Distribuované operacní systémy

Filip Zavoral

Pomocné ucební texty k stejnojmenné prednášce

v. 12.3.2001

Upozornení: tyto ucební texty nejsou presným obrazem na prednášce probírané látky – nekteré pasáže tu jsou popsány nebo zobrazeny velmi strucne ci heslovite, naopak jiné pasáže jsou rozebrány podrobneji než bývá obvyklé na prednášce. Proto je nelze považovat za plnohodnotný a úplný zdroj informací pro úcely složení zkoušky.

Obsah

1. ÚVOD 6
   - 1.1 POTENCIÁLNÍ VÝHODY DISTRIBUOVANÝCH SYSTÉMU 6
   - 1.2 NEVÝHODY DISTRIBUOVANÝCH SYSTÉMU 7
   - 1.3 DEFINICE A FUNKCE DISTRIBUOVANÝCH OPERACNÍCH SYSTÉMU 8
     - *1.3.1 Transparentnost 8*
     - *1.3.2 Prizpusobivost 9*
     - *1.3.3 Spolehlivost 10*
     - *1.3.4 Výkonnost 10*
     - *1.3.5 Rozširitelnost 11*
   - 1.4 ARCHITEKTURY A MODELY DISTRIBUOVANÝCH SYSTÉMU 11
   - 1.5 HARDWARE 12
2. MEZIPROCESOVÁ KOMUNIKACE 15
   - 2.1 KLIENT/SERVER MODEL 15
     - *2.1.1 Request/reply protokol 16*
     - *2.1.2 Struktura zpráv 17*
     - *2.1.3 Synchronní a asynchronní komunikace 17*
     - *2.1.4 Prímé zasílání zpráv vs. zasílání zpráv pres schránku 19*
     - *2.1.5 Prímé a neprímé adresování - linky a porty 20*
   - 2.2 SPOLEHLIVOST KOMUNIKACE 20
     - *2.2.1 Spolehlivost serveru 21*
     - *2.2.2 Spolehlivost klienta 22*
     - *2.2.3 Vztah komunikace a virtuální pameti 22*
   - 2.3 REMOTE PROCEDURE CALL 23
3. SKUPINOVÁ KOMUNIKACE 25
   - 3.1 UNICAST, BROADCAST & MULTICAST 25
   - 3.2 PRACOVNÍ SKUPINY 25
   - 3.3 ATOMICITA 26
   - 3.4 USPORÁDÁNÍ ZPRÁV 27
   - 3.5 DORUCOVACÍ PROTOKOLY 29
     - *3.5.1 Kauzální závislost 29*
     - *3.5.2 Kauzální usporádání dorucovaných zpráv 29*
     - *3.5.3 Vektorové hodiny 29*
     - *3.5.4 Dorucovací protokol pro jednu skupinu 30*
     - *3.5.5 Dorucovací protokol pro prekrývající se skupiny 31*
4. SYNCHRONIZACE 32
   - 4.1 USPORÁDÁNÍ UDÁLOSTÍ A LOGICKÉ HODINY 32
   - 4.2 SYNCHRONIZACE FYZICKÝCH HODIN 34
     - *4.2.1 Cristianuv algoritmus 35*
     - *4.2.2 Berkeley algoritmus 35*
     - *4.2.3 Distribuovaný algoritmus 36*
   - 4.3 VZÁJEMNÉ VYLOUCENÍ PROCESU 36
     - *4.3.1 Centralizovaný algoritmus 36*
     - *4.3.2 Lamport 37*
     - *4.3.3 Ricart & Agrawala 38*
     - *4.3.4 Suzuki & Kasami 38*
     - *4.3.5 Token ring 38*
     - *4.3.6 Porovnání jednotlivých algoritmu 39*
     - *4.3.7 Vyloucení procesu s prioritami 39*
   - 4.4 VOLBA KOORDINÁTORA 39
     - *4.4.1 Bully algoritmus 39*
     - *4.4.2 Kruhový algoritmus 40*
     - *4.4.3 Kruhový algoritmus - O(n log n) 40*
   - 4.5 SYNCHRONIZACE UKONCENÍ A DETEKCE GLOBÁLNÍHO STAVU 41

