Katedra Informatiky
Fakulta Matematiky, Fyziky a Informatiky
Univerzita Komenského, Bratislava

Atomic Commit Protocol

(Bakalárska práca)

Andrej Borsuk

Vedúci: Dr. Tomáš Plachetka

Bratislava, 2009

Čestne prehlasujem, že som túto bakalársku prácu vypracoval samostatne s použitím citovaných zdrojov.

.....

Poďakovanie

Chcel by som sa poďakovať vedúcemu mojej bakalárskej práce RNDr. Tomášovi Plachetkovi za ústretovosť, pomoc a vedenie pri tvorbe. Takisto sa chcem poďakovať mojej rodine za prejavenú trpezlivosť.

Abstrakt

Práca sa zaoberá protokolom pre atomický commit, t.j. protokolom, ktorý umožňuje dohodnúť sa viacerým účastníkom v distribuovanom systéme na rovnakom spoločnom rozhodnutí. Po úvodných kapitolách, ktoré poskytujú stručný prehľad problematiky sa v práci rozoberajú niektoré riešenia tohoto problému, ich vlastnosti, výhody a nevýhody. Súčasťou práce je aj implementácia tohoto protokolu na simulovanom distribuovanom systéme.

Kľúčové slová: **databázy, distribuované systémy, protokol**

Obsah

Úvod

Databázy sú v dnešnej dobe často základným stavebným kameňom mnohých informačných systémov. Slúžia na spoľahlivé uchovávanie informácii a bezpečnú manipuláciu s nimi. Kým pri menších a stredne veľkých informačných systémoch je jedna centralizovaná databáza postačujúca, rôzni internetoví giganti ako Google, eBay alebo Amazon si s tým prirodzene nevystačia. Na rad prichádzajú distribuované databázové systémy. Umožňujú lepšie rozloženie záťaže, poskytujú vyššiu spoľahlivosť systému a pri správnom nastavení môžu priniesť aj vyššiu efektivitu. Prirodzene, distribuovaný databázový systém je o niečo zložitejší a vyžaduje si riešenie niekoľkých problémov navyše. Jedným z nich je problém atomického commitu. Kým v centralizovanom databázovom systéme je databázový systém sám sebe pánom, v distribuovanom systéme je potrebné, aby sa dohodli viacerí účastníci. Spôsob, akým účastníci dospejú k rovnakému a konzistentnému rozhodnutiu popisuje atomic commit protocol. Navyše je potrebné, aby bol tento protokol odolný voči rôznym poruchám a výpadkom a aby po odstránení týchto chýb a obnove bol schopný dokončiť svoj beh zachovávajúc pri tom konzistenciu údajov.

Prvá kapitola je venovaná úvodu do problematiky. Rozoberajú sa v nej základné pojmy a modely z oblasti databáz potrebné pre pochopenie ďalších kapitol. Druhá kapitola je venovaná problému atomického commitu. Ukazuje rôzne záludnosti tohoto zdanlivo jednoducho vyzerajúceho problému

a následne stanovuje požiadavky, ktoré by mal spĺňať protokol pre atomický commit. V tretej kapitole sa rozoberá najjednoduchšie riešenie tohoto problému - dvojfázový commit. Uvádza jeho priebeh, vlastnosti protokolu i možné modifikácie. Ďalšia kapitola je venovaná implementácii jednej z verzií dvojfázového commitu na simulovanom distribuovanom systéme. Ďalšie dve kapitoly sú venované iným riešeniam tohoto problému a to konkrétne trojfázovému commitu a ďalším protokolom vychádzajúcim z dvojfázového alebo trojfázového commitu.

Kapitola 1

Distribuované databázy

1.1 Základné definície

Databáza sa skladá z *dátových jednotiek*. Každá dátová jednotka má svoju *hodnotu*. Hodnoty dátových jednotiek v určitom čase spolu tvoria *stav* databázy.

Databázový systém (DBS) je súbor hardwaru a softwaru, ktorý umožňuje vykonávať príkazy, ktoré pristupujú k databáze, tzv. *databázové operácie*. Najdôležitejšími operáciami sú Read a Write, pričom Read(x) vráti hodnotu uloženú v dátovej jednotke x a Wrtie(x,y) zmení hodnotu x na y.

Databázový systém vykonáva každú operáciu *atomicky*. To znamená, že ich vykonáva ako keby sa vykonávali *sekvenčne*, teda jedna za druhou. Toto je možné dosiahnuť aj tým, že sa operácie budú vykonávať skutočne sekvenčne, no oveľa bežnejším prípadom je to, že sa vykonávajú *paralelne*. To znamená, že sa vykonáva viac operácií v jednom momente. Výsledok však musí byť rovnaký ako pri sekvenčnom vykonávaní.

Databázová transakcia (ďalej len transakcia) je postupnosť databázových operácií, ktoré sa majú vykonať.

Databázový systém takisto podporuje *riadiace transakčné operácie*, me-

novite *Start*, *Commit* a *Abort*. Zavolaním operácie Start program dáva najavo, že chce začať vykonávať novú transakciu. Koniec transakcie je možné určiť buď pomocou operácie Commit alebo Abort. Ak program zavolá Commit, znamená to, že transakcia skončila správne a bez problémov a všetky zmeny, ktoré zapríčinila, sa majú stať trvalými. Ak naopak zavolá Abort, znamená to, že transakcia skončila nekorektne a všetky zmeny majú byť zrušené.

