aplikacemi. Oba tyto způsoby však vyžadují, aby obě komunikující strany byly běžící, schopny komunikace a znaly vzájemně své adresy (i způsob komunikace). To dalo vzniknout dalšímu integračnímu principu - zprávově orientovaný middleware MOM, jednotlivé systémy komunikují

pomocí výměny zpráv přes prostředníka (middleware). Této infrastruktury je využito při integraci aplikací. Nad jednotlivými aplikacemi je vystavěn integrační systém jako nadstavba. Tato nadstavba obsahuje pouze logiku, jak využívat služeb podřízených aplikací, které poskytuje navenek(viz obr. 3.4). Uživatelé pak nemusí pracovat s množstvím samostatných aplikací, ale pouze s tímto integračním systémem.

4 KOMUNIKACE V DISTRIBUOVANÝCH SYSTÉMECH

Distribuované systémy jsou organizované jako samostatné procesy běžící na různých uzlech počítačové sítě, neobsahují žádnou sdílenou paměť přes kterou by bylo možné vyměňovat informace nebo implementovat synchronizační primitiva. Musí proto komunikovat skrze síť pomocí zpráv (přímo, nebo pomocí front), vzdáleného volání procedur, metod (RPC, RMI) nebo datových streamů. Použití těchto komunikačních prostředků může být různé podle použití:

- **stream** : multimédia nebo transakční logy v případě databázových replikací (např. PostgreSQL, MySQL)
- **vzdálené volání**: volání procedur (RPC) v případě Network File System, síťový souborový systém vyvinutý společností Sun Microsystems (NFS)
- **zasílání zpráv** : MPI, Message Passing Interface, knihovna implementující protokol pro podporu paralelního řešení výpočetních problémů v počítačových clusterech pomocí zasílání zpráv (MPI)
- **zasílání zpráv do front** : zprávy zasílané pomocí Apache ActiveMQ, IBM WebSphere Message-Queuing System a dalších

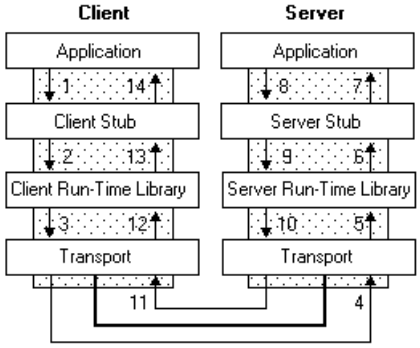
Způsob komunikace mezi procesy v DS je klíčový pro jejich návrh, pochopení i provoz. Komunikace v distribuovaných systémech je mnohem složitější, než v paralelních programech se sdílenou pamětí. Reálně je distribuovaný systém složen z desítek až tisíců komunikujících procesů, proto je třeba dbát na návrh komunikace při implementaci distribuovaného systému. Způsob komunikace ovlivňuje výběr a použití algoritmů pro synchronizaci, řazení požadavků, vzájemné vyloučení nad sdílenými prostředky a dalších, které se v paralelních systémech řeší pomocí sdílené paměti, nebo speciálních instrukcí jednotlivých procesorů. Metody synchronizace procesů, vzájemného vyloučení a další vzájemná interakce procesů proto musí být řešeny na úrovni algoritmů prostřednictvím zvolených a dostupných komunikačních prostředků.

4.1 Vzdálená volání

Přirozeně se nabízí implementovat provádění akcí vzdáleně skrze počítačovou síť tak, jak je v programovacích jazycích zvykem, klasickým voláním procedur (RPC), funkcí nebo metodou objektů (RMI), z toho pak vychází abstrakce v síťových (distribuovaných) systémech. Funkce v programovacích jazycích se v programovém kódu volají stejně jako by byly lokální. Speciální programová vrstva (knihovna), toto volání přenáší prostřednictvím počítačové sítě na vzdálený uzel, kde je funkce provedena.

Tento mechanizmus pro vzdálené volání procedur (RPC) verze 2 nejaktuálněji popisuje RFC 5531 [9], pro RMI je popsán ve whitepaperu společnosti Oracle [10]. V této kapitole nabídne práce popis pouze pro RPC. Snahou RPC je se co nejvíce přiblížit použití běžného volání procedur na lokálním systému.

Lokální a vzdálený uzel nemají společný zásobník pro předávání parametrů, ani adresní prostor pro parametry předávané odkazem. Problém tvoří i použití vstupně/výstupních zařízení (soubory, IO porty atp. - ty není možné v proceduře použít), která nejsou připojená k oběma uzlům. Je proto nutné zřídit mechanizmus, který bude tuto vlastnost řešit.



Obr. 4.1: Vzdálené volání procedur

V prostředí RPC tento problém řeší tzv. stub (adaptér), který je volán místo zamýšlené procedury. Stub (klientský) provede akce potřebné pro volání procedury, tak aby je dokázal provést na vzdáleném uzlu. Pomocí jmenných služeb vyhledá serverový stub, hodnoty volané procedury zkopíruje, ukazatele dereferencuje a vše zabalí a přenese po síti zakódované tak, aby zakryl i možné rozdíly mezi architekturami obou uzlů. K přenášeným parametrům ještě přidá informace o tom jaká procedura má být volána, verzi protokolu a další potřebné informace. Vše na vzdáleném uzlu po síti přijme serverový stub, který data dekóduje, rozbalí a připraví v paměti pro lokální volání procedury. Ze zprávy se podle identifikace procedury pak zavolá procedura implementovaná na serveru. Po skončení procedury, vezme server stub data jež mají být vrácena, zabalí je, zakóduje a pošle zpět po síti na klientský stub, který data zrekonstruuje jako lokální, vrácené procedurou. Celý proces volání vzdálené procedury zobrazuje obr. 4.1.

4.2 Komunikace pomocí zpráv

Vzdálené volání procedur nebo metod poskytuje pohodlnou abstrakci pro vývoj síťových a distribuovaných systémů. To ovšem nemusí být dostačující pokud je třeba

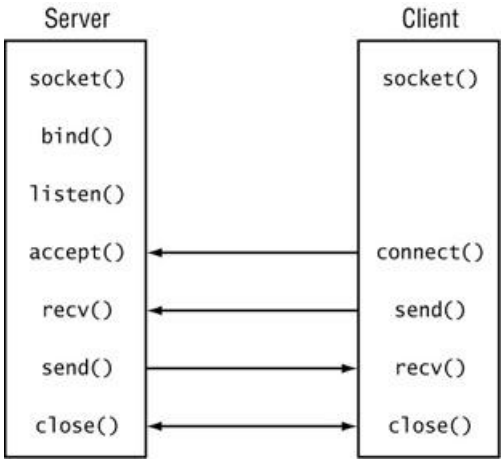
jiná úroveň abstrakce, nebo pokud sémantika volání vzdálených procedur nevyhovuje určení DS. Pak se nabízí implementace komunikace uzlů pomocí zpráv.

Zprávy jsou předávány přímo mezi komunikujícími uzly. Zprávy jsou zasílány pomocí protokolů transportní vrstvy ISO/OSI [11] pomocí různých protokolů a technologií (sériové linky, ethernet, infiniband a dalších). Práce se bude dále zabývat komunikací pomocí zpráv v prostředí internetu, kde je zasílání zpráv implementované pomocí Berkeley socketů, které jsou použity také v KIVFS. Berkeley sockety (dále sockety) vznikly jako programovací prostředek pro snadnou implementaci síťových aplikací přenositelných mezi platformami s důrazem na transparentnost. Se sockety se pracuje podobně jako s klasickými vstupně/výstupními zařízeními, které jsou v prostřední OS UNIX (kde Berkeley sockety vznikly) implementovány jako soubory. Pro práci se sockety jsou definována primitiva, která jsou popsaná v tab. 4.1 a obr. 4.2 pak zobrazuje jejich použití.

Primitivum	Význam / Popis
`socket`	vytváří koncový bod spojení - socket
`bind`	páruje lokální adresu a socket
`listen`	umožňuje přijímat nová spojení
`accept`	blokuje volajícího dokud nepřijde nové spojení
`connect`	navazuje spojení k vzdálené adrese
`send`	posílá data přes dané spojení
`receive`	přijímá data přes dané spojení
`close`	zavírá, uvolňuje spojení

Tab. 4.1: Přehled primitiv používaných u socketů

V každém programu je nejprve vytvořen `socket`, dále se liší použití v aplikaci serveru (přijímá spojení) od aplikace klienta (navazuje spojení). Pomocí `socket` se v systému vytvoří komunikační koncový bod (síťový socket, unix socket, spojový, nespojový). Poté následuje na straně serveru volání `bind`, které zajistí párování adresy a portu (v IP sítích) s vytvořeným socketem. V případě spojových služeb následuje volání `listen`, které oznamuje operačnímu systému, že má na socketu přijímat nová spojení. Volání `accept` blokuje volajícího do doby, než bude na daném socketu navázáno nové spojení. Poté `accept` vrací socket nového spojení, přes který probíhá následná komunikace. Na straně klienta není třeba volat `bind` ani `listen`. K přiřazení místního portu dochází při volání `connect`, kdy je vyžadována identifikace cílového spojení (vzdálená adresa a port). Volání je blokující dokud nedojde k navázání spojení, kdy dochází k synchronizaci klienta a serveru na voláních `connect` a `accept`. Když je spojení sestaveno, lze přes něj pomocí `send` a `receive` vyměňovat datové zprávy. Spojení se následně uzavírá voláním `close`.



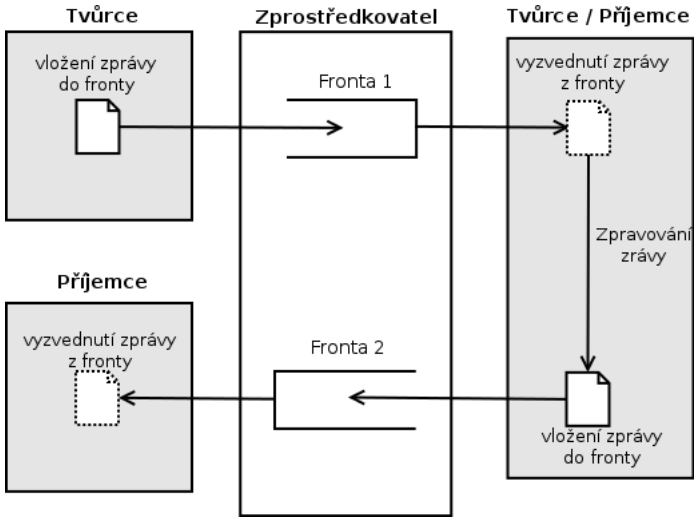
Obr. 4.2: Sémantika funkcí Berkeley socketů

Nad touto infrastrukturou tvořenou sockety lze vystavět složitější konstrukce, frameworky a prostředky jako je například MPI, nebo starší obdoba PVM, které se používají v klastrech, jak bylo zmíněno v předchozích kapitolách. Zasílání zpráv je použito i v jednodušších protokolech jako je HTTP, FTP, DNS a dalších internetových protokolech.

4.3 Komunikace pomocí front zpráv

Komunikace pomocí front zpráv je vystavěna nad prostým zasíláním zpráv. Model je tvořen třemi aktéry, **tvůrcem** zprávy, jejím **příjemcem** (či jejich libovolným množstvím) a **zprostředkovatelem**, který spravuje jednu nebo více front. Schéma komunikace popisují obrázky 4.3 a 4.4. Komunikace pomocí front nabízí komunikační infrastrukturu pro asynchronní výměnu zpráv mezi uzly distribuovaného systému. Tvůrce vytvoří zprávu (ve specifickém formátu např. XML, YAML, JSON) a vloží ji do fronty. Zpráva může obsahovat jednoznačný identifikátor aktérů (tvůrce a příjemce) a samotné zprávy, podle kterého může být řízeno vyzvedávání zpráv z fronty. Příjemce zpráv, který je přihlášen k dané frontě, nebo frontám si zprávy z fronty vyzvedává a následně provádí definované akce. Odpověď na danou zprávu bývá realizována vložením zprávy s odpovědí do k tomu určené fronty. Prostředník, který realizuje fronty zpráv a jejich výměnu, registraci tvůrců a příjemců je speciální software. Tento software může mít různou architekturu, jednou z možností je centralizovaný broker (případně jejich klastr) nebo překryvná síť, kde dochází ke směrování zpráv pomocí routerů.

Komunikace pomocí zpráv nabízí specifické vlastnosti. Komunikující strany nemusí být spojené přímo, příjemce zprávy nemusí být připojen v době jejího odeslání



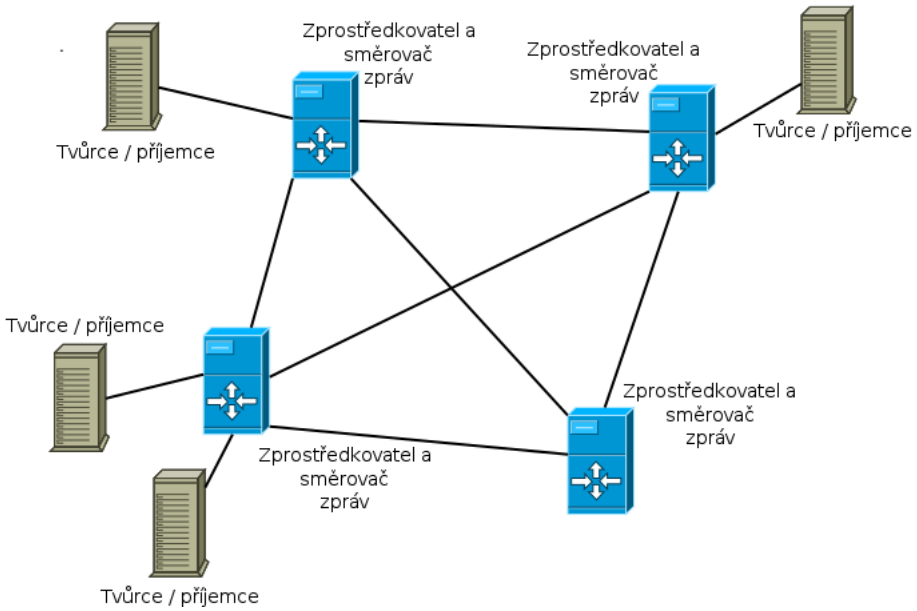
Obr. 4.3: Schéma komunikace pomocí front zpráv (centralizované)

nebo aktivně přijímat zprávy. Oba aktéři se nemusí vzájemně znát (podmínkou je znalost prostředníka). To umožňuje libovolně přeskupovat jak tvůrce tak příjemce zpráv, aniž by o tom kterákoli strana musela být informována. Dalším aspektem tohoto způsobu komunikace je, že tvůrci je potvrzeno pouze zařazení do fronty. O tom kdy bude zpráva vyzvednuta(zda vůbec) příjemcem není informován, ani o tom, že je zpráva zpracována. Toto může být implementováno pomocí dalších zpráv přes fronty, které o tom budou tvůrce informovat. Ani tam není zaručeno, kdy budou tyto zprávy zpracovány.

Tento způsob komunikace se používá, pokud zpracování zprávy nebo provedení dané úlohy nevyžaduje synchronní zpracování, typicky když se jedná o operaci časově náročnou, nebo není možné zaručit aby oba komunikující uzly byly schopné komunikovat přímo a současně. Zpráva může obsahovat libovolná data, multimédia, XML dokument, popis volání akce a další. Velikost zprávy může být omezena, toto záleží na implementaci komunikujících stran nebo brokera, poté musí komunikující strany podporovat fragmentaci zpráv a následné sestavení. K tomu je třeba zajistit spolehlivou unikátní identifikaci zpráv.

Pro komunikaci pomocí front zpráv je definována následující základní sada primitiv (tab. 4.2), tato primitiva mohou být rozšířena pro kontrolu a sledování, zjištění počtu zpráv ve frontě(frontách), průměrná doba jakou stráví zpráva ve frontě, počet registrovaných tvůrců a příjemců front a další.

Primitivum `put` vkládá zprávu do fronty, vložení zprávy je potvrzované. Příjemce vyzvedává zprávu z fronty pomocí primitiva `get`, který blokuje volajícího dokud není ve frontě dostupná zpráva. Pokud není fronta prázdná vrací zprávu, která je ve frontě nejdelší dobu a odebírá ji z fronty. Primitivum `poll` provádí stejnou akci,



Obr. 4.4: Schéma komunikace pomocí front zpráv (decentralizované)

Primitivum	Význam / Popis
`put`	vloží zprávu do vybrané fronty
`get`	blokuje dokud není zpráva ve frontě, pak odebere první zprávu
`poll`	neblokující odebrání zprávy z fronty
`notify`	nastavuje akci, když je do fronty vložena zpráva

Tab. 4.2: Přehled primitiv používaných u komunikace pomocí front zpráv

jen volajícího neblokuje, pokud je fronta prázdná skončí a informuje o tom. Dále je možné vložit akci v případě přidání nové zprávy do fronty, která může probudit nebo nastartovat proces, který zprávu z fronty vyzvedne.

4.4 Stream komunikace

Komunikace pomocí proudů dat představuje další možnost komunikace, široce využívanou například u multimédií. Přenos zvuku nebo obrazu je vhodnější přenášet jako nepřerušovaný proud dat, který obsahuje za sebou organizované vzorky zvuku nebo obrazové informace.

Prakticky lze datový proud realizovat i pomocí množství menších zpráv představující například vzorek zvuku nebo jejich kratší posloupnost. Informace přenášené pomocí proudu dat jsou časově závislé. Informace má pro příjemce hodnotu jen pokud dorazí k příjemci v daném pořadí a s minimálním zpožděním. Z toho často

vychází i další aspekt a to k jakému účelu je daný proud využíván. V případě online hovorů je hlavním parametrem nízká latence spojení, pokud se jedná o video hovor je často požadována i vyšší přenosová rychlost. Oproti tomu ztrátu menšího množství dat je možné tolerovat, protože si aktéři informaci zopakují. V případě přenosu například televizního vysílání není prioritou nízká latence, protože je možné využít vyrovnávací paměť, ze které bude vysílání se zpozděním přehráváno, ale hlavním požadavkem bude přenosová rychlost kvůli obrazové kvalitě vysílání.

Komunikace pomocí proudů dat je často doplněna komunikací pomocí zpráv pro synchronizaci a ovládání proudu, jak bylo uvedeno v příkladu s televizním vysíláním. Také je možné dalším komunikačním kanálem vysílat televizní program, měnit kanály, ovládat rychlost proudu nebo měnit jeho parametry.

K řízení (komunikaci mezi uzly) DS není příliš vhodný, protože je náročnější na synchronizaci.

4.5 Strategie komunikace

Ať už využívá distribuovaný systém jakýkoli způsob komunikace, je možné dále rozlišovat různé strategie komunikace synchronní a asynchronní.