4.5.1 Dijkstra-Scholten (DS) algoritmus....41
4.5.2 Znackový (TM) algoritmus....42
4.5.3 Detekce globálního stavu....42

5. TRANSAKCNÍ ZPRACOVÁNÍ....44

5.1 VLASTNOSTI TRANSAKCÍ....44
5.1.1 Vnorené transakce....45
5.1.2 Zotavení z chyb....45
5.2 IMPLEMENTACE TRANSAKCÍ....46
5.2.1 Stabilní pamet....46
5.2.2 Lokální pracovní prostor....46
5.2.3 Intencní seznam....47
5.3 TRANSAKCNÍ KOMUNIKACNÍ PRIMITIVA....47
5.3.1 Dvoufázový commit (potvrzování)....48
5.4 KONTROLA KONKURENCE....49
5.4.1 Zámky....49
5.4.2 Dvoufázové uzamykání....49
5.4.3 Optimistická kontrola konkurence....50
5.4.4 Casové znacky....50

6. DISTRIBUOVANÁ SDÍLENÁ PAMET (DSM)....52

6.1 KONZISTENCNÍ MODELY....52
6.1.1 Striktní konzistence (strict consistency)....52
6.1.2 Sekvencní konzistence (sequential consistency)....52
6.1.3 Kauzální konzistence (causal consistency)....54
6.1.4 PRAM konzistence (PRAM consistency)....54
6.1.5 Slabá konzistence (weak consistency)....55
6.1.6 Uvolnovací konzistence (release consistency)....56
6.1.7 Prístupová konzistence (entry consistency)....56
6.1.8 Shrnutí konzistencních modelu....57
6.2 DISTRIBUOVANÉ STRÁNKOVÁNÍ....57
6.2.1 Nalezení vlastníka stránky:....58
6.2.2 Nalezení kopií:....58
6.2.3 Alokace stránek:....58
6.3 DISTRIBUOVANÉ SDÍLENÉ PROMENNÉ....59
6.3.1 Munin....59
6.4 DISTRIBUOVANÉ OBJEKTY....59

7. IDENTIFIKACE OBJEKTU....60

7.1 IDENTIFIKACNÍ SYSTÉM....60
7.1.1 Jména....61
7.1.2 Struktura jmen....61
7.1.3 Cesty....62
7.1.4 Adresy a mapování....63
7.2 SYSTÉMOVÁ JMÉNA....63
7.2.1 Interní identifikátory....63
7.2.2 Lokálne implementacní identifikátory....64
7.2.3 Mapování systémových jmen....64
7.3 KAPABILITY....64
7.3.1 Kapability s podpisem....65
7.3.2 Kapability s redundancí....65
7.4 UŽIVATELSKÁ JMÉNA....66
7.4.1 Servry jmen....66
7.4.2 Správa prostoru jmen....66
7.4.3 Rozklad jmen....66
7.4.4 Agenti....67
7.4.5 Vyhledávání jmen....68

8. PROCESY....69

8.1 VLÁKNA, NITE A THREADY....69
8.1.1 Použití vláken....70

8.1.2 Implementace vláken ... 70
8.1.3 Vlákna a zprávy ... 71
8.2 SYSTÉMOVÉ MODELY ... 71
8.2.1 Workstation model ... 71
8.2.2 Procesor pool model ... 72
8.3 VZDÁLENÉ SPOUŠTENÍ PROCESU ... 72
8.4 ALOKACE PROCESORU ... 73
8.4.1 Klasifikace alokacních algoritmu ... 73
8.4.2 Implementace alokacních algoritmu ... 74
8.4.3 Deterministický grafový algoritmus ... 74
8.4.4 Up-down algoritmus (Mutka-Livny) ... 74
8.4.5 Hierarchický algoritmus ... 75
8.4.6 Distribuovaný heuristický algoritmus ... 76
8.4.7 Bidding (obchodní) algoritmus ... 76

9. MIGRACE PROCESU ... 77

9.1 MECHANISMUS MIGRACE ... 77
9.1.1 Dorucování zpráv ... 77
9.1.2 Stav procesu ... 78
9.1.3 Komunikace s okolím behem migrace a po migraci ... 78
9.1.4 Implementace migracní strategie a vlastní migrace ... 79
9.1.5 Vícenásobná migrace ... 79
9.1.6 Transparence ... 79
9.1.7 Reziduální dependence ... 80
9.1.8 Virtuální pamet ... 80
9.1.9 Cíle návrhu migrace ... 81
9.1.10 Kdy migrovat ... 81
9.1.11 Výber cíle migrace - migracní strategie ... 81
9.2 PREHLED NEKTERÝCH SYSTÉMU ... 82
9.2.1 DEMOS/MP ... 83
9.2.2 Charlotte ... 84
9.2.3 V ... 86
9.2.4 MOSIX ... 87
9.2.5 Sprite ... 89
9.2.6 Migrace objektu - systém Emerald ... 92
9.3 KOMUNIKACE PRI MIGRACI ... 93
9.4 VYVAŽOVÁNÍ ZÁTEŽE ... 95
9.4.1 Párový algoritmus (Bryant & Finkel) ... 95
9.4.2 Vektorový algoritmus (Mosix) ... 96
9.4.3 Bidding algoritmus (Stankovic & Sidhu) ... 96
9.4.4 SLA algoritmus (Stankovic) ... 97
9.4.5 BDT algoritmus (Stankovic) ... 97
9.4.6 Centralizovaný algoritmus ... 97
9.4.7 Lokální algoritmus ... 97
9.4.8 Porovnání distribuovaných vyvažovacích algoritmu ... 97