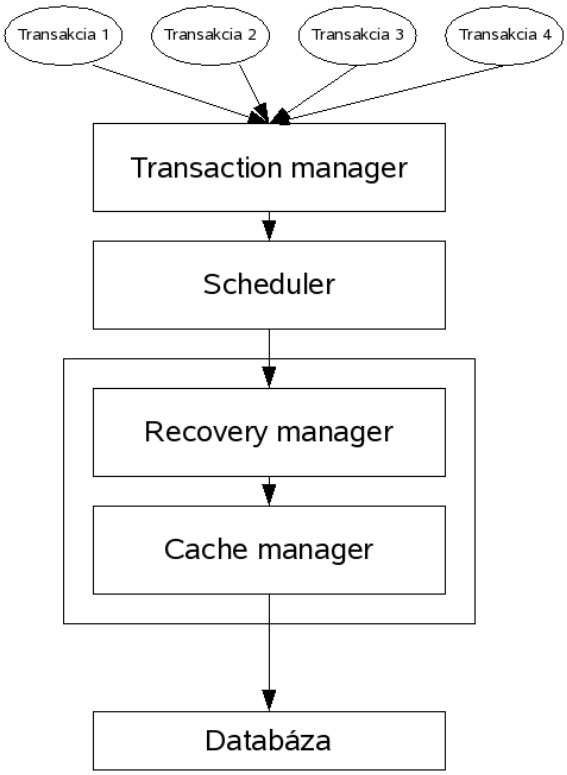
1.2 Databázový model

V nasledujúcich riadkoch si predstavíme jednoduchý model databázového systému. Kvôli nasledujúcim kapitolám je vhodné, aby čitateľ mal aspoň základný prehľad z akých častí sa databázový systém skladá ako ako funguje spracovávanie transakcií. Tento model sa bude skladať zo štyroch častí, ako je možné vidieť aj na obrázku 1.1.

Prvou časťou je *transaction manager*, ktorý prijíma databázové a transakčné operácie a posúva ich scheduleru. V distribuovaných databázových systémoch takisto určuje, ktorý uzol by mal vykonať danú operáciu.

Scheduler je zodpovedný za paralelný beh operácií rôznych transakcií. Jednotlivé operácie transakcií usporadúva do poradia tak, aby ich vykonávanie bolo serializovateľné a obnoviteľné. V niektorých databázových systémoch môžu byť na poradia vykonávania aj ďalšie požiadavky, napríklad, aby sa vykonávanie vyhýbalo kaskádovým abortom alebo aby sa generovali len striktné plány.

Hlavným cieľom *recovery managera* je, aby databáza v ktoromkoľvek momente obsahovala všetky dáta z transakcií, ktoré boli potvrdené a neobsahovala dáta z transakcií, ktoré boli zrušené. Podporuje operácie Start, Commit, Abort, Read a Write. Na komunikáciu s cache managerom používa operácie Fetch a Flush.



Obrázok 1.1: Databázový model

Úložiská databázových systémov možno rozdeliť z hľadiska dĺžky uchovávania informácií na dva typy — prechodné, napríklad operačná pamäť a trvalé, napríklad pevný disk. Prechodné úložisko (nazývané aj cache) je rýchlejšie a efektívnejšie pre prácu s dátami, preto *cache manager* do neho číta potrebné dáta. Slúži mu na to operácia Fetch(x). Na druhú stranu sa dáta v prechodnom úložisku nezachovajú v prípade poruchy systému. Na uloženie dát do trvalého úložiska cahce manager podporuje operáciu Flush(x).

1.3 Distribúcia dát

Doposiaľ sme uvažovali len o centralizovanom databázovom systéme a jeho vlastnostiach. No v rozsiahlych systémoch sa často stáva, že jeden databázový server nemusí byť dostačujúci. V takom prípade je často potrebné siahnuť po riešení, kedy sa viacero jednotlivých databázových serverov spojí do distribuovaného databázového systému.

Distribuovaný databázový systém je súbor viacerých jednotlivých databáz rozdelených na niekoľko fragmentov. Každý fragment je uložený na jednom alebo viacerých počítačoch, ktoré sú spojené sieťou, prostredníctvom ktorej prebieha komunikácia.

V distribuovaných (relačných) databázových systémoch častokrát dochádza k situácii, že určitú reláciu potrebujeme rozdeliť na podrelácie, zvané *fragmenty*, ktoré sú v systéme distribuované. Tento jav nazývame *fragmentácia*. Poznáme dva základné typy fragmentácie dát: horizontálnu fragmentáciu a vertikálnu fragmentáciu.

Horizontálna fragmentácia označuje fragmentáciu, keď sú distribuované záznamy tej istej relácie. Príklad: Každá pobočka má svoju vlastnú databázu kde má záznamy o svojich zamestnancoch. Ak chceme získať záznamy všetkých zamestnancov všetkých pobočiek, musíme urobiť zjednotenie jednotlivých fragmentov.

Vertikálna fragmentácia označuje fragmentáciu, keď niektoré atribúty sú uložené v jednej databáze, kým iné atribúty v inej. Príklad: Banka má v jednej databáze uložené iba prihlasovacie údaje, v ďalšej má údaje o kontách a ich výškach, ďalšia databáza môže obsahovať logy a štatistické záznamy. Ak chceme získať všetky údaje, musíme urobiť join (spojenie) jednotlivých relácií.

V distribuovaných databázových systémoch výrazne naberá na význame ďalšia možnosť. Tou je ukladanie kópií dát na rôzne databázy, tzv. *re-*

plikácia dát. Replikácia dát môže mať veľmi pozitívny dopad na spoľahlivosť systému, keďže výpadok jednej databázy nemusí okamžite spôsobiť nedostupnosť určitých dát, ak sú tieto dáta replikované. Problémy spojené s replikáciou dát v distribuovaných databázových systémoch však presahujú rámec tejto bakalárskej práce.

Kapitola 2

Atomický commit

2.1 Rozdiely oproti centralizovanému systému

Zamyslime sa teraz nad operáciou Commit nejakej distribuovanej transakcie T. Oproti operáciám Read(x) a Write(x, y), ktoré sa týkajú iba uzlu, kde je x uložené, operácia Commit sa týka všetkých uzlov, ktoré spracúvajú transakciu. Teda transaction manager musí poslať Commit všetkým uzlom, kde transakcia T pristupovala k dátovým jednotkám. To isté platí aj pre operáciu Abort. Teda jedna operácia (Commit alebo Abort) sa vykonáva na viacerých miestach v distribuovanom databázovom systéme. To je základný rozdiel medzi spracovávaním transakcií v centralizovanom a distribuovanom databázovom systéme.

Problém je však zložitejší ako sa na prvý pohľad môže zdať. Prvým z dôvodov je, že transakcia nie je potvrdená v momente, keď transaction manager posiela Commit, ale až keď data manager Commit vykoná. Môže sa stať, že transacton manager pošle Commit, ale scheduler ho odmietne a namiesto toho zruší transakciu. V takom prípade musí byť zrušená vo všetkých uzloch, kde pristupovala k dátovým jednotkám.