4.5.1 Synchronní komunikace

Synchronní komunikace představuje blokující přístup, kdy komunikující strana čeká dokud není její požadavek vyřízen. Nad voláním daného požadavku dochází k synchronizaci, strana která odesílá požadavek je zablokována a čeká dokud strana, která tento požadavek plní nedokončí jeho obsluhu. To představuje časový souběh požadavku a jeho vyřízení (to si žádá jistý čas, kdy je volající blokován). Tento princip je využíván například u volání vzdálených procedur, kdy klientský program čeká na obsloužení vzdáleného volání a vrácení jeho výsledku. Synchronní komunikace je využívána v mnoha případech, například načítání dat z databáze. Databázový klient čeká dokud není jeho dotaz dokončen a vráceny výsledky. Z pravidla se využívá synchronní komunikace v případech, kdy na výsledku požadavku závisí další činnost a nelze bez jeho dokončení pokračovat v další práci, nebo pokud je třeba znát výsledek co nejdříve. Synchronní komunikace může sama o sobě sloužit jako synchronizace dvou aktérů nad jednou zprávou. Synchronní komunikace se využívá v mnoha síťových protokolech, například HTTP, POP3, SMTP a další.

4.5.2 Asynchronní komunikace

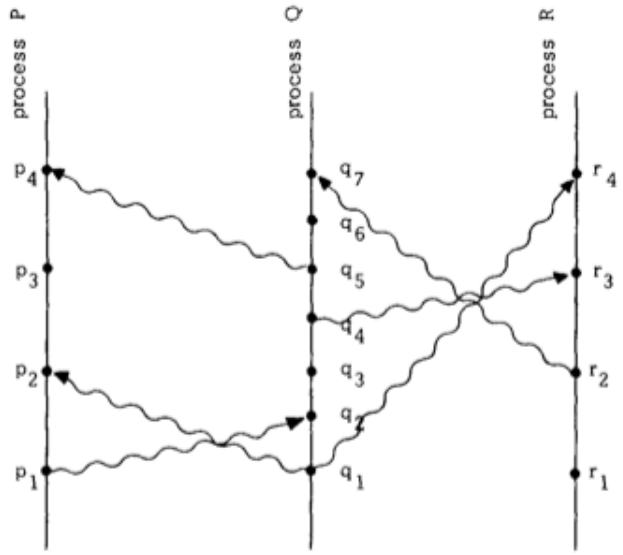
Druhou možností je asynchronní přístup. Strana požadující provedení libovolné akce není omezena nebo blokována po dobu jejího provádění, ale může pokračovat v další činnosti. Příkladem takového přístupu je komunikace pomocí front zpráv. Aktér vyžadující provedení nějaké akce, vloží do fronty svůj požadavek a pokračuje v další činnosti, o výsledku operace může být informován další zprávou, nebo také vůbec. Tento přístup lze aplikovat i v případě výše zmíněných vzdálených volání. Synchronně proběhne pouze volání procedury, která požadavek přijme ke zpracování a vyzvednutí výsledku je provedeno dalším synchronním voláním požadujícím výsledky, pokud ještě není výsledek dostupný, je o tom volající informován a opakuje dotaz po dalším obecně libovolném intervalu. Výhodou asynchronní komunikace je, že aktér požadující nějakou službu nebo akci není na tomto požadavku blokován a nečeká na jeho vyřízení, to může být výhodné v případech kdy je akce příliš časově náročná, její vyřízení není akutní, nebo probíhá dávkové zpracování požadavků, klient nečeká na vyřízení, ale je možné ho opět asynchronně informovat o výsledku akce.

4.5.3 Komunikace v KIVFS

Pro použití v KIVFS byl zvolena synchronní komunikace pomocí zpráv, která je pro implementaci souborového systému přirozená. Jednotlivé operace souborového systému představují zprávy a klientská aplikace nemůže pokračovat v práci dokud není daná operace zpracována na straně serveru.

5 ČAS A SYNCHRONIZACE

V předchozích kapitolách byly popsány distribuované systémy a možnosti komunikace jednotlivých aktérů. Jak již bylo řečeno, distribuované systémy mají za cíl sdílet a nabízet zdroje, dochází tak k opakovanému konkurenčnímu přístupu k těmto zdrojům. Ke zdrojům se přistupuje, jak pro získávání dat, tak i pro jejich modifikaci (zápis, přepis). Uzly distribuovaného systému musí, stejně jako je tomu u paralelních systémů, o sdílené prostředky nejprve požádat. Je proto důležité mít informaci o tom, v jakém pořadí se tyto požadavky přišly, aby bylo možné určit, který z konkurenčních uzlů dostane zdroj přidělen.



Obr. 5.1: Události a zasílání zpráv v DS

V reálném světě se k určení pořadí událostí využívají fyzické hodiny. Problém s těmito hodinami je jak s jejich synchronizací a přesností, jejich měřením, tak korekcí jejich běhu. Za nejpřesnější zdroj času jsou považovány atomové hodiny, ty ovšem nelze mít obsažené v každém uzlu distribuovaného systému, jak pro finanční náročnost, tak i proto, že atomové hodiny musí být synchronizovány k nějakému fyzickému času. Atomové nebo jiné hodiny přesného času poskytují pouze takt (periodicky odpočítávají daný časový interval), proto musí být s přesným časem synchronizovány. V distribuovaných systémech je situace o to složitější, že počet aktérů v systému může být v řádu desítek až tisíců, jejich lokální hodiny jsou různě přesné a zpoždění při jejich komunikaci není nulové ani konstantní. Je proto nutné zavést jiné prostředky pro určování času a synchronizaci, které nemají výše zmíněné nedostatky. Těmito prostředky jsou logické hodiny a další synchronizační algoritmy.

5.1 Fyzické hodiny

Fyzické hodiny se v distribuovaných systémech nehodí pro určování pořadí akcí nebo synchronizaci akcí, ale jsou i tak velice důležité. Určuje se podle nich platnost vystavených certifikátů, autentizačních oprávnění, časy přístupu, zápisu či modifikace souborů, platnost záznamů ve vyrovnávacích pamětech a mnoho dalších. Je proto důležité udržovat hodiny fyzického času synchronní mezi uzly distribuovaného systému. To lze realizovat například pomocí protokolu NTP (Network Time Protocol, protokol pro synchronizaci času po síti).

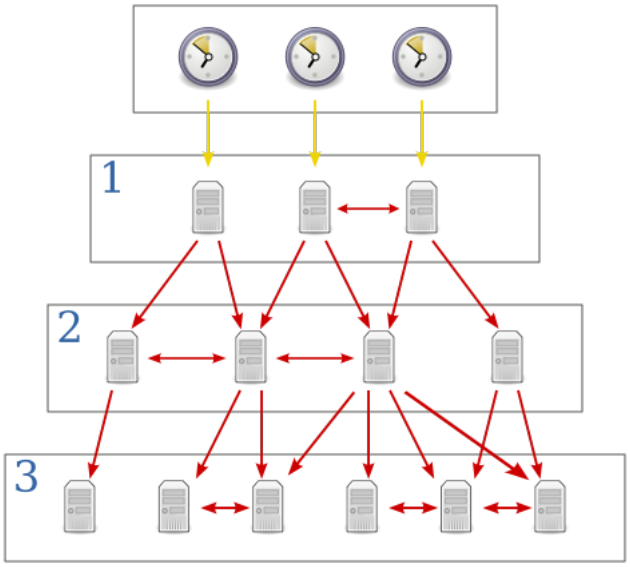
5.1.1 Protokol NTP

Protokol NTP je jeden z nejstarších protokolů internetu, jeho práci specifikují RFC 778, 891, 956, 958 a 1305. Protokol je určen pro synchronizaci hodin počítačů po paketové síti (využívá UDP/IP) s proměnným zpožděním. Vzhledem k době kdy byl navržen (1981), tak počítá s různou kvalitou propojovacích linek, možnou nedostupností všech spojů a klade důraz na efektivitu komunikace. NTP rozlišuje klienty (požadují časové údaje) a servery (poskytují časové údaje). Servery NTP tvoří hierarchickou strukturu podle kvality zdrojů času (obr. 5.2). Každý stupeň je označen číslem, který určuje kvalitu zdroje času označené jako stratum. Hodnoty, které jsou pro stratum definované jsou 0 až 15, reálně se objevují hodnoty do čísla 5.

- **stratum 0** mají přesné zdroje času (atomové hodiny), nejsou přímo připojené k síti, ale k serveru stratum 1
- **stratum 1** mají počítače, které jsou přímo připojené k přesným zdrojům času (např. sériovou linkou), jsou označovány jako timeservery
- **stratum 2** mají počítače, které přijímají časové informace od serverů se stratem 1
- **stratum 3 až 14** mají další počítače, se vzrůstající vzdáleností od přesných zdrojů času hodnota stratum stoupá
- **stratum 15** mají servery, které nejsou schopny poskytovat přesný čas (po výpadku sítě nebo při opětovné synchronizaci)

NTP používá pro určení času 64 bitů dlouhé číslo s pevnou desetinnou čárkou, prvních 32 bitů určuje počet sekund od 1.1. 1900 a druhých 32 bitů určuje desetinnou část. Teoreticky může tedy NTP poskytovat přesný čas s přesností až 2^{-32}. Přesnost sekund na 32 bitů by se mohla zdát nedostačující (dochází k pravidelným přetečením po 2^{32} sekundách). S tím však současné implementace protokolu počítají a zavádí tzv. éry, které udávají počet přetečení.

Synchronizace času probíhá tak, že klient si zjistí časové údaje (přesný čas, stratum, přesnost) z dostupných serverů, z přijatých dat vyřadí ty servery, jejichž přesný



Obr. 5.2: Hierarchie časových serverů v NTP

čas se od ostatních příliš liší. Pokud je nastaven jen jeden server je brán pouze jeho přesný čas. Z přijatých časů pak počítá odchylku svého času od vypočteného přesného času a pomocí klouzavého algoritmu přibližuje své hodiny k přesné hodnotě. Pokud je odchylka delší než 128 ms pak NTP mění čas skokem, v případě že je odchylka větší než cca 17 minut odmítne NTP klient provést synchronizaci a oznámí chybu. Jak je patrné z faktu, že NTP se snaží neměnit čas skokově, ale postupně, tak synchronizace probíhá i několik minut, než se rozdíl ustálí okolo stanovené přesnosti.

5.2 Logické hodiny

Pro distribuované systémy je podstatné v jakém pořadí se jednotlivé dvě události staly nebo stanou, aby byly porovnatelné a bylo možné určit, která událost přecházela a která následovala, například při konkurenčním přidělování zdrojů nebo provádění změn (zápis). Pro takové řazení není třeba znát čas události, ale její pořadí vůči ostatním. Toho lze jednoduše docílit, tím že budou událostem přiřazována pořadová čísla, na přiřazování těchto čísel musí existovat jednoznačný algoritmus, který toto umožní napříč distribuovaným systémem. Řadit lokální události není náročné s použitím běžných synchronizačních primitiv známých z paralelních programů.V distribuovaných systémech jsou proto implementovány logické hodiny, které operace číslují. K synchronizaci logických hodin dochází pomocí zasílání zpráv. Každý uzel obsahuje vlastní logické hodiny, globální logický čas je synchronizován mezi procesy při jejich výměně zpráv.

Pro logické hodiny si nadefinujeme relaci **událo se před**, označíme si ji $\rightarrow$. Představme si dvě události a a b. Zápis $a \rightarrow b$ pak označuje, že operace a se udála před operací b. Pokud dojde k další události c zapíše se to následujícím způsobem $b \rightarrow c$. Relace **událo se před** je tranzitivní, pokud $a \rightarrow b \wedge b \rightarrow c$, pak platí $a \rightarrow c$. Pokud nelze určit, která událost se udála dříve $a \nrightarrow b \wedge b \nrightarrow c$, pak se jedná o dvě konkurenční (současné) události . V distribuovaném prostředí je třeba porovnávat události, které se staly na různých uzlech v různých procesech, mezi kterými existuje komunikační zpozdění, které je nekonstantní. Situaci popisuje obr. 5.1, kde jsou svislými čárami označené procesy, tečkami události a šipkami zasílané zprávy. Logické hodiny mají za úkol stanovit, v jakém pořadí se dvě události staly.

Každá implementace logických hodin musí obsahovat dvě základní pravidla

1. **R1**: pravidlo určuje jak jsou logické hodiny synchronizované při provádění akce (odeslání, příjem, nebo lokální)
2. **R2**: pravidlo určuje jak jsou synchronizovány logické hodiny z globálního pohledu

5.2.1 Skalární Lamportovy hodiny

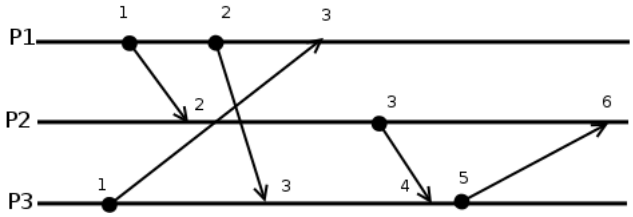
Nejjednodušší formou logických hodin jsou hodiny skalární, ty obsahují pouze jedno bezrozměrné číslo. Každý uzel distribuovaného systému sekvenčně čísluje všechny operace, které provádí. Při komunikaci s jiným uzlem si informaci o stavu svých lokálních hodin porovnají a aktualizují na aktuálně vyšší hodnotu. Konkrétní algoritmus skalárních logických hodin popsal Leslie Lamport [12] v roce 1978. Zavádí pro každý proces P_i lokální logické hodiny C_i , které přiřazují každé události a procesu číslo $C_i\langle a\rangle$ a globální hodiny C. Jak bylo popsáno výše relace $a \rightarrow b$ značí, že událost a nastala před b, pak pro hodiny platí $C(a) < C(b)$, hodnota hodin pro událost a je menší než pro událost b.

Hodnota hodin C_i se řídí následujícími pravidly:

1. **R1**: před provedením každé operace se zvýší hodnota hodin $C_i = C_i + d$, d je konstanta, obyčejně se používá 1, každá zpráva je pak označena hodnotou hodin $C_i(msg)$
2. **R2**: při přijetí zprávy od jiného uzlu jsou provedené následující akce, C_{msg} značí stav hodin přijaté zprávy
   (a) $C_i = max(C_i, C_{msg})$ (aktualizace lokálních hodin)
   (b) provede se pravidlo **R1** (přiřazení lokálního logického času přijaté zprávě)
   (c) zpracuje se daná zpráva

Použití Lamportových hodin ukazuje obr. 5.3.

Skalární logické hodiny mají pro distribuované systémy několik zajímavých vlastností. Nabízejí pouze částečné řazení požadavků. Jsou schopné řadit požadavky ko-



Obr. 5.3: Lamportovy skalární logické hodiny

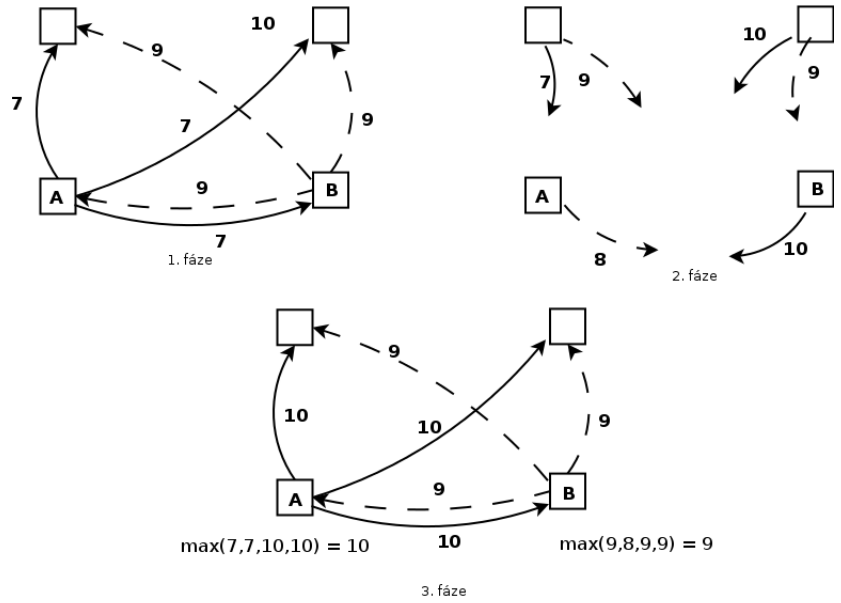
munikujících uzlů, nejsou schopné řadit konkurenční požadavky, kdy nedochází ke komunikaci, nebo požadavky právě nekomunikujících uzlů, viz obr. 5.3 $p_1(3)$, $p_2(6)$ a $p_3(5)$. K synchronizaci logických hodin dochází pouze při komunikaci mezi uzly.

Absolutní řazení Nedostatek částečného řazení lze řešit pomocí skupinové komunikace, nebo centralizovaně (v systému existuje arbitr, který přiřazuje logický čas všem akcím). Dále bude popsán distribuovaný algoritmus, kde každý proces vyšle ostatním požadavek o přiřazení hodnoty logických hodin se svým návrhem, ostatní procesy odpoví svým návrhem. Z přijatých návrhů vybere nejvyšší hodnotu a následně ji oznámí ostatním, kteří si podle vítězné hodnoty aktualizují své lokální hodiny. Algoritmus znázorňuje obr. 5.4. Na obrázku jsou dva procesy A a B požadující synchronizaci svého požadavku. Plná čára představuje komunikaci uzlu A a přerušovaná uzlu B. Algoritmu prochází třemi fázemi.

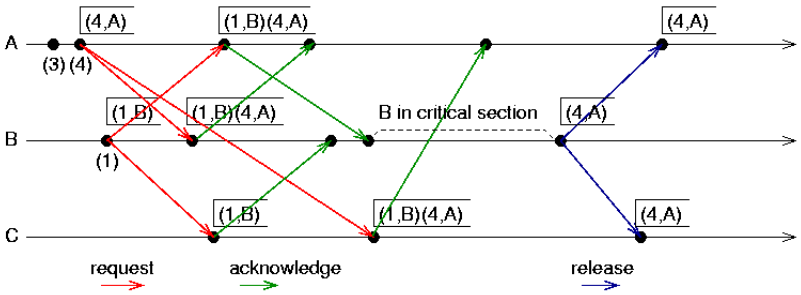
1. fáze: uzel pošle požadavek na ostatní uzly s návrhem logických hodin
2. fáze: uzly přijmou požadavek (podle pravidla **R2**) a odpoví odesílateli stavem svých logických hodin (návrh logického času)
3. fáze: uzel vybere maximum, které oznámí ostatním uzlům a pod tímto časem je akce zpracována

Tento algoritmus přiřazuje řídká pořadová čísla požadavkům (jak znázorňuje obr. 5.4), ale zaručuje jednoznačné určení pořadí všem akcím.

Algoritmus absolutního řazení je možné v obměněné formě použít pro vzájemné vyloučení více uzlů (procesů) nad sdíleným prostředkem. Uzel (proces), který chce vstoupit do kritické sekce pošle žádost ostatním uzlům s časovou značkou svých logických hodin, ostatní uzly si tento požadavek zařadí do fronty. Pokud je požadavek na vrcholu fronty, pošlou žádajícímu uzlu potvrzení. Jakmile obdrží žádající uzel potvrzení od všech uzlů, může vstoupit do kritické sekce. Při jejím opuštění pošle zprávu opět všem o vystoupení z kritické sekce. Ostatní uzly si z fronty odstraní požadavek a pošlou potvrzení dalšímu uzlu, jehož požadavek je ve frontě za právě uvolněným. Uzel může vstoupit do kritické sekce jen pokud má potvrzený vstup od všech uzlů.



