Univerzita Komenského v Bratislave
Fakulta Matematiky, Fyziky a
Informatiky
Katedra Informatiky

Distribuovaný transakčný systém

Diplomová práca

Michal Štrbka

Školiteľ: Dr. Tomáš Plachetka

Bratislava, 2007

Čestne prehlasujem, že som túto diplomovú prácu vypracoval samostatne s použitím uvedenej literatúry.

V Bratislave, 7.Mája 2007 .....

Michal Štrbka

Poďakovanie

Ďakujem svojej rodine za podporu počas štúdia a v neposlednom rade svojmu diplomovému vedúcemu Dr. Tomášovi Plachetkovi za neoceniteľné rady pri písaní diplomovej práce.

Obsah

Kapitola 1

Úvod

Napriek tomu, že výpočtová sila počítačov neustále rastie, ešte stále existuje mnoho problémov, na ktoré je výkon súčasných počítačov nepostačujúci. Kvôli zrýchleniu výpočtov sa ľudia snažia písať čo najefektívnejšie algoritmy. Ale pre mnohé problémy aj tie najefektívnejšie algoritmy, bežiace na jedinom počítači, majú veľkú časovú zložitosť, často aj niekoľko dní, mesiacov, či dokonca rokov. Ako teda dosiahnuť ich rýchlejší výpočet? Jednoduchá a logická odpoveď je zapojiť do výpočtu viac počítačov. Preto je nutné prácu jednotlivým počítačom (programom, ktoré na nich bežia a rátajú daný problém) rozdeliť. Rozdeľovať prácu "manuálne" by asi nebol optimálny prístup, preto je nutné následne upraviť aj algoritmus výpočtu tak, aby sa práca rozdeľovala "automaticky". Pod slovom automaticky rozumiem, že počítače si navzájom rozdeľujú prácu samy podľa dopredu dohodnutých pravidiel (napr. workstealing, factoring, ...). To nás vedie k tomu, že počítače, konkrétne aplikácie, ktoré na nich bežia, musia spolu vedieť komunikovať.

Ak sa lepšie pozrieme na stavbu ľubovoľného paralelného programu, všimneme si, že sa skladá z dvoch častí. Jedna je samotné riešenie problému. Táto časť je pre každý problém rôzna. Avšak druhá časť programu slúži na komunikáciu medzi jednotlivými aplikáciami, aby si medzi sebou mohli vymieňať medzivýsledky ich výpočtov. Po krátkej úvahe si uvedomíme, že tá sa nemusí vôbec meniť, pretože riešenie problému nie je závislé od toho, akým spôsobom spolu aplikácie komunikujú. Preto je možné vytvoriť jednu knižnicu, pomocou ktorej budú aplikácie v distribuovanom systéme spolu komunikovať.

Aké vlastnosti by ale mala mať táto knižnica? Okrem možnosti posielania správ medzi uzlami, určite by mala byť odolná voči chybám (fault tolerant). Odolnosťou voči chybám myslím odolnosť voči výpadkom procesov. Predstavme si, že spustíme

náročný výpočet na niekoľkých počítačoch a počas výpočtu jeden z počítačov prestane komunikovať, alebo prestane úplne počítať. Ak sú výpočty na počítačoch závislé od ostatných, tak nedokážeme ani len predvídať, aké chyby mohli nastať výpadkom jedného z uzlov. Nájdenie takejto chyby by zabralo veľa času a námahy a navyše by bolo potrebné celý výpočet zopakovať (a dúfať, že už nedôjde k ďalším výpadkom).

1.1 Cieľ diplomovej práce

Cieľom mojej diplomovej práce je zadefinovať teoretický model pre posielanie správ, ktorý je odolný voči výpadkom komunikujúcich uzlov. Následne na základe tohto teoretického modelu navrhnúť architektúru a implementáciu distribuovaného systému pre posielanie správ, ktorý bude odolný voči výpadkom procesov.

1.2 Členenie diplomovej práce

V druhej kapitole oboznamujem čitateľa s dvomi najpoužívanejšími knižnicami pre posielanie správ. V tretej kapitole zadefinujem teoretický model pre posielanie správ. Vo štvrtej popisujem architektúru distribuovaného systému, ktorý bude vykonávať úlohu teoretického modelu a v piatej navrhnem programy vykonávané v každom procese, ktoré zabezpečia funkčnosť systému. Správnosť týchto programov aj dokážem. Napokon v poslednej kapitole zhrniem výsledky mojej diplomovej práce a navrhnem ďalšie možné optimalizácie.

Kapitola 2

Súčasné komunikačné knižnice

V súčasnosti najznámejšie a najpoužívanejšie knižnice pre posielanie správ sú MPI (Message Passing Interface) a PVM (Parallel Virtual Machine).

2.1 PVM (Parallel Virtual Machine)

Vývoj PVM [PVMTut] začal v lete roku 1989. Medzičasom bola už vyvinutá verzia PVM 3.4. Základnou myšlienkou PVM je "virtuálny stroj" — množina heterogénnych hostiteľov, tzv. pvmd (pvm daemon), spojených sieťou, ktorý sa pre koncového užívateľa tvári ako jeden veľký paralelný počítač. Jedným z hľadísk tvorby virtuálneho stroja bolo, ako si procesy medzi sebou vymieňajú dáta. Toto je v PVM urobené pomocou posielania správ. Pri tvorbe PVM sa ďalej kládol dôraz na jednoduchosť používania a pochopenia PVM rozhrania. Portabilita bola považovaná za dôležitejšiu ako výkon z dvoch dôvodov: komunikácia cez internet je pomalá; a výskum bol zameraný najmä na heterogénnosť a odolnosť voči chybám.

2.2 MPI (Message Passing Protocol)

Vývoj MPI [MPForum] sa začal v apríli roku 1992. Impulzom pre vytvorenie MPI bolo, že každý výrobca MPP (Massively Parallel Processor) vytváral vlastné API pre posielanie správ. Kvôli tomuto nebolo možné písať portabilné paralelné aplikácie. MPI bolo mienené ako štandard, ktorý bude implementovaný každým výrobcom MPP na jeho systém. Hlavným cieľom MPI bolo dosiahnuť čo najväčší výkon. MPI má preto veľkú množinu komunikačných rutín. Avšak aplikácie

používajúce MPI-1 neboli prenosné na sieť pracovných staníc, pretože nebola žiadna štandardná metóda na spúšťanie MPI na oddelených hostiteľoch. Preto bol vytvorený štandard MPI-2, ktorý rieši tieto nedostatky.

Kapitola 3

Sémantika systému pre posielanie správ

Jedným z fundamentálnych teoretických komunikačných modelov je kanálový model [JGF96]. Procesy v kanálovom modeli navzájom komunikujú prostredníctvom kanálov, do ktorých správy zapisujú, resp. z nich správy čítajú. Kanály sú neohraničené FIFO (First-In-First-Out[1]) dátové štruktúry.

Tento model preto nemôže byť priamo namapovaný na súčasnú komunikačnú štruktúru. Komunikačné linky majú nulovú kapacitu, sú to teda nulové FIFO. Preto je na súčasné komunikačné vedenia potrebné vytvoriť nové (iné) modely, ktoré sa budú dať na ne namapovať a zároveň budú rovnako silné ako kanálový model.

Zároveň od nového modelu (a na jeho základe implementovanej knižnice) požadujem, aby bol odolný voči výpadkom.

Ako vyplýva z publikácie [Pla06], MPI je slabšie ako kanálový model, preto MPI nespĺňa moje požiadavky na teoretický model.

Z manuálu pre PVM [PVMTut], ako aj z publikácie [PVMvsMPI] vyplýva, že PVM nie je úplne odolné voči výpadkom. PVM kontroluje výpadky pvmd[2] pomocou pravidelných "heart-beat" správ, čo nie je celkom prijateľný prístup (active polling), alebo detekovaním timeoutu v prípade priamej žiadosti od iného pvmd. Takže zistenie výpadku nie je najrýchlejšie možné. Navyše v PVM nesmie dôjsť k výpadku hlavného pvmd. V prípade výpadku hlavného pvmd spadne celý virtuálny stroj. Preto PVM nespĺňa moje požiadavky na distribuovanosť systému, a

[1] prvý-dnu-prvý-von

[2] pvm daemon

teda absolútnu nezávislosť od výpadku ľubovoľného uzla[3].

3.1 Komunikačný model

Pre svoju prácu som si zvolil model [Pla06], ktorý sa dá namapovať na súčasnú komunikačnú štruktúru. Zvolený model je obdobou databázového modelu, kde na výmenu správ slúžia operácie CREATE, vytvorenie správy, DESTROY, odstránenie správy, SEND, poslanie správy, RECV, prijatie správy. Tieto operácie zodpovedajú operáciám databázového modelu INSERT, DELETE, WRITE, READ.

Vybraný model je rovnako silný ako kanálový model a je výpočtovo silnejší ako MPI[Pla06] (Message Passing Protocol). Vyplýva to z toho, že model je rovnako silný ako kanálový model a ten je silnejší ako MPI.

Avšak model nie je odolný voči chybám (fault tolerant), preto ho ešte rozšírim o ďalšie operácie START, spustenie nového procesu a END, odstránenie procesu.

3.2 Špecifikácia

Definícia (Poslanie základnej operácie) Vykonanie základnej operácie znamená predanie operácie z procesu do systému pre posielanie správ.

Definícia (Reprezentácia základných operácií) Všetky základné operácie sú sedmice [op, x, Y, m, f, s, t], kde op $\in$ {CREATE, DESTROY, SEND, RECV, START, END}; x je identifikátor procesu, ktorý vykonal operáciu; Y je množina identifikátorov procesov; m je správa; f je booleovská funkcia definovaná na m (filter); s je NULL alebo referencia na semaforový objekt, ku ktorému môže pristupovať systém; t je časová pečiatka poslania operácie (t.j. čas, kedy bola operácia prečítaná systémom)

Definícia (Oblasť procesu) Oblasť procesu je pamäťový priestor, kde sú uložené správy prislúchajúce danému procesu. K správe môže pristupovať (t.j. čítať, zapisovať) iba proces, v ktorého oblasti sa správa nachádza. Proces môže vytvoriť a zrušiť správu iba pomocou operácií CREATE a DESTROY. Systém vytvára, ruší a pristupuje k správam v oblastiach procesov iba ako je definované v tejto sekcii.

[3]PVM sa správa ako centralizovaný systém.

SC(x) bude označenie pre oblasť procesu x, SC(*) bude znamenať zjednotenie oblastí všetkých procesov.

Definícia (Párovanie operácií) Hovoríme, že dve základné operácie $BO_1 = [op_1, x_1, Y_1, m_1, f_1, s_1, t_1]$, $BO_2 = [op_2, x_2, Y_2, m_2, f_2, s_2, t_2]$ sú pár práve vtedy, keď

$$(op_1 = SEND \wedge op_2 = RECV \wedge x_1 \in Y_2 \wedge x_2 \in Y_1 \wedge f_2(m_1) \wedge (\forall BO_1^{'} = [po_1^{'}, x_1^{'}, Y_1^{'}, m_1^{'}, f_1^{'}, s_1^{'}, t_1^{'}] \in SC(*) : (BO_1^{'} \equiv BO_1 \vee op_1^{'} \neq SEND \vee x_1^{'} \notin Y_2 \vee x_2^{'} \notin Y_1 \vee \neg f_2(m_1^{'}) \vee t_1^{'} \geq t_1)) \wedge (\forall BO_2^{'} = [op_2^{'}, x_2^{'}, Y_2^{'}, m_2^{'}, f_2^{'}, s_2^{'}, t_2^{'}] \in SC(*) : (BO_2^{'} \equiv BO_2 \vee op_2^{'} \neq RECV \vee x_1 \notin Y_2^{'} \vee x_2^{'} \notin Y_1 \vee \neg f_2^{'}(m_1) \vee t_2^{'} \geq t_2)))$$

Neformálne, operácia send $BO_1 = [op_1, x_1, Y_1, m_1, f_1, s_1, t_1]$ je v páre s operáciou recv $BO_2 = [op_2, x_2, Y_2, m_2, f_2, s_2, t_2]$ vtedy, keď množina adresátov Y_1 obsahuje x_2, množina odosielateľov Y_2 obsahuje x_1, filtrovania funkcia f_2 akceptuje správu m_1 a ani jedna z BO_1, BO_2 nemôžu byť nahradené staršou operáciou, ktorá by spĺňala predošlé podmienky.