Ďalším rozdielom sú poruchy v distribuovaných systémoch. V centralizovaných systémoch sa jedná o poruchu celého systému, teda buď systém funguje celý správne a spracúva transakcie, alebo došlo k poruche a žiadne transakcie v tom momente nemôžu byť spracované. V distribuovanom systéme však môžeme mať čiastočné poruchy. Niektoré časti môžu fungovať, zatiaľ čo iné nie.

Táto vlastnosť vedie k celkovej väčšej spoľahlivosť distribuovaných systémov, keďže porucha nemusí automaticky viesť k pádu celého systému. Čiastočné poruchy sú však zdrojom netriviálnych problémov, ktoré je potrebné vyriešiť. Tieto problémy pomáha zvládnuť *Atomic Commit Protocol*, ktorý zabezpečuje konzistenciu dát a je čo najviac odolný voči poruchám, ktoré sa vyskytnú.

2.2 Poruchy a ich správa

Distribuovaný databázový systém si je možné predstaviť ako graf. Jeho uzlami budú jednotlivé databázové servery a hranami sieťové prepojenia. Každá z týchto častí môže byť náchylná na poruchu a preto sa protokol musí vysporiadať aj s takýmito situáciami. Poďme sa pozrieť bližšie na to, ako môžu poruchy jednotlivých častí ovplyvniť beh distribuovaného databázového systému.

Ak príde k pádu jednotlivého databázového serveru, predpokladáme, že server nie je schopný spracovávať žiadne transakcie a to až do doby reštartu systému. Inými slovami predpokladáme, že buď je server plne funkčný alebo nefunkčný a neuvažujeme že by pracoval s chybami. Výpadok jedného alebo časti serverov nemusí končiť pádom celého systému. Distribuovaný systém

môže fungovať ďalej, i keď na určitú dobu budú niektoré dáta nedostupné. No môže sa stať, že v momente pádu niektorého serveru sa môžu iné servery nachádzať v stave, v ktorom nevedia o rozhodnutí, ktoré urobil ten ktorý spadol. Keďže v takej situácii nemôžu samovoľne schváliť alebo zrušiť transakciu, dostanú sa do blokovaného stavu, čo je nepríjemná situácia. Protokol pre atomický commit by sa mal snažiť v čo najväčšej miere (ideálne úplne) tejto situácii vyhnúť.

To, ako poruchy jednotlivých sieťových prepojení ovplyvňujú distribuovaný databázový systém, značne závisí od topológie siete. Môže sa napríklad stať, že v dôsledku prerušenia niektorých sieťových prepojení určitý uzol zostane izolovaný alebo sa sieť môže rozpadnúť na niekoľko komponentov. V prípade, že nie je možné určitému uzlu doručiť správu, odosielateľ sa správa rovnako ako by príjemca bol spadnutý a naopak. V praxi je možné tieto poruchy detekovať, čo znamená, že uzol sa dozvie o tom, že správa doručená nebola.

2.3 Predpoklady protokolu atomického commitu

Na predchádzajúcich stranách bolo rozoberané aký problém má protokol pre atomický commit riešiť a na aké úskalia môže naraziť. V tomto momente je vhodné zhrnúť požiadavky na protokol, ktorý bude vhodným protokolom pre atomický commit. Tieto požiadavky je možné zhrnúť nasledovne:

Protokolom pre atomický commit je taký algoritmus, kde sa jeho účastníci dohodnú či danú transakciu schvália alebo nie. Presnejšie povedané, každý

účastník má možnosť jedného z dvoch *hlasov* (*áno* alebo *nie*) a musí dospieť k jednému z dvoch *rozhodnutí* — *schváleniu* (commit) alebo *zrušeniu* (abort) transakcie. Protokol pre atomický commit navyše spĺňa nasledovné podmienky:

1. Všetci účastníci, ktorí dospejú k rozhodnutiu, dospejú k rovnakému rozhodnutiu.

2. Účastník nemôže zmeniť rozhodnutie, ak už k nejakému dospel.

3. K rozhodnutiu *schváliť* (commit) transakciu je možné iba vtedy, ak *všetci* účastníci hlasovali *áno*.

4. Ak v systéme nie sú žiadne poruchy a všetci účastníci hlasovali za *áno*, tak rozhodnutie bude *schválenie* transakcie (commit).

5. Ak uvažujeme poruchy, ktoré je algoritmus ochotný tolerovať, tak v ľubovoľnom momente vykonávania a po odstránení porúch na dostatočný čas všetci účastníci dospejú k rozhodnutiu.

Jeden účastník protokolu, tzv. *koordinátor* máva významnejšie postavenie. Zvykne zbierať hlasy ostatných *účastníkov* alebo je významnejší tým, že je iniciátorom behu protokolu.

Kapitola 3

Dvojfázový commit

3.1 Popis protokolu

Najjednoduchším atomickým protokolom je *dvojfázový commit* (two-phase commit, 2PC). Meno dostal na základe toho, že sa skladá z dvoch fáz, pričom v prvej fáze účastníci hlasujú o schválení/zrušení transakcie a v druhom sa dozvedia výsledok hlasovania. Jeho priebeh možno rozdeliť do nasledujúcich bodov:

1. Koordinátor pošle správu `VOTE_REQ` o začatom hlasovaní
2. Keď účastník dostane správu `VOTE_REQ`, odpovie na ňu správou, ktorá obsahuje hlas účastníka, t.j. `VOTE_YES` alebo `VOTE_NO`. Ak účastník zahlasuje za `VOTE_NO` tak sa rozhodne pre Abort a skončí beh protokolu
3. Koordinátor pozbiera správy od všetkých účastníkov. Ak všetky hlasy účastníkov aj koordinátora boli `VOTE_YES` tak sa rozhodne pre Commit a pošle všetkým účastníkom správu `COMMIT`. Ináč sa rozhodne pre Abort a takisto pošle správu `ABORT` všetkým účastníkom, ktorý hlasovali `VOTE_YES` (tým, ktorí hlasovali `VOTE_NO` správu nie je potrebné zasielať). Koordinátor týmto ukončí beh protokolu.

4. Každý účastník, ktorý hlasoval `VOTE_YES` čaká na správu od koordinátora. Podľa prijatej správy sa rozhodne pre Commit alebo Abort a ukončí beh protokolu.