Obr. 5.4: Absolutní řazení akcí v distribuovaném systému



Obr. 5.5: Vzájemné vyloučení pomocí Lamportova algoritmu

5.2.2 Vektorové hodiny

Vektorové hodiny zavádí vyšší přesnost. Jsou odvozené od hodin skalárních, které rozšiřují. Místo jednoduchého čítače událostí obsahují tolik čítačů kolik je komunikujících uzlů. Každý uzel má vlastní čítač zpráv a společně tvoří dohromady vektor, který bude značen vt_i. Jednotlivé prvky tohoto vektoru představují čítač operací jednotlivých procesů. Každý proces má vyhrazen vlastní prvek. Stejně jako v případě skalárních hodin jsou pro vektorové hodiny definovány pravidla pro jejich nastavení.

1. **R1**: před provedením každé operace se zvýší hodnota hodin $vt_i[i] = vt_i[i] + d$, d je konstanta, obyčejně se používá 1, každá zpráva je pak označena celým vektorem
2. **R2**: při přijetí zprávy od uzlu k jsou provedené následující akce, vt_{msg} je vektor

hodin přijaté zprávy

(a) $vt_i[k] = max(vt_i[k], vt_{msg}[k])$ (aktualizace lokálních hodin)

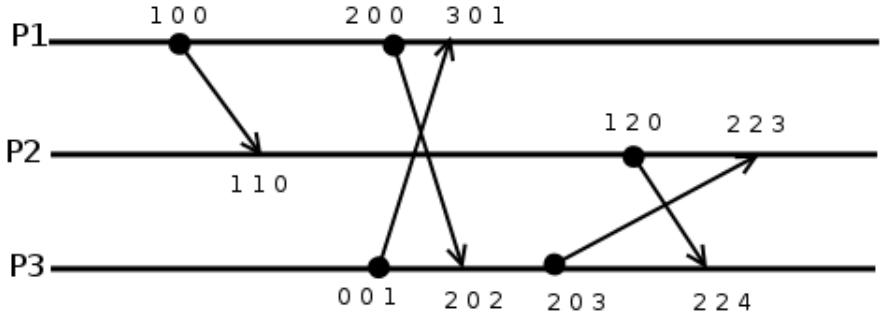
(b) provede se pravidlo **R1** (přiřazení lokálního logického času přijaté zprávě)

(c) zpracuje se daná zpráva

Funkci vektorových hodin popisuje názorně obr. 5.6. Porovnání dvou logických vektorových časových značek probíhá následujícím způsobem (vt_i a vt_j jsou dvě různé vektorové časové značky).

- události jsou shodné jestliže platí: $\forall x : vt_i[x] = vt_j[x]$
- událost (vt_i) nastala dříve nebo souběžně, pokud platí: $\forall x : vt_i[x] \leq vt_j[x]$
- událost (vt_i) nastala dříve, pokud platí: $vt_i \leq vt_j \wedge \forall x : vt_i[x] < vt_j[x]$
- události jsou konkurenční (souběžné), pokud platí: $\neg(vt_i < vt_j) \wedge \neg(vt_j < vt_i)$

Pokud jsou události souběžné, nelze spolehlivě určit jejich pořadí. V případě, že je třeba toto pořadí určit lze situaci řešit pomocí nastavení různých vah uzlům (například číselný identifikátor) kdy událost uzlu s vyšší váhou je prohlášena za předcházející, nebo je událost označena za neplatnou a zrušena, uzly ji poté zkusí opakovat a k souběhu by již dojít nemělo.



Obr. 5.6: Vektorové logické hodiny

5.2.3 Maticové hodiny

Dalším rozšířením vektorových hodin můžeme získat hodiny maticové, logické hodiny jsou reprezentovány maticí $n \times n$, nezáporných celých čísel. Uzel P_i spravuje matici označenou jako $mt_i[1..n, 1..n]$. Prvky matice pak mají následující význam:

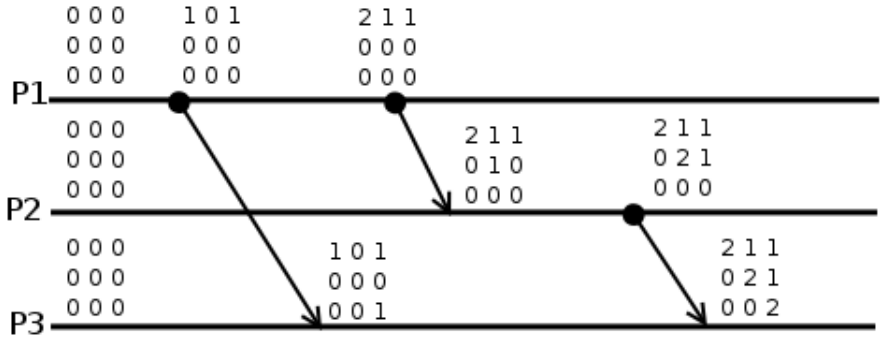
- $mt_i[i, i]$ reprezentují lokální události, čítač operací které se udály lokálně
- $mt_i[i, j]$ reprezentují poslední známý čítač lokálních hodin procesu P_j ($mt_j[j, j]$)
- $mt_i[j, k]$ reprezentují pohled na vztah lokálních hodin procesů P_j ($mt_j[j, j]$) a P_k ($mt_k[k, k]$)

Celá matice $mt_i[]$ pak reprezentuje pohled procesu P_i na globální logický čas distribuovaného systému. Jak bylo řečeno, každé logické hodiny musí obsahovat dvě

pravidla určující práci logických hodin:

1. **R1**: před provedením akce aktualizuje uzel své logické hodiny: $mt_i[i,i] = mt_i[i,i] + d$
2. **R2**: každá zpráva je označena maticovou časovou značkou mt_{msg}, když proces P_i obdrží zprávu od procesu P_j provede následující sekvenci úkonů
   (a) $1 \leq k \leq n$: $mt_i[i,k] = max(mt_i[i,k], mt_{msg}[j,k])$, to aktualizuje všechny prvky $mt_i[i,*]$ prvky přijatými od procesu P_j v časové známce mt_{msg}
   (b) $1 \leq k,l \leq n$: $mt_i = max(mt_i[k,l], mt_{msg}[k,l])$
3. zpracuje se daná zpráva

Maticové hodiny jsou striktně konzistentní, umožňují vždy rozhodnout, která událost se udála dříve.



Obr. 5.7: Maticové logické hodiny

5.2.4 Logický čas v KIVFS

Pro realizaci logických hodin byly zvoleny skalární logické hodiny implementované pomocí algoritmu absolutního řazení s hlasováním, kdy je každému požadavku přiřazena unikátní hodnota logických hodin.

6 GLOBÁLNÍ STAV

Globální stav distribuovaného systému se skládá z lokálních stavů každého uzlu (LS - stav proměnných, procesů, CPU, paměti atp.) a stavu komunikačních kanálů (CS - zpráv na cestě).

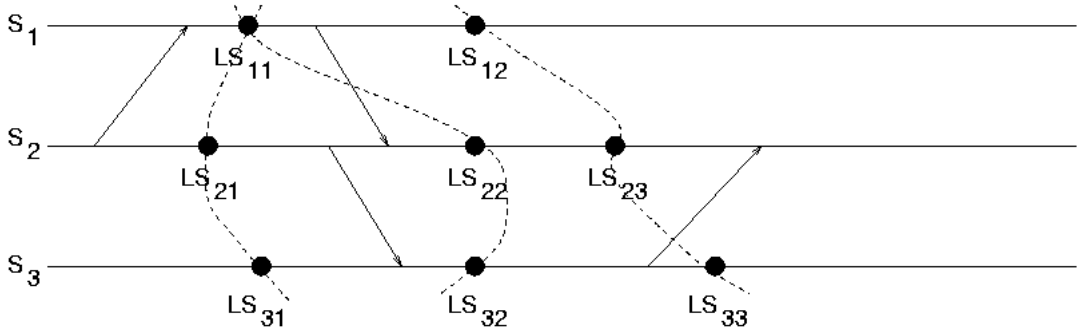
$$GS = \{\bigcup_{i} LS_i^{x_i}, \bigcup_{j,k} CS_{jk}^{y_j,z_k}\} \tag{6.1}$$

Lokální stav je označen jako LS_i^x, pokud jde o stav uzlu i v čase x po události e_i^x a před událostí e_i^{x+1}. Stav kanálů je definován jako všechny nezpracované odeslané a přijaté (na cestě) zprávy mezi uzly distribuovaného systému. Definován je také následovně pro kanál C_{ij} z uzlu P_i do P_j, e_i^x udává událost v uzlu i v čase x.

$$CS_{ij}^{x,y} = \{m_{ij} \mid send(m_{ij}) \leq e_i^x \wedge recv(m_{ij} \not\leq e_j^y)\} \tag{6.2}$$

Tento zápis označuje všechny zprávy m_{ij}, které byly odeslány po události e_i^x a do události e_j^y nebyly přijaté.

Algoritmy, které zjišťují globální stav distribuovaného systému, pracují s takzvanými snímky (snapshoty), které tvoří řez distribuovaným systémem, jak ukazuje obr. 6.1. Jedním z algoritmů pro zjištění globálního stavu distribuovaného systému je tzv. Chandy-Lamport algoritmus [13], který zde bude popsán. Tento algoritmus je určen pro FIFO kanály.



Obr. 6.1: Řezy distribuovaným systémem

Řez distribuovaným systémem je konzistentní, za následujících podmínek:

- **C1** $send(m_{ij}) \in LS_i \Rightarrow m_{ij} \in CS_{ij} \oplus recv(m_{ij}) \in LS_j$
- **C2** $send(m_{ij}) \notin LS_i \Rightarrow m_{ij} \notin CS_{ij} \wedge recv(m_{ij}) \in LS_j$

Pravidlo **C1** říká, že zpráva m_{ij} která je zaznamenána jako odeslaná v lokálním stavu LS_i, pak musí být zaznamenána jako součást stavu kanálu CS_{ij} nebo jako

součást lokálního stavu LS_j příjemce zprávy (zpráva se nemůže při tvorbě řezu ztratit). Pravidlo **C2** říká, že zpráva, která není zaznamenána jako odeslaná v LS_i, nemůže být součástí stavu kanálu CS_{ij} ani lokálního stavu příjemce LS_j (zprávy se nemohou jen tak v systému objevit).

6.1 Chandy-Lamport algoritmus

Chandy-Lamport algoritmus používá k zjištění globálního stavu značkovací zprávu (tzv. marker) a předpokládá FIFO organizaci kanálů. Kromě organizace kanálů předpokládá algoritmus další vlastnosti distribuovaného systému:

- v systému neexistuje selhání, každá zpráva dorazí příjemci právě jednou
- komunikační kanály jsou jednosměrné a organizované jako FIFO fronta
- mezi každými dvěma uzly existuje přímé spojení
- kterýkoli proces může inicializovat proces zjišťování globálního stavu
- zjištění globálního stavu nesmí ovlivnit běh systému
- každý uzel je schopen zaznamenávat svůj lokální stav a stav příchozího kanálu

Uzel, jenž chce inicializovat zjištění globálního stavu, vyšle ke všem dalším uzlům značkovací zprávu. Jakmile uzel obdrží značkovací zprávu zaznamená svůj lokální stav. Kanál, ze kterého značkovací zprávu obdržel, označí jako prázdný a odešle vlastní značkovací zprávu všem ostatním uzlům. Od této chvíle zaznamenává všechny zprávy na kanálech, dokud na nich neobdrží znovu značkovací zprávu.
Algoritmus lze snadno modifikovat pro více paralelních zjišťování globálního stavu (značkovací zpráva bude specifikovat, o který průběh algoritmu se jedná). Následuje přehlednější popis algoritmu.

1. uzel, který zjišťuje globální stav
   (a) zaznamená vlastní lokální stav
   (b) vyšle do všech odchozích kanálů značkovací zprávu
2. uzel, který obdrží značkovací zprávu poprvé na daném kanálu
   (a) zaznamená vlastní lokální stav
   (b) vyšle do všech odchozích kanálů značkovací zprávu
   (c) začne zaznamenávat stav kanálu
3. uzel, který obdrží značkovací zprávu z uzlu příslušící kanálu kam ji sám vyslal
   (a) ukončí zaznamenávání zpráv na příslušném kanálu
   (b) když se toto stane na všech kanálech, tak odešle zaznamenaný stav kanálů a zaznamenaný lokální stav uzlu, který proces započal
4. uzel, který inicializoval proces zjišťování globálního stavu, má všechny potřebné informace pro sestavení globálního stavu

Lze dokázat, že takto získaný řez (globální stav) je konzistentní následujícím způsobem. Algoritmus splňuje pravidla **C1** a **C2**. Pokud proces P_j přijme zprávu m_{ij} před přijetím značkovače, je tato zpráva součástí lokálního stavu, jinak je zaznamenána jako stav kanálu, tím je splněna podmínka **C1**. Organizace kanálu jako FIFO zaručuje, že žádná zpráva po značkovači nebude zapsána do lokálního stavu, ale do stavu kanálu, tím je splněna podmínka **C2**.

6.2 Využití v KIVFS

Pro potřeby obnovení uzlu KIVFS po selhání je třeba synchronizovat jeho stav a zařadit ho do skupiny. K tomu je využita znalost konzistentního řezu, kterým je obnovení realizováno. Po aplikaci záznamů zpráv z jiného uzlu je systém pozastaven systému vůči vnějšku a zprávy vnitřních kanálů jsou zpracovány, pak může být obnovený uzel zařazen (lokální stav uzlů je synchronní, vnitřní kanály a vnější jsou pozastaveny).

7 KONZISTENCE A JEJÍ MODELY

V distribuovaných systémech jsou data často replikována na více místech za účelem větší propustnosti systému a stabilitě v případě selhání některého z uzlů systému. Klasickým problémem replikovaných dat je jejich konzistence (je nežádoucí nabízet v jeden okamžik různá data pro stejný dotaz). To v praxi znamená zajistit, že pokud jsou měněna replikovaná data, pak musí existovat postup, jak změny propagovat a provést ve správném (stejném) pořadí na všech replikách. Podle způsobu propagace změn, rozlišujeme několik modelů konzistence. Základní dělení konzistenčních modelů je definováno podle pohledu na systém, pro kterého aktéra v distribuovaném systému musí být pohled na data konzistentní. Rozlišuje se pak konzistence z pohledu klientských aplikací a z pohledu dat. Následně lze dělit modely konzistence podle způsobu propagace změn dat. Dále budou popsány modely konzistence z pohledu dat.

7.1 Striktní konzistence

Striktní konzistence je z modelů konzistence nejsilnější, je definována následujícím pravidlem pro čtení a zápis v distribuovaném systému. Jakékoli čtení dat x vrací hodnotu odpovídající poslednímu zápisu dat x. To znamená, že všechny změny musí být zapisovány všude ve stejném čase a pořadí a čtecí operace na jakémkoli uzlu dostane vždy nejaktuálnější informaci. Jedná se o stejný princip jako je ACID v transakčním zpracování. Změny jsou propagované všude v jeden čas nebo v případě chyby vůbec.

7.2 Sekvenční konzistence

Sekvenční konzistence je slabší než striktní. Předpokládá řazení operací zápisu na všech uzlech ve stejném pořadí, ale nezaručuje jejich synchronní provádění. Mezi provádění operací na jednotlivých uzlech může vznikat zpoždění. Toto zpoždění je často označováno jako oblast nekonzistence, kdy operace čtení v jeden čas může vracet různé hodnoty dat. Toto zpoždění je minimálně doba nutná pro přenos zprávy mezi uzly. Lze implementovat s využitím atomického broadcastu.

7.3 Příčinná konzistence

Příčinná konzistence představuje slabší model než sekvenční. Představuje model, kde jsou operace řazeny za sebou v tom pořadí v jakém na sebe vzájemně působí,

ovlivňují se. Pokud se operace vzájemně neovlivňují, pořadí jejich změn není zaručeno. U operací je třeba určovat možnou příčinnou závislost a udržovat pak graf závislých operací.

7.4 FIFO konzistence

FIFO konzistence představuje model, ve kterém jsou zápisy každého uzlu viděny v tom pořadí, v jakém byly prováděny. Vzájemně však mohou být viděny různě, není určeno jejich vzájemné pořadí (na rozdíl od sekvenčního modelu). To pro implementaci představuje číslování operací z každého zdroje a jejich provádění ve stejném pořadí. Pokud lze předpokládat FIFO organizaci kanálů je implementace o to snadnější.

7.5 Slabá konzistence

Slabá konzistence zavádí využití synchronizačních operací, které jsou prováděny sekvenčně, řízené například logickými hodinami. Synchronizační proměnná představuje primitivum na kterém se propagují provedené změny pouze výsledkem poslední operace, tím je sledováno zjednodušení procesu a minimalizace komunikace. Mezioperace jsou viditelné pouze lokálně, až jejich výsledek je vidět globálně.

7.6 Uvolňovací konzistence

Tento model konzistence zavádí kritickou sekci, změny jsou propagovány v době opuštění této kritické sekce, před opuštěním musí být dokončeny všechny operace zápisu. Přístup ke kritické sekci musí být FIFO konzistentní a zavádí operace stejně jako v případě paralelního programování pro vstup (začátek práce se sdílenou proměnnou) a výstup z této sekce.

7.7 Přístupová konzistence

Přístupová konzistence je podobná uvolňovací, jejím hlavním rozdílem je, že zápis do sdílených proměnných musí být dokončen při vstupu do této sekce a zabrání výhradního přístupu.

7.8 Konzistence KIVFS

V systému KIVFS je třeba zaručit synchronní stav databází metadat, proto byla zvolena striktní konzistence implementovaná pomocí distribuovaných transakcí. Metadata musí být striktně konzistentní, protože se jimi řídí konzistence dat souborového systému.

8 TRANSAKCE V DISTRIBUOVANÝCH SYSTÉMECH

V této kapitole se bude práce věnovat transakcím v distribuovaných systémech. Jejich sémantikou a v prvé řadě řízením. Klíčovou vlastností distribuovaných systémů transakčního zpracování je, že transakce probíhá synchronně na všech uzlech systému a její výsledek se musí projevit na všech uzlech systému nebo nikde. Transakce musí splňovat ACID kritéria, která jsou na ní kladena. Pro potřeby této práce bude postačující zjednodušený pohled na transakce, pro které si nadefinujeme zjednodušující kritéria:

- předpokládá se, že požadavky na distribuovaný systém jsou synchronizované, tím je míněno, že každý požadavek má unikátní logickou časovou značku získanou pomocí algoritmů pro absolutní pořadí
- každý uzel systému obsahuje frontu požadavků k vyřízení, tyto požadavky budou zpracovávány jako transakce
- transakce pro uživatele probíhá jako synchronní operace (s dalším požadavkem čeká na její dokončení)
- uzel, který přijal požadavek je koordinátorem dané transakce

Transakční primitiva popsaná v kapitole 3.2.1 jsou základem transakčního zpracování. Každá transakce má následující životní cyklus:

- **Nová transakce** - koordinátor ohlásí ostatním uzlům začátek transakce
- **Bežící transakce** - transakce je podle instrukcí nebo zpráv koordinátora prováděna na uzlech DS
- **Končící transakce** - transakce je ukončena primitivem `KONEC_TRANSAKCE` pro její provedení (pokud je transakce úspěšně ukončena její změny jsou zveřejněny), nebo `ZRUSENI_TRANSAKCE` pro její zrušení, transakce nezměnila žádná data systému

O tom jestli je transakce úspěšně provedena na všech uzlech systému musí panovat jednoznačná shoda, hlasování o této shodě si řídí koordinátor dané transakce. K dispozici jsou algoritmy pro spolehlivé provedení transakce, jeden z nich je dvoufázový a druhý je třífázový.