10. SPRÁVA PROSTREDKU ... 98

10.1 SPRÁVCI PROSTREDKU ... 98
10.1.1 Centralizovaná správa prostredku ... 98
10.1.2 Distribuovaná správa prostredku ... 99
10.1.3 Správa prostredku pomocí agentu ... 99
10.2 ZABLOKOVÁNÍ (DEADLOCK) ... 100
10.2.1 Modely deadlocku ... 100
10.2.2 Algoritmy detekce deadlocku ... 101
10.2.3 Prevence deadlocku ... 102
10.3 OCHRANA PROSTREDKU ... 103
10.3.1 Klasifikace bezpecnosti DoD - the Orange Book ... 103

11. SPRÁVA SOUBORU ... 104

11.1 DISKOVÉ SLUŽBY ... 104
11.1.1 Bloková struktura souboru ... 104

11.1.2 Systémová identifikace *104*
11.1.3 Vyrovnávací pamet (cache) *104*
11.2 SOUBOROVÉ SLUŽBY 105
11.2.1 Sémantika sdílení souboru *106*
11.3 ADRESÁROVÉ SLUŽBY 107
11.3.1 Prostor jmen adresáru *108*
11.4 REPLIKACE 108
11.4.1 Aktualizacní protokoly *108*

1. Úvod

Až do poloviny 80. let byly pocítace velké a drahé, dokonce i minipocítace stály desítky tisíc dolaru. Vetšina organizací mela jeden centrální pocítac (mainframe), na který bylo pripojeno, podle výkonu, nekolik až nekolik desítek terminálu. Od zacátku 80. let se ale situace zacala výrazne menit. V té dobe se zacaly hromadne vyrábet výkonné mikroprocesory. Bežne dostupnými se staly mikropocítace založené nejdríve na 8-bitových, pozdeji na 16- a 32-bitových a v soucasné dobe už i na 64-bitových mikroprocesorech. Mnoho z nich melo výkonnost srovnatelnou se stredne velkým strediskovým pocítacem, avšak za zlomek jeho ceny. Druhou zmenou, která ovlivnila vývoj pocítacového prumyslu, byl vývoj rychlých lokálních sítí (LAN). Tyto síte dokáží prenášet informace mezi pocítaci rádove za milisekundy, vetší objemy dat jsou prenášeny rychlostí desítek milionu bitu za sekundu.

Dusledkem vývoje techto technologií byl od poloviny 80. let nárust sítí osobních pocítacu, na kterých jsou provozovány **sítové** a **distribuované operacní systémy**. V sítových operacních systémech má každý pocítac svuj operacní systém. Každý z techto operacních systému má nadstavbu umožnující komunikaci s ostatními systémy v síti. Uživatelé mají na zreteli existenci více pocítacu - mohou se explicitne prihlásit ke vzdálenému serveru, kopírovat soubory z jednoho pocítace na druhý apod.

Distribuovaný systém poskytuje vyšší stupen transparentnosti a sdílení prostredku nežli sítové systémy. Distribuované systémy je možno rozdelit na **distribuované systémy na uživatelské úrovni** (distribuované aplikace, distribuované výpocetní prostredí) a **distribuované operacní systémy**. V distribuovaných systémech na uživatelské úrovni je podpora distribuce umístena v softwarové vrstve nad (nedistribuovaným) operacním systémem. Príkladem takového systému je OSF Distributed Computing Environment (DCE). Distribuované operacní systémy implementují podporu distribuovaného zpracování prímo v jádru operacního systému. Celý systém se tváří jako tradicní jednoprocesorový systém, prestože jeho jednotlivé komponenty jsou fyzicky rozmísteny na jednotlivých pocítacích. V pravém distribuovaném operacním systému se uživatel nemusí (casto ani nemuže) zabývat urcováním na kterém pocítaci ci procesoru jeho programy beží nebo kde jsou fyzicky uloženy jeho soubory. O to se automaticky a (v ideálním prípade) efektivne stará sám operacní systém. Príkladem takového systému je napr. Amoeba, Chorus, Spring apod.