Definícia (Vykonanie operácie CREATE) Vykonanie operácie [CREATE, x, Y, m, f, s, t] pozostáva z nasledujúcich akcií vykonaných v atomickom kroku:

1. Systém vytvorí novú správu v SC(x)
2. Ak $s \neq NULL$, tak systém vykoná semaphore-signal(s)
3. Systém odstráni operáciu z SC(x)

Definícia (Vykonanie operácie DESTROY) Vykonanie operácie [DESTROY, x, Y, m, f, s, t] pozostáva z nasledujúcich akcií vykonaných v atomickom kroku:

1. Systém odstráni m z SC(x)
2. Ak $s \neq NULL$, tak systém vykoná semaphore-signal(s)
3. Systém odstráni operáciu z SC(x)

Definícia (Vykonanie operácie START) Vykonanie operácie [START, x, Y, m, f, s, t] pozostáva z nasledujúcich akcií vykonaných v atomickom kroku:

1. Systém vytvorí nový proces
2. Systém vytvorí novú správu m v SC(x)

3. Systém skopíruje do m hodnotu NEW-ID = x', kde x' je identifikátor nového procesu. (Tento identifikátor je jednoznačný v celom systéme).

4. Ak $s \neq NULL$, tak systém vykoná semaphore-signal(s).

5. Systém odstráni túto operáciu z SC(x)

Poznámka: Predpokladám, že každý proces pozná svoj vlastný identifikátor. Teda nový proces vie, že má identifikátor x' a proces s identifikátor x sa meno nového procesu dozvie zo správy m.

Definícia (Vykonanie operácie END) Vykonanie operácie BE = [END, x, Y, m, f, s, t] pozostáva z nasledujúcich akcií vykonaných v atomickom kroku:

1. Systém odstráni všetky operácie okrem BE z SC(x)

2. Systém odstráni všetky správy z SC(x)

3. Ak $s \neq NULL$, tak systém vykoná semaphore-signal(s)

4. Systém odstráni BE z SC(x)

5. Systém odstráni proces s identifikátorom x

Definícia (Vykonanie operácie SEND) Vykonanie operácie [SEND, x, Y, m, f, s, t], kde $Y = \emptyset$ alebo $\exists z \in Y$ také, že neexistuje proces s identifikátorom z, pozostáva z nasledujúcich akcií vykonaných v atomickom kroku:

1. Systém odstráni správu SC(x)

2. Ak $s \neq NULL$, tak systém vykoná semaphore-signal(s)

3. Systém odstráni operáciu z SC(x)

Definícia (Vykonanie operácie RECV) Vykonanie operácie [RECV, x, Y, m, f, s, t], kde $Y = \emptyset$ alebo $\exists z \in Y$ také, že neexistuje proces s identifikátorom z, pozostáva z nasledujúcich akcií vykonaných v atomickom kroku:

1. Systém vytvorí správu m v SC(x)

2. Do m priradí hodnotu SENDER-NOT-EXISTS

3. Ak $s \neq NULL$, tak systém vykoná semaphore-signal(s)

4. Systém odstráni operáciu z SC(x)

Definícia (Vykonanie páru operácií SEND a RECV) Systém môže vykonať operáciu BR = [RECV, x, Y, m, f, s, t] v čase t iba ak existuje párová operácia BS = $[SEND, x^{'}, Y^{'}, m^{'}, f^{'}, s^{'}, t^{'}]$ v SC(*) v čase t. Ak sa systém rozhodne vykonať BR, musí vykonať aj párovú operáciu BS v tom istom atomickom kroku, ktorý pozostáva z nasledujúcich akcií:

1. Systém vytvorí novú správu m v SC(x)
2. Systém skopíruje obsah $m^{'}$ do m
3. Systém odstráni $m^{'}$ z SC($x^{'}$)
4. Ak $s \neq NULL$, tak systém vykoná semaphore-signal(s)
5. Ak $s \neq NULL$, tak systém vykoná sempahore-signal($s^{'}$)
6. Systém odstráni BS z SC($x^{'}$)
7. Systém odstráni BR z SC(x)

Poznámka Ak nastane situácia, že systém bude chcieť vykonať operáciu BR = [RECV, x, Y, m, f, s, t] takú, že $\exists z \in Y$ také, že neexistuje proces s identifikátorom z a zároveň existuje párová operácia BS = $[SEND, x^{'}, Y^{'}, m^{'}, f^{'}, s^{'}, t^{'}]$, systém sa nedeterministicky rozhodne, podľa ktorej definície sa bude správať.

Definícia (Postup spracovania) Systém v konečnom čase prečíta každú poslanú operáciu a v konečnom čase vykoná každú operáciu CREATE a DESTROY. Ďalej, ak existuje v nejakom čase pár operácií (SEND a RECV, ktoré spĺňajú podmienky definície pre párovanie operácií uvedenej vyššie) v SC(*), tak systém napokon vykoná aspoň jednu z týchto operácií.

Kapitola 4

Principiálne fungovanie systému

V predošlej kapitole som zadefinoval, ako sa má správať systém pracujúci s nestabilnými procesmi, resp. linkami. Toto správanie sa systému je spoločné pre centralizovaný, ako aj pre distribuovaný systém. Avšak nič to nehovorí o technickom pozadí vykonávania jednotlivých operácií. Ako zistiť, či niektorý z procesov prestal komunikovať? Ako pridať do siete nový proces a zabezpečiť, aby o tom vedeli všetky ostatné procesy? V tejto kapitole popíšem základnú myšlienku fungovania môjho distribuovaného systému. V ďalšej kapitole presne popíšem protokoly pre pridávanie procesu do siete, pre odobratie procesu zo siete a pre posielanie správ.

4.1 Technické pozadie systému

Najdôležitejšie rozhodnutie pri návrhu môjho systému je, akým spôsobom detekovať, či niektorý proces prestal komunikovať. Zároveň je dôležité túto informáciu zistiť čo najskôr (najlepšie hneď) a potom ju distribuovať po sieti. Na riešenie tohto problému som sa rozhodol použiť protokol TCP/IP. Dôležitou vlastnosťou tohto protokolu je, že ak dôjde k prerušeniu komunikácie medzi dvoma socketmi (či už z dôvodu porušenia linky, alebo padnutia jedného zo socketov), operačný systém pošle socketu správu o odpojení sa druhej strany. Teda z momentálnych praktických možností je toto najrýchlejší a najspoľahlivejší spôsob, ako zistiť výpadok procesu. Z tohto dôvodu som sa rozhodol, že komunikácia medzi procesmi bude prebiehať na protokole TCP/IP.

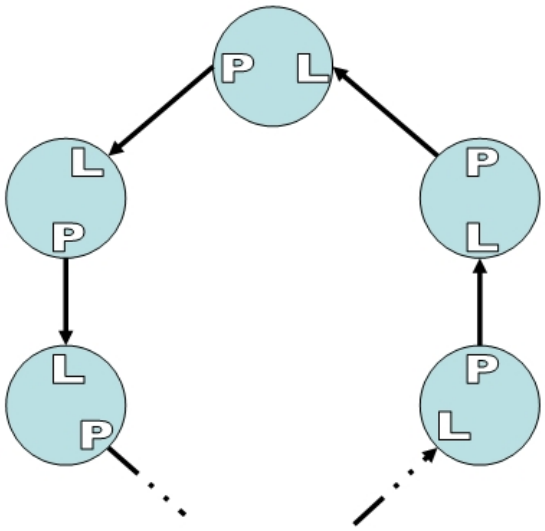
4.2 Naivný návrh systému a jeho nedostatky

Systém pre posielanie správ sa má navonok správať ako kompletný graf, čiže každý proces vie komunikovať s každým iným procesom v sieti. Jednoduché riešenie by bolo, aby každý proces mal otvorený jeden socket pre každý iný proces v sieti. V takomto prípade by bolo zabezpečené, že každý proces vie komunikovať s každým a z vlastnosti TCP/IP popísanej vyššie by všetky procesy hneď detekovali výpadok niektorého z procesov. Toto riešenie ale nie je dobré, lebo v sieti s N procesmi by pre každý proces muselo byť otvorených $N - 1$ socketov. To riešenie je neefektívne kvôli potrebe veľkých zdrojov vzhľadom na veľkosť siete, pri N procesoch až $N * (N - 1)$ socketov. Po technickej stránke toto riešenie tiež nie je dobré. Na jednom počítači (operačnom systéme) je obmedzený počet otvorených socketov (socket je dvojica file descriptor a port, počet portov je v súčasnosti 65 535). Teda toto riešenie by obmedzilo môj systém len na 65536 procesov v sieti. Toto samo osebe nemusí byť skutočnou praktickou limitáciou. Avšak takáto architektúra aj v prípade menších sietí podstatne zvýši čas potrebný na pridanie a odobratie uzla, nehovoriac o obrovskej komunikačnej réžii, ktorú samotné takéto zapojenie spôsobí v komunikujúcich uzloch a v sieti, ktorá ich spája.

4.3 Architektúra systému

V časti vyššie som ukázal jednoduché riešenie, v ktorom bol počet socketov v jednom procese závislý od počtu procesov v sieti. V tejto časti navrhnem architektúru systému, kde bude počet socket v jednom procese konštantný bez ohľadu na počet procesov v sieti a celkový počet socketov v sieti bude lineárne závislý od veľkosti siete.

Pre zníženie počtu procesov v sieti je potrebné použiť inú štruktúru ako úplný graf. Navyše chcem dosiahnuť, aby počet socketov v jednom procese bol konštantný. Z toho vyplýva, že nemôže existovať spojenie medzi všetkými procesmi naraz, ale každý proces môže priamo komunikovať iba s malou skupinou procesov. Dobrou štruktúrou je hamiltonovský kruh. Kruh (alebo viac disjunktných kruhov) je jediný graf, ktorý vznikne, ak každý uzol v grafe je stupňa práve dva. Preto každý proces má otvorené sockety iba pre hamiltonovské hrany. Tento prístup zabezpečuje, že bez ohľadu na veľkosť siete, každý proces potrebuje mať otvorené iba 2 sockety. Navyše celkový počet socketov v sieti s N procesmi je $2 * N$. Neskôr sa počet socketov v jednom procese ešte zvýši, ale toto zvýšenie bude len o konš-



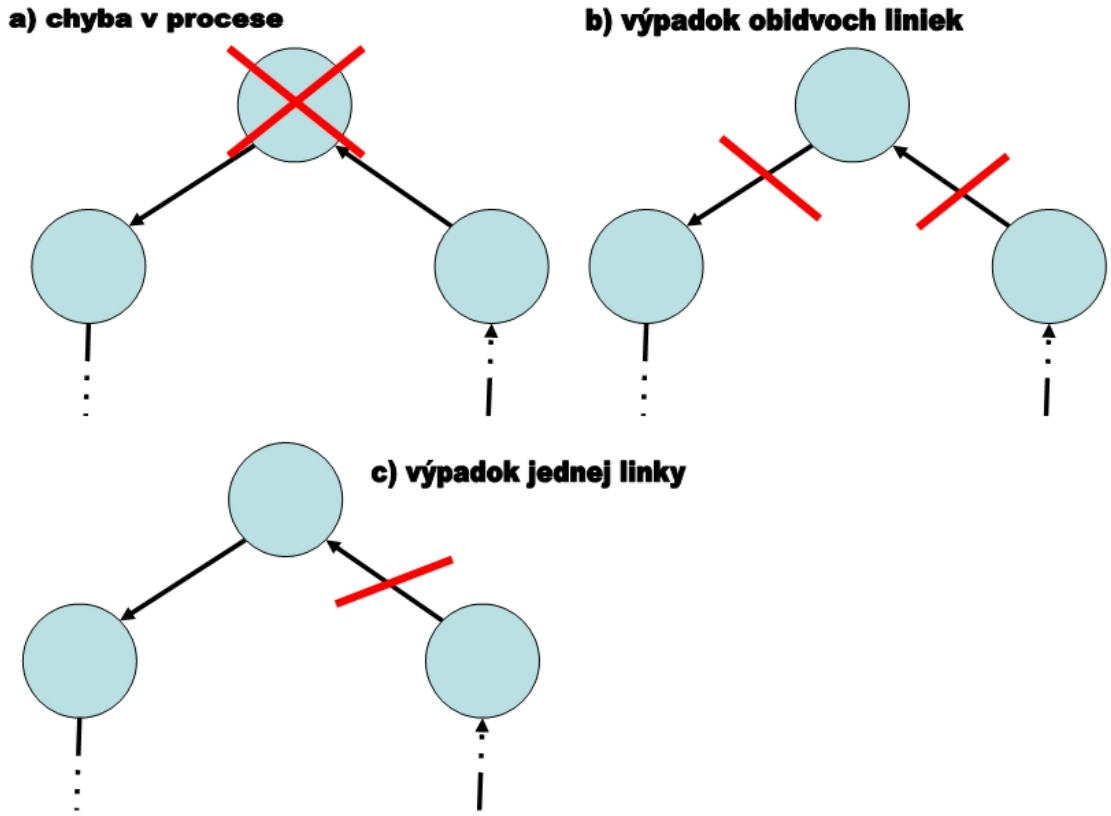
Obrázok 4.1: Kruh procesov. Každý proces vie rozoznať pravého (označenie P) a ľavého (označenie L) suseda. Komunikácia s pravým susedom znamená pre proces zapisovanie a čítanie z "pravého socketu", komunikácia s ľavým susedom znamená zapisovanie a čítanie z "ľavého socketu".

tantu c, takže počet socketov v jednom procese bude konštantný bez ohľadu na veľkosť siete a celkový počet socketov v sieti s N procesmi bude O(N). V každom procese budú bežať dva thready, z ktorých každý bude počúvať na jednom zo socketov. Tým sa zabezpečí vzájomná nezávislosť socketov.