8.1 Dvoufázové provedení

Dvoufázové provedení[14] distribuované transakce je jednoduší varianta než třífázový. Dvoufázové provedení popisuje obr. 8.1. V alogirmu vystupují následující aktéři

- koordinátor transakce, který řídí její běh a ukončení

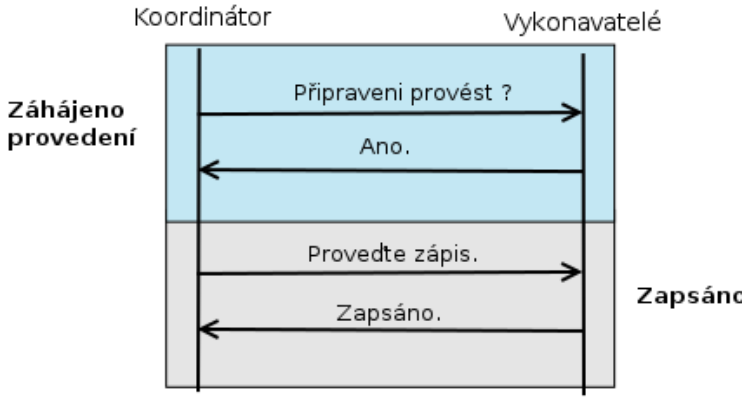
- uzly (vykonavatelé transakce), kde transakce probíhá

Průběh provedení transakce podrobně popisují následující odstavce.

První fáze V první fázi sbírá koordinátor potvrzení pro provedení (zapsání) dané transakce. Pokud jsou uzly připravené a nic nebrání provedení, všem zúčastněným uzlům pošle zprávu **připraveni k provedení**, následně koordinátor čeká dokud nedostane odpověď od všech uzlů. Uzly odpovídají pouhým **ANO** nebo **NE**.

Druhá fáze Ve druhé fázi samotného provedení transakce rozhodne koordinátor na základě přijatých odpovědí. Pokud jsou všechny odpovědi kladné, pak prohlásí transakci za úspěšnou a vyšle uzlům zprávu **provést**. Pokud je mezi odpověďmi záporné potvrzení je transakce prohlášena za neúspěšnou a koordinátor odešle všem zprávu **zrušení transakce**, čímž budou změny provedené v transakci zahozeny. Uzly poté ještě potvrdí koordinátorovi přijaté rozhodnutí a dokončení transakce.

Dvoufázové provedení je blokující dokud koordinátor neobdrží všechny potvrzovací zprávy od uzlů a zároveň jsou uzly blokovány. Tento nedostatek je řešen pomocí třífázového provedení, nebo hlasováním.



Obr. 8.1: Dvoufázové provedení

8.2 Třífázové provedení

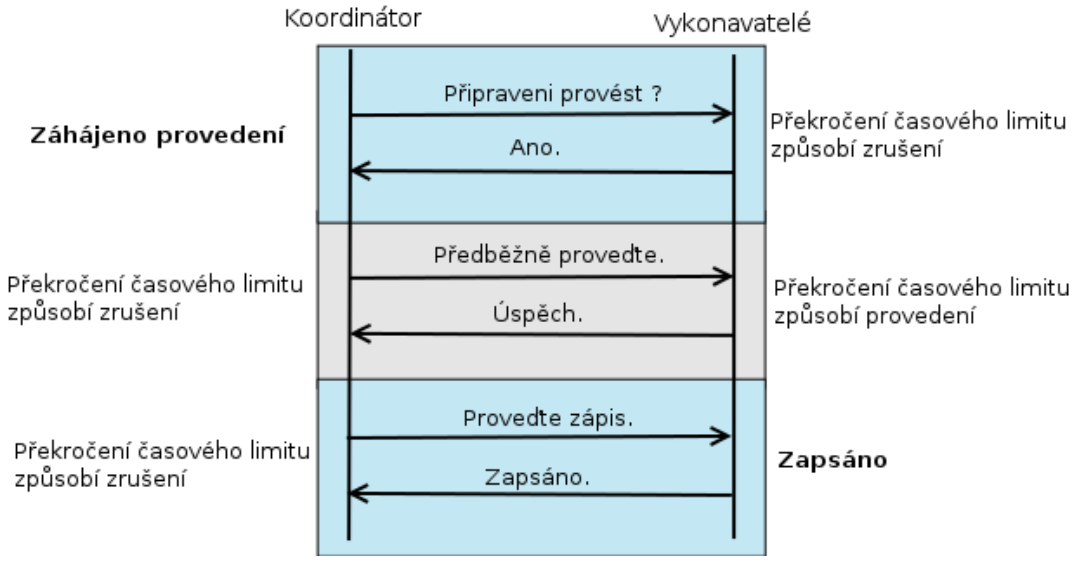
Třífázový algoritmus provedení[15] transakce vychází z dvoufázového, ale odstraňuje některé jeho nedostatky. Rozdíl je patrný z obr. 8.2. Třífázové provedení zavádí časové limity jako základní vlastnost.

První fáze Stejně jako v případě dvoufázového provedení posílá koordinátor zprávu **připraveni na provedení**, pokud tuto zprávu uzly neobdrží do konce časového limitu je transakce automaticky zrušena. Uzly opět odpovídají pouze **ANO** nebo **NE**, jestli jsou připraveni.

Druhá fáze Ve druhé fázi provedení rozhodne koordinátor předběžném zápisu na základě přijatých potvrzení. Stejně jako v předchozím algoritmu způsobí jediné záporné potvrzení o zrušení transakce. Pokud koordinátor od některého z uzlů neobdrží včas odpověď, prohlásí automaticky transakci za neúspěšnou a zruší ji. Pokud je výsledek kladný pošle koordinátor zprávu s požadavkem na předběžný zápis. Ztráta nebo zpoždění této zprávy znamená na straně uzlů kladný výsledek hlasování a změnu zapíší.

Třetí fáze Ve třetí fázi požaduje koordinátor potvrzení zápisu, pokud uzly neobdrží tuto žádost pak automaticky potvrzují zápis. Ztrátu tohoto potvrzení pak koordinátor chápe jako selhání a transakci ruší.

Výhodou tohoto způsobu provádění transakcí je definování časových limitů a automatická reakce systému na její překročení.



Obr. 8.2: Třífázové provedení

8.3 Modifikace dvoufázové metody s hlasováním

Protože se může počet uzlů systému měnit (vlivem odstávky, přidáním dalšího uzlu, přerušením spojení), byl navržen modifikovaný algoritmus dvoufázového provedení v systému KIVFS. Právo veta jediným uzlem bylo nahrazeno hlasováním, kdy hlas uzlu, který je odpojený je brán jako záporný, ale dokud je členem skupiny více než polovina serverů, kteří s provedením transakce souhlasí je transakce přesto provedena. Tím je odstraněn nedostatek zrušení transakce v případě selhání menšího než poloviční počtu uzlů. Uzly, které nebyly hlasování přítomny, nebo jejich odpovědi dorazily mimo časový limit musí projít procesem zotavení. Toto bude implementováno v rámci realizační části práce, kterou popisuje kapitola 14.3.

9 ZOTAVENÍ ZE SELHÁNÍ

V předchozí kapitole byly popsány algoritmy pro potvrzení konzistentního zápisu transakce v distribuovaném prostředí. Byl řešen i případ kdy od jednoho z uzlů nebo od koordinátora nedorazí některá ze zpráv, nebo dorazí se zpožděním větším než je definovaný limit. Ztrátu nebo zpoždění uzlu může způsobit nízká kvalita propojovacích linek, ztráta nebo pád některého uzlu. Distribuovaný systém proto musí obsahovat mechanizmy, které se s touto chybou budou schopny vyrovnat a umožní zotavení ze selhání (návrat uzlu zpět do systému). Dokud nebude uzel v konzistentním stavu nemůže se podílet na další práci v distribuovaném systému. Pro obnovení konzistentního stavu lze použít dva postupy, které je možné kombinovat, jeden je pomocí kontrolních bodů a druhý je pomocí zaznamenávání zpráv (žurnálu).

9.1 Zotavení pomocí kontrolních bodů

Kontrolní body slouží pro obnovu stavu po selhání nebo havárii distribuovaného systému. Jednotlivé uzly si mohou dělat vlastní kontrolní body, nebo je lze vytvářet koordinovaně a vytvářet kontrolní body schopné obnovit celý systém ve stejný čas. Kontrolní bod představuje zakonzervovaný stav uzlu nebo systému. Kontrolní bod celého distribuovaného systému je obdobou globálního stavu, konkrétně se jedná o trvale zapsaný globální stav. Jak již bylo zmíněno v kapitole o globálním stavu, globální stav distribuovaného systému je sjednocením lokálních stavů uzlů a kanálů. Kontrolních bodů může být obecně libovolný počet, systém pak musí být schopen z kteréhokoli kontrolního bodu obnovit svůj stav. Proto je zde důležitá konzistence kontrolních bodů, ta je definována stejně jako v případě globálního stavu. Proto se přirozeně jako nejvhodnější nabízí koordinovaný algoritmus kontrolních bodů, který vychází z algoritmu pro zjišťování globálního stavu. Jediným rozdílem je, že v případě zřizování kontrolního bodu se stav uzlů a kanálů nezasílá koordinátorovi, ale zapisuje se do trvalého úložiště pro možnost pozdější obnovy. Pokud jsou kontrolní body zřizované bez koordinace, je v případě obnovy nutné najít takovou množinu kontrolních bodů všech uzlů, aby dohromady tvořily konzistentní stav. Nekoordinovaný kontrolní bod může být zřizován různým způsobem, například periodicky po časových intervalech nebo počtu zpráv.

9.2 Zotavení pomocí záznamů

Další možností je zotavení systému pomocí přehrávání záznamů zpráv. Každá zpráva je zaznamenána na trvalé datové úložiště. V případě pádu je konzistentního stavu

docíleno přehráním záznamů z trvalé lokální paměti, nebo lze záznamy pořídit zasláním ze vzdáleného uzlu.

9.2.1 Pesimistické pořizování záznamů

Pesimistické pořizování záznamů, je postavené na základním pravidle, které říká, že žádná událost která je závislá na předchozí nesmí být provedena, dokud ta předchozí není zaznamenána. Z toho plyne, že zaznamenávání událostí a zpráv musí být synchronní a je pro další závislé události blokující, tzn. žádná další závislá zpráva nesmí být zpracována, pokud není původní zpráva zaznamenána. To může způsobit značné zpomalení systému, zvláště pak pokud záznamové médium není dostatečně rychlé pro zápis. Toto omezení je možné zeslabit použitím speciálního hardware, který podporuje rychlejší zápis, například SSD disky nebo výkonná disková pole.

9.2.2 Optimistické pořizování záznamů

Optimistické zaznamenávání implementuje v paměti dočasnou frontu pro zápis požadavků, které jsou připravené pro zápis na trvalé úložiště. Zápis záznamů pak probíhá periodicky nebo s možným zpožděním oproti provádění dalších operací, které tím nejsou blokovány. Předpokládá se, že k zápisu záznamu dojde dříve než k selhání uzlu. Pokud není tento předpoklad naplněn je třeba to algoritmicky řešit, protože nelze s jistotou určit kolik záznamů se nepodařilo zapsat a jaký je skutečný aktuální stav uzlu.

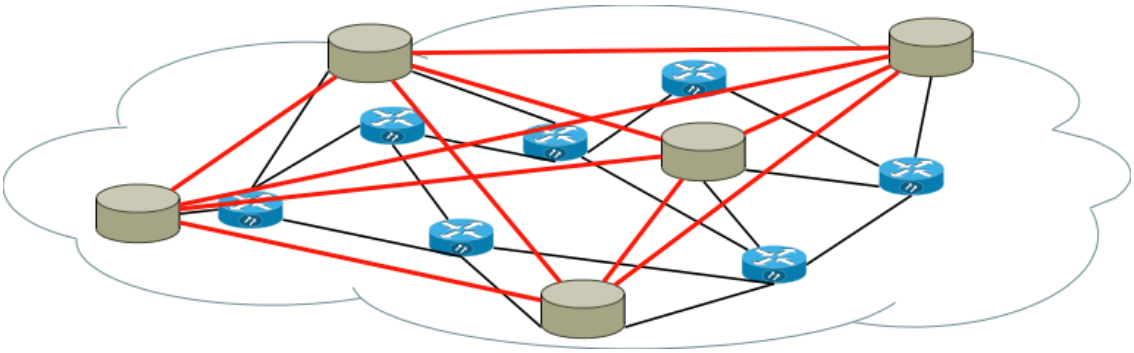
9.3 Zotavení v KIVFS

Při návrhu mechanizmu obnovy v KIVFS bylo přistoupeno k zotavení pomocí optimistického pořizování záznamů, pokud je těchto záznamů velké množství (proces obnovy by byl časově náročný), je možné ho urychlit obnovením databáze metadat ze zálohy jiného uzlu (obdoba kontrolního bodu), čímž dojde k redukci počtu zpráv nutných k obnově.

10 SMĚROVÁNÍ POŽADAVKŮ

Distribuované systémy mohou být svojí geografickou polohou rozsáhlé systémy s různými propojovacími linkami (různě smluvně dohodnuté linky, zahraniční transporty, obecně se jedná o technologicky heterogenní prostředí), které mohou mít i různé kvalitativní parametry. S touto praktickou překážkou se musí vyrovnat požadavek na co nejrychlejší doručování dat distribuovaného systému směrem k uživateli. To vede k potřebě nalézt nejvhodnější cestu pro doručování dat, čehož je možné docílit vytvořením překryvné sítě v rámci distribuovaného systému, která bude využívat kvalitnějších linek (obr. 10.1, černé trasy jsou fyzické, červené překryvné). Problém směrování požadavků je možné převést na problém hledání nejkratších cest v grafu.

Distribuovaný systém si lze představit jako orientovaný ohodnocený graf, kdy uzly distribuovaného systému tvoří uzly grafu a linky počítačové sítě tvoří ohodnocené orientované hrany. Hrany je vhodné považovat za orientované, protože linky mohou být asymetrické (směr od a do uzlu nemá stejnou přenosovou rychlost). To může být způsobeno technickým řešením propojení, nebo smluvním vztahem. I linky mezi jednotlivými síťovými operátory mohou být různé, jednotliví operátoři mají možnost spojení přes peeringová centra, často také využívají přímých propojů mezi sebou a někteří funguji jako transportní pro jiné, tedy data v počítačových sítích mohou k uzlu přicházet od ostatních z různých směrů, po různých linkách. Tyto linky vyhledáváme jako nejkratší cesty v grafu.



Obr. 10.1: Počítačová síť a její překrytí

10.1 Metrika cest

V prostředí distribuovaných systémů, nelze kvalitu propojovacích linek porovnávat pouze pomocí rychlosti, i když ta je zde stále jedním z klíčových parametrů. Distribuované systémy kladou velký důraz jak na rychlost, tak na odezvu linek, ale i na

jejich kvalitu. V distribuovaných systémech, kde probíhají synchronní operace, je odezva klíčovou vlastností, která pak ovlivňuje další množství parametrů systému jako celku. Může způsobit zpomalení systému až jeho zastavení z důvodu množících se příchozích požadavků ve frontě, které nemohou být kvůli pomalému synchronnímu zpracování odbavovány s dostatečnou rychlostí. Ze stejných důvodů je klíčová vlastnost i stabilita těchto linek. Stabilitou je rozuměn vývoj rychlosti a zpoždění v čase (v ideálním případě konstantní), ale i dostupnost dané linky. Z těchto parametrů je pak určována výsledná metrika spojení.

10.2 Algoritmy hledání nejkratší cesty

V teorii grafů existuje množství algoritmů pro výpočet nejkratší cesty. V této kapitole budou popsány vybrané algoritmy používané i jinými směrovacími systémy a určena jejich časová a paměťová náročnost. Vybrané algoritmy jsou těmi základními, další jsou většinou obdobné nebo obsahující pouze optimalizace pro specifické případy.

10.2.1 Dijkstrův algoritmus

Dijkstrův algoritmus[16] hledání nejkratších cest v kladně ohodnoceném grafu nalézá nejkratší cesty v grafu z uzlu v_0 do všech ostatních uzlů grafu. Algoritmus je popsán pomocí v pseudokódu jako algoritmus 10.1. Algoritmus pracuje s množinou vrcholů V, hran E a počátečním vrcholem s. Interně pracuje s množinou $T \subseteq V$, která obsahuje trvalé vrcholy (ty do kterých je nejkratší cesta již známa) a množinou $N \subseteq V$ doposud nenavštívených vrcholů. Hodnota $d[v]$ udává nejkratší délku cesty z s do v přes trvalé vrcholy (je horním odhadem nejkratší cesty). Algoritmus probíhá tak, že nejprve nastaví vzdálenost do všech vrcholů na nekonečno, kromě zdrojového vrcholu s a množinu N naplní všemi vrcholy. Dále prochází postupně všechny vrcholy z N (v pseudokódu vybírá vrchol s nejkratší vzdáleností) a hledá nejkratší vzdálenost ze všech sousedů (jako součet vzdálenosti do sousedního vrcholu v z s a vzdálenost k do v), která je uložena do pole vzdáleností. Algoritmus končí, když je množina nenavštívených vrcholů prázdná. Z algoritmu lze odvodit nejhorší časovou složitost nalezení nejkratších cest $O(\mid V^2 \mid)$. Paměťová náročnost algoritmu je velikost incidenční matice V^2 a uložení cest V^2 v nejhorším případě celkem $2V^2$.