Architektura vetšiny distribuovaných systému je založena na modelu **klient/server**. V asymetrickém klient/server modelu jsou vybrané pocítace urceny pro jednotlivé servry (napr. file server, autentifikacní server apod.), zatímco na ostatních pocítacích beží uživatelské programy. V symetrickém klient/server modelu muže být každý pocítac jak serverem pro jiné pocítace, tak i klientem jiných serveru. Tento model je pro použití v distribuovaných operacních systémech vhodnejší pro vyšší stupen decentralizace, lepší využití dostupných prostredku a lepší rozširitelnost.

1.1 Potenciální výhody distribuovaných systému

Asi pred ctvrt stoletím platilo zhruba pravidlo, že výpocetní síla CPU je prímo úmerná druhé mocnine ceny. Zaplacením dvojnásobné ceny se výkon pocítace zvýšil ctyrikrát. To vedlo k velkému rozšírení sálových pocítacu, protože vetšina spolecností kupovala nejvýkonnejší dostupné pocítace. Avšak s rozvojem mikroprocesorové technologie toto pravidlo prestalo platit - za nekolik stovek dolaru je možno získat procesor s vetším výkonem než mely nejvetší sálové pocítace zacátku 80. let. Zaplacením dvojnásobku ceny se výkon pocítace zvetší jen nepatrne. Proto se zacaly používat architektury založené na propojení vetšího množství levných procesoru do jednoho systému. Takto vytvorené systémy mají mnohem lepší pomer ceny a výkonu nežli centralizované systémy.

Propojením velkého množství procesoru lze dokonce dosáhnout takového celkového výkonu systému, jaký nelze dosáhnout použitím jednoho sálového pocítace za jakoukoliv cenu. S použitím

soucasné technologie je možné vytvorit systém složený z 1000 CPU o výkonu 20 MIPS (miliónu instrukcí za sekundu). Celý systém pak muže mít výkon až 20000 MIPS. Aby tohoto výkonu dosáhl jediný procesor, musel by vykonat 1 instrukci za 0.05 ns, což je rychlost rádove 100krát vetší nežli u nejrychlejších soucasných procesoru.

Dalším duvodem pro vytvárení distribuovaných systému je to, že nekteré aplikace jsou svým charakterem samy distribuované. Napríklad u továrního systému rídícího roboty a stroje podél celé výrobní linky je rozumné, aby každý robot nebo skupina robotu byly rízeny vlastním pocítacem. Hardwarovým a softwarovým propojením techto pocítacu dostaneme prumyslový distribuovaný systém. Podobne v bankovnictví propojením pocítacu ve všech pobockách banky spolu s penežními automaty dostaneme bankovní distribuovaný systém.

Další potenciální výhodou distribuovaných systému je spolehlivost. Pri rozložení výpocetní záteže na nekolik procesoru zpusobí výpadek jednoho procesoru prerušení práce pouze jednoho pocítace, zatímco ostatní pracují dál. V ideálním prípade 5% vypnutých pocítacu zpusobí 5% zmenšení výkonu celého systému. U centralizovaných sálových pocítacu vetšinou výpadek nekterého duležitého obvodu zpusobí výpadek celého systému. U nekterých kritických aplikací, napr. rízení jaderného reaktoru nebo rízení letového provozu, muže být spolehlivost nejduležitejším duvodem pro výber distribuovaného rídícího systému.

Rozširitelnost systému je další výhodou oproti centralizovaným systémum. V prípade, že nedostacuje kapacita a výkon sálového pocítace, jsou možná pouze dve rešení - vymenit celý pocítac za nový nebo pridat druhý sálový pocítac. Obe dve rešení mají své velké nevýhody. Oproti tomu u distribuovaných systému stací pridat nekolik dalších komponent do systému (napr. procesory, disky, terminály apod.), aby bylo dosaženo požadovaného výkonu.

Konecne distribuované systémy umožnují vzdálené provádení úloh a rozdelení záteže pokud možno rovnomerne mezi propojenými pocítaci. Navíc, pokud jsou v systému procesory ruzných typu nebo výkonu, lze každé úloze priradit nejvhodnejší procesor.