Ľahko vidieť, že daný protokol spĺňa podmienky 1–4 z podkapitoly 2.3. Na druhú stranu bez ďalších úprav nespĺňa podmienku číslo 5. Je tomu tak z dvoch dôvodov. Prvým dôvodom je, že v určitých fázach protokolu účastník čaká na správu. No môže sa stať, že druhý účastník spadne pred zaslaním tejto správy. Účastník protokolu vie zdetekovať, že uzol od ktorého čaká správu, spadol. V takom prípade musí vykonať špeciálnu, takzvanú *timeout akciu*, ktorá mu umožní vyjsť z tejto situácie.

Druhým dôvodom je, že rozhodnutie musí byť konzistentné s ostatnými účastníkmi a to aj v prípade porúch. Účastník si musí uchovávať informácie potrebné k tomu, aby po oprave porúch bolo možné zistiť k akému rozhodnutiu dospeli ostatní účastníci v danom behu protokolu.

3.1.1 Timeout akcie

Účastník čaká na správu od iných účastníkov v krokoch 2–4. V kroku 2 a kroku 3 je možné beztrestne zvoliť Abort, keďže v tomto momente ešte žiadny z účastníkov nedospel k žiadnemu rozhodnutiu. Iné je to však v kroku 4. V tomto kroku účastník čaká na správu Commit alebo Abort. K rozhodnutiu nemôže dospieť sám a ak mu správa o rozhodnutí nedôjde, zostáva mu jediná možnosť. Spýtať sa ostatných účastníkov na rozhodnutie. Tento postup sa nazýva *ukončovací protokol* pre dvojfázový commit.

Najjednoduchším riešením by bolo to, ak by uzol, ktorý nedostal správu čakal až do odstránenia poruchy. Po jej odstránení bude môcť komunikovať s koordinátorom a teda dospieť k správnemu rozhodnutiu. Veľkou nevýhodou

je, že účastník môže zostať na dlhý čas blokovaný.

Lepším riešením je, že sa účastník A spýta ostatných účastníkov a to tak, že im pošle správu `DECISION_REQ`. Účastník B, ktorému bola takáto správa zaslaná sa môže nachádzať v troch stavoch.

1. Účastník B sa rozhodol pre Commit a teda pošle naspäť účastníkovi A správu `COMMIT`
2. Účastník B sa rozhodol pre Abort a teda pošle naspäť účastníkovi A správu `ABORT`
3. Účastník B nedospel k rozhodnutiu a teda nemôže pomôcť účastníkovi A rozhodnúť sa

K nepríjemnej situácii dochádza ak všetci účastníci, s ktorými môže komunikovať účastník A sa nachádzajú v stave 3. V tom prípade zostávajú blokovaní až do odstránenia porúch v systéme. Každopádne ukončovací protokol redukuje pravdepodobnosť, že sa protokol dostane do blokovaného stavu. Zároveň si môžeme všimnúť, že takýto protokol už spĺňa podmienku 5 z podkapitoly 2.3. Stačí totiž, aby bolo možné komunikovať len s jedným účastníkom, ktorý už k rozhodnutiu dospel. Taký určite existuje — je to minimálne koordinátor — a preto po odstránení porúch účastníci dospejú k rovnakému rozhodnutiu.

3.1.2 Obnova

Majme účastníka A, ktorý mal poruchu a snaží sa dospieť k rozhodnutiu po obnove. Predpokladajme, že má zapísané všetky potrebné informácie z doterajšieho behu protokolu. V určitých situáciách sa dokáže účastník rozhodnúť sám. Napríklad ak došlo k poruche ešte pred tým než odoslal svoj hlas, môže si byť istý, že ostatní dospeli k rozhodnutiu zrušiť transakciu. Podobne je to

aj ak už dostal správu `COMMIT` alebo `ABORT` od koordinátora.

No v situácii, ak hlasoval áno, ale nedostal správu `COMMIT` alebo `ABORT` od koordinátora sa nevie rozhodnúť len sám. V skutočnosti sa nachádza v rovnakej situácii ako účastník čakajúci na správu od koordinátora. Preto v tomto momente je možné použiť ukončovací protokol na zistenie rozhodnutia ostatných účastníkov, pričom pre protokol platia rovnaké vlastnosti ako v predchádzajúcej podkapitole.

3.2 Modifikácie dvojfázového commitu

Uvedený priebeh popísaný v časti 3.1 nie je jedinou možnosťou ako môže dvojfázový commit prebiehať. Uvedieme si dve základné varianty ako je možné daný protokol modifikovať.

Prvá modifikácia šetrí počet poslaných správ. Očíslujme si účastníkov protokolu od 1 po n a nech účastník číslo 1 je koordinátor. Účastník číslo 1 pošle svoj hlas účastníkovi číslo 2. Ten na základe hlasu ktorý dostal a na základe svojho hlasu (ak dostal správu `VOTE_YES` a sám hlasuje `VOTE_YES`, tak pošle `VOTE_YES`, inak `VOTE_NO`), pošle hlas účastníkovi 3. A tak ďalej účastník i pošle hlas účastníkovi i+1, ktorý reťazovo posiela správu nasledujúcemu účastníkovi. Posledný účastník takto dostane rozhodnutie, na ktorom sa podieľali všetci účastníci protokolu a teda môže rozhodnúť, či bude transakcia schválená alebo zrušená. Toto rozhodnutie pošle naspäť, pričom rozhodnutie sa reťazovo preposiela medzi jednotlivými účastníkmi. Po tom, čo správa prejde "cez ruky" všetkým účastníkom sa vráti aj ku koordinátorovi. Ľahko možno vidieť, že počet poslaných správ je 2(n-1). Čo je pokles, keďže v kla-

sickom protokole je počet poslaných správ 3(n-1)[1]. Naproti tomu sa však výrazne zvýši počet kôl protokolu na 2(n-1) keďže žiadne dve správy nemôžu byť poslané paralelne. Možnosť využitia tejto modifikácie v praxi je mizivá.