10.2.2 Floyd-Warshall algoritmus

Oproti Dijkstrovu algoritmu nalézá tento algoritmus [17] nejlepší cesty mezi všemi vrcholy grafu. V každém svém průchodu nalézá nejkratší cesty mezi všemi vrcholy

```
function Dijkstra(E, V, s):
   for each vertex v in V:
      d[v] := infinity
   d[s] := 0
   N := V
   while N is not empty:
      u := extract_min(N)
      for each neighbor v of u:
         alt = d[u] + l(u, v)
         if alt < d[v]
            d[v] := alt
```

Algoritmus 10.1: Dijsktrův algoritmus - pseudokód

a zpřesňuje odhad nejkratší cesty, dokud není zřejmé, že nalezená cesta je skutečně nejkratší , zastavovací podmínkou je tak v nejhorším případě počet cyklů roven počtu uzlů. Algoritmus je opět popsán pseudokódem jako algorimus 10.2. Algoritmus pracuje s maticí *path* velikosti $N \times N$ ($path[i, j]$ je hodnota nejkratší cesty z i do j) a funkcí $edgeCost(i, j)$, která vrací ohodnocení hrany z vrcholu i do vrcholu j. Základní hodnoty matice *path* jsou nastaveny pomocí funkce $edgeCost(i, j)$, $path[i, j] = edgeCost(i, j)$. V každém cyklu pak nastavuje hodnoty matice $patch[i, j]$ na kratší vzdálenost pokud je možné se do vrcholu j z i dostat kratší cestou přes vrchol k. Jinými slovy algoritmus v každém cyklu zkouší cestu přes větší počet vrcholů. Z algoritmu lze odvodit časovou složitost výpočtu nejkratších cest $O(\mid V^3 \mid)$ a paměťová náročnost je v nejhorším případě V^3, protože u každého vrcholu v matici cest je třeba si pamatovat posloupnost vrcholů, přes které vede cesta.

```
int path[][];
function Floyd-Warshall():
   for k := 1 to n
      for i := 1 to n
         for j := 1 to n
            path[i][j] = min ( path[i][j], path[i][k]+path[k][j] );
```

Algoritmus 10.2: Floyd-Warshall algoritmus - pseudokód

10.2.3 Bellman-Ford algoritmus

Předchozí algoritmy měly jako podmínku kladně ohodnocené hrany, Bellman-Ford [18] algoritmus umí pracovat i s grafy, kde jsou hrany ohodnoceny záporně a hledá nejkratší cestu z vrcholu s do ostatních vrcholů. Tento algoritmus je podobný Dijkstrovu algoritmu. Algoritmus je opět uveden v pseudokódu jako algoritmus 10.3. Nejprve

jsou nastaveny počáteční hodnoty vzdáleností, následuje zkoumání všech hran zda není možné se do vrcholu v dostat lepší hranou z vrcholu u. Následuje vyhledávání záporných cyklů. Pokud graf obsahuje záporný cyklus, je možné opakovaným počtem jeho průchodů získat stále kratší cestu. Z algoritmu lze odvodit časovou složitost výpočtu nejkratších cest $O(\mid E \mid\mid V \mid)$, paměťová náročnost je pak stejná jako u Dijkstrova algoritmu $2V^2$.

```
function Bellman-Ford(E, V, s):
   for each vertex v in V:
       if v=s then v.distance := 0
       else v.distance := infinity
       v.predecessor := null
   for i from 1 to size(V)-1:
       for each edge uv in edges:
           u := uv.source
           v := uv.destination
           if u.distance + uv.weight < v.distance:
               v.distance := u.distance + uv.weight
               v.predecessor := u
   for each edge uv in edges:
       u := uv.source
       v := uv.destination
       if u.distance + uv.weight < v.distance:
           error "Graph contains a negative cycles"
```

Algoritmus 10.3: Bellman-Ford algoritmus - pseudokód

10.2.4 Použití v KIVFS

Z uvedených algoritmů byly pro implementaci vybrány Dijkstrův a Floyd-Warshallův. Bellman-Ford algoritmus se od Dijkstrova neliší výpočetní ani paměťovou složitostí, přináší pouze možnost hledat cesty i v grafu se záporně ohodnocenými hranami, která je pro využití v KIVFS zbytečná, protože cesty budou vždy kladně ohodnocené.

11 DISTRIBUOVANÝ SOUBOROVÝ SYSTÉM

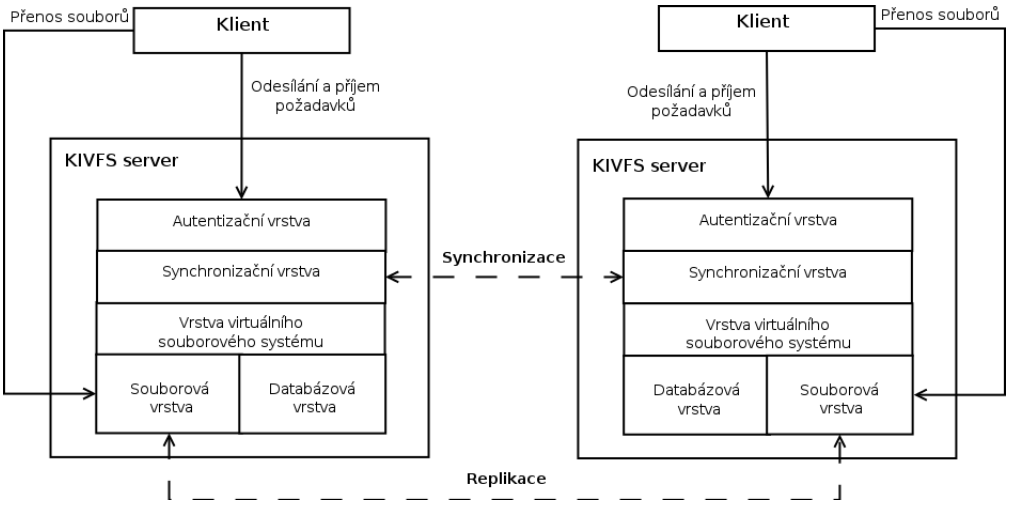
Distribuovaný souborový systém je zvláštní případ distribuovaných systémů. Podle kategorií které byly definovány v kapitole 3, je zařazen mezi distribuované informační systémy. Nabízí vzdálený přístup k souborů, které mohou být fyzicky rozložené po různých uzlech, ale tuto skutečnost překrývá a nabízí je vůči uživateli transparentně jako jeden zdroj - adresářový strom. V rámci uzlů distribuovaného systému mohou probíhat replikace dat, migrace a zálohování. Pro uchování informací o struktuře a umístění dat obsahuje obyčejně databázi metadat systému s jejichž pomocí řídí práci s daty a další procesy v DFS (replikace, migrace). Tato databáze může být distribuovaná, nebo centralizovaná s replikami. Praktická část diplomové práce popisuje implementaci mechanizmů, které zajišťují synchronní replikaci metadat souborového a požadavků systému KIVFS. Distribuovaný systém obyčejně implementuje také systém autentizace a autorizace uživatelů. K tomu implementuje vlastní mechanizmy nebo využívá již existující (LDAP [20], Kerberos [19]).

12 KIVFS

KIVFS je experimentální distribuovaný souborový systém vyvíjený na katedře informatiky a výpočetní techniky. Při jeho implementaci a návrhu byly využity poznatky z výše popsaných kapitol. Distribuovaný souborový systém lze v obecných distribuovaných systémech klasifikovat jako systém distribuované trvalé paměti (trvalé protože si zachovává stav i po restartu systému nebo v případě jeho vypnutí). Dále pak lze KIVFS klasifikovat jako systém transakčního zpracování, který obsahuje funkci synchronizace a replikace požadavků.

Distribuované transakční zpracování je motivované potřebou mít stav metadat na všech uzlech ve shodném stavu. Z toho plyne i model konzistence dat, operace s metadaty KIVFS jsou striktně konzistentní, protože jejich stav je závislý na operaci přidělování sekvenčně generovaných identifikátorů záznamů (k tomu by dostačovala i sekvenční konzistence), ale je třeba také zaručit, že operace budou provedeny na všech aktivních uzlech, což je možné jen v případě striktní konzistence.

Potřeba synchronizace a implementace logických hodin byla uvedena v kapitole 5. Pro řízení přístupu k metadatům souborového systému je třeba znát pořadí požadavků, aby bylo možné data zrekonstruovat pomocí stejné posloupnosti požadavků. Pomocí logických hodin je možné určit stav daného uzlu v době selhání a počet požadavků o který se systém liší od ostatních uzlů. Na základě této informace je možné řídit obnovu zasláním rozdílových zpráv.



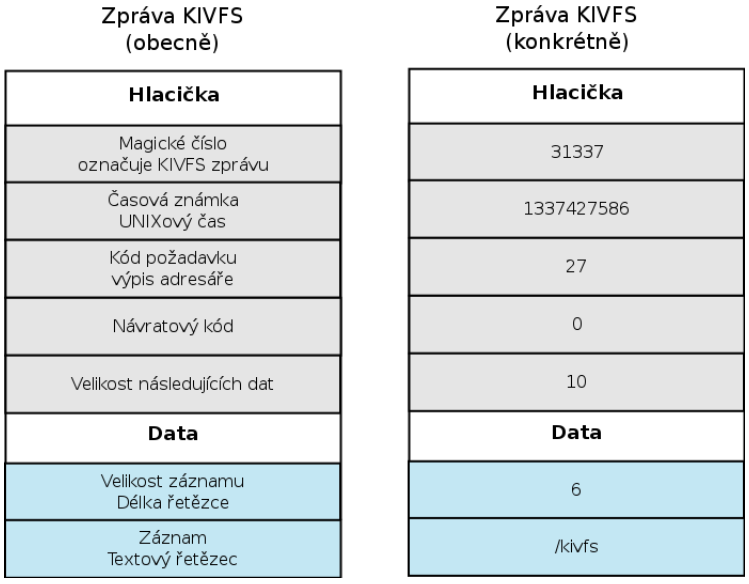
Obr. 12.1: Schéma KIVFS

12.1 Architektura KIVFS

Distribuovaný souborový systém KIVFS je implementován jako skupina serverů (nazývané také jako vrstvy) běžících na všech uzlech, které poskytují služby souborového systému. Každý server má svoji specifickou funkci. K implementaci pomocí samostatných serverů bylo přistoupeno z důvodu snadnější implementace, kdy každý server je při svém vývoji nezávislý na ostatních. Dále to dává tvůrcům volnost volby implementace jednotlivých služeb a porovnání jejich funkce, kvalitativních a výkonnostních parametrů.

Jednotlivé servery mezi sebou komunikují pomocí síťových spojení vlastním protokolem nad protokolem TCP/IP. Struktura KIVFS je naznačena na obr. 12.1, kde jsou zobrazené jednotlivé komponenty. Serverové části jsou naprogramovány v jazyce C, ale díky nezávislému protokolu zpráv není tímto programovacím jazykem implementace omezena, což je demonstrováno existujícími klientskými aplikacemi, zvláště pak mobilními (v jazycích C#, Java a Objective-C).

Struktura přenášených dat pomocí zpráv je zobrazena na obr. 12.2. Serverová část je určena k běhu na operačním systému GNU/Linux, konkrétně na distribuci Debian, její implementace je psána ale dostatečně obecně, aby ji bylo možné bez větších obtíží upravit pro jiný UNIXový systém. Servery poskytující jednotlivé funkce KIVFS jsou programované kompletně v uživatelském prostoru, tím nevzniká přílišná závislost na daném operačním systému. Jednotlivé komponenty mají závislosti pouze na použitých knihovnách, například pro autentizaci nebo práci s relační databází.



Obr. 12.2: Zpráva KIVFS

Každá komponenta plní zadanou funkčnost souborového sytému. Funkce jednotlivých komponent, které jsou logicky uspořádané podle obr. 12.1, jsou následující:

- **Autentizační vrstva** má za úkol autentizaci uživatele a zabezpečenou komunikaci s klientskými aplikacemi. Pro zabezpečení přenosu používá protokol SSL. Autentizace uživatelů probíhá skrze systém Kerberos. Po bezpečném ověření pomocí služby Kerberos spravuje tato vrstva sezení autentizovaných uživatelů, do zbytku systému propaguje identifikátor autentizovaného sezení, kterým se následně prokazují všechny požadavky na souborový systém.
- **Synchronizační vrstva** operuje napříč celou skupinou serverů KIVFS, synchronizuje požadavky na změnu metadat souborového systému i zprávy systému spravující repliky, zámky a další interní struktury. Řídí také obnovu uzlů po návratu do skupiny. Další úlohou této vrstvy je sbírání informací o dostupnosti jednotlivých serverů a hledání nejkratších cest podle zvolených metrik a jejich předání vrstvě virtuálního souborového systému.
- **Vrstva virtuálního souborového systému** představuje abstrakci souborového systému, neimplementuje žádnou skutečnou funkčnost, ale slouží jako prostředník, který přerozděluje požadavky mezi podřízené vrstvy.
- **Databázová vrstva** má za úkol uchovávat metadata souborového systému, metadaty se v tomto kontextu rozumí kompletní adresářová struktura, atributy jednotlivých souborů, mapování virtuální adresářové struktury na skutečné soubory, informace o aktuálnosti a dostupnosti replik, informace o svazcích, ze kterých se souborový systém skládá a další
- **Souborová vrstva** řídí ukládání dat na lokální systém souborů jednotlivých uzlů do určených svazků, poskytuje uživateli přímý přístup k datům, provádí replikaci souborů a řídí jejich přenos, šifrování a deduplikaci dat.

Komponenty KIVFS komunikují s okolním světem i mezi sebou pomocí zpráv. Spojení je vždy inicializováno klientem, žádná z komponent nenavazuje spojení směrem ke klientovi, to umožňuje použití KIVFS i na klientech, kteří jsou technologicky omezeni v přijímání spojení z vnější sítě. Toto omezení může způsobovat například firewall, nebo překlad adres v privátních sítích (NAT), který je velice častý například u mobilního připojení.

13 REALIZACE

Zadáním diplomové práce bylo nastudovat problematiku synchronizace požadavků a jejich trasování v prostředí distribuovaných systémů, z tohoto zadání vyplynuly i další implementované funkčnosti.

V rámci práce byl implementován modul (samostatná služba, middleware) zajišťující zadanou funkčnost. Modul je v architektuře KIVFS zasazen mezi autentizační vrstvou a vrstvou virtuálního souborového systému. Hlavní úlohou tohoto modulu (v KIVFS nazýván synchronizační vrstvou) je zajištění synchronního a konzistentního stavu metadat souborového systému, zajištění vzájemného vyloučení, distribuce zámků, dále pak hledání nejvhodnějších cest při přístupu uživatelů k souborům. Ze zadání vyplynula implementace distribuovaných transakcí, zaznamenávání operací v DFS a následné zotavení po selhání. V kontextu práce se selháním rozumí, plánované i neplánované opuštění distribuovaného systému a zotavením následný návrat do tohoto systému v plně funkčním a synchronním stavu.

13.1 Knihovna libkivfscore

Pro snadnou implementaci jednotlivých serverových komponent KIVFS, tak i klientských aplikací byla ve spolupráci s kolegy vytvořena obecná knihovna `libkivfcore`, kde je implementována veškerá společná funkčnost. Knihovna je používána pro implementaci jak serverů KIVFS (je součástí jejich zdrojových kódů `server/core`), tak pro implementaci klientských aplikací a ovladačů souborového systému. Knihovna implementuje komunikaci jednotlivých komponent KIVFS pomocí zasílání zpráv, jsou v ní definovány používané datové struktury a abstraktní datové typy pro práci s daty distribuovaného souborového systému. Knihovna je složena z několika částí, jejich výčet včetně popisu je následující:

- **definice požadavků** - tato část knihovny definuje sadu možných příkazů KIVFS, odpovědi a chybové stavy
- **funkce pro práci se zprávami** - vytvoření zprávy, zrušení zprávy, jejich zasílání po síti nebo zpracování zprávy na interní datové typy a naopak
- **správa sezení** - umožňuje spravovat autentizovaná sezení a jejich kontextové informace (příslušný socket, autentizační informace a další)
- **správa vláken** - nabízí pohodlnou infrastrukturu pro práci s vlákny, která jsou v KIVFS využívána pro obsluhu požadavků
- **definice datových typů** - které lze zasílat pomocí zpráv, nebo které se používají interně v jednotlivých vrstvách a klientech
- **zpracování konfigurace** - to umožňují knihovní funkce pro zpracování konfiguračního souboru a načtení potřebných voleb

- **záznamy aplikace** - ty jsou pořizované pomocí vlastních funkcí, které umožňují zasílat zprávy do `syslog` démona, zapisovat do souboru nebo vypisovat na obrazovku, pokud běží jednotlivé vrstvy na popředí

13.1.1 Příklad užití knihovních funkcí

Následuje komentovaný příklad použití knihovních funkcí13.1 pro zaslání lokálních metrik trasovací části, který představuje použití funkcí `libkivfscore` v praxi. V ukázce jsou použity jak funkce pro práci s abstraktními datovými typy (`kivfs_create_list`, `kivfs_append_to_list`...), tak funkce pro vytváření zpráv pomocí formátovacích řetězců (`kivfs_request`) a zasílání těchto zpráv (`kivfs_send`), včetně funkcí pro uvolňování paměti po těchto strukturách.

```
#include <kivfs.h>
...
// vytvoří obecný prázdný seznam
routes = kivfs_create_list();
// iteruje přes cílové uzly
for (i_hop = hops; i_hop != NULL; i_hop = i_hop->next) {
   // kopie informací o uzlu, identifikátor a ip adresa
   id = i_hop->id;
   ip = strdup(i_hop->ip);
   // vytváří položku seznamu tras s ohodnocením bez dalších skoků
   // protože se jedná o lokální ceny linek
   route = kivfs_route(kivfs_route_node(id, 1, ip), \\
                       i_hop->sumCost, kivfs_create_list());
   // přidá položku do seznamu
   kivfs_append_to_list(routes, route);
}
// vytvoří zprávu z připravených dat
reply = kivfs_request(NULL_ID, KIVFS_ROUTE_PULL_REPLY, "%routes", routes);
// odešle zprávu klientovi
rCode = kivfs_send(sockFd, reply);
// uvolní paměť zprávy
kivfs_free_msg(reply);
// zruší seznam v paměti
kivfs_destroy_list(routes, kivfs_free_route);
```

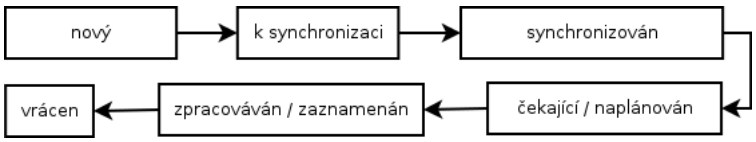
Obr. 13.1: Zaslání lokálních metrik serveru

14 SYNCHRONIZAČNÍ VRSTVA KIVFS

Hlavní úlohou synchronizační vrstvy KIVFS je zajištění konzistentního a synchronně replikovaného stavu metadat souborového systému. Jak již bylo řečeno v kapitole o architektuře KIVFS synchronizační vrstva pracuje na všech uzlech před vrstvou VFS, DB a FS. Zároveň se synchronizací (řízením logických hodin) dochází k replikaci požadavků. Metadata na všech uzlech musí být konzistentní a shodná, protože podle aktuálního stavu se odvozuje stav příští, ovlivňuje data pro příští transakce (např. generátor interních identifikátorů).

Práce a přístup k metadatům implementuje a řídí databázová vrstva. Metadaty se pak řídí přístup k souborům (čtení, zápis, zámky), fungování klientských vyrovnávacích pamětí, identifikátory souborů nebo repliky vlastních dat. Replikace souborových dat je řešena v implementaci souborové vrstvy a využívá služeb synchronizační vrstvy. Z výše uvedených důvodů plyne zvolený model konzistence. Při návrhu synchronizační vrstvy byl zvolen striktní konzistenční model, který je realizován pomocí synchronních distribuovaných transakcí, jak bude popsáno dále.

Kromě prosté synchronizace požadavků a jejich provádění zavádí synchronizační vrstva podporu zotavení po pádu uzlu a implementuje trasovací mechanizmy a algoritmy pro zjišťování nejlepších tras, které pak předává serveru souborového systému pro poskytování dat. Uvnitř synchronizační vrstvy jsou jednotlivé funkčnosti implementovány sadou vláken, jak bude popsáno dále. Každý požadavek procházející přes tuto vrstvu má životní cyklus podle obr. 14.1.