Ekonomika	Mikroprocesory poskytují lepší pomer ceny a výkonu
Výkon	Distribuovaný systém muže dosáhnout lepšího celkového výkonu než potenciálne jakýkoliv sálový pocítac
Inherentní distribuovanost	Nekteré aplikace jsou svým charakterem samy distribuované
Spolehlivost	Pri selhání cásti systému muže zbytek systému pokracovat v cinnosti
Rozširitelnost	Kapacita a výkon systému muže být po cástech zvyšován
Sdílení	Sdílení spolecných dat a periferií
Vzdálený beh procesu	Sdílení a vyvažování výpoctové záteže

Tab. 1 -Výhody distribuovaných systému

1.2 Nevýhody distribuovaných systému

Prestože distribuované systémy mají mnoho výhod, mají i nekteré slabiny. Jednou z nejvetších slabin je software. V soucasné dobe s tvorbou a použitím distribuovaného softwaru není príliš zkušeností. Stále existují otevrené otázky o vhodném operacním systému, programovacích jazycích, vhodných aplikacích a nástrojích pro jejich tvorbu. Není jasno o tom, co vše mají uživatelé vedet o distribuci, která rozhodování má provádet systém sám a která rozhodnutí ponechat na uživateli apod.

Software pro distribuované systémy, obzvlášte distribuované operacní systémy, musí rešit mnohem více problému a provádet mnohem více rozhodování než jejich centralizované protejšky a ekvivalenty. Proto je pro jejich beh zapotrebí dostatecne kvalitní hardware - výkonné pocítace s dostatkem operacní pameti i diskového prostoru.

Další problémy zpusobuje vlastní propojení jednotlivých pocítacu. Komunikace muže být nespolehlivá, mohou se ztrácet jednotlivé zprávy, což vyžaduje další software, který tyto problémy

bude rešit. Navíc pri intenzivní komunikaci se muže sít zahltit. Pokud sít nebude dostatecne pruchozí, znegují se témer všechny výhody, které byly použitím distribuovaného systému dosaženy.

Snadné sdílení dat vyvolává další problém - bezpecnost. Je zapotrebí zajistit, aby utajovaná data nebyla použita neoprávneným uživatelem, aby data jednoho uživatele nebyla zmenena nebo dokonce smazána jiným uživatelem apod. Nedozírné následky nedostatecne vyrešené bezpecnosti jsou zrejmé na první pohled napr. v bankovnictví, v oblasti výzkumu zvláštní duležitosti, ve vojenství apod.

Pres výše uvedené problémy prevažuje názor, že výhody distribuovaných systému prevažují jejich nevýhody, a že distribuované systémy budou hrát v nejbližších letech stále duležitejší roli.

Software	Málo existujícího software, velké požadavky na hardware
Propojení do síti	Sít muže být zahlcena
Bezpecnost	Snadný prístup k datum vyvolává nutnost utajení

Tab. 2 - Nevýhody distribuovaných systému

1.3 Definice a funkce distribuovaných operacních systému

Distribuovaný operacní systém je systém, který beží na množine procesoru, které nesdílejí spolecnou pamet , pricemž poskytuje uživateli dojem jednoho pocítace.

V literature bývá tato vlastnost nekdy oznacována jako **jednotný obraz systému** (**single system image**) nebo také **virtuální uniprocesor**. Krome této základní vlastnosti by distribuovaný operacní systém mel:

- efektivne rídit pridelování prostredku dostupných na síti
- skrýt fyzické rozmístení prostredku
- poskytovat mechanismus ochrany prístupu systémových prostredku pred neautorizovanými uživateli
- umožnovat spolehlivou a bezpecnou komunikaci

Základem každého distribuovaného systému je jednotný, globální mechanismus meziprocesové komunikace (IPC) takový, aby každý proces mohl jednotne komunikovat s jakýmkoliv jiným procesem. Tato komunikace by mela být zcela transparentní bez ohledu na fyzické umístení jednotlivých procesu. Komunikace procesu musí být v nekterých prípadech synchronizována. Protože však v distribuovaných systémech neexistuje spolecná pamet ani spolecné hodiny, navíc pri komunikaci vznikají prodlevy, nelze použít synchronizacní metody urcené pro centralizované operacní systémy. Z tohoto duvodu by každý distribuovaný systém mel poskytovat jednotný synchronizacní mechanismus.

Vzhledem k tomu, že úlohy jednoho uživatele mužou bežet na nekolika procesorech a zároven úlohy nekolika uživatelu mužou bežet na jednom procesoru, musí distribuovaný operacní systém zabezpecit jednotný systém ochrany a autentifikace. Ze stejného duvodu je nezbytné, aby pro všechy propojené pocítace existoval jednotný identifikacní systém, který by zarucoval jednotný prístup ke všem prostredkum systému (napr. souboru) z libovolného pocítace.

Pro dosažení všech výše uvedených vlastností je zapotrebí pri tvorbe distribuovaného systému zabezpecit následující podmínky.