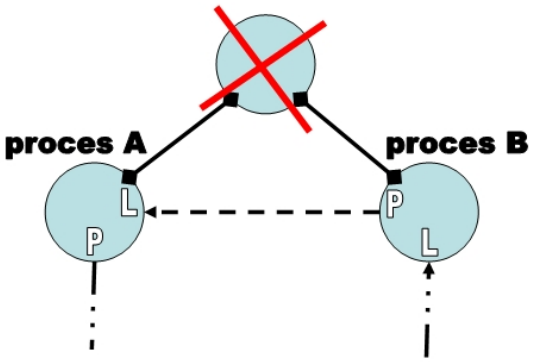
4.4 Detekcia výpadku procesu a oprava kruhu

V kruhu procesov môžu nastať 3 rôzne chyby spôsobujúce, že proces prestane komunikovať s aspoň jedným zo susedných procesov. Buď spadne samotný proces, alebo sa preruší sieťové spojenie medzi procesom a zvyškom kruhu, čo sa prejaví ako prerušenie obidvoch liniek, alebo sa preruší iba jedna linka. Prvé dva prípady sa pre kruh javia rovnako, a to tak, že s procesom sa vôbec nedá komunikovať. Preto budem ďalej uvažovať už iba o spadnutí procesu. V treťom prípade chybu na sockete detekuje iba jeden zo susedných procesov a túto chybu označím ako chybu linky (resp. spadnutie linky).

Ak dôjde ku chybe procesu, o opravu kruhu sa vždy stará ľavý sused L spadnutého procesu. Proces L sa pripojí k pravému susedovi P spadnutého procesu, a



Obrázok 4.2: Typy výpadkov v kruhu. Chyby a) a b) sa v kruhu javia rovnako (proces prestane úplne komunikovať).



Obrázok 4.3: Výpadok procesu. Proces B sa pripojí k procesu A odošle informáciu o spadnutí procesu do kruhu cez svojho ľavého suseda.

potom pošle svojmu ľavému susedovi správu o spadnutí procesu. Keď správa obíde celý kruh, všetky ostatné procesy v kruhu sú informované o spadnutí procesu.

Ak dôjde ku chybe linky, o opravu sa stará proces, pre ktorý sa situácia javí, že spadol jeho pravý sused[1], teda proces naľavo od linky (označím ako proces B). Pre proces B sa teraz javí situácia rovnako ako aj v ostatných prípadoch možných porúch. Preto proces B kontaktuje pravého suseda A "spadnutého procesu"[2]. Pošle procesu A žiadosť o obnovu kruhu z dôvodu výpadku procesu medzi nimi. Proces A donúti svojho ľavého suseda spadnúť a s procesom B obnovia kruh[3]. Následne proces B pošle správu po svojom ľavom susedovi o spadnutí procesu. Každý proces, ktorý dostane správu o spadnutí procesu, vymaže zo svojho zásobníka všetky ukončené správy od tohto procesu.

4.4.1 Výpadky viacerých procesov

V predošlej časti som popísal ako dva procesy opravia kruh, keď dôjde k nejakej chybe. Ako sa však majú správať, ak v danom momente[4] bude spadnutých viacej procesov? A ako sa správať, keď niektorý proces spadne počas vykonávania opravného protokolu?

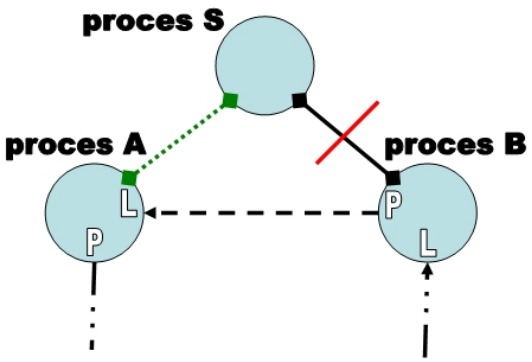
V kruhu môže dôjsť k výpadku procesov v ľubovoľnom poradí a v ľubovoľnom čase. V tejto časti sa budem zaoberať výpadkami, ktoré nastanú pred ukončením

[1] Je zrejmé, že druhému procesu sa situácia javí, ako keby spadol jeho ľavý sused

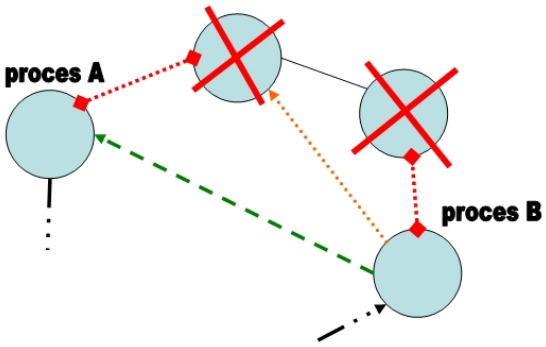
[2] Proces síce nie je spadnutý, ale B to nemá odkiaľ vedieť.

[3] V skutočnosti ho už obnovili

[4] V čase, keď sa proces snaží nadviazať spojenie s pravým susedom spadnutého procesu.



Obrázok 4.4: Výpadok linky. Proces B sa pripojí k procesu A informuje ho o chybe. Proces A donúti svojho ľavého suseda spadnúť a opraví sa kruh (spojenie medzi procesmi A a B). B následne odošle informáciu o spadnutí procesu do kruhu cez svojho ľavého suseda.



Obrázok 4.5: Výpadok viacej procesov. Proces B sa pokúsi obnoviť spojenie s pravým susedom svojho pravého (spadnutého) suseda. Keď sa mu to nepodarí, postupne skúša ďalšie procesy, až kým nenarazí na žijúci proces.

opravného protokolu[5]. Teda výpadky procesov môžu byť hromadné[6], alebo sekvenčné[7], alebo kombinácia obidvoch.

Opravný protokol začína prvý ľavý sused B spadnutého procesu. Tento sa snaží spojiť s pravým susedom spadnutého procesu. Keď sa mu to nepodarí, snaží sa spojiť s nasledujúcim procesom v kruhu vpravo. Takto postupne vyskúša všetkých až kým nenarazí na najbližší (v zmysle prehľadávania kruhu smerom vpravo od procesu vykonávajúceho opravný protokol) žijúci proces, s ktorým obnoví kruh. Ak nikoho nenájde, napokon obnoví kruh sám so sebou.

Ak počas vykonávania opravného protokolu spadne proces A, proces B sa jednoducho vráti na začiatok opravného protokolu, kde hľadá najbližší žijúci proces v kruhu smerom vpravo. Teda situácia je podobná ako pri výpadku viacerých procesov naraz.

Ak počas vykonávania opravného protokolu spadne proces B, proces A sa vráti na začiatok opravného protokolu, t.j. bude očakávať spojenie od iného procesu. Zároveň ľavý sused procesu B detekuje spadnutie procesu B, preto začne opravný protokol. Pre tento proces sa už ale situácia javí ako spadnutie viacerých procesov naraz.

Opravný protokol vykonávajú iba dva najbližšie procesy k spadnutej skupine procesov (jeden proces je tiež skupina procesov). Preto ak dôjde k spadnutiu viacerých disjunktných skupín procesov (medzi nimi je z každej strany aspoň jeden žijúci proces), nemá to nijaký vplyv na opravu kruhu (všetky opravy budú prebiehať naraz).

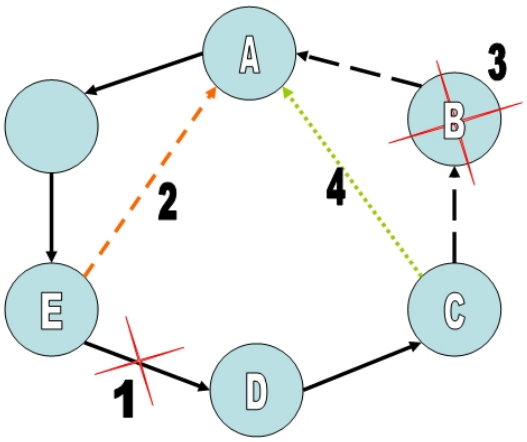
4.4.2 Problém s obnovou kruhu

Pri obnove kruhu môže dôjsť k rôznym komplikáciám kvôli neúspešnosti nadviazania spojenia. Po spadnutí niekoľkých liniek môže nastať situácia, že skupina procesov obnoví kruh, pričom z neho vylúči iné procesy. Keď sa tie potom pokúsia o opravu kruhu, môže nastať situácia ako na obrázku 4.6. Navyše na prvý pohľad sa táto situácia javí procesu A rovnako ako na obrázku 4.4. Tento problém je riešiteľný. Keď dôjde k takejto situácii (obrázok 4.6), proces A si najprv vyžiada

[5]Výpadok procesu po ukončení protokolu bol v predošlej časti. Ak sú procesy v kruhu, je to rovnaký stav ako po vykonaní opravného protokolu.

[6]Spadne viacero procesov skôr, ako sa spustí opravný protokol, napr. dôjde k fyzickému prerušeniu siete, čo spôsobí rozdelenie kruhu na dve disjunktné siete procesov, pričom obidvom týmto sieťam sa zdá, že spadlo naraz niekoľko procesov.

[7]Procesy padajú postupne počas opravných protokolov.



Obrázok 4.6: Problém s obnovou kruhu. Po prerušení linky medzi procesmi E a D, proces E sa pokúsi spojiť postupne s procesmi C a B, čo sa mu nepodarí, preto obnoví kruh až s procesom A. A donúti spadnúť proces B. Preto sa potom proces C pokúsi obnoviť kruh s procesom A. Ten však už má nového suseda, ktorý sa v kruhu nachádza viac vľavo, preto prikáže procesu C spadnúť. Situácia sa rovnako zopakuje aj pre proces D.

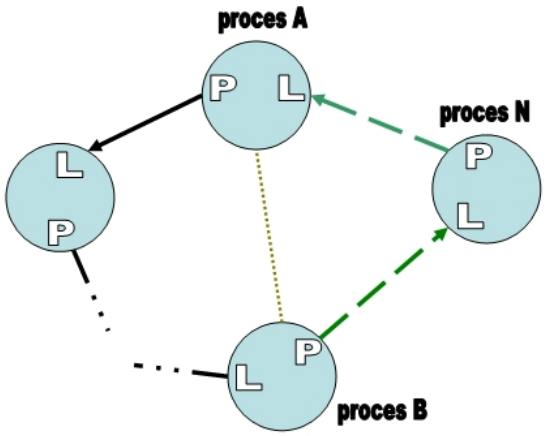
od procesu C informácie o tom, kde v kruhu sa proces C nachádza (C má ešte stále relatívne staré informácie). Ak je C v kruhu naľavo od ľavého suseda A (t.j. od procesu E), tak sa jedná o situáciu z obrázka 4.4 a rieši sa ako som popísal vyššie. Ak je však C napravo od procesu E, tak sa jedná o situáciu z obrázka 4.6. V tomto prípade proces A donúti proces C "dobrovoľne" spadnúť.

4.5 Pridanie nového procesu

V predošlej kapitole som popísal fungovanie systému a sémantiku jeho operácií. Operáciou START systém vytvorí nový proces. V skutočnosti treba tento proces pridať do kruhu medzi niektoré dva existujúce procesy[8].