Druhá modifikácia je o čosi praktickejšia. Modifikujeme v nej počet kôl a teda (teoreticky) aj rýchlosť protokolu. Funguje na nasledovnom princípe. V klasickom protokole posiela účastník svoj hlas iba koordinátorovi. Modifikujem protokol tak, že účastník pošle svoje rozhodnutie všetkým ostatným účastníkom. Rovnako aj koordinátor pošle svoj hlas pri správe `VOTE_REQ`[2]. V takomto prípade odpadá povinnosť koordinátora zasielať správy `COMMIT`, resp. `ABORT`, keďže každý z účastníkov dospeje k výsledku pozbieraním hlasov od všetkých účastníkov. Rovnako ako v klasickom protokole aj tu môžeme v prípade poruchy spustiť ukončovací protokol. Dokonca je možné urobiť obdobu ukončovacieho protokolu, kde okrem možnosti spýtať sa na výsledok pribudne možnosť spýtať sa na jednotlivý hlas účastníka. Potom by aj v určitých situáciách, v ktorých by klasický protokol zostal blokovaný, by takto modifikovaný protokol mohol skončiť bez blokovania. Znížili sme počet kôl protokolu z 3 na 2. Nevýhodou je, že sa zvýšil počet správ. Počet správ je pri takejto modifikácii kvadratický v závislosti od počtu účastníkov. Túto modifikáciu protokolu, resp. jej simuláciu som sa rozhodol implementovať.

[1] n-1 správ `VOTE_REQ`, n-1 správ `VOTE_YES` alebo `VOTE_NO` a n-1 správ `COMMIT` alebo `ABORT`

[2] Môžeme predpokladať, že ak koordinátor pošle `VOTE_REQ`, tak jeho hlas je áno. V opačnom prípade môže rovno poslať správu `ABORT`

Kapitola 4

Implementácia

4.1 Použité technológie

Ako súčasť mojej bakalárskej práce som implementoval simulátor protokolu pre atomický commit. Implementáciu som realizoval v jazyku ANSI C, kvôli čo najvyššej portabilite. Zároveň som využil diplomovú prácu Romana Pauera [Pau07], ktorý vytvoril simulátor distribuovaného systému spolu s knižnicou Thread Parallel Library (ďalej len TPL) [Pla03] pre toto prostredie. Táto knižnica ponúka funkcie na vytváranie viacero paralelných procesov a posielanie správ medzi nimi. Tieto funkcie mali prirodzene veľké využitie v simulátore atomického protokolu.

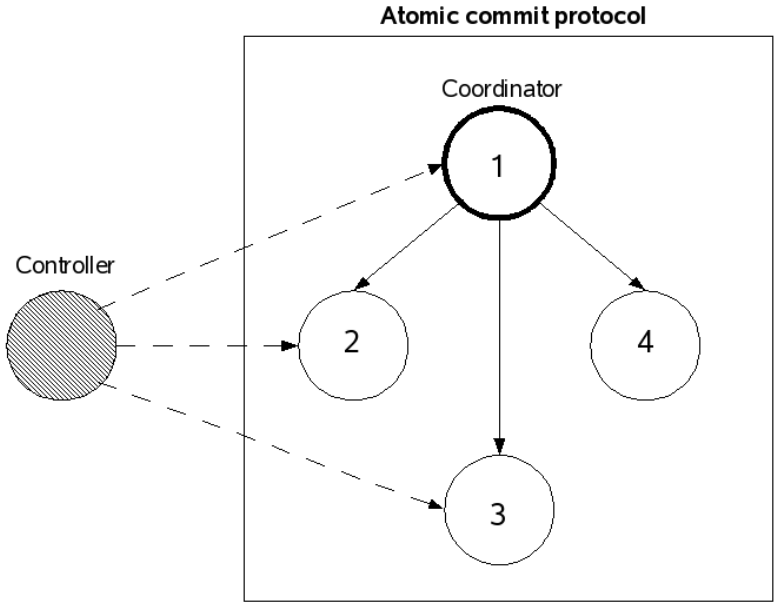
4.2 Model

Na znázornenie práce distribuovaného databázového systému sa zvyčajne používa nasledovný model: Distribuovaný databázový systém si je možné predstaviť ako graf, kde uzly sú databázové servery a hrany sieťové prepojenia. Uzly medzi sebou komunikujú posielaním správ. Na odoslanie správy slúži funkcia `send()`, ktorá je neblokovaná. To znamená, že po odoslaní beh

programu pokračuje ďalej a nečaká na odpoveď alebo potvrdenie prijatia správy. Na prijatie správy slúži funkcia `receive()`, ktorá je blokovaná, teda beh programu sa pozastaví a čaká na to, kým mu bude zaslaná nejaká správa. Detekciu poruchy si je možné predstaviť ako zaslanie špeciálnej správy od zdroju poruchy. Takéto správanie je v skutočnosti možné implementovať na úrovni sieťových protokolov.

Špecialitou databázového modelu je to, že po poruche a jej odstránení sa obnovený uzol vracia na svoje pôvodné miesto, čím sa tento model líši napríklad od modelu procesov v operačnom systéme, kde obnovený proces dostane iný identifikátor a teda je už iným účastníkom, hoci môže vykonávať ten istý program. Model povoľuje aj pridávanie alebo odoberanie ďalších uzlov, ale v databázovom modeli to obyčajne nie je skutočnosť, na ktorú sa upiera väčšia pozornosť.

Spomínaný simulátor však využíva striktnejší model. Spoločnými prvkami oboch modelov je abstrakcia grafu, komunikácia pomocou neblokovaného `send()` a blokovaného `receive()`. Simulátor podporuje iba jedinú topológiu siete a to kompletný graf. Na druhej strane nie je možné za behu do neho pridávať alebo z neho uberať uzly. Zhodenie jedného uzlu je chybou a správanie v takom prípade je nedefinované. Ak chceme teda simulovať poruchy jednotlivých uzlov, musíme to urobiť inak. Uzol nezhodíme skutočne, ale jeho výpadok len simulujeme. Uzol sa prepne do stavu poruchy, v ktorom na správy, ktoré prijme odpovedá správou o poruche. Správu však nespracuje ako by ju spracoval pri štandardnom behu protokolu, ale ju odignoruje. Zároveň pri prepnutí uzlu do stavu poruchy zašle správu o poruche všetkým uzlom. Týmto je možné simulovať detekciu poruchy u ostatných uzlov.