1.3.1 Transparentnost

Pravdepodobne nejduležitejší podmínkou dosažení iluze jednotného systému je transparentnost. Té muže být dosaženo ve dvou úrovních - na úrovni komunikace s uživatelem nebo na úrovni komunikace procesu s jádrem systému. Dosažení transparentnosti na systémové úrovni klade mnohem

vetší nároky na jádro operacního systému, avšak výsledný výkon a schopnosti celého systému bývají výrazne lepší.

Idea transparentnosti muže být vztažena na nekolik oblastí distribuovaných systému.

Prístupová transparentnost	Proces má stejný prístup k jak k lokálním tak ke vzdáleným prostredkum
Lokacní transparentnost	Uživatel (proces) nemuže ríci kde jsou jednotlivé prostredky umísteny
Migracní transparentnost	Prostredky se mohou premistovat mezi jednotlivými pocítaci
Exekucní transparentnost	Procesy mohou bežet na libovolném procesoru zapojeném do systému
Replikacní transparentnost	Uživatel nemuže ríci kolik kopií daného objektu existuje
Konkurencní transparentnost	Prostredky mohou být automaticky využívány zároven nekolika uživateli
Paralelismová transparentnost	Ruzné cinnosti mohou být provádeny paralelne bez vedomí uživatele

Tab. 3 - Druhy transparentnosti v distribuovaném systému

Prístupová transparentnost znamená, že proces má stejnou metodu prístupu jak k lokálním tak ke vzdáleným prostredkum. Napr. prístup k nejakému souboru je stále stejný i když je soubor fyzicky umísten na jiném pocítaci. **Lokacní transparentnost** zabezpecuje, že uživatelský proces nemusí znát aktuální umístení jednotlivých prostredku. Ve jménech prostredku nesmí být zakódováno jejich umístení, napr. systém je jmény souboru tvaru *pocítac1:cesta/jméno_souboru* nesplnuje podmínku lokacní transparentnosti.

Migracní transparentnost znamená, že prostredky musí být schopné premístení na libovolné jiné místo beze zmeny jména a bez vlivu na procesy, které práve daný prostredek využívají nebo používají. Operacní systém musí zabezpecit, aby po dobu premistování prostredku (napr. souboru na jiný server) byly požadavky pozdrženy a poté presmerovány na nové místo. Podobným zpusobem funguje i **exekucní transparentnost** - procesy musejí být schopny bežet na libovolném procesoru zapojeném do systému. Tato vlastnost umožnuje premistování behu programu a sdílení a vyvažování výpoctové záteže.

Jestliže distribuovaný systém splnuje podmínku **replikacní transparentnosti**, pak umožnuje vytvárení kopií souboru nebo jiných prostredku bez vedomí uživatele. Replikované prostredky jsou velmi duležité zejména u rozsáhlých systému, kde prístup napr. k souboru fyzicky umístenému na jiném kontinente by trval neprimerene dlouhou dobu. Replikace prostredku je také významná z duvodu spolehlivosti - v prípade, že jedna z kopií selže, ostatní kopie mohou být nadále využívány.

Distribuované systémy vetšinou umožnují prístup více uživatelum najednou. Systém s **konkurencní transparentností** by mel umožnovat prístup uživatele k libovolným prostredkum bez ohledu na to, kolik uživatelu tyto prostredky práve využívá.

Nejzajímavejší, ale zároven nejméne probádaná vlastnost je **paralelismová transparentnost**. Pokud je v systému velké množství volných procesoru, uživatel by mel mít možnost je využívat co možná nejvíce podle jeho potreb. Operacní systém by mel teoreticky sám využít všech dostupných procesoru a provádet danou úlohu na tolika procesorech, aby celkový cas provádení byl co nejmenší. Bohužel, soucasné vedomosti jsou tomuto ideálu na hony vzdáleny - v naprosté vetšine soucasných systému úloha, která by mela využívat zároven nekolik procesoru, musí být speciálne naprogramována s explicitními požadavky na procesory.

1.3.2 Prizpusobivost

Další klícovou podmínkou dobre fungujícího distribuovaného systému je prizpusobivost (flexibilita). Vzhledem k tomu, že distribuovaný systém beží na mnohem složitejší strukture hardwaru

než klasické operacní systémy, mel by být daleko méne citlivý na zmenu prostredí. Prizpusobivost muže být opet vztažena na nekolik oblastí distribuovaných systému.