Pozícia nového procesu nebude náhodná. Keďže je proces pridávaný do systému operáciou START, ktorú odošle konkrétny proces x, nový proces sa stane v kruhu jeho ľavým susedom. Nový proces sa najprv pripojí na svojho pravého suseda P (ktorým je proces A). Ten prikáže svojmu ľavému susedovi L (procesu

[8]Ak je v systéme iba jeden proces, má vytvorený kruh sám so sebou a nový proces sa pridá "medzi" jeho dva sockety.



Obrázok 4.7: Pridanie procesu. Prerušované čiary sú novovytvorené linky medzi novým procesom a dvomi pôvodnými procesmi v kruhu. Bodkovaná čiara je linka, ktorá sa zruší po úspešnom pridaní nového procesu.

B), aby sa pripojil na nový proces. B pošle správu po kruhu smerom vľavo (t.j. k svojmu ľavému susedovi) o pridaní nového procesu. Každý proces si do svojho zoznamu procesov pridá identifikátor nového procesu a prepošle správu ďalej. Keď správa obíde celý kruh, každý proces vie o novom procese, a teda proces je "pridaný" do kruhu. V tomto okamihu už spojenie medzi procesmi P a L nie je potrebné, preto sa rozpoja.

4.5.1 Výpadok procesu počas pridávania nového procesu

Počas vykonávania protokolu môže dôjsť k výpadkom procesov buď vykonávajúcich protokol[9], alebo k výpadku niektorého iného procesu.

Nech spadne proces mimo procesov vykonávajúcich pridávací protokol. Spadnutý proces môže byť susedom procesu A, alebo procesu B, alebo žiadneho z nich. Ak nie je susedom žiadneho z nich, jeho výpadok nemá nijaký vplyv na pridávací protokol. Ak je susedom procesu A[10], tak A vykoná obidva protokoly (opravný protokol na pravom sockete a pridávací protokol na ľavom sockete), lebo programy pre tieto protokoly nepoužívajú rovnaké zdroje. Protokoly sú preto v tomto prípade na sebe nezávislé. Rovnako to platí aj v prípade, že spadne ľavý sused

[9]Nový proces, proces x, na ktorý sa pripája a ľavý sused procesu x.

[10]Spadnutý proces je pravým susedom procesu A. Ľavým susedom je proces B, ale povedali sme, že on nespadol.

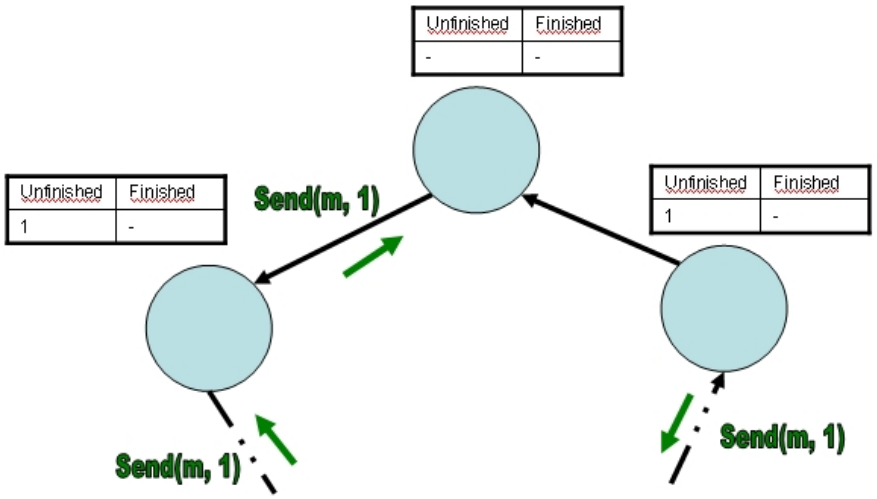
procesu B.

Ak spadne počas pridávacieho protokolu proces A, proces N dobrovoľne spadne tiež. Proces B zruší pridávací protokol a prejde k oprave kruhu.

Ak spadne počas pridávacieho protokolu proces B, proces A detekuje výpadok, donúti proces N spadnúť, zruší pridávací protokol a začne (bude očakávať spojenie) opravovací protokol.

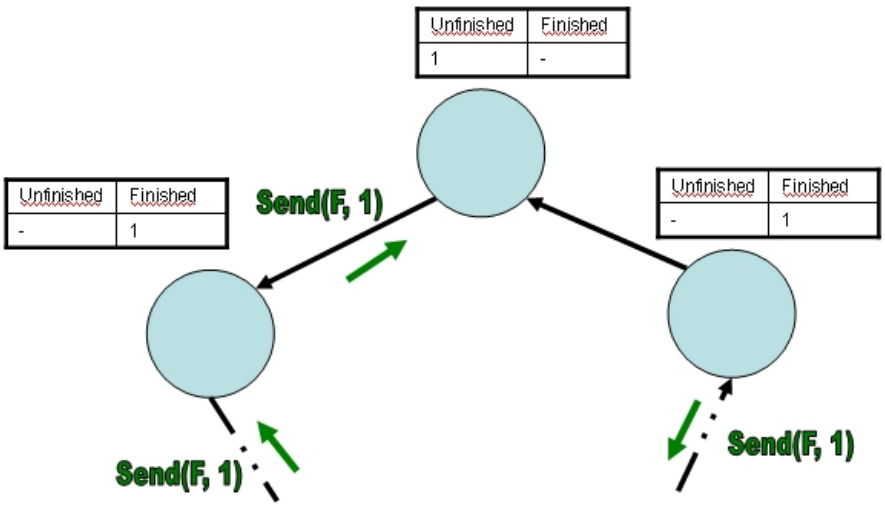
4.6 Komunikácia medzi procesmi

Navonok sa systém musí správať ako úplný graf, ale realizácia systému je iba kruh. Preto komunikácia procesov musí prebiehať po kruhových hranách tak, že proces vždy pošle správu k svojmu ľavému susedovi a ten ju prepošle ďalej. Preposielanie správy sa skončí, keď obehne celý kruh. Vtedy má posielajúci proces istotu, že ak existuje adresát, tak správu dostal[11]

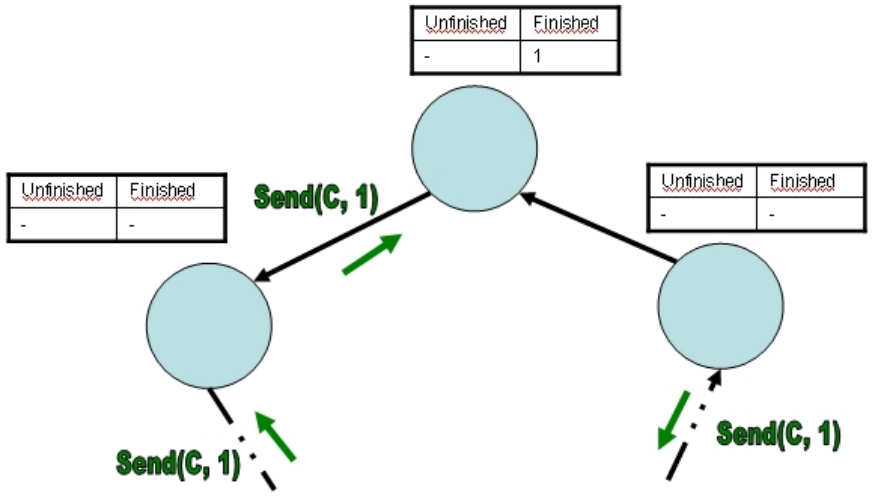


Komunikácia medzi procesmi. Prvá fáza: proces posiela správu a jej identifikačné číslo. Každý proces si označí správu v zásobníku ako UNFINISHED-SEND.

[11]Možným riešením by bolo pridať konštantný počet c socketov každému procesu na priamu komunikáciu s inými procesmi. Procesy by otvárali priame spojenia len na komunikáciu a 2 sockety hamiltonovského kruhu by slúžili iba na detekciu výpadkov. Potom by však bolo potrebné vyriešiť problém, keby s jedným procesom chcelo komunikovať c + i procesov ($i > 0$).



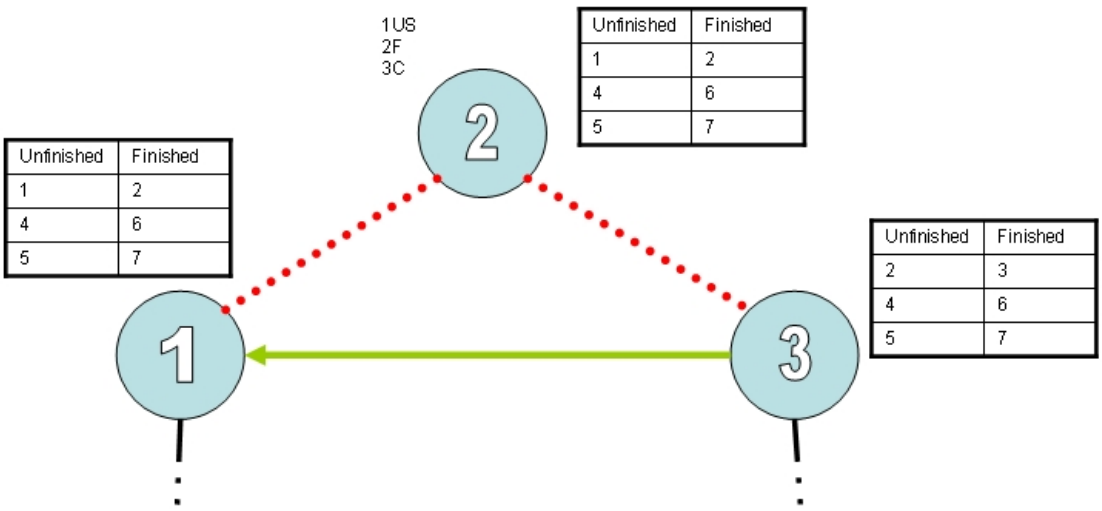
Komunikácia medzi procesmi. Druhá fáza: proces posiela správu F (FINISHED-SEND) a identifikačné číslo správy. Každý proces si správu označí ako FINISHED-SEND.



Komunikácia medzi procesmi. Tretia fáza: proces posiela správu C (COMPLETED-SEND) a identifikačné číslo. Každý proces odstráni správu zo zásobníka.

Počas posielania správy po kruhu môže dôjsť k výpadkom procesov a tým ku strate posielanej správy. Preto bude posielanie správy po kruhu fungovať trojfázovo. Najprv sa po kruhu pošle samotná správa - obsah správy m a jednoznačné číslo správy ID. Každý proces si ju odloží do svojho zásobníka[12] označenú ako neukončené posielanie (UNIFINISHED-SEND) a prepošle ďalej po kruhu smerom

[12]Pod zásobníkom budeme v celom texte rozumieť dátovú štruktúru FIFO



Obrázok 4.8: Výpadok procesu počas posielania správ po kruhu a stavy zásobníkov jednotlivých procesov. V procese 2 boli počas výpadku 3 správy (identifikátor správy, označenie), ktoré sa výpadkom procesu stratili: (1, UNFINISHED-SEND), (2, FINISHED-SEND), (3, COMPLETED-SEND). Bodkované čiary znamenajú prerušené linky, šípka znamená smer obnovenia spojenia medzi procesmi 1 a 3.

vľavo. Ak proces dostane tú istú správu dvakrát, vie, že správa už obehla celý kruh a začne druhú fázu posielania. V druhej fáze sa už posiela iba číslo správy s príkazom ukončenia posielania. Keď proces dostane správu m' = FINISHED-SEND a ID, prezrie svoj zásobník. Správu s číslom ID označí ako FINISHED-SEND a prepošle ďalej m' a ID. Ak proces dostane túto správu druhýkrát, vie, že už prešla celý kruh a začne tretiu fázu. V nej posiela správu m'' = COMPLETED-SEND a ID správy. Proces, ktorý dostane správu m'', vymaže zo svojho zásobníka správu s číslom ID a prepošle m'' ďalej po linke. Ak správu s číslom ID v zásobníku nemá, je posielanie správy ukončené, pričom ak adresát správy stále žije, dostal správu m.