Obrázok 4.1: Model programu

4.3 Štruktúra programu

Program pri štarte vytvorí niekoľko procesov pomocou funkcie `tpl_initialize()`. Každý z týchto procesov predstavuje jedného účastníka protokolu. Každý účastník sa však skladá z dvoch vlákien. Prvé *listener*, ktoré čaká a počúva správy, ktoré mu prídu. Druhé *sender* je vlákno, ktoré spracúva prijaté správy, spravuje transakcie a zároveň posiela správy ostatným procesom, ak je to potrebné. Inicializácia účastníka vyzerá nasledovne

```
[10pt]
    my_id = id;
    nodes_count = nodes;
    node_init();
```

```
debug_print(DEBUG_START_STOP, "Listener %d created\n", my_id);
pthread_create(&listener, NULL, node_listen, NULL);
```

Okrem toho jeden z procesov využijeme a spravíme z neho tzv. *controller*. Tento nebude účastníkom protokolu ale bude slúžiť na ovládanie ostatných procesov v tom zmysle, že im bude zasielať správy o vytvorení transakcie alebo o potrebe ukončiť beh celého programu. Ostatné procesy následne schvália alebo zamietnu transakciu pomocou protokolu pre atomic commit alebo bezchybne ukončia program.

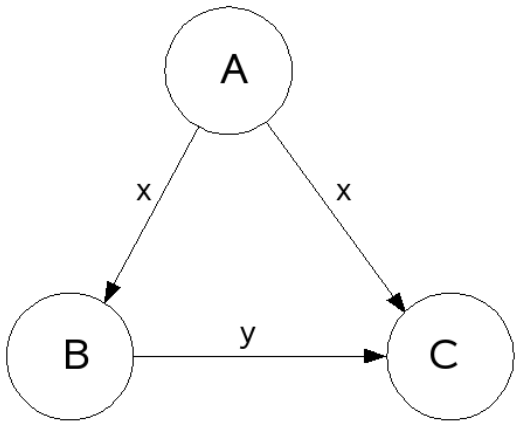
4.4 Problémy a ich riešenia

Prvým problémom bolo úspešné ukončenie všetkých procesov v tomto prostredí. Zdanlivé riešenie je, že ak proces chce skončiť, upovedomí o tom ostatné procesy a skončí. Ale nie je to správne riešenie. To, že proces chce skončiť a nepotrebuje už nič od ostatných účastníkov neznamená, že oni nepotrebujú nič od neho. Preto ak chce proces skončiť tak upovedomí o tom, že chce skončiť ostatné procesy. Ostatné procesy si toto upovedomenie zaznamenajú. Proces sa následne skutočne ukončí až vtedy, ak dostane správu od všetkých ostatných procesov, že chcú svoj beh tiež ukončiť. V tomto momente už všetky procesy môžu bezpečne skončiť.

Ďalším problémom je identifikátor transakcie. Každá transakcia potrebuje mať svoj jednoznačný identifikátor, aby sa dalo odlíšiť, z ktorého behu protokolu je určitá správa. V centralizovanom systéme to môže byť jedna premenná, ktorá sa postupne zvyšuje. No v distribuovanom systéme nie je možné mať globálne počítadlo. Riešením by mohlo byť, že by počítadlo bolo u niektorého procesu a ostatné procesy by zisťovali jeho stav pomocou správ. No ak systém nechceme takto centralizovať, môžeme siahnuť po nasledujúcom riešení. Každý proces bude mať vlastné počítadlo. Identifikátor novovytvorenej transakcie vytvoríme tak, že aktuálnu hodnotu vynásobíme

počtom účastníkov a pripočítame offset — identifikátor procesu, napríklad nasledovne:

```
long node_get_transaction_id(void)
{
    long id_transaction = node_transaction_counter*NODES+my_id;
    node_transaction_counter++;
    return id_transaction;
}
```



Obrázok 4.2: Poradie správ

Ďalšou vecou, ktorú si je potrebné uvedomiť je to, v akom poradí správy prichádzajú. To že nejaká správa bola odoslaná skôr v žiadnom prípade neznamená, že bude aj skôr doručená. Všimnime si príklad na obrázku 4.2. Nech uzol A poslal tú istú správu x uzlom B a C. Uzol B po prijatí správy následne odoslal správu y uzlu C. Ktorú správu prijme C skôr, x alebo y? Nič nezaručuje to, že budú prijaté v prirodzene očakávanom poradí, t.j. správa

x prvá. Za určitých okolností môže dôjsť k tomu, že správa y príde skôr. Túto skutočnosť je treba mať pri implementácii na pamäti. Jedno z riešení je ukladať si správy, ktoré momentálne protokol "nepotrebuje", do takzvanej fronty a v patričný čas ich vybrať. V určitých prípadoch však môžeme siahnuť aj po inom riešení. Je garantované, že "oneskorená" správa po určitom čase príde. Napríklad ak príde najskôr správa o hlase (napr. `VOTE_YES`), správa o začiatku hlasovania (`VOTE_REQ`) po určitom čase s určitosťou príde. V takomto prípade si môžeme predvytvoriť štruktúru na zapamätanie si hlasov jednotlivých uzlov. Ak potom neskôr príde správa o začatí hlasovania o transakcii T, jednoducho prepojíme dáta o transakcii s dátami o hlasovaní, ktoré sme v doterajšom priebehu protokolu nazbierali.