Autonomie	Každý pocítac je potenciálne schopný samostatné funkcnosti
Decentralizované rízení a rozhodování	Každý procesor vykonává rozhodnutí nezávisle na ostatních
Migrace procesu a prostredku	Procesy i prostredky mohou být premísteny na jiný pocítac
Vyvažování výpoctové záteže	Úlohy mohou být premísteny na méne vytížený procesor

Tab. 4 - Ruzné vlastnosti prizpusobivých distribuovaných systému

Jednotlivé pocítace zapojené do systému by mely být **autonomní**, tj. mely by být schopny samostatné funkcnosti. To je ve vetšine moderních distribuovaných systému zajišteno architekturou **microkernel**. Na každém pocítaci je implementováno pouze nejnutnejší jádro systému obsahující meziprocesovou komunikaci, správu vnitrní pameti, nízkoúrovnovou správu procesu a nízkoúrovnové služby pro styk s periferiemi. Ostatní služby operacního systému, jako napr. filesystem, jsou implementovány jako servery na uživatelské úrovni. Tímto zpusobem je napr. možné vytvorit distribuovaný operacní systém s nekolika fileservery - jeden podporující souborové služby napr. MS-DOSu, druhý umožnující prístup k souborum na UNIXové stanici apod.

Duležitou vlastností prizpusobivých distribuovaných systému je decentralizované rízení další cinnosti a decentralizované rozhodování. V prípade, že každý procesor vykonává svá rozhodnutí nezávisle na ostatních procesorech, je celý systém mnohem méne citlivý na ruzné poruchy síte, pridávání a odstranování pracovních stanic, výpadky servru a pod.

Plnohodnotný distribuovaný systém by mel umožnovat **migraci procesu a prostredku** mezi jednotlivými pocítaci. V prípade, že nejaký server spravuje vetší množství prostredku, než na co stací jeho kapacita, lze cást prostredku premístit na jiný pocítac. Stejne tak procesoru, na kterém beží velké množství aktivních procesu, lze odlehcit premístením urcitého množství procesu na méne vytížený procesor (sdílení záteže, load sharing). Migrace také umožnuje premístení prostredku z pocítace, který má být ze systému docasne ci trvale vyrazen, a tím zachování plné funkcnosti celého systému.

Transparentní migrace procesu je podmínkou toho, aby distribuovaný systém umožnoval **vyvažování výpoctové záteže** (load balancing), nekterými autory nazývané též distribuované plánování (distributed scheduling). Vyvažování záteže spocívá v automatické distribuci procesu v celém systému tak, aby výsledná zátež na jednotlivých procesorech byla zhruba stejná.

1.3.3 Spolehlivost

Jednou z puvodních motivací pro tvorbu distribuovaných systému bylo dosažení vyšší spolehlivosti nežli u klasických centralizovaných systému. Základní ideou zvýšení spolehlivosti je to, že v prípade výpadku nekterých komponent systému si ostatní rozdelí jejich práci. Jestliže napr. nespolehlivost jednoho servru je 1%, pak by teoreticky nespolehlivost ctyr spražených servru mela být $0.01^4 = 0.00000001$, tedy jedna milióntina procenta. Soucasné distribuované systémy však mají služby jednotlivých servru natolik vzájemne propojeny, že zvyšovýní spolehlivosti probíhá mnohem pomaleji, pokud vubec. V jedné casto citované poznámce Leslie Lamport, klasik v oblasti distribuovaných systému, definoval distribuovaný systém jako: *"Systém, který pro me odmítá vykonávat jakoukoliv cinnost, protože pocítac, o kterém jsem nikdy neslyšel a ani nepotrebuji jeho služby, byl náhodne vypnut*". Prestože tato poznámka byla silne nadnesena, je pravdou, že v oblasti spolehlivosti distribuovaných systému je ješte mnoho oblastí zasluhujících si dukladný výzkum.

1.3.4 Výkonnost

K tomu, aby byl systém prakticky použitelný, nestací jen dosažení flexibility, transparentnosti a spolehlivosti. Když systém bude mít všechny tyto vlastnosti, ale bude nepoužitelne pomalý, bude

k nicemu. Konkrétne beh urcité aplikace v distribuovaném systému by nemel být výrazne pomalejší než v klasickém jednoprocesorovém systému stejné trídy. Bohužel, soucasné distribuované systémy toto pravidlo príliš nesplnují - pro dosažení odpovídající rychlosti behu aplikace je zapotrebí výkonnejší hardware. Zpomalení je zpusobeno jednak pomalejším prenosem zpráv po síti, jednak složitejším softwarem.

Pri návrhu systému je zapotrebí dát velký pozor na granularitu výpocetní jednotky. V prípade, že se budou vzdálene provádet príliš malé kousky programu, jako napr. scítání dvou císel, neúmerne vzroste potreba komunikace a nekolikanásobne se zvýší režie potrebná pro inicializaci, rozbehnutí a ukoncení jednotlivých cástí programu. Na druhé strane prenášení príliš velkých kusu programu na vzdálený pocítac muže trvat neprimerene dlouho.