Čo však robiť, ak niektorý proces prestane z nejakého dôvodu komunikovať (porušenie linky, spadnutie procesu, atď.) počas posielania správy? Keďže procesy komunikujú po kruhu, prebieha komunikácia aj cez proces 2. Preto spadnutím procesu 2 sa môžu niektoré správy stratiť. Môžu nastať tieto prípady:

1. Proces 1 už dostal správu (m, ID) (t.j. prvá fáza), ale proces 3 ešte nie.

2. Obidva procesy 1 a 3 dostali správu (m, ID).

3. Proces 1 už dostal správu ($m^{'}$, ID) (t.j. druhá fáza), ale proces 3 nie.

4. Obidva procesy 1 a 3 dostali správu ($m^{'}$, ID).

5. Proces 1 už dostal správu ($m^{''}$, ID) ale proces 3 nie.

6. Obidva procesy dostali správu ($m^{''}$, ID).

Po spadnutí procesu 2 procesy 1 a 3 obnovia kruh (popísané vyššie). Nech množina všetkých správ v zásobníku s označením UNIFINISHED-SEND procesu 3 je U_3 a procesu 1 je U_1. Podobne nech množina všetkých správ v zásobníku s označením FINISHED-SEND procesu 3 je F_3 a procesu 1 je F_1.

Potom množina $LU = U_1 \setminus (U_3 \cup F_3)$ (LostUnfinished) je množina správ s označením UNFINISHED-SEND, ktoré neprešli cez proces 2 [13]. Pre množinu LU zopakuje proces 1 posielanie správy m, čím sa obnoví posielanie stratených správ s označením UNFINISHED-SEND.

$LF = F_1 \cap U_3$ (LostFinished) je množina správ s označením FINISHED-SEND, ktoré neprešli cez proces 2. Pre množinu LF zopakuje proces 1 posielanie správy $m^{'}$, čím obnoví posielanie stratených správ s označením FINISHED-SEND.

$LC = F_3 \setminus (F_1 \cup U_1)$ (LostCompleted) je množina správ s označením COMPLETED-SEND, ktoré neprešli cez proces 2. Pre množinu LC proces 1 zopakuje posielanie správy $m^{''}$, čím obnoví posielanie správ s označením COMPLETED-SEND.

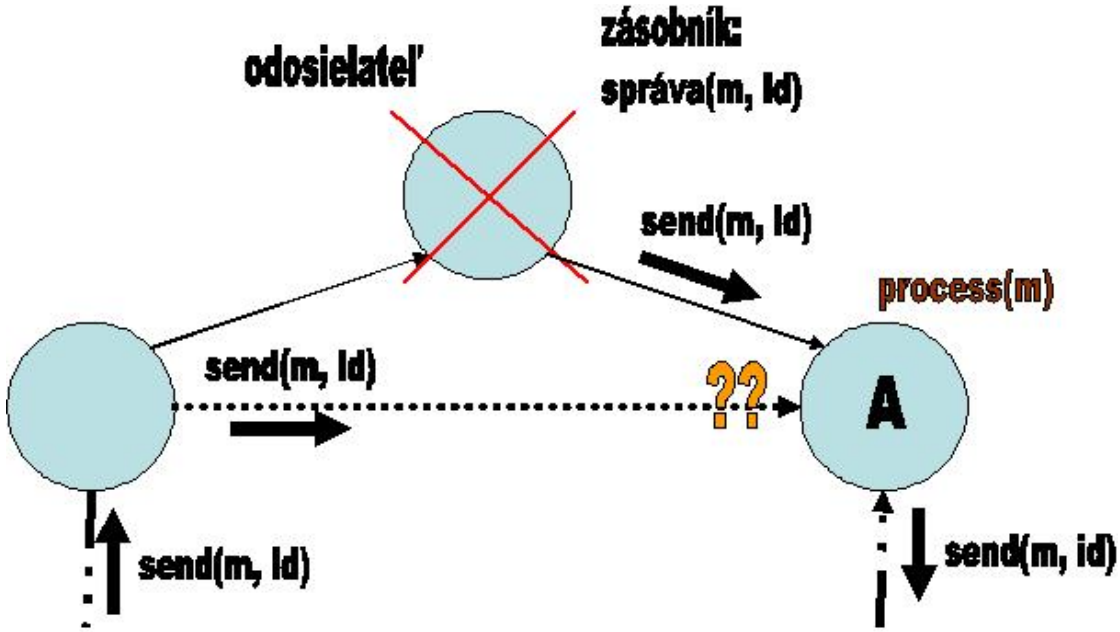
4.6.1 Nutnosť trojfázového protokolu

Vyššie som popísal, že posielanie správ po kruhu bude fungovať v trojfázovom protokole. V tejto časti ukážem, prečo nestačí na posielanie správ po kruhu menej ako 3 fázy.

Jednofázový protokol a jeho nedostatky

V jednofázovom protokole by sa správa poslala po kruhu iba raz, a keď by obišla celý kruh, bolo by isté, že informáciu dostal každý žijúci proces v kruhu. Ak by však počas posielania správy spadol niektorý proces, mohla by sa správa stratiť. Vysielajúci proces by preto po zistení, že spadol niektorý z procesov v kruhu, musel správu znovu preposlať. Avšak v prípade, že spadnú proces, v ktorom sa

[13] Buď ich proces 1 nestihol poslať pred spadnutím procesu 2, alebo proces 2 spadol skôr, ako ich stihol preposlať



Obrázok 4.9: Chyba jednofázového protokolu. Proces A prijme správu m, odošle ju ďalej a potom spracuje. Po spadnutí odosielateľa mu správa prichádza znovu. Avšak proces A sa nevie rozhodnúť, či je sa jedná o novú správu, alebo o nejakú, ktorú už spracoval. Preto správu považuje za novú a celý proces sa môže opakovať bez skončenia.

práve nachádza správa a má ju preposlať, aj vysielajúci proces, správa by bola nenávratne stratená, pričom časť procesov v kruhu správu dostala a časť nie.

Ďalším problémom v jednofázovom protokole by bolo, ak vysielajúci proces odoslal správu a spadol skôr, ako by táto správa obišla celý kruh. Jeho pravý sused by po obnove kruhu poslal k ľavému susedovi správu, ktorú už mal, no on o tom nevie[14]. Ten potom túto správu považuje za novú správu. Takto by potom správa mohla putovať po kruhu donekonečna.

Dvojfázový protokol a jeho nedostatky

V dvojfázovom protokole posielajúci proces posiela najprv správu, a potom potvrdzovaciu správu o úspešnom poslaní. Potvrdzovacia správa sa začne posielať, až keď skončí prvá fáza, t.j. keď niektorí z procesov prijme rovnakú správu dvakrát.

[14]Nie je reálne, aby si proces pamätal všetky správy, ktoré cez neho išli. Pamätať si môže nanajvýš správy, ktoré posielal on sám

Ak niektorý proces spadne počas prvej fázy, jeho susedia si porovnajú obsahy svojich zásobníkov a podľa toho sa zopakuje posielanie správy. Zlepšenie v dvojfázovom protokole je, že žiadny proces nemôže odstrániť správu zo zásobníka kým určite neobišla celý kruh (Až potom začne druhá fáza, ktorá dovolí procesu spracovať správu), preto sa teoreticky nemôže stať, že správa bude donekonečna obiehať v systéme. Avšak aj tu ešte existuje situácia, kedy to nie je spoľahlivé. Ak odosielateľ pošle potrvdzovaciu správu o úspešnosti poslania, jeho najbližší sused potvrdenie prepošle a správu spracuje. Ak však odosielateľ spadne skôr, ako potvrdzovacia správa príde k jeho pravému susedovi, zopakuje sa rovnaká situácia ako v jednofázovom protokole. Môže dôjsť k opakovanému posielaniu správy.

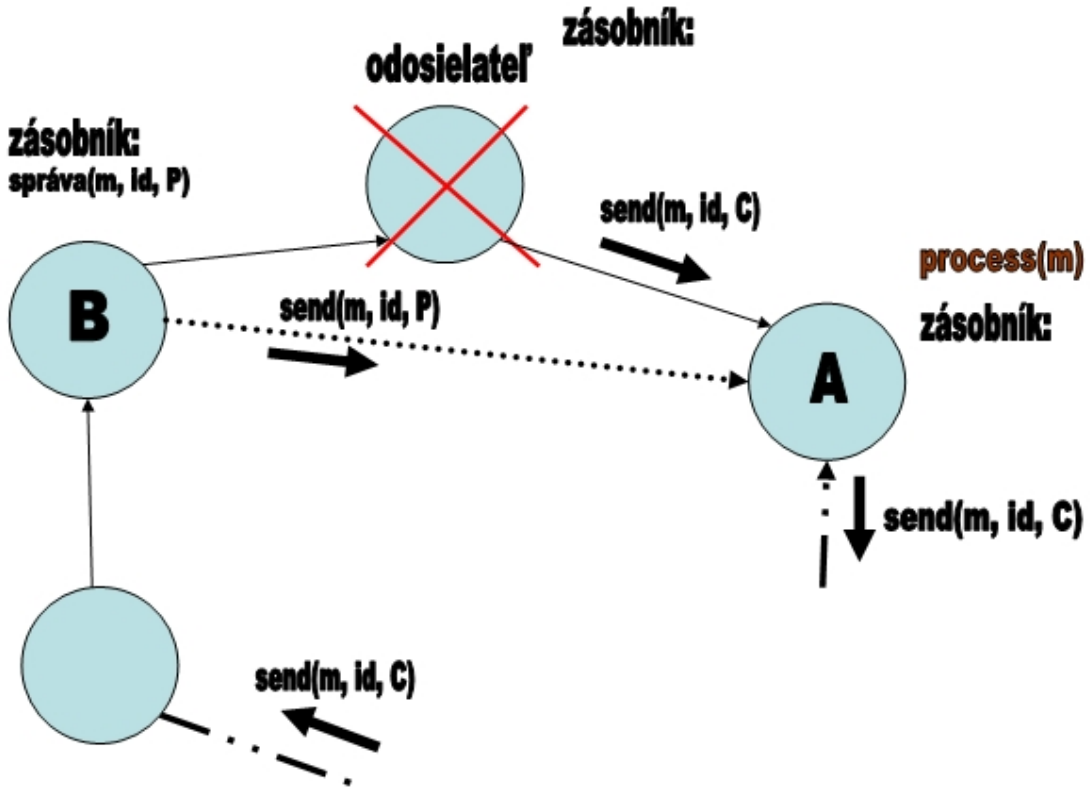
Prečo je trojfázový protokol postačujúci

V trojfázovom protokole (obrázok 4.10) nemôže nastať situácia popísaná vyššie. V prvej fáze sa posiela správa, v druhej fáze PRECOMMIT a v tretej fáze COMMIT. Druhá a tretia fáza začínajú v momente, ak niektorý z procesov dostane správu predošlého protokolu dvakrát. Samotnú správu môže proces "prečítať" (odstrániť zo zásobníka), až keď prijme COMMIT. Avšak sú prípady, kedy je potrebné čítať aj z nedokončených správ, kvôli informáciám, ktoré v sebe nesú (Kapitola Protokoly, podkapitola Protokol pre pridanie procesu, Dôkaz správnosti programu.).

Predpokladajme rovnakú situáciu ako v predošlej časti, ale procesy sú v tretej resp. druhej fáze. Keďže nová fáza protokolu začne, až keď skončí predošlá, nemôže sa stať, že proces B bude mať zásobníku iba správu prvej fázy protokolu a proces A už preposlal COMMIT. Teda dva procesy nemôžu byť v danom momente jeden v prvej fáze a druhý proces v tretej fáze. Po spadnutí procesu a obnove kruhu, proces B pošle procesu A správu (m, id, P), lebo nemá istotu, či ju spadnutý proces spracoval. Proces A však nemá v zásobníku žiadnu správu s daným id. Tým pádom proces A vie, že k danej správe už dostal COMMIT, a preto správu (m, id, P) ignoruje.

4.7 Posielanie a prijímanie správ

V predošlej časti som popísal ako funguje preposielanie správ medzi procesmi v kruhu. Toto preposielanie však beží na procese a nijako nedostane obsah správ do aplikácie, ktorá očakáva správu.



Obrázok 4.10: Trojfázový protokol. Keď proces A dostane správu (m, id, P) a v zásobníku nemá správu s týmto id, vie, že k danej správe už dostal COMMIT, preto túto správu ignoruje a posielanie považuje za skončené.

4.7.1 Poslanie správy

Posielanie správy vykoná aplikácia vykonaním systémovej neblokovacej operácie (t.j. operácia mojej knižnice) Send(Sender, SetOfReceivers, Message, Tag, FilterFunction()), takže aplikácia pokračuje vo svojom výpočte. Proces (knižnica) prijme parametre vstupujúce do príkazu Send() a zabalí ich do svojej správy, pričom ku správe ešte priloží aj čas vykonania operácie Send() a nastaví správe prvú fázu posielania. Správu vloží do svojho zásobníka pre neukončené posielanie a pošle svojmu ľavému susedovi. Ďalej už posielanie správy funguje ako je popísané vyššie.