4.5 Hlavná časť protokolu

Pozrime sa na najdôležitejšiu časť implementácie protokolu. Po odstránení rôznych pomocných funkcií, ktoré pomáhajú pri odladovaní a odstraňovaní chýb vyzerá telo hlavnej funkcie nasledujúco:

```
void p2pcb_handle_message(message m)
{
  transaction_list *list;
  transaction *t = NULL;
  long transaction_id;
  int node_id;

  node_id = node_get_id();
  list = node_get_transaction_list();

  switch (m.tag)
  {
```

```
case CMESSAGE_TRANSACTION_BEGIN:
  transaction_id = node_get_transaction_id();
  t = transaction_begin(transaction_id, node_id, node_id);
  transaction_list_add(list, t);
  p2pcb_clear_votes(transaction_id);
  p2pcb_save_vote(transaction_id, node_id, VOTE_YES);
  node_send_all_transaction_message(
    MESSAGE_2PC_VOTE_REQ, transaction_id
  );
  return;
case MESSAGE_2PC_VOTE_REQ:
  transaction_id = (long)(m.message);
  t = transaction_begin(transaction_id, node_id, m.sender);
  transaction_list_add(list, t);
  p2pcb_save_vote(transaction_id, m.sender, VOTE_YES);
  node_send_all_transaction_message(
    p2pcb_get_vote_message_tag(
      p2pcb_vote(transaction_id)), transaction_id
  );
  return;
case MESSAGE_2PC_VOTE_YES:
  transaction_id = (long)(m.message);
  p2pcb_save_vote(transaction_id, m.sender, VOTE_YES);
  return;
case MESSAGE_2PC_VOTE_NO:
  transaction_id = (long)(m.message);
  p2pcb_save_vote(transaction_id, m.sender, VOTE_NO);
  return;
case MESSAGE_2PC_ABORT:
```

```
    case MESSAGE_2PC_COMMIT:
      return;
    default :
      return;
  }
}
```

Protokol je v podstate pomerne jednoduchý. Ak procesu príde správa týkajúca sa protokolu, zavolá funkciu `p2pcb_handle_message()`. Tá sa so správou vysporiada. V prípade, že ide o začatie transakcie, tak sa vytvorí dátová štruktúra pre transakciu a pridá medzi ostatné transakcie. Zároveň sa inicializuje hlasovanie a koordinátor zaznačí svoj hlas. Následne pošle správu o hlasovaní `MESSAGE_2PC_VOTE_REQ` ostatným účastníkom. Tí si vytvoria štruktúru pre transakciu a pridajú do vlastnej dátovej štruktúry určujúcej o stav a schválenie jednotlivých transakcií. Následne pošlú správu ostatným účastníkom. Funkcia `p2pcb_save_vote()` okrem uloženia hlasu účastníka robí aj to, že skontroluje aktuálny stav hlasovania. To znamená, že v prípade, že všetci hlasovali za schválenie transakcie, transakcia sa schváli. Ak niekto hlasoval za zrušenie transakcie, transakcia sa zruší. Správy `MESSAGE_2PC_COMMIT` a `MESSAGE_2PC_ABORT` ako už bolo spomínané sa v tejto verzii protokolu nevyužívajú.

Kapitola 5

Trojfázový commit

5.1 Výhody trojfázového protokolu

V kapitole 3 bolo spomínané, že za určitých okolností môže zostať dvojfázový commit blokovaný. Táto vlastnosť je nežiadúca, hoci na jej výskyt treba súhru viacerých okolností. Keďže k jeho zablokovaniu dochádza len veľmi zriedkavo, resp. nikdy, v praxi sa používa najčastejšie nejaká verzia dvojfázového protokolu. Trojfázový commit (three-phase commit, 3PC) túto nepríjemnú vlastnosť nemá za predpokladu, že uvažujeme iba výpadky uzlov a neuvažujeme výpadky jednotlivých sieťových prepojení.

5.2 Popis protokolu

Trojfázový commit prebieha nasledovne.

1. Koordinátor pošle správu `VOTE_REQ` o začatom hlasovaní všetkým účastníkom
2. Účastník po obdržaní správy `VOTE_REQ` pošle ako odpoveď `VOTE_YES` alebo `VOTE_NO`, podľa hodnoty svojho hlasu. Ak pošle `VOTE_NO` zároveň transakciu zruší (Abort) a skončí beh protokolu.

3. Koordinátor pozbiera správy od všetkých účastníkov. Ak niektorý z účastníkov, prípadne sám koordinátor hlasoval `VOTE_NO` tak sa rozhodne pre Abort a pošle všetkým účastníkom, ktorý hlasovali `VOTE_YES` správu `ABORT`. Ináč pošle správu `PRE-COMMIT` všetkým účastníkom.

4. Účastník, ktorý hlasoval `VOTE_YES` obdrží buď správu `ABORT` alebo `PRE-COMMIT`. V prvom prípade transakciu zruší (Abort) a skončí beh protokolu. V druhom prípade pošle koordinátorovi správu `ACKNOWLEDGMENT`.

5. Koordinátor zozbiera všetky správy `ACKNOWLEDGMENT`. Následne transakciu schváli (Commit) a pošle všetkým správu `COMMIT`. Beh protokolu sa týmto pre koordinátora končí.

6. Účastník čaká na správu `COMMIT`. Po jej obdržaní transakciu schváli (Commit) a ukončí beh protokolu.

Trojfázový commit sa veľmi podobá dvojfázovému commitu, akurát pribudla fáza pre-commitu. Na prvý pohľad sa môže zdať, že pridaním tejto fázy nič nové nezískame, ba naopak. Posledné správy protokolu `ACKNOWLEDGMENT` a `COMMIT` sa zdajú byť nadbytočné, keďže príjemca čaká iba na danú správu a odosielateľ nemôže poslať nič iné, iba danú správu. Hlbšia analýza tohoto protokolu však odhalí, že tieto správy tam majú svoj dôvod.

5.3 Neblokovanie protokolu

Na objasnenie si zavedieme nasledovné pravidlo o neblokovaní [BHG86]:
NP: Ak sa niektorý fungujúci uzol nachádza vo fáze neistoty[1]*, tak sa žiadny uzol (fungujúci alebo spadnutý) nemohol rozhodnúť pre Commit*

[1] *Fáza neistoty* je úsek v behu protokolu, počas ktorého sa uzol bez ďalších informácií nemôže rozhodnúť ani pre Commit ani pre Abort

Dvojfázový commit toto pravidlo nespĺňa. Uvažujme typickú situáciu, keď koordinátor rozošle účastníkom správu `COMMIT`. Účastník sa rozhodne pre potvrdenie (Commit) transakcie. No ostatní účastníci v tomto momente nemusia vedieť rozhodnutie koordinátora. Preto dvojfázový commit toto pravidlo porušuje.