1.3.5 Rozširitelnost

Všechny soucasné distribuované systémy jsou konstruovány pro práci nekolika desítek až stovek procesoru. Potenciálne je však možné, že budoucí komplexní distribuované systémy budou o nekolik rádu vetší a rešení, která fungovala dobre pro 200 procesoru budou naprosto nevhodná pro 200?000?000 procesoru. Prestože soucasné znalosti o takto obrovských systémech jsou minimální (a zkušenosti s nimi prakticky žádné), jeden základní princip je znám: vyhnout se cemukoliv centralizovanému - servrum, tabulkám, algoritmum apod. Základními charakteristikami distribuovaných algoritmu jsou:

1. Žádný pocítac nemá úplnou informaci o celkovém stavu systému
2. Procesy se rozhodují pouze na základe lokálne dostupných informací
3. Výpadek jednoho pocítace nesmí zpusobit nefunkcnost algoritmu
4. Nelze spoléhat na existenci presných globálních hodin

Význam prvních trech charakteristik je zrejmý. Ctvrtá je možná méne zrejmá, avšak stejne duležitá. Jakýkoliv algoritmus založený na myšlence "Presne ve 12:00 si všechny pocítace zaznamenají velikost výstupní fronty" musí nutne selhat, protože nelze všechny hodiny presne sesynchronizovat. Na lokální síti je sice možné sesynchronizovat hodiny na rozdíl nekolika milisekund, avšak v celostátním ci celosvetovém merítku je to nemožné.

koncept	Príklad
Centralizované komponenty	Jednotný mail server pro všechny uživatele
Centralizované tabulky	Jednotný telefonní seznam
Centralizované algoritmy	Smerovací algoritmus na základe úplné informace

Tab. 5 - Potenciální úzká místa velkých distribuovaných systému

1.4 Architektury a modely distribuovaných systému

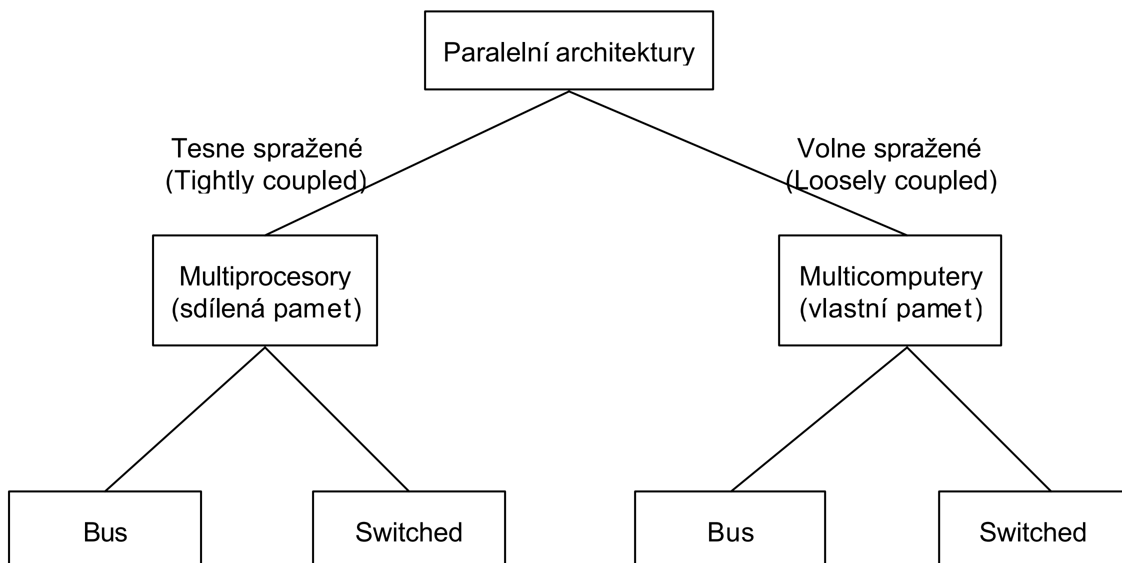
```
workstation / processor pool / hybrid architecture
process / object / UNIX-like model
```

1.5 Hardware

Prestože jsou všechny distribuované systémy tvoreny urcitým množstvím procesoru, je nekolik rozdílných možností jejich organizace a propojení. Podle prístupu k pameti je mužeme rozdelit na **multiprocesory**, které sdílejí spolecnou pamet, a **multicomputery**, které spolecnou pamet nesdílejí.