4.7.2 Prijatie správy

Pre prijímanie správ je v každom procese vytvorený ešte jeden zásobník. Ak chce aplikácia prijať správu, vykoná blokovaciu operáciu Recv(Receiver, SetOfSenders, &Message, Tag, FilterFunction()). Operácia Recv() prezrie zásobník pre prijaté správy a vyberie z neho najstaršiu správu spĺňajúcu podmienky určené parametrami funkcie Recv() (to sú SetOfSenders a FilterFunction()). Ak v zásobníku takáto správa nie je, aplikácia sa blokuje na operácii Recv(), kým do zásobníka nepribudne správa spĺňajúca podmienky, alebo kým nespadne niektorý proces z tých, čo sú uvedené v parametri SetOfSenders.

Samotný proces po prijatí správy od svojho pravého suseda zaktualizuje zásobník s nedokončenými správami. Ak je prijatá správa COMMIT a proces je medzi adresátmi, zaradí správu do zásobníka pre prijaté správy.

Ak proces prijme správu o spadnutí procesu, vymaže zo zásobníka pre prijaté správy všetky správe posielané práve týmto spadnutým procesom. Ak navyše aplikácia vykonáva Recv() a medzi odosielateľmi je spadnutý proces, vloží do zásobníka správu s chybovou správou, ktorú operácia Recv() prečíta.

Kapitola 5

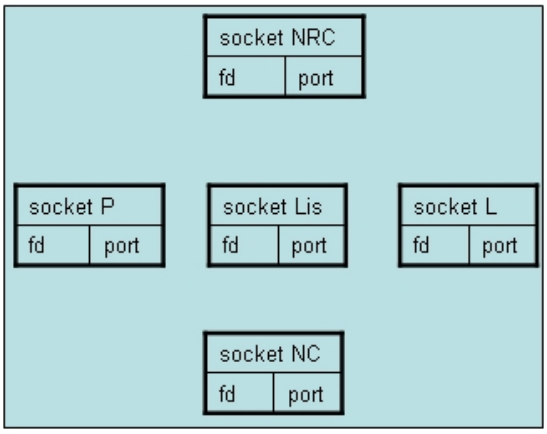
Protokoly

V predošlej kapitole som popísal, akým spôsobom sa obnoví kruh po výpadku procesu, ako sa pridá nový proces do kruhu a ako vyzerá protokol pre posielanie správ. V tejto kapitole popíšem programy pre pridanie nového procesu, pre odobratie procesu a pre posielanie správ, ktoré bežia v každom procese v kruhu.

Každý proces bude používať 5 socketov. Pre jednoduchosť si ich označím Lis, L, P, NC, NRC. Socket Lis je v procese len na prijímanie nového spojenia. Po pripojení iného procesu na socket Lis sa spojenie presmeruje na sockety NC alebo NRC (podľa toho, ktorý z nich nie je momentálne používaný). Sockety L (ľavý sused) a P (pravý sused) sú používané pre udržiavanie kruhu a pre preposielanie správ prichádzajúcich od pravého suseda. V procese bežia 3 thready. Prvý thread T_{Lis} počúva na sockete Lis a prijíma spojenia. Druhé dva thready T_P a T_L sa starajú o sockety P a L, pričom v niektorých programoch používajú aj pomocné sockety NC a NRC.

Thready T_P a T_L vykonávajú na svojich socketoch neblokovací recv()[1]. To znamená, že po prijatí nových dát, operačný systém pošle programu systémovú správu FD_RECV. Po zrušení spojenia, operačný systém pošle správu FD_CLOSE. Na základe týchto správ sa vedia thready jednoznačne rozhodnúť, ktorý program majú začať vykonávať. V programoch protokolov mením vlastnosť socketov na blokovací recv().

[1] To, či sa na sockete vykonáva blokovací alebo neblokovací recv(), je vlastnosť daného socketu. V procese využívam obidva spôsoby prijímania správ.



Obrázok 5.1: Sockety v procese. Sockety P, L a Lis sú kontrolované každý vlastným threadom. Sockety NC a NRC sú pomocné sockety počas vykonávania opravného protokolu a protokolu pridávania nového procesu.

5.1 Zdravý stav kruhu

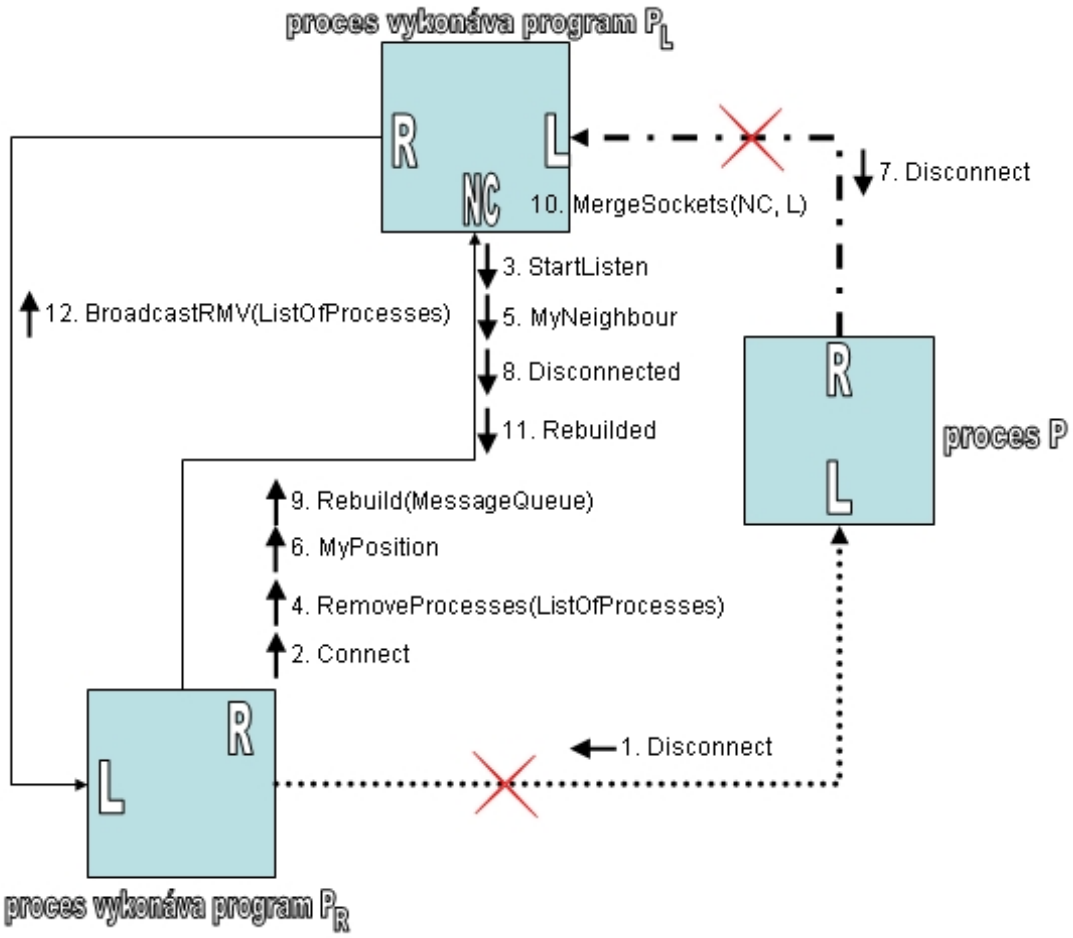
Kvôli neskoršiemu dokazovaniu správnosti programov je potrebné vedieť určiť, kedy je systém v nami požadovanom stave.

Definícia (Zdravý stav kruhu) Systém je v zdravom stave, ak pre každý proces platí jedna z týchto dvoch možností:

1. Proces má práve dvoch susedov.
2. Proces vytvára kruh sám so sebou.

5.2 Prijatie spojenia

Thread T_{Lis} po spustení procesu pripraví socket Lis na prijímanie spojení, vykonaním operácií Bind(fd, IP, Port), a potom "vykonaním" blokovacej operácie Listen(fd, ..). Po pripojení sa klienta na socket Lis, operačný systém o tom pošle programu systémovú správu FD_ACCEPT. Po prijatí správy FD_ACCEPT, T_{Lis} prijme spojenie a vytvorí nový socket vykonaním operácie Soc = accept(Lis, clientinfo, l). Socket Lis potom prejde znovu do stavu listen(). Socket Soc je ďalej priradený do socketu NC alebo NRC. Sockety NC a NRC sú ďalej používané threadom



Obrázok 5.2: Grafické zobrazenie poradia krokov, ktoré vykoná program pre opravu kruhu.

T_L. Thread T_{Lis} vykonáva tento program počas celého behu procesu bez ohľadu na to, aký protokol práve vykonávajú ostatné dva thready.

5.3 Program pre odobratie procesu

V kruhu sú možné dva spôsoby porušenia kruhu. Buď proces prestane úplne komunikovať (prerušilo sa mu spojenie s obidvoma susedmi, alebo proces spadol), alebo sa poruší linka len s jeho ľavým susedom[2]. Ak dôjde k výpadku iba jednej linky, v

[2]Teda za spadnutý proces budem považovať ten, ktorému sa preruší spojenie medzi ním a jeho ľavým susedom.

opravnom protokole sa naviac vykoná iba krok 5. Disconnect(L) v programe P_L.

5.3.1 Program

Celý program pre odobratie procesu sa skladá z dvoch procedúr, kde každá beží na inom threade (vzhľadom na pozíciu v procese, teda T_L a T_P). Každý neúspešný read(), write() a connect()[3] vráti tok riadenia na prvý krok programu. Neúspešný príkaz 7. write(Kill, L) nevracia tok riadenia na prvý krok programu.

Program P_L

1. Listen(NC)
2. write(StartListen, NC)
3. read(NC) /*RemoveProcesses*/
4. write(MyNeighbour, NC)
5. read(R) /*MyPosition*/
6. if PositionNotOk then goto ErrorCode
7. write(Kill, L)
8. Disconnect(L)
9. write(Disconnected, NC)
10. read(NC)
11. MergeSockets(NC, L)
12. write(Rebuilded, L)
13. RestartLostMessages()
14. goto EndProcotocol
15. **ErrorCode**

[3]Operácie sú neúspešné, ak sa spojenie socketov preruší, a preto nie je možné vykonať danú operáciu.

16. write(Kill, NC)
17. Disconnect(NC)
18. **EndProtocol**

Program P_R

1. receive Disconnect
2. FindRightNeighbour()
3. Connect(R)
4. read(R) /*StartListen*/
5. write(RemoveProcesses, R)
6. read(R) /*MyNeighbour*/
7. write(MyPosition, R)
8. read(R) /*Disconnected, Kill*/
9. if Kill then goto ErrorCode
10. write(Rebuild, R)
11. read(R)
12. write(BroadcastRMV, L)
13. goto EndProtocol
14. **Label ErrorCode**
15. Disconnect(AllSockets)
16. Kill
17. **Label EndProtocol**

5.3.2 Dôkaz správnosti programu

Majme kruh s procesmi očíslovanými 0..n, kde susedmi procesu 0 sú 1 (pravý sused) a n (ľavý sused), procesu 1 sú 2 (pravý sused) a 0 (ľavý sused), atď. Predpokladajme, že aspoň jeden proces nespadne, kým sa neukončí vykonávanie opravného protokolu. Opravný protokol vždy začína ľavý sused spadnutého procesu. Nech teda spadne proces $i \in \{0..n\}$. Potom proces $(i+n)mod(n+1)$ (ľavý sused) bude vykonávať program P_L a proces $(i+1)mod(n+1)$ (pravý sused) program P_R. V ďalšom texte budem tieto procesy volať podľa toho, aký program vykonávajú. Ak počas vykonávania opravného protokolu nespadne žiadny iný proces, procesy P_R a P_L dostanú systém do zdravého stavu.

Dôkaz je zrejmý z obidvoch programov. Každý read() je blokovací, t.j. tok riadenia stojí na príkaze read() dovtedy, kým nie je možné niečo prečítať. Proces P_R prijme správu o prerušení linky medzi ním a jeho pravým susedom (procesom i). Proces má v pamäti stav celej siete, a preto vie, ktorý proces je pravým susedom procesu i (funkcia FindRightNeighbour()). Pravým susedom procesu i je P_L. P_R sa pripojí na P_L a prijme správu, že P_L je pripravený na komunikáciu. P_R mu oznámi spadnutie procesu i. Ak i je ešte stále pripojený na P_L a P_R je v kruhu vľavo od procesu i, tak P_L zruší spojenie a donúti proces i spadnúť úplne. Procesy P_L a P_R potom vytvoria medzi sebou kruhovú hranu a pošlú po kruhu informáciu o spadnutí procesu. Keď sa skončí posielanie, každý proces v sieti vie, že proces i spadol a opravný protokol je skončený.