Obr. 14.1: Životní cyklus požadavku v KIVFS

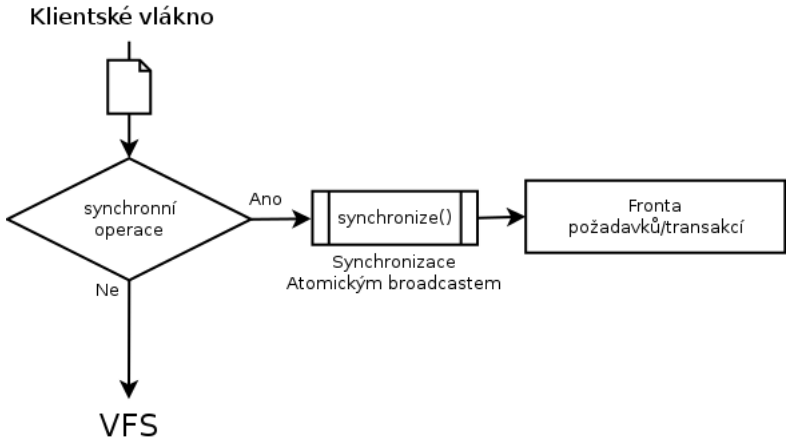
14.1 Synchronizace

K synchronizaci požadavků dochází použitím algoritmu Lamportových logických hodin s hlasováním, který implementuje skalární logické hodiny. Všechny požadavky, které mění metadata souborového systému a pracují se zámky musí probíhat synchronně na všech uzlech KIVFS, tak aby databázová data byla striktně konzistentní. Pro synchronizaci požadavků je vyhrazeno jedno vlákno, které distribuuje požadavky ostatním uzlům, další vlákno je vyhrazeno pro aktualizaci logických hodin a příjem požadavků z ostatních uzlů. Implementace je naznačena na obr. 14.2.

Pro každého připojeného klienta se spouští vyhrazené vlákno, které předává požadavky k synchronizaci nebo je předává přímo VFS serveru. Pokud je přes toto vlákno přijat požadavek na práci s metadaty, je tento požadavek synchronizován (zařazen do fronty požadavků) s ostatními uzly. Klientské vlákno je uspáno dokud není požadavek proveden na všech uzlech, teprve poté je klientské aplikaci vrácen výsledek dané operace.

Všechny operace jsou zpracovávány jako transakce, což bude popsáno v následující kapitole. Průběh vyjednání logické časové značky prezentuje algoritmus 14.1 publikovaný na konferenci In Informatics 2009[21].

Uzel který obdrží požadavek od klientské aplikace, si do pracovní proměnné uloží aktuální hodnotu logických hodin a osloví ostatní uzly s požadavkem na synchronizaci (vyjednání časové značky požadavku a uložení do lokálních front), pošle všem svůj návrh časové značky. Ostatní uzly pokud je časová značka vyšší odpoví kladně, jinak záporně. Uzel vyžadující synchronizaci poté sečte přijeté hlasy, pokud přijal nadpoloviční počet kladných potvrzení přiřadí požadavku navrhovanou časovou značku a potvrdí ji i ostatním uzlům. V případě, že časová značka není potvrzena proces se opakuje. Pokud se nepodaří požadavek zařadit do určitého počtu pokusů je klientské aplikaci oznámena chyba.



Obr. 14.2: Přijetí požadavku v KIVFS

14.2 Transakce a záznamy

Po synchronizaci požadavku (přidělení logické časové značky) je požadavek zařazen do fronty požadavků `requestQueue`, viz výpis na obr. 14.4, které mají být synchronně provedeny. Každý synchronizovaný požadavek je z této fronty vyzvednut

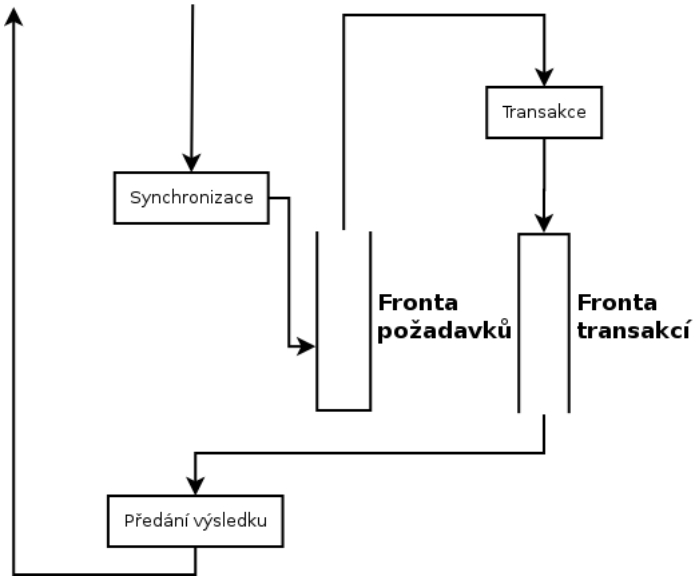
```
SYNCHRONIZACE_POKUD_JE_POŽADAVEK_PŘIJAT_LOKÁLNĚ:
serverId = 1;
begin
   recvFromClient (request);
   timeTmp = time + 1;
point START:
   int ack = 0;
   for server in (server2 server3 ...)
      timeHelp = sendQueryTime (server, timeTmp)
      if (timeHelp == timeTmp)
         ack++;
      else-if (timeTmp != timeOut)
         timeTmp = timeHelp
         goto START
   if (ack > (serverCount mod 2))
      time = timeTmp
      push(request, time)
      sendRequest(request, time)
   else
      sendAbort(request)
end

SYNCHRONIZACE_POKUD_JE_POŽADAVEK_ZE_VZDÁLENÉHO_UZLU:
serverId = 2 or 3 or ...
begin
   recvQueryTime(serverTmp, timeTmp)
   if (time < timeTmp)
      return timeTmp
   else
      return time+1
   recvRequest(request, timeTmp) or recvAbort()
end
```

Algoritmus 14.1: Synchronizace

jediným vyhrazeným transakčním vláknem. Požadavek je zpracováván jako transakce, aby nebyla databáze metadat uvedena do nekonzistentního stavu. Vzhledem k tomu, že je transakcím vyhrazeno speciální vlákno jsou transakce v rámci každého uzlu prováděny sériově v daném pořadí podle časové značky logických hodin.

Uzel, který přijal požadavek přímo od klientské aplikace je automaticky koordinátorem transakce. Koordinátor osloví ostatní uzly zprávou `ZAČÁTEK_TRANSAKCE` s časovou značkou a čeká na potvrzení zahájení transakce na metadatových serverech. Následně každý uzel provede daný požadavek. Koordinátor transakce následně zjistí s jakým návratovým kódem skončili operace na lokálních uzlech pomocí zprávy `PŘEDBĚŽNÉ_PROVEDENÍ`. Pokud byl návratový kód chybový v nadpoloviční většině zahájí zrušení transakce zprávou `ZRUŠENÍ_TRANSAKCE`, v opačném případě potvrdí transakci na všech uzlech `POTVRZENÍ_TRANSAKCE`. Pokud se nepodaří započetí transakce je bez vykonání požadavku zrušena pomocí `ZRUŠENÍ_TRANSAKCE`.

Podpora transakčních primitiv je implementována v databázové vrstvě, která



Obr. 14.3: Fronty pro synchronizaci v KIVFS

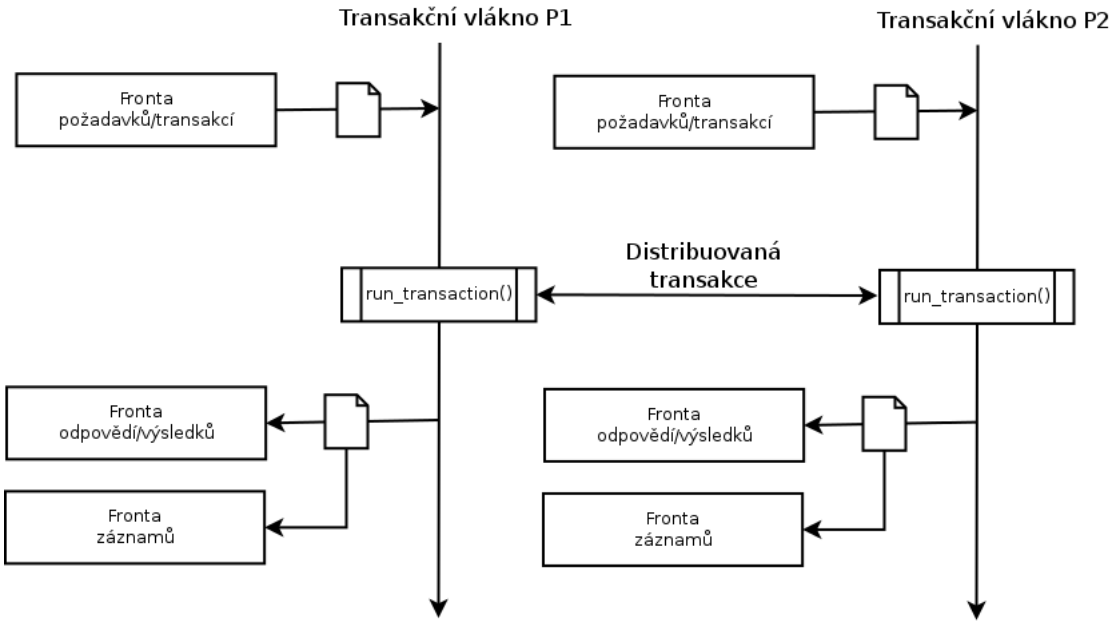
tuto funkci zprostředkovává synchronizační vrstvě, využívá k tomu služeb relační databáze (aktuálně konkrétně MySQL), nad kterou je realizována. Uzly, které transakci provádí před přeposláním požadavku na vrstvu virtuálního souborového systému, zašlou žádost o začátek transakce, která je převedena na transakci uvnitř relační databáze a po vykonání požadavku ji potvrdí nebo zruší.

```
struct orderQueueItem {
   int id;
   int server;
   kivfs_msg_head_t *head;
   void *data;
   struct orderQueueItem *next;
   struct orderQueueItem *prev;
};
```

Obr. 14.4: Položka fronty pro požadavky

Po skončení zpracování je výsledek (odpověď) požadavku zařazená do fronty provedených požadavků, odkud si ji vyzvedne původní klientské vlákno a výsledek přepošle zpět klientské aplikaci (obr. 14.3). Proces provedení transakce a její záznam zobrazuje obr. 14.5. Zaznamenávání zpráv je realizováno optimisticky, pokračování dalších transakcí není blokováno do doby zápisu do trvalé paměti (relační databáze). Pro zápis je vyhrazeno samostatné vlákno, které čeká nad frontou záznamů a požadavky zapisuje přes VFS vrstvu pomocí databázové vrstvy do trvalé paměti relační databáze. K optimistickému zaznamenávání bylo přistoupeno, aby nebyl sys-

tém synchronizace blokován dalšími operacemi. Použitím vyhrazeného vlákna se pak minimalizuje zpoždění při zápisu oproti provedení transakce.



Obr. 14.5: Transakční zpracování v KIVFS

14.3 Zotavení po selhání

Zotavení po selhání systému je implementováno pomocí přehrávání záznamů zpráv. Jak bylo popsáno v kapitole výše, každá provedená transakce je zaznamenána lokálně pomocí databázové vrstvy. V případě pádu uzlu je pak možné požádat ostatní uzly o přehrání zpráv, pomocí kterých se zotavující uzel dostane do synchronního stavu se zbytkem systému. Uzel nesmí být vpuštěn mezi ostatní pokud není synchronní. Mohlo by tak dojít k nekonzistentnímu stavu metadat souborového systému obnovovaného ale i dalších uzlů. Zde je využito poznatků o globálním stavu DS, konzistentního řezu, systém je při obnově na krátkou dobu zastaven, aby byla provedena konzistentní obnova.

Při startu synchronizační vrstvy si zjišťuje každý uzel logický čas poslední transakce z databázové, tento čas použije jako svůj výchozí pro obnovu. V případě přidání nového uzlu si tento uzel přehraje z výchozího stavu všechny záznamy od začátku. Tento proces je možné zkrátit použitím zálohy metadat jiného bežícího uzlu.

Pro zotavení se používá vyhrazené vlákno, které slouží k přehrání záznamů zpráv z relační databáze a nabízí další operace pro zapojení uzlu zpět do skupiny (zjištění poslední zaznamenané zprávy, zastavení příjmu nových zpráv). Zotavující se

uzel pošle ostatním uzlům požadavek s časovou značkou poslední provedené transakce a získá zpět poslední zaznamenanou časovou značku vzdáleného uzlu. Pokud se značky mezi sebou liší vyžádá si rozdílové záznamy od uzlu s nejvyšší časovou značkou, které lokálně aplikuje a zaznamenává. Následně opět zjišťuje časové značky ostatních serverů a maximální odchylku od vlastní poslední časové značky.

Tento proces se opakuje dokud není rozdíl roven nule. Následně je započat proces včlenění do skupiny. Obnovující se server pošle požadavek ostatním na zastavení příjmu nových požadavků a dokončení rozpracovaných. Opět se opakuje proces přehrání záznamů od uzlu s nejvyšší časovou značkou, aby se aplikovaly i do té doby rozpracované požadavky, které jsou teprve ve frontách nebo v procesu synchronizace. Když je proces dokončen, je vyslána zpráva s požadavkem na spojení. Dokud nejsou všechna spojení navázána, nejsou nové požadavky přijímány. Po navázání spojení je vyslán požadavek obnovení přijímání nových požadavků. Proces zotavení reprezentuje algoritmus 14.2. Tím je dosaženo připojení v konzistentním globálním stavu, požadavky klientů jsou zadrženy v komunikačních kanálech, synchronizační a transakční kanály jsou vyprázdněné a stav metadat je na všech serverech shodný, viz. kapitola 6 popisující globální stav systému.

```
function REPLAY_LOG:
while (timeDiference NOT 0)
   localTime = getMaxTime()
   maxTime = localTime
   for server in (server2 server3 ...)
      tmpTime = getMaxTime(server)
      if (tmpTime > maxTime)
         maxTime = tmpTime
   getLogMessages(server, localTime, maxTime)
   localTime = getMaxTime()
   for server in (server2 server3 ...)
      tmpTime = getMaxTime(server)
      if (tmpTime > maxTime)
         maxTime = tmpTime
timeDiference = maxTime - localTime

REPLAY LOG
for server in (server2 server3 ...)
   stopNewRequest(server)
REPLAY_LOG
for server in (server2 server3 ...)
   connectRequest(server)
for server in (server2 server3 ...)
   connect(server)
for server in (server2 server3 ...)
   startNewRequest(server)
```

Algoritmus 14.2: Obnovení

Ukázka zdrojového kódu 14.6 prezentuje zápis funkce `REPLAY_LOG`.

```
// dokud není uzel synchronní přehráváme záznamy od uzlu
// s nejvyšším časem logických hodin
while (isSynchronized != KIVFS_OK) {
   // poslední zaznamenaná lokální zpráva
   lastLocalOp = kivfsGetLastTransactionId(vfsPort);
      // pokud je nalezen uzel s nejvyšším stavem logických hodin
      // přehrajeme od něj záznamy
      if ((tmpServer = getBestNode(srvList, recvPort, nodeId)) != NULL) {
         isSynchronized = replayLogsFromNode(tmpServer, recvPort, \\
          lastLocalOp, vfsPort);
      }
      // aktualizace vlastních logických hodin
      updateClock(lastLocalOp = kivfsGetLastTransactionId(vfsPort));
}
```

Obr. 14.6: Obnovení

14.4 Směrování požadavků

Jedním z cílů práce bylo navrhnout a implementovat systém trasování požadavků. Při návrhu architektury KIVFS bylo rozhodnuto, že trasování, resp. jeho část zjišťující kvalitu a stav linek bude implementována právě v synchronizační vrstvě, protože tato vrstva má z principu spojení mezi všemi uzly systému. V návrhu KIVFS bylo také rozhodnuto, že poskytování dat uživateli bude souborová vrstva realizovat spojením přímo s klientskou aplikací, proto není samotné předávání dat realizováno synchronizační vrstvou, ale souborovým systémem.

Podklady, trasovací tabulky, získává souborová vrstva od synchronizační. Implementovány byly dvě metody jak tyto informace předávat. Jednou je, že pokud synchronizační vrstva spočítá odlišnou trasu pro dané spojení aktivně o tom vyrozumí souborovou vrstvu přes VFS. Druhá metoda umožňuje souborové vrstvě dotazovat se na aktuální trasovací informace. Při implementaci bylo zvoleno, aby se předávaly informace jen o nejkratších cestách pro ušetření komunikační režie.

Trasování požadavků v KIVFS pak vytváří překryvnou síť nad tou skutečnou. Síťové cesty jsou zanedbány a převedeny na jednotlivé hrany s vypočtenými metrikami, kde probíhá vlastní trasování v rámci souborových serverů KIVFS.

14.4.1 Výpočet a význam metrik

S ohledem na požadavky distribuovaného souborového systému byl navržen následující systém hodnocení cest. Systém zohledňuje jak hodnoty měřené v reálném čase, tak umožňuje nastavit vlastní statickou konstantu pomocí konfiguračních voleb. Potřeba statického ovlivnění dané cesty může být dána provozními vlastnostmi v reálném nasazení. Může se jednat o tranzitní linky mezi státy, které bývají často

zvlášť zpoplatněné nebo o jiná spojení takto formálně znevýhodněná přestože jejich reálné parametry mohou být vhodnější. Nebo naopak administrátor může tímto parametrem znevýhodnit linky, které sice mají výborné parametry, ale nemají je dostatečně stabilní - linky jejichž kvalita je kolísavá.

Metrika cesty je v KIVFS složena z následujících parametrů:

- **rychlost** - jedná se o rychlost přenosu dat mezi dvěma uzly
- **odezva** - jedná o rychlost za jakou dorazí požadavek k cíli a pak zpět k odesílateli

Nepřímo je zohledněna také zátěž uzlu, protože má negativní vliv na odezvu výpočetní jednotky jako takové, pokud je uzel zatížený pak nereaguje na tyto požadavky se stejnou rychlostí jako nezatížený uzel a měření je tímto parametrem zatíženo. Vzhledem k abstrakci kdy je spojení mezi jednotlivými uzly distribuovaného systému reprezentováno jednoduchým číslem, přestože reálně se trasa mezi uzly skládá z mnoha skoků a různě kvalitních linek je třeba z měřených složek metriky získat skalární číslo, které by se dalo s ostatními jednoznačně porovnat, k tomu byl sestaven následující vzorec.

$$p = \frac{velikost\_dat}{velikost\_hlavicky} \tag{14.1}$$

$$M = \frac{cas\_prazdne\_zpravy * p}{cas\_zpravy\_s\_daty} * c \tag{14.2}$$

Tento vzorec počítá s poměrem zpoždění (upravený poměrným počtem požadavků) a rychlosti dané linky, oba údaje jsou zadány jako časy potřebné k odeslání a přijetí výsledku. Zvýhodňuje tak linky s lepší odezvou. Korekční parametr p slouží pro normalizaci - soulad objemu přenesených dat. Vzorec tak udává poměr času nutný k přenesení stejného objemu dat různými metodami, po malých zprávách a v jednom bloku. Konečné vynásobení konstantou c je další normalizací aby bylo možné pracovat s výsledkem jako celým číslem ve zvoleném rozsahu hodnot. Pokud je jako klíčový parametr rychlost linky, lze pracovat pouze s údajem udávajícím rychlost dané linky - časem nutným pro přenos známého objemu dat.