Naopak, pridanie pre-commit fázy pomáha toto pravidlo splniť. Ak účastník obdrží správu `PRE-COMMIT`, vie, že všetci ostatní účastníci hlasovali za Commit transakcie. Tým pádom už nie je vo fáze neistoty. Hoci výsledok už vie, nerozhodne sa pre Commit transakcie okamžite, ale chvíľu počká. Tu prichádzajú na rad "nadbytočné" správy `ACKNOWLEDGMENT` a `COMMIT`. Koordinátor po obdržaní správy `ACKNOWLEDGMENT` od všetkých účastníkov vie, že žiadny účastník už nie je vo fáze neistoty. Preto môže transakciu schváliť. Takisto účastník po obdržaní správy `COMMIT` vie, že žiadny účastník už nie je vo fáze neistoty, keďže správu `COMMIT` koordinátor pošle po zozbieraní správ `ACKNOWLEDGMENT` od všetkých účastníkov. Z toho vidno, že trojfázový commit spĺňa pravidlo o neblokovaní.

Splnenie tohto pravidla skutočne zabezpečí neblokovanie protokolu. V prípade poruchy všetci zostávajúci účastníci musia len zistiť, či sa všetci nachádzajú vo fáze neistoty alebo nie. Ak sú všetci vo fáze neistoty, môžu transakciu zrušiť (Abort), keďže podľa pravidla o neblokovaní *neexistuje* účastník, ktorý by sa rozhodol pre Commit. Naopak ak zistia, že niektorý z účastníkov rozhodnutie už vie, rozhodnú sa aj oni rovnako. Formálny dôkaz neblokovania a ďalšie podrobnosti môže čitateľ nájsť v [BHG86].

Kapitola 6

Iné protokoly pre atomický commit

6.1 Decentralized non-blocking atomic commit

Myšlienkou je tento protokol veľmi podobný implementovanej modifikácii dvojfázového protokolu. Rozdiel je v tom, že je to obdoba trojfázového protokolu, ktorá prostredníctvom broadcastovania správ redukuje počet kôl správ na tri. V prvom kole koordinátor iniciuje správou o začatí hlasovania protokol, v druhom kole každý účastník pošle svoj hlas všetkým (inými slovami broadcastuje) a v treťom kole zasa účastníci broadcastujú správu `PRE-COMMIT`. Po prijatí správy `PRE-COMMIT` od všetkých účastníkov účastník transakciu schváli.

Veľkou výhodou tohoto protokolu je to, že sa neblokuje a oproti trojfázovému commitu má výrazne nižší počet kôl. Riešenie prípadov ak nastane porucha je pomerne komplikované, a preto ho nemá význam v tejto práci bližšie ro-

zoberať. Viac informácií o tomto protokole je možné sa dočítať v [GS95].

6.2 Modular decentralized three phase commit

Tento protokol je opäť decentralizovanou modifikáciou trojfázového commitu. Rozdiel oproti predchádzajúcemu je taký, že v druhom kole sa neposielajú hlasy všetkým, ale iba určitej podmnožine účastníkov. V treťom kole potom táto podmnožina brodcastuje správu `PRE-COMMIT`. Veľkosť tejto podmnožiny je určená počtom, koľko jednotlivých pádov účastníkov je protokol ochotný tolerovať. Tento počet je zvyčajne konštanta (2 alebo 3), čo vedie k výraznému ušetreniu počtu správ a teda odľahčeniu protokolu. Podrobnejšie tento protokol rozoberá [GL96]

6.3 Asynchronous non-blocking coordinator logical log

Tento protokol prišiel so zaujímavou myšlienkou. Od schedulera totiž vyžaduje, aby generoval iba rozvrhy vyhýbajúce sa kaskádovým abortom. V takom prípade nemôže dôjsť k abortu transakcie neschválením nejakým účastníkom, ale len poruchami v systéme. Tým pádom je možné z protokolu odstrániť fázu hlasovania. Dostaneme tak veľmi rýchly (2 kolá) protokol, ktorý sa neblokuje. Ďalšie informácie o tomto protokole je možné nájsť v [AP98]

Porovnanie jednotlivých protokolov[1]:

Protokol	**Počet kôl**	**Počet správ**	**Počet správ(broadcast)**
2PC	3	3n	n+2
3PC	5	5n	2n+3
DNB-AC	3	n(2n+1)	2n+1
MD3PC	3	3n(f+1)	n+f+2
ANB-CLL	2	n(n+1)	n+1

[1]n označuje celkový počet účastníkov protokolu, f označuje počet koľko pádov účastníkov je protokol ochotný tolerovať

Záver

V práci sme zhrnuli poznatky z problematiky atomického commitu pre distribuované databázy. Rozobrali sme základné protokoly pre atomický commit, teda dvojfázový commit v 3 a trojfázový commit v 5. Popísali sme ich vlastnosti a vysvetlili problém blokovania protokolu. Zároveň sme stručne rozobrali niektoré ďalšie protokoly vychádzajúce z týchto dvoch základných, čo určite do značnej miery naznačilo možnosti vylepšovania a experimentovania s protokolmi pre atomický commit.

Súčasťou práce je takisto aj základná implementácia decentralizovanej verzie dvojfázového commitu. Implementácia ďalších protokolov je už o niečo jednoduchšia, keďže mnohé implementačné problémy súvisiace so simuláciou distribuovaného databázového systému boli úspešne zvládnuté. Táto simulácia môže pomôcť pri skúmaní praktických vlastností ako porovnávanie efektivity jednotlivých protokolov. Prípadne môže slúžiť ako inšpirácia pre čitateľa zaujímajúceho sa o programovanie distribuovaných aplikácií.

Literatúra

[AP98] Maha Abdallah and Philippe Pucheral. A single-phase non-blocking atomic commitment protocol. In *In 9th International Conference on Data and Expert Systems Applications(DEXA*, 1998.

[BHG86] Philip A Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency control and recovery in database systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[CT96] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43:225–267, 1996.

[GL96] Rachid Guerraoui and Mikel Larrea. Reducing the cost for non-blocking in atomic commitment, 1996.

[GS95] Rachid Guerraoui and André Schiper. The decentralized non-blocking atomic commitment protocol. In *Proc. of the 7th IEEE Symposium on Parallel and Distributed Systems*, pages 2–9, 1995.

[Pau07] Roman Pauer. Centralizovaný systém pre výmenu správ medzi paralelnými procesmi. 2007.

[Pla03] T. Plachetka. *Event-Driven Message Passing and Parallel Simulation of Global Illumination*. Phd thesis, University of Paderborn, Faculty of Electrotechnics, Informatics and Mathematics, 2003.