Ak sa o opravu kruhu pokúsi proces, ktorý to nemôže urobiť (obrázok 4.6), toto je detekované pri zistení jeho pozície vzhľadom na aktuálneho suseda procesu P_L. Ak je pozícia procesu P_R nesprávna, P_L prejde na príkaz Label ErrorCode, potom vykoná príkaz write(Kill, NC) a následne odpojí socket NC, čím sa pre proces P_L opravný protokol skončil. Proces P_R po prijatí správy Kill, prejde na príkaz Label ErrorCode, odpojí všetky svoje spojenia, a potom dobrovoľne spadne. Systém sa teraz dostal do zdravého stavu.

Počas opravného protokolu však môže dôjsť k ďalším výpadkom procesov. Program opravného protokolu vykonávajú iba dvaja najbližší susedia spadnutého procesu. Preto, ak spadne ešte iný proces, ktorý nie je susedom jedného z procesov P_L alebo P_R, tak to nemá žiadny vplyv na vykonávanie opravného protokolu. Vplyv na beh programu môžu mať výpadky týchto procesov:

1. spadne pravý sused procesu P_L

2. spadne ľavý sused procesu P_R

3. spadne proces P_L

4. spadne proces P_R

Prípad 1: spadne pravý sused procesu P_L Keďže v každom procese sú dva thready pre ľavý a pravý socket, proces P_L dokáže naraz bežať obidva opravovacie programy. Teda v skutočnosti vo svojom ľavom threade vykonáva program P_L a vo svojom pravom threade vykonáva program P_R. Obidva programy môžu byť vykonávané naraz v jednom procese, pretože nepoužívajú spoločné zdroje, teda sú na sebe úplne nezávislé. Zároveň to zabezpečuje, že ak je proces sám v sieti, dokáže vytvoriť kruh sám so sebou, bez špeciálnej úpravy programu pre tento prípad. Čiže proces P_L vykonáva dva opravné programy, kde v jednom prípade vykonáva program pre pravého suseda a v druhom prípade vykonáva program pre ľavého suseda.

Prípad 2: spadne ľavý sused procesu P_R Tento prípad je symetrický k predošlému, a preto preň platí rovnaká úvaha a z nej vyplývajúci záver.

Prípad 3: spadne proces P_L Ak spadne proces P_L potom, ako proces P_R pošle BroadcastRMV, systém je už v zdravom stave. Teda systém (proces P_R) sa správa rovnako ako po spadnutí procesu i, akurát že program P_L začne vykonávať ľavý sused spadnutého procesu P_L. Ak proces P_R spadne skôr, ako proces P_R pošle BroadcastRMV, P_R to detekuje na niektorej operácii $p \in \{read(), write(), connect()\}$. Keď je operácia p neúspešná, vykonávanie programu sa vráti ku kroku 2, kde FindRightNeighbour() vráti pravého suseda procesu, s ktorým P_R naposledy komunikoval. Neúspešná operácia connect() sa tiež považuje ako komunikovanie s procesom. Takže proces P_R si zistí ďalší proces v poradí kruhu, na ktorý sa pokúsi pripojiť. Tento proces napokon vykoná program P_L a spolu s procesom P_R dokončia opravný protokol. V prípade výpadku k navzájom susedných procesov naraz, alebo sekvenčne vždy pred odoslaním BroadcastRMV (alebo kombinácia výpadkov), proces P_R sa po k krokoch (neúspešných pokusoch opraviť kruh) pripojí na proces, s ktorým sa mu podarí obnoviť linku. Ak sú v kruhu procesy 0..n, tak proces P_R obnoví kruh po maximálne n neúspešných pokusoch.

Prípad 4: spadne proces P_R Nech procesy P_L a P_R vykonávajú opravný protokol. Nech proces P_L vykoná operáciu Merge(NC, L). V tomto momente je systém v zdravom stave. Ak proces P_R spadne po tomto kroku, proces P_L to detekuje na write(Rebuilded, L), alebo neskôr tak ako pri prvom detekovaní výpadku suseda. Preto proces P_L prechádza na prvý krok svojho programu a očakáva pripojenie sa iného procesu. Opravovací program však začne vykonávať ľavý sused procesu P_R. Avšak toto je rovnaké ako predošlý prípad. Ak proces P_R spadne skôr ako proces P_L vykoná Merge(NC, L), proces P_L detekuje výpadok na niektorej operácii $p \in \{read(), write()\}$. Vtedy sa vráti k prvému kroku programu (Listen()) a čaká, kým sa niektorý proces nepokúsi o obnovu kruhu. Protokol určite skončí, lebo v sieti je $n + 1$ procesov, preto po maximálne n výpadkoch procesov, proces P_L začne vykonávať aj program P_R a obnoví kruh. □

5.4 Protokol pre pridanie procesu

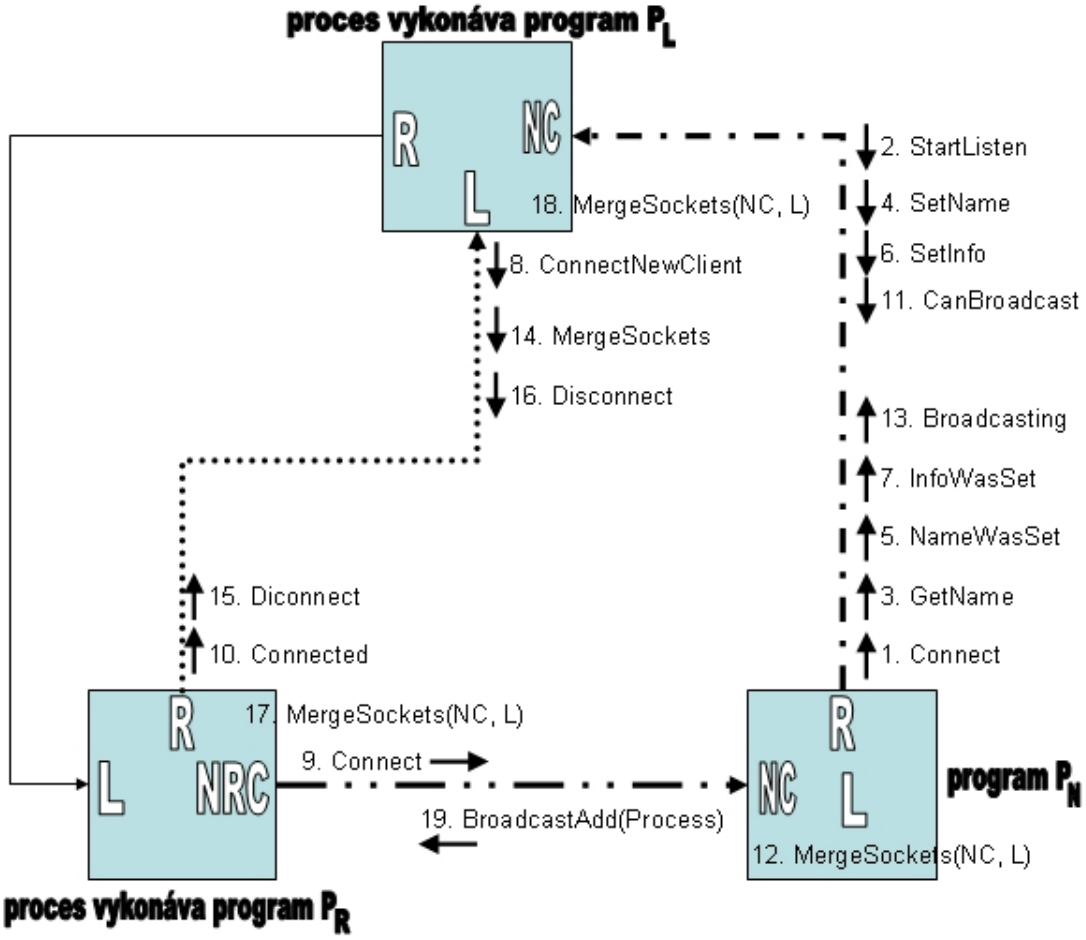
Pri pridávaní procesu do kruhu sa nový proces pripája na svojho budúceho pravého suseda. Počas protokolu je nutné, aby nový proces dostal jednoznačné meno, aby si nastavil informácie o sieti tak, ako to má jeho pravý sused. Takisto si musí skopírovať stav zásobníka s nedokončenými správami. Potom prijme spojenie svojho ľavého suseda. Napokon sa rozpojí linka medzi pôvodnými susedmi. Nový proces pošle do siete správu o jeho pridaní.

5.4.1 Program

Program pre pridanie nového procesu pozostáva z troch rôznych procedúr. Procedúry P_N (nový proces), P_L (pravý sused nového procesu) a P_R (ľavý sused nového procesu). Každá neúspešná operácia read(), write() a connect() pošle tok riadenia na Label ErrorCode. Navyše v programoch P_L a P_R sa po každom úspešnom read() vykoná kontrola, či neprišla spáva KillProtocol namiesto očakávanej správy (uvedenej v komentári /*..*/). Ak prišla správa KillProtocol, tok riadenia tiež prejde na riadok Label ErrorCode.

Program P_N

1. Connect(R)
2. read(R) /*StartListen*/



Obrázok 5.3: Grafické zobrazenie poradia krokov, ktoré vykoná program pre pridanie nového procesu do kruhu.

3. write(GetName, R)
4. read(R) /*SetName*/
5. write(NameWasSet, R)
6. read(R) /*SetInfo*/
7. write(InfoWasSet, R)
8. T_{Lis}: Listen(NC)
9. read(R) /*CanBroadcast*/
10. Merge(NC, L)
11. write(Broadcasting, R)
12. BroadcastAdd() /*posielanie správy o pridaní procesu*/
13. goto EndProgram
14. **Label ErrorCode**
15. Disconnect(AllSockets)
16. Kill
17. **Label EndProgram**

Program P_L

1. T_{Lis}:Listen(NC)
2. write(StartListen, NC)
3. read(NC) /*GetName*/
4. write(SetName, NC)
5. read(L) /*NameWasSet*/
6. write(SetInfo, NC)
7. read(NC) /*InfoWasSet*/

8. write(ConnectNewClient, L)
9. read(L) /*Connected*/
10. write(CanBroadcast, NC)
11. read(NC) /*Broadcasting*/
12. write(MergeSockets, L)
13. Disconnect(L)
14. Merge(NC, L)
15. goto EndProgram
16. **Label ErrorCode**
17. write(KillProtocol, L)
18. Disconnect(NC)
19. **Label EndProgram**

Program P_R

1. read(R) /*ConnectNewClient*/
2. Connect(NRC)
3. write(Connected, R)
4. read(R) /*MergeSockets*/
5. Disconnect(R)
6. Merge(NRC, R)
7. goto EndProgram
8. **Label ErrorCode**
9. write(KillProtocol, R)
10. Disconnect(NRC)
11. **Label EndProgram**

5.4.2 Dôkaz správnosti programu

Ak počas pridávania programu nespadne žiadny proces, procesy P_L, P_R a P_N dostanú systém do zdravého stavu, pričom systém bude mať o jeden proces viac. Proces P_N sa pripojí na P_L, následne prijme správy obsahujúce jeho nové jednoznačné meno v systéme, informácie o celej sieti a stav zásobníka správ v posielaní, ktoré sú v procese P_L, a pošle svoje informácie (na akej IP a porte počúva) procesu P_L (V krokoch 1.–7. programu P_L a krokoch 1.–7. programu P_N). P_L potom pošle procesu P_R informácie, ako sa pripojiť na nový proces (kroky 8.–9. programuP_L, kroky 1.–3. programu P_R a krok 8. programu P_N). Potom P_L informuje P_N o tom, že už má spojenie na oboch svojich susedov, a preto môže začať pracovať ako účastník kruhu (kroky 10.–11. programu P_L a kroky 9.–11. programu P_N). Program P_N vtedy začne BroadcastAdd(), avšak ten je pozdržaný v procese P_R, kým on nedokončí zvyšok svojho programu. Na záver procesy P_L a P_R rozpoja svoje spojenie a každý označí proces P_N za svojho ľavého, resp. pravého suseda.