14.4.2 Zjišťování metrik jednotlivých tras

V rámci synchronizační vrstvy KIVFS je zjišťování vlastností jednotlivých tras řešeno samostatným vláknem, které periodicky komunikuje s ostatními uzly a zkouší jednotlivé parametry linek. Pro tyto testy parametrů spojení mezi uzly jsou definované zvláštní požadavky KIVFS. Zjišťuje se pomocí nich rychlost a odezva vzdáleného uzlu.

Test na rychlostní parametr linky je realizován pomocí zprávy definující rychlostní test `KIVFS_ROUTE_SPEED`. V rámci toho uzel, který chce daný parametr zjistit vyšle k cílovému uzlu balík dat o definované velikosti, pro účely vývoje a testování byla tato hodnota experimentálně stanovena na 512 kB. Cílový uzel odpoví zpět zprávou bez dat. Tento způsob byl zvolen, protože v případě, že se jedná o asymetrické spojení, pak by tato hodnota byla touto asymetrií ovlivněna a získávala by se hodnota průměrná pro oba směry. Vzhledem k tomu že v tomto případě cílový uzel provádí stejné měření metriky, pak při výměně těchto metrik je získaná informace kompletní a pro přenos dat opačným směrem může být volena jiná cesta.

Test odezvy probíhá obdobným způsobem. Zdrojový uzel posílá do cíle prázdnou zprávu `KIVFS_ROUTE_PING`, na kterou cíl odpovídá opět prázdnou zprávou. Tím se minimalizuje vliv rychlosti přenosu zprávy na výslednou hodnotu. Zdrojový kód pro zaslání požadavku je přiložen jako obr 14.7.

Pokud je cílový uzel nedostupný, je mu nastavena hodnota na nekonečno k tomu je použita vyhrazená konstanta `KIVFS_INF` (maximální hodnota datového typu) reprezentující nekonečno.

```
// vytvoříme požadavek KIVFS_ROUTE_PING
request = kivfs_request(NULL_ID, KIVFS_ROUTE_PING, NULL);
// záznam času a odeslání požadavku
(void) gettimeofday(&ts, (struct timezone *) NULL);
kivfs_send(sockFd, request);
kivfs_free_msg(request);

// pokud získáme odpověď vypočteme dobu odezvy
if (kivfs_recv(sockFd, &reply) == KIVFS_OK) {
   (void) gettimeofday(&tr, (struct timezone *) NULL);
   kivfs_free_msg(reply);
   time = diffusec(&ts, &tr);
} else {
// jinak vracíme nekonečno
   return KIVFS_INF;
}
return time;
```

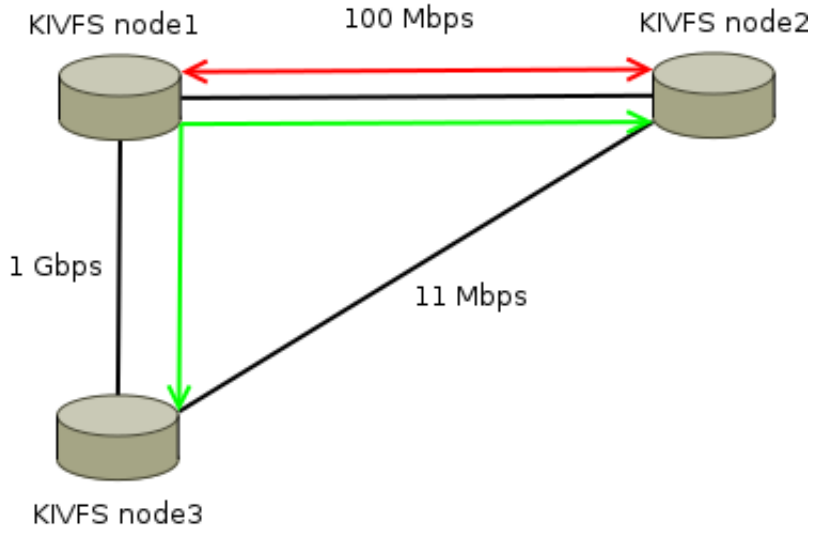
Obr. 14.7: Test odezvy

14.4.3 Výpočet nejkratší cesty

Každý uzel KIVFS zjišťuje parametry jednotlivých tras nezávisle na ostatních ze svého vlastního pohledu, periodicky pak dochází k výměně těchto informací pomocí zpráv. Získané informace jsou reprezentovány jako incidenční matice, výsledné trasy pak jako spojový seznam kde každý list seznamu reprezentuje jeden skok v dané trase. Z naměřených hodnot jsou periodicky vyhrazeným vláknem přepočítávány

jednotlivé trasy. K tomu je použit algoritmus dle konfigurace. Pokud je po novém výpočtu zjištěna změna trasovacích tabulek, je zaslána nová tabulka souborové vrstvě přes síťové spojení. Za změnu není považována změna výsledné metriky, ale tabulky jsou kontrolovány na jednotlivé skoky, tzn. změna je propagována pouze pokud se jedná o změny v organizaci jednotlivých skoků mezi uzly.

Obrázek 14.8 znázorňuje výsledné trasy cest z uzlu 2 do ostatních, červená je cesta k uzlu 1 a zelená k uzlu 3 vedoucí přes uzel 1, černé čáry zobrazují přímé cesty a jejich rychlost.



Obr. 14.8: Znározněné cesty z uzlu 2

14.4.4 Použité algoritmy

Během realizace diplomové práce a její praktické části byly implementovány dva algoritmy, Dijkstrův a Floyd-Warshall pro výpočet nejkratší cesty, Bellman-Ford algoritmu nebyl implementován, protože proti Dijkstrovu nepřináší odlišnou časovou složitost a jeho výhoda záporných cest není v tomto případě nutná, protože ohodnocení hran je pouze kladné, nebo symbolicky nekonečno v případě nedostupného spojení. Algoritmy je možné zvolit konfigurační volbou `route` (obr. 15.5), v části diskutující výsledky diplomové práce budou jednotlivé algoritmy porovnány v provozu, teoretické rozdíly byly již popsány v příslušných kapitolách.

14.5 Organizace zdrojových souborů

Vzhledem k rozsáhlosti projektu (více než tři tisíce pět set řádek včetně hlavičkových souborů) jsou zdrojové kódy rozděleny do několika zdrojových souborů, jejichž

popsaný seznam následuje.

- `include/` je adresář, který obsahuje hlavičkové soubory, kde jsou definovány funkce a datové struktury, soubory jsou rozděleny podle zdrojových souborů (zdrojovému souboru odpovídá se hlavičkový se stejným názvem)
- `kivfs-route.c` o obsahuje implementace algoritmů vyhledávání nejkratších cest a práci s pomocnými strukturami pro výpočet Floyd Warshall metody
- `kivfs-route-net.c` obsahuje funkce pro výměnu zpráv pomocí protokolu KIVFS po zjištování nejkratších cest a výměnu trasovacích tabulek
- `kivfs-route-struct.c` obahuje funkce pro práci se strukturami používanými pro trasování, výpis tabulek, alokace a dealokace struktur, kopírování a nastavování hodnot
- `kivfs-route-threads.c` definuje kód vláken, která zajišťují získávání a výpočet tras a inicializaci používaných sdílených struktur
- `kivfs-sync-server.c` je hlavní zdrojový soubor obsahující funkci `main()`, kde jsou startována ostatní vlákna a řešená obnova uzlu při jeho startu
- `kivfs-sync-clock.c` obsahuje implementaci logických hodin, jejich inicializaci a aktualizaci logického času
- `kivfs-sync-net.c` definuje funkce pro výměnu zpráv pomocí protokolu KIVFS, které využívá synchronizační část práce, navázání spojení s protějšky, zjištění aktuálního stavu zaznamenaných zpráv
- `kivfs-sync-order.c` obsahuje implementaci řazené fronty, funkce pro vložení požadavku, vyzvednutí, alokaci a uvolnění potřebných struktur
- `kivfs-sync-struct.c` definuje struktury využívané pro synchronizaci (seznam serverů, seznam socketů a stavové informace) a práci s nimi
- `kivfs-sync-threads.c` obsahuje kód vláken, která zajišťují synchronizaci a transakce v KIVFS, inicializaci proměnných a některé další funkce, které pracují se sdílenými proměnnými vláken
- `Makefile` je soubor pro překlad pomocí nástroje `make`, obsahuje potřebné cíle pro překlad synchronizačního serveru, tvorbu distribučních balíčků a instalaci

15 PŘEKLAD, KONFIGURACE A SPUŠTĚNÍ

Synchronizační vrstva KIVFS byla implementována v jazyce C jako samostatná serverová aplikace, která komunikuje s ostatními aktéry pomocí Berkeley socketů a zasíláním zpráv. Jak bylo popsáno výše, synchronizační vrstva využívá funkcí knihovny `libkivfscore`, která je její jedinou závislostí. Pro překlad je připraven v adresáři projektu (`server/sync`) soubor `Makefile`, který řídí překlad pomocí nástroje `make`. V souboru `Makefile` jsou definovány cíle pro překlad, vyčištění adresáře projektu od přeložených objektových souborů, instalaci do adresářové struktury (`/opt/kivfs/`) a tvorbu balíčků pro distribuci GNU/Linux Debian.

15.1 Překlad libkivfscore

Pro překlad synchronizační vrstvy je třeba mít na serveru předem nainstalovanou knihovnu `libkivfscore` ve standardních cestách (`/usr/lib`) s jejími hlavičkovými soubory (`/usr/include/`) a základní nástroje pro překlad (`gcc`, `libc`, `libc-dev`).

Knihovna `libkivfscore` obsahuje další závislosti na knihovnách OpenSSL a Kerberos (implementace Heimdal), které jsou používané pro zabezpečenou komunikaci a autentizaci. Všechny potřebné balíčky pro překlad a instalaci `libkivfscore` lze nainstalovat na distribuci GNU/Linux Debian jediným příkazem, jak ukazuje obr. 15.1 (další závislosti jsou řešeny automaticky).

```
apt-get install build-essential libssl-dev heimdal-dev
```

Obr. 15.1: Instalace balíčků nutných pro překlad

Zdrojové soubory `libkivfscore` jsou součástí balíku zdrojových souborů serverových modulů KIVFS (`server/core`). V adresáři knihovny je připravený soubor `Makefile`, který řídí překlad pomocí nástroje `make`. Překlad a instalaci je možné provést pomocí příkazu `make` jak znázorňuje obr. 15.2.

```
fs-1:/opt/kivfs/src/core# make && make install
make && make install
Compiling all C source files: Done!
Generating KIVFS Core shared library: Done!
Compiling all C source files: Done!
Generating KIVFS Core shared library: Done!
Shared library libkivfscore.so has been installed to your system.
```

Obr. 15.2: Překlad knihovny libkivfcore

15.2 Překlad synchronizační vrstvy

Překlad synchronizační vrstvy se provádí pomocí programu `make` s vhodně zvoleným parametrem cíle podle tabulky 15.1. Synchronizační vrstvu je možné instalovat přímo ze zdrojových souborů do operačního systému, nebo pomocí balíčku pro distribuci Debian.

bez parametru	provede překlad, výsledný soubor je `kivfs-sync-server`
install	provede překlad pokud je to nutné a nainstaluje synchronizační vrstvu do vyhrazeného adresáře v `/opt/`
debian	provede potřebný překlad a vygeneruje balíček pro distribuci GNU/Linux Debian
clean	vyčití projekt o objektové a jiné dočasné soubory

Tab. 15.1: Definované cíle pro program make

Příklad překladu je vidět na obr. 15.3.

```
fs-1:/opt/kivfs/src/sync# make
Compiling kivfs-sync.c... Done.
Compiling kivfs-sync-net.c... Done.
Compiling kivfs-sync-order.c... Done.
Compiling kivfs-sync-server.c... Done.
Compiling kivfs-sync-struct.c... Done.
Compiling kivfs-sync-threads.c... Done.
Compiling kivfs-sync-clock.c... Done.
Compiling kivfs-route-net.c... Done.
Compiling kivfs-route-threads.c... Done.
Compiling kivfs-route.c... Done.
Compiling kivfs-route-struct.c... Done.
Linking kivfs-sync-server... Done.
```

Obr. 15.3: Překlad synchronizačního serveru

Pokud je preferována instalace pomocí balíčků distribuce (podporovaný je Debian) je třeba provést překlad příkazem `make debian`, který vygeneruje balíček `kivfs-sync_i386.deb`, který je možné instalovat pomocí nástroje `dpkg`. Překlad s instalací balíčku zobrazuje obr. 15.4. Balíček instaluje konfiguraci do adresáře `/opt/kivfs/etc/` a binární soubory do `/opt/kivfs/bin/`. V případě instalace pomocí `make` jsou cesty stejné. Konfiguraci která je součástí balíčku je třeba upravit podle daných potřeb, jednotlivé volby budou popsány dále.

```
fs-1:/opt/kivfs/src/sync# make debian
Compiling kivfs-sync.c... Done.
...
Compiling kivfs-route-struct.c... Done.
Linking kivfs-sync-server... Done.
Generating package... Done.

fs-1:/opt/kivfs/src/sync# dpkg -i kivfs-sync_i386.deb
```

Obr. 15.4: Překlad, tvorba balíčku a instalace

15.3 Konfigurace synchronizační vrstvy

Konfigurace synchronizační vrstvy je součástí konfiguračního souboru `kivfs.conf`, který obsahuje pro synchronizační vrstvu samostatnou sekci, jak je ukázáno v komentované konfiguraci 15.5. Vzorový konfigurační soubor je uložen v adresáři `server/conf/` zdrojových souborů a je součástí generovaného balíčku.

```
#definice sekce
[sync]
#port kde naslouchá příchozím požadavkům klientů
port=30002
#adresa kde naslouchají další lokální servery KIVFS
ip=127.0.0.1
#porty ostatních serverů a služeb
vfsport=30003
syncport=30010
routeport=30011
recoveryport=30012
transactionport=30013
#odkaz na lokální uzel
local=node1
#definice uzlů a jejich metrik
node1=147.228.63.46
node2=147.228.63.47
costnode2=1000
node3=147.228.63.48
#výber trasovacího algoritmu
#0 - žádný výpočet
#1 - Dijkstra
#2 - Floyd Warshall
route=1
```

Obr. 15.5: Konfigurace synchronizační vrstvy

15.4 Spuštění synchronizační vrstvy

Synchronizační vrstvu je možné spouštět samostatně, nebo jako jednu ze služeb pomocí `kivfs-daemon`, který spouští všechny servery KIVFS a v případě jejich selhání a ukončení je spouští znovu. Pro systém KIVFS je dostupný i startovací skript `/etc/init.d/kivfs` pro distribuci GNU/Linux Debian, pomocí kterého může být KIVFS startováno automaticky při startu systému. Startovací skript podporuje parametry `start`, `stop` a `restart`. Jediným parametrem všech vrstev je cesta ke konfiguračnímu souboru. Spuštění samostatné vrstvy zobrazuje výstup 15.6, synchronizační vrstva vyžaduje běh vrstvy virtuálního souborového systému a databázové vrstvy.

```
fs-1:~# /opt/kivfs/bin/kivfs-sync-server /opt/kivfs/conf/kivfs.conf
Will listen on 30002
VFS listen on 30003
SYNC listen on 30010
ROUTE listen on 30011
RECOVERY listen on 30012
TRANSACTION listen on 30013
LocalNode: node1 (1)
node1: 147.228.63.46
Static cost to node2: 1000
node2: 147.228.63.47
node3: 147.228.63.48
```

Obr. 15.6: Spuštění synchronizačního serveru

Přestože je systém KIVFS připravován pro GNU/Linux distribuci Debian, je možné ho pouze s drobnými modifikace přenést na jinou distribuci GNU/Linuxu nebo UNIXový operační systém. Z pohledu synchronizační vrstvy je třeba přenést pouze knihovnu `libkivfscore`.

16 TESTOVÁNÍ

Výsledky realizační části diplomové práce byly prakticky ověřeny pomocí několika testovacích scénářů. Byl také zkoumán vliv nově implementovaných funkcí na rychlost systému a reálné rychlostní parametry algoritmů hledání nejkratších cest, výsledky testů jsou rozebrány v dalších kapitolách.

16.1 Testovací prostředí

Systém KIVFS byl testován na čtyřech vyhrazených fyzických serverech se shodnou konfigurací popsanou v tabulce 16.1. Klientské aplikaci byl vyhrazen jeden server, na ostatních třech běžel systém KIVFS.

Procesor	Intel(R) Pentium(R) D CPU 3.40GHz
Paměť	512 MB
Disk	20 GB
Operační systém	GNU/Linux Debian Lenny 5.0.4
Síť	1Gbps

Tab. 16.1: Konfigurace testovacích serverů

16.2 Testovací scénáře

Realizovaná funkčnost byla ověřena pomocí testovacích scénářů, které obsahovaly sadu kroků pro prověření správnosti realizace a funkčnosti systému KIVFS.

16.2.1 Práce se soubory

V tomto testu byl systém KIVFS nastartován ve výchozím stavu. Po nastartování a navázání všech spojení následuje připojení klientskou aplikací k libovolnému serveru. Na server bylo potom nahráno a následně staženo několik souborů (pět až dvacet), polovina souborů byla také smazána. Následně byl další klientskou aplikací proveden obdobný test přes jiný server, doposud nahrané soubory byly staženy a přepsány, opět staženy a smazány.

16.2.2 Schopnost obnovy pomocí záznamů

Tento test měl za cíl ověřit schopnost obnovy odstaveného uzlu a jeho návrat zpět do funkčního stavu. Test se sestával opět ze startu systému ve výchozím konzistentním stavu, následovalo nahrání a stažení několika souborů (pět až deset), poté byl jeden ze serverů ukončen. Pokračovalo se opět nahráním a stažením dalších souborů ve stejném množství. Následně byl odstavený server znovu nastartován, po jeho plném startu následovalo stahování souborů nahraných do KIVFS v době jeho odstávky, právě přes tento server. Test byl ukončen opětovným nahráním a následným stažením několika dalších souborů přes obě aktuálně bežící klientské aplikace.

16.2.3 Výpočet cest a jejich předání

Tento test byl pasivní bez účasti uživatele, systém KIVFS byl nastartován a byly sledovány záznamy systému do kterých trasovací vlákna zapisovala tabulku metrik a nalezené nejkratší cesty. Tyto nalezené cesty byly manuálně ověřeny. Ze záznamů systému bylo pak možné dosledovat i předání nových tras souborovému serveru.

17 VÝSLEDKY

Z testovacích scénářů byly naměřeny hodnoty, které byly následně srovnány a porovnány s předchozí implementací.

17.1 Vliv synchronizace na přenos souborů

Součástí testování synchronizační vrstvy bylo zjištění vlivu synchronizace požadavků a jejich transakčního zpracování na odezvy a rychlost celého systému.

17.1.1 Popis měření

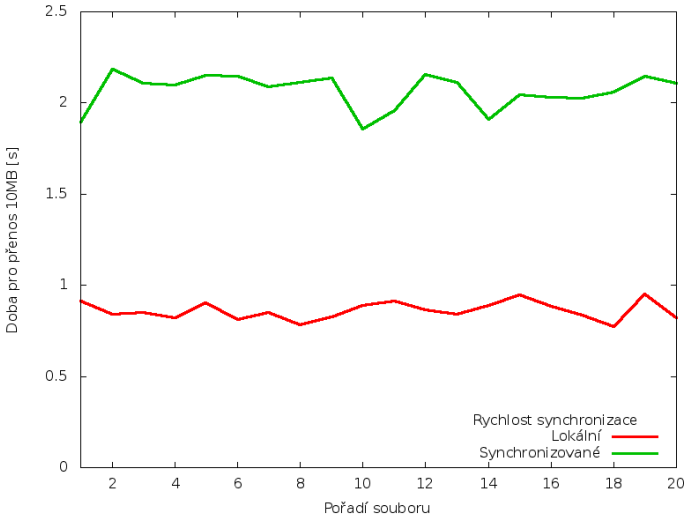
Souborový systém KIVFS je možné provozovat samostatně na jediném serveru bez synchronizační vrstvy v lokálním módu, provoz v tomto módu byl porovnán s provozem na třech serverech s použitím synchronizační vrstvy. Vliv synchronizace byl měřen na přenosu souborů na server. Konfigurace dalších vrstev byla v obou měřených případech schodná. Na servery byl opakovaně nahráván stejný soubor o velikosti 10 MB plný náhodných dat. Nahrání celých souborů bylo opakováno dvacetkrát, jak v lokálním módu, tak s použitím synchronizační vrstvy. Z naměřených výsledků byl pak vypočítán průměr obou měření.

17.1.2 Výsledky měření

Naměřené hodnoty jsou zobrazeny v tabulce 17.1 a časové hodnoty znázorňuje graf obr. 17.1. Z hodnot je patrné, že synchronizace má negativní vliv na celkovou přenosovou rychlost systému. Vzhledem k další implementaci v ostatních vrstvách KIVFS představuje v měřeném případě upload jednoho souboru výměnu alespoň deseti synchronizovaných zpráv. Tento počet je pro každý soubor základem, v případě větších souborů dochází ještě k dalším zprávám na rezervaci místa a jeho řízení. Vliv synchronizace na rychlost přenosu je pak s rostoucí velikostí souboru klesající (viz tab. 17.2), protože požadavků na synchronizaci je v poměru k objemu dat méně. Z naměřených hodnot lze spočítat průměrný časový rozdíl 1,2 sekundy, který jde na vrub synchronizace a transakčního zpracování.

Pořadí	Lokální [s]	Rychlost [MB/s]	Synchronizace [s]	Rychlost [MB/s]
1	0,9147	10,9325	1,8940	5,2797
2	0,8426	11,8680	2,1837	4,5793
3	0,8514	11,7454	2,1068	4,7464
4	0,8236	12,1418	2,0996	4,7629
5	0,9048	11,0522	2,1495	4,6522
6	0,8147	12,2745	2,1467	4,6582
7	0,8521	11,7357	2,0870	4,7916
8	0,7836	12,7616	2,1110	4,7370
9	0,8277	12,0817	2,1391	4,6749
10	0,8879	11,2625	1,8589	5,3794
11	0,9119	10,9661	1,9593	5,1038
12	0,8639	11,5754	2,1553	4,6398
13	0,8436	11,8540	2,1110	4,7372
14	0,8878	11,2638	1,9090	5,2384
15	0,9490	10,5374	2,0469	4,8855
16	0,8831	11,3237	2,0310	4,9236
17	0,8354	11,9703	2,0280	4,9311
18	0,7757	12,8916	2,0595	4,8555
19	0,9517	10,5075	2,1452	4,6616
20	0,8240	12,1359	2,1068	4,7465
Průměr	0,8615	11,6441	2,0664	4,8492

Tab. 17.1: Vliv synchronizace na rychlost přenosu souboru o velikosti 10 MB



Obr. 17.1: Doba nutná pro přenos 10 MB souboru

Velikost souboru	Lokální přenos [MB/s]	Synchronní přenos [MB/s]
10 MB	11,6443	4,8492
100 MB	32,4541	24,1171
1000 MB	41,0332	39,8875

Tab. 17.2: Rychlost přenosu - různé velikosti

17.2 Rychlost obnovy uzlu

Jedním z testovacích scénářů bylo obnovení uzlu ze záznamů transakcí. V systém KIVFS byl jeden uzel odstaven a na ostatních probíhal další provoz, tím bylo docíleno různého stavu metadat na běžících uzlech a odstaveném. Následně byla měřena doba obnovy odstaveného uzlu pomocí přehrání záznamů z databázového serveru v závislosti na počtu zpráv.

17.2.1 Popis měření

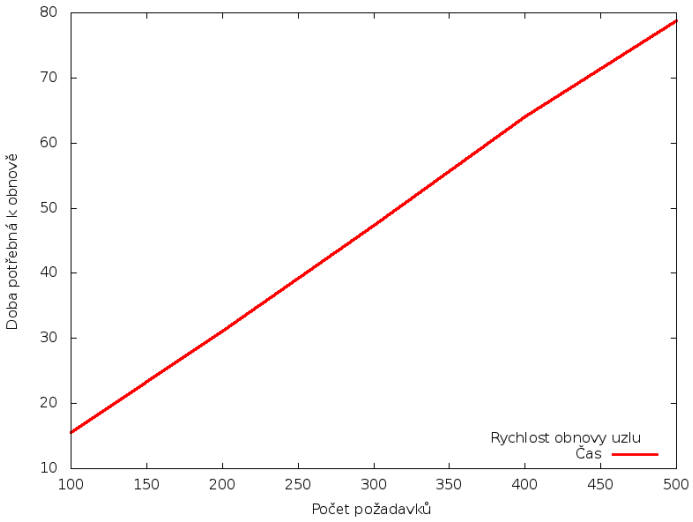
Z tří uzlů distribuovaného systému byl jeden odstaven, na ostatních bylo proveden upload deseti až padesáti souborů s krokem deset souborů. Po nahrání daného počtu souborů byl na odstaveném uzlu proveden proces obnovy a tento čas byl měřen. Proces obnovy byl měřen od startu obnovy do plného přehrání všech zaznamenaných zpráv. Rychlost spojení ostatních uzlů mezi sebou byla v tomto případě zanedbatelná.

17.2.2 Výsledky měření

Výsledky měření jsou zaneseny v tabulce 17.3 a znázorněny grafem obr. 17.2. Z grafu je patrná lineární přímá úměra mezi počtem požadavků potřebných pro obnovu a časem který je k obnově třeba. Synchronizační vrstva si během tohoto procesu vyžádá záznamy od databázového serveru a následně je přímo posílá na lokální databázovou vrstvu. Podle naměřených hodnot je třeba k obnově jedné zprávy 0,15 sekundy. Pokud připadá na nahrání souboru deset zpráv znamená to 1,5 sekundy pro obnovení jednoho souboru. V případě většího množství zpráv nutných k obnově uzlu, může být výhodnější přenést na tento uzel zálohu databáze jiného uzlu a pouze ji doplnit přehráním rozdílových zpráv.

Počet souborů	Počet požadavků	Čas obnovy $[s]$	Obnova požadavku $[s]$
10	100	15,5008	0,155008
20	200	31,0896	0,155448
30	300	47,4163	0,158054
40	400	63,9862	0,159966
50	500	78,7555	0,157511

Tab. 17.3: Rychlost obnovy uzlu



Obr. 17.2: Rychlost obnovy uzlu

17.3 Měření metrik a výpočet tras

Dalším testem synchronizační vrstvy bylo porovnání rychlosti výpočtu nejkratších cest mezi uzly pomocí Dijkstrova a Floyd Warshallova algoritmu.

17.3.1 Popis měření

Vzhledem k tomu, že trasovací funkce jsou implementovány nezávisle na zbytku systému je možné ověřit a měřit tuto funkčnost mimo KIVFS. Byla vytvořena sada pěti až sta virtuálních uzlů s krokem po pěti uzlech a byla měřena rychlost s jakou algoritmus vypočte nové nejkratší cesty.

17.3.2 Výsledky měření

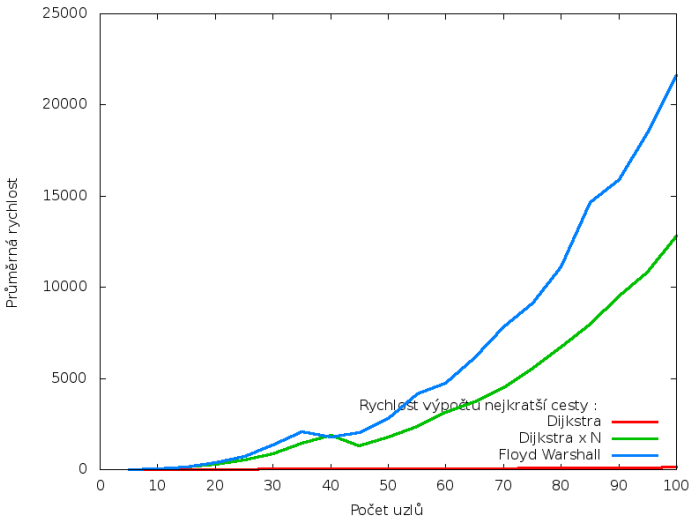
Výsledky měření jsou zobrazené v tabulce 17.4 a názorněji v grafu na obr. 17.3. Jak bylo popsáno v kapitolách popisující jednotlivé algoritmy, oba mají rozdílnou časovou složitost. Dijkstrův algoritmus má složitost odpovídající druhé mocnině počtu vrcholů a Floyd Warshallův algoritmus má složitost odpovídající třetí mocnině, to bylo prakticky ověřeno v testech.

Z naměřených hodnot je patrné, že druhý algoritmus je proti Dijkstrovu náročnější násobně podle počtu vrcholů, poměr je dokonce vyšší, protože je zde režie nutná pro vytvoření pracovní matice, která je oproti použité incidenční matici paměťově náročnější. Implementace Dijkstrova algoritmu využívá pouze incidenční matici a výstupní trasovací tabulku, je tedy paměťově méně náročná. Výhodou Floyd Warshallova algoritmu je nalezení cest mezi všemi vrcholy grafu, k dosažení stejného výsledku by bylo třeba volat Dijkstrův algoritmus opakovaně pro každý vrchol.

Pak by byla výpočetní i paměťová náročnost srovnatelná, což lze pozorovat v grafu i tabulce kde je zobrazena rychlost výpočtu Dijkstrova algoritmu vynásobená počtem uzlů, pro výpočet všech nejkratších cest v grafu. Přesto je Dijkstrův algoritmus rychlejší a proto byl zvolen jako výchozí.

Počet uzlů	Dijkstra $[\mu s]$	Dijkstra $\times N$ $[\mu s]$	Floyd Warshall $[\mu s]$
5	3	15	12
10	5	50	59
15	9	135	167
20	14	280	380
25	21	525	742
30	29	870	1331
35	41	1435	2092
40	47	1880	1787
45	29	1305	2046
50	36	1800	2828
55	43	2365	4137
60	52	3120	4761
65	57	3705	6150
70	64	4480	7831
75	74	5550	9149
80	84	6720	11115
85	94	7990	14636
90	106	9540	15914
95	114	10830	18467
100	128	12800	21613

Tab. 17.4: Rychlost výpočtu nejkratší cesty



Obr. 17.3: Rychlost výpočtu nejkratší cesty

18 ZÁVĚR

V diplomové práci byly popsány podstatné teoretické principy a algoritmy distribuovaných systémů nutné pro implementaci synchronizační vrstvy. Popsány byly algoritmy pro synchronizaci fyzických i logických hodin, algoritmy hledání nejkratších cest, byl popsán globální stav systému a možnosti obnovy pomocí záznamů. Na základě těchto algoritmů byla navržena a implementována programová část práce.

Zadané funkčnosti bylo v práci dosaženo v plném rozsahu, v systému KIVFS byla zavedena synchronizace a replikace požadavků, pro které bylo zavedeno transakční zpracování s použitím hlasování. Synchronizované požadavky jsou nyní zaznamenávány pomocí databázové vrstvy do trvalé paměti a mohou tak sloužit v případě odstávky některého uzlu pro jeho obnovu v případě návratu do skupiny serverů.

Pro směrování požadavků bylo implementováno měření metrik linek mezi servery a na jejich základě pak výpočet nejkratších cest.

Realizovaná funkčnost byla ověřena pomocí modelových testovacích scénářů. Následně byl zkoumán vliv nových funkcí na výkon systému. Synchronizace požadavků je funkční a nemůže tak dojít k konzistentnímu stavu dat. Synchronizace generuje při práci se systémem vyšší zpoždění, které je ovšem při práci s velkými soubory zanedbatelné. Díky přehrávání záznamů je odstavený uzel schopen obnovy do konzistentního stavu.

Algoritmy hledání nejkratší cesty byly prakticky porovnané z pohledu reálné rychlosti výpočtu. Dijkstrův algoritmus hledání nejkratších cest byl vyhodnocen jako nejvhodnější. Pomocí hledání nejkratších cest je systém KIVFS schopen doručovat data rychleji.

Realizovaný modul KIVFS je možné použít při dalších experimentech a vývoji tohoto distribuovaného souborového systému.

Implementovanou funkčnost lze dále rozšířit o zavedení detekce nestabilní linky, nebo rozdělení front požadavků podle svazků KIVFS aby mohly transakce příslušející k rozdílným svazkům probíhat paralelně a snížilo se tak zpoždění vzniklé synchronizací.

LITERATURA

[1] KSHEMKALYANI, Ajay D. *Distributed computing: principles, algorithms, and systems.* Cambridge: Cambridge University Press, 2008, 736 s. ISBN 978-0-521-87634-6.

[2] TANENBAUM, Andrew S. *Distributed systems: principles and paradigms.* New Jersey: Prentice-Hall, 2002, 803 s. ISBN 01-308-8893-1.

[3] *Encyclopædia Britannica Online, s. v. "Moore's law"* Dostupné z URL: . <http://www.britannica.com/EBchecked/topic/705881/Moores-law>.

[4] ANSA *A Systems Designer's Introduction to the Architecture* [online] Dostupné z URL: <http://www.ansa.co.uk/ANSATech/91/RC25300.pdf>.

[5] DONALDSON, D. a HAYTON, R. *FlexiNet and FollowMe Developments* [online] ANSA Technical Committee, 21. července 1998 Dostupné z URL: <http://www.ansa.co.uk/ANSATech/ANSAhtml/98-ansa/external/9807tb/9807ffp.pdf>.

[6] BARAK, A a SHILOH, A. *The MOSIX Cluster Operating System for High-Performance Computing on Linux Clusters, Multi-Clusters and Clouds* white paper, 2012. Dostupné z URL: <http://www.mosix.org/pub/MOSIX_wp.pdf>.

[7] CESNET, z.s.p.o. *MetaCentrum* Dostupné z URL: <http://www.metacentrum.cz/>.

[8] NOWINSKI, K., LESYNG, B., NIEZGÓDKA, M., BALA, P., *Project EUROGRID (abstract)* PIONIER 2001 Conference Proceedings, pp. 187–191, Poznan 2001, ISBN 83-913639-2-9

[9] THURLOW, R. *RPC: Remote Procedure Call Protocol Specification Version 2*, RFC 5531, Sun Microsystems, Network Working Group, květen 2009 Dostupné z URL: <http://www.ietf.org/rfc/rfc5531.txt>.

[10] ORACLE CORPORATION *Java Remote Method Invocation - Distributed Computing for Java.* Dostupné z URL: <http://www.oracle.com/technetwork/java/javase/tech/index-jsp-138781.html>.

[11] ZIMMERMANN, H. *OSI Reference Model-The IS0 Model of Architecture for Open Systems Interconnection*, IEEE TRANSACTIONS ON COMMUNICATIONS, svazek COM-28, číslo 4, Duben 1980

[12] LAMPORT, L. *Time, Clocks, and the Ordering of Events in a Distributed System* Communications of the ACM, svazek 21, číslo 7, červenec 1978

[13] CHANDY, K. M., LAMPORT, L. *Distributed Snapshots: Determining Global States of Distributed Systems* ACM Transactions on Computer Systems, svazek 3, číslo 1, únor 1985

[14] BERNSTEIN, P. A., Vassos HADZILACOS, V., Nathan GOODMAN, N. *Concurrency Control and Recovery in Database Systems* Kapitola 7, Addison Wesley Publishing Company, 1987, 370 s. ISBN 0-201-10715-5

[15] SKEEN, D. a STONEBRAKER, M. *A Formal Model of Crash Recovery in a Distributed System* IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, svazek SE-9, číslo 3, květen 1983

[16] Dijkstra, E. W, *A note on two problems in connexion with graphs* Numerische Mathematik, 1959, svazek 1, číslo 1, s. 269-271

[17] Floyd, R. W., *Algorithm 97: Shortest path* Communications of the ACM, svzek 5, číslo 6, červen 1962 Page 345

[18] Bellman, R., *On a routing problem*, Quarterly of Applied Mathematics číslo 16. 1958, s. 87–90.

[19] NEUMAN, C., *The Kerberos Network Authentication Service*, RFC 4120, Network Working Group, červenec 2005 Dostupné z URL: <http://www.ietf.org/rfc/rfc4120.txt>.

[20] SERMERSHEIM, J., *Lightweight Directory Access Protocol (LDAP): The Protocol*, RFC 4511, Novell, Inc., Network Working Group, červen 2006 Dostupné z URL: <http://www.ietf.org/rfc/rfc4511.txt>.

[21] JUNÁK, M., MATĚJKA, L., PEŠIČKA, L., PIVNIČKA, M., SKUPA, J., STREJC, R., STEINER, V., ŠAFAŘÍK, J. *KIV-DFS-Experimental Distributed File System* In Informatics 2009. Košice: Technical University, 2009. s. 45-50. ISBN: 978-80-8086-126-1

SEZNAM SYMBOLŮ, VELIČIN A ZKRATEK

ANSA Advanced Networked Systems Architecture

ACID požadavky na bezpečný transakční přístup, A - atomičnost, C - konzistence, I - izolace, D - trvanlivost

OS operační systém

DS distribuovaný systém

DFS Distributed File System, distribuovaný souborový systém

NFS Network File System, síťový souborový systém vyvinutý společností Sun Microsystems

NTP Network Time Protocol, protokol pro synchronizaci času po síti

MPI Message Passing Interface, knihovna implementující protokol pro podporu paralelního řešení výpočetních problémů v počítačových clusterech pomocí zasílání zpráv

PVM Parallel Virtual Machine, rozhraní pro paralelní programy využívající zejména spojované spojení pomocí Berkeley socketu a účelem spojení výkonu jednotlivých uzlů sítě do klastru - virtuálního superpočítače

RPC Remote Procedure Call, vzdálené volání procedur

RMI Remote Method Invocation, vzdálené vykonávání metod

MOM Message Oriented Middleware

API Application Program Interface, aplikační programové rozhraní

VFS Virtual File System, virtuální systém souborů

LDAP Lightweight Directory Access Protocol, protokol adresářových služeb

Kerberos autentizační protokol umožňující bezpečnou autentizaci po nezabezpečených linkách

SEZNAM OBRÁZKŮ

SEZNAM TABULEK

SEZNAM ALGORITMŮ