Ak počas pridávacieho protokolu dôjde k výpadku niektorého procesu okrem tých, ktoré vykonávajú protokol, nemá to nijaký vplyv na pridávací protokol, lebo oprava kruhu sa týka len procesov, ktorých sused spadol. Na pridávací protokol môžu mať vplyv tieto výpadky:

1. spadne pravý sused procesu P_L
2. spadne ľavý sused procesu P_R
3. spadne proces P_N
4. spadne proces P_L
5. spadne proces P_R

Prípad 1: spadne pravý sused procesu P_L Keďže v každom procese sú dva thready pre ľavý a pravý socket. Rovnako ako pri programoch pre odobratie procesu (programy P_R a P_L pre pravý a ľavý socket) aj program P_R pre odobratie procesu a program P_L pre pridanie procesu nepoužívajú rovnaké zdroje. Teda obidva tieto programy môžu bežať naraz na jednom procese. Preto spadnutie pravého suseda procesu P_L nemá žiadny vplyv na dokončenie pridávacieho protokolu. (Ak pravým susedom procesu P_L bol proces P_R, teda boli v kruhu len oni dvaja, spadá pod prípad "5. spadne proces P_R".)

Prípad 2: spadne ľavý sused procesu P_R Tu platí rovnaká úvaha ako v predošlom prípade. Proces P_R dokáže vykonávať naraz obidva protokoly, preto spadnutie jeho ľavého suseda nemá na pridávací protokol vplyv.

Prípad 3: spadne proces P_N Počas pridávacieho protokolu môže dôjsť k prerušeniu iba jednej linky medzi novým procesom a niektorým jeho susedom, alebo k prerušeniu obidvoch liniek (výpadok siete, alebo spadnutie procesu P_N).

Prerušenie linky medzi procesmi P_R a P_N:

1. Nepodarí sa vykonať operáciu Connect(NRC)
2. Operácia Connect(NRC) je úspešná a linka sa preruší až potom

Ak sa procesu P_R nepodarí pripojiť na proces P_N, tok riadenia programu skočí na riadok Label ErrorCode. Následne odošle procesu P_L správu KillProtocol, a odpojí spojenie zo socketu NRC (v skutočnosti tam momentálne nie je žiadne spojenie, lebo sa nepodarilo pripojiť). Proces P_L zatiaľ čaká na príkaze 9. read(L). Keď mu príde správa KillProtocol, jeho tok riadenia tiež skočí na riadok Label ErrorCode. Odošle procesu P_R správu KillProtocol, ktorú však proces ignoruje, lebo momentálne nie je vo vykonávaní pridávacieho protokolu, a potom P_L preruší spojenie socketu NC. Proces P_N toto prerušenie detekuje na príkaze 9. read(R). Operácia read() je neúspešná, preto tok riadenia skočí na riadok Label ErrorCode, následne odpojí všetky pripojené sockety a nakoniec proces dobrovoľne spadne. Keďže nedošlo k rozpojeniu kruhu medzi procesmi P_L a P_R, systém je v zdravom stave.

Ak sa procesu P_R podarí spojenie a linka medzi ním a novým procesom sa preruší až potom, pričom druhá linka zostane nového procesu nepreruší, dokončia všetky procesy pridávací protokol. Systém však v tomto momente nie je v zdravom stave. Avšak proces P_R po skončení protokolu detekuje prerušenie spojenia na pravom sockete, a preto spustí opravný protokol, kde proces P_R považuje za spadnutý práve P_N, preto opravný protokol vykoná s procesom P_L. Ako som dokázal vyššie, opravný protokol vráti systém do zdravého stavu, pričom donúti proces P_N dobrovoľne spadnúť.

Prerušenie linky medzi procesmi P_L a P_N:

1. Linka sa preruší skôr, ako program P_L vykoná krok 7. read(NC)
2. Linka sa preruší skôr, ako program P_L vykoná krok 11. read(NC)

3. Linka sa preruší neskôr

Ak sa linka medzi procesmi P_L a P_N preruší skôr ako vykonajú príkazy 7. read(NC), resp. 7. write(InfoWasSet, R), obidva procesy to detekujú na niektorej z predošlých operácií read() a write() a ich toky riadenia prejdú na príkaz Label ErrorCode. P_L potom pošle procesu P_R správu KillProtocol, P_R ukončí protokol, a následne P_L rozpojí spojenie socketu NC. Proces P_N rozpojí spojenia všetkých svojich socketov (momentálne len socket R) a dobrovoľne spadne. Systém je v zdravom stave, pretože nedošlo k rozpojeniu spojenia medzi procesmi P_L a P_R.

Ak sa linka preruší skôr, ako program P_L vykonal príkaz 11. read(NC), je už proces P_R spojený s procesom P_N, ale spojenie medzi procesmi P_L a P_R je ešte zachované. Proces P_N detekuje výpadok linky najneskôr na príkaze 11. write(Broadcasting, R), kde následne prejde na príkaz Label ErrorCode a odpojí všetky svoje sockety (R aj L) a potom dobrovoľne spadne. Proces P_L pošle procesu P_R správu KillProtocol (ľavý socket je ešte stále pripojený na proces P_R), a potom odpojí socket NC. Proces P_R po prijatí správy KillProtocol odpojí socket NRC a ukončí protokol.

Ak sa linka preruší po kroku 11. read(NC) programu P_L, procesy P_L a P_N to detekujú až po skončení protokolu. Systém v tej chvíli nie je v zdravom stave, avšak proces P_N začne opravný protokol s predpokladom, že proces P_L spadol. Ako som dokázal už vyššie, opravný protokol vráti systém do zdravého stavu.

Spadne proces P_L Ak dôjde k prerušeniu linky iba medzi procesmi P_L a P_R pred krokom 4. read(R) programu P_R, tak proces P_R predčasne ukončí protokol a následne spustí opravný protokol s pravým susedom procesu P_L (Tak ako je to popísané v časti 4.3 Program pre odobratie procesu.). Protokol donúti proces P_L spadnúť, čo donúti následne spadnúť aj proces P_N. Prerušenie linky neskôr ako program P_R vykoná príkaz 4. read(R) je považované za dobrovoľné, lebo to nijako neporušuje zdravosť systému (k tomuto prerušeniu by totiž došlo v nasledujúcich krokoch). Prerušenie linky medzi procesmi P_R a P_N som riešil vyššie.

Spadne proces P_R Spadnutie procesu P_R pred vykonaním kroku 12. write(MergeSockets, L) (zodpovedá to príkazu 4. read(R) programu P_R) vedie k prerušeniu protokolu a následnému dobrovoľnému spadnutiu procesu P_N. Neskôr ešte môže dôjsť k výpadku pravej linky procesu P_R (t.j. medzi ním a procesom P_N), alebo ľavej linky. Výpadok pravej linky som vyriešil vyššie. Ak dôjde k výpadku ľavej linky, môžu

nastať tieto situácie:

1. Proces P_N začal BroadcastAdd(), ale proces P_R ešte nepreposlal správu prvej fázy.

2. Proces P_N začal BroadcastAdd() a proces P_R stihol preposlať správu aspoň jednej fázy.

V prvom prípade ľavý sused procesu P_R ešte nevie o procese P_N, preto v opravnom protokole kontaktuje hneď proces P_L. Avšak počas opravného protokolu, proces P_L pošle svojmu ľavému susedovi správu Kill, preto proces P_N dobrovoľne spadne a po dokončení opravného protokolu bude systém v zdravom stave. V druhom prípade ľavý sused procesu P_R už dostal správu aspoň prvej fázy, preto funkcia opravného protokolu FindRightNeighbour() vráti proces P_N (okrem tabuľky procesov skontroluje aj zásobník s nedokončenými správami, kde skontroluje práve správy o pridávaní nového procesu). Ľavý sused procesu P_R sa potom spojí s procesom P_N, P_N donúti proces P_R spadnúť (ak sa tak už nestalo) a spolu obnovia kruh. Systém sa potom nachádza v zdravom stave. □

5.5 Protokol pre posielanie správ

Kvôli neexistencii priamych spojení medzi jednotlivými procesmi (s výnimkou susedných procesov) musí byť komunikácia medzi nimi sprostredkovaná pomocou ostatných procesov v kruhu. Tie okrem preposielania správ musia zabezpečiť, aby sa správy počas výpadku niektorého z nich nestratili. Každý proces vykonáva jednoduchý program, ktorý prijme správu od pravého suseda, správu spracuje (upraví jej stav v zásobníku), a potom ju ďalej prepošle k ľavému susedovi.

5.5.1 Program

Samotný program pre preposielanie správ je jednoduchá sekvencia krokov:

1. read(R)

2. if $Phase(m) \geq PRECOMMIT$ && NotInStack(m) then goto EndProgram

3. UpdateStack(m) /*zvýši fázu správy v zásobníku, alebo ju zo zásobníka odstráni*/

4. UpdatePhase(m) /*ak správu dostal druhýkrát, tak jej zvýši fázu*/

5. write(L, message)

6. Label EndProgram

5.5.2 Dôkaz správnosti programu

Program najprv prijme správu od svojho pravého suseda.

Následne proces zaktualizuje svoj zásobník pre nedokončené posielanie tak, že správu do zásobníka pridá (ak je to nová správa), alebo jej zvýši fázu (PRECOMMIT), alebo ju zo zásobníka odstráni (COMMIT). Ak je správa COMMIT a navyše je proces medzi adresátmi, vloží ju do zásobníka pre prijaté správy.

Ak proces už správu v tejto fáze prijal predtým, tak teraz jej zvýši fázu (4. UpdatePhase(m)) a pošle ju ľavému susedovi.

Proces posielanie skončí, keď dostane dvakrát správu s fázou COMMIT (2. krok programu).

Keď dôjde k opakovanému posielaniu správy po obnove kruhu (situáciu som analyzoval v časti "Nutnosť trojfázového protokolu"), detekuje sa to na 2. kroku programu a posielanie sa preruší.□

Veta Ak od istého okamihu nenastane žiaden ďalší výpadok, tak protokoly pre pridanie a odobratie procesu privedú systém do zdravého stavu.

Dôkaz: Dôkaz vyplýva z dôkazov funkčnosti samotných protokolov. □

Kapitola 6

Záver

V tejto práci som vychádzal z teoretického asynchrónneho komunikačného modelu, ktorý som rozšíril o odolnosť voči výpadkom uzlov.

Navrhol som architektúru distribuovaného systému spĺňajúceho sémantiku mnou navrhnutého teoretického modelu. Hlavnou myšlienkou architektúry je využitie grafovej štruktúry kruh. Z vlastnosti grafu, že je kruh (alebo niekoľko disjunktných kruhov), vyplýva, že všetky uzly sú stupňa 2. Preto je možné urobiť relatívne jednoduchú implementáciu tohto distribuovaného systému.

Mojou hlavnou snahou v tejto práci bolo najmä navrhnúť systém a dokázať jeho funkčnosť. Preto som sa nezaoberal časovou a správovou zložitosťou jednotlivých protokolov. Logickým pokračovaním pokračovaním práce by mohla byť optimalizácia protokolu pre posielanie správ. Jednou z možností by bolo využiť "socket pool"[1] v každom procese (okrem kruhových socketov), na priamu komunikáciu medzi jednotlivými procesmi. V tom prípade treba riešiť situáciu, keď bude chcieť s procesom komunikovať viac procesov, než má k dispozícii socketov. Rovnako aj distribuovanie informácie medzi všetky ostatné procesy (Broadcast) je v takomto prípade trochu problematický (ale určite nie neriešiteľný) problém.

Napokon možnou optimalizáciou môže byť aj zvolenie inej grafovej štruktúry, ktorá priamo povedie k zníženiu počtu správ.

[1] súbor socketov

Literatúra

[Pla06] Tomáš Plachetka. Unifying Framework for Message Passing, SOFSEM 2006, http://www.dcs.fmph.uniba.sk/~plachetk/PUBLICATIONS/sofsem06.pdf

[JGF96] Peyton Jones, S.L. Grodon, and S. Finne, Concurrent Haskell. In 23rd ACM Symposium on Principles of Programming Languages (1996)

[PVMTut] PVM: Parallel Virtual Machine. A Users' Guide and Tutorial for Networked Parallel Computing

[PVMvsMPI] G. A. Geist, J. A. Kohl, P. M. Papadopoulos. PVM and MPI: a Comparison of Features, 30. 5. 1996

[1] Vittoria Gianuzzi. Fault tolerant Aspects of the PVM Based Virtual Time Machine PV^2M

[MPForum] http://www.mpi-forum.org/

[2] http://msdn2.microsoft.com

[3] http://www.netlib.org/pvm3/

[4] Wikipedia - the free encyclopedia. http://www.wikipedia.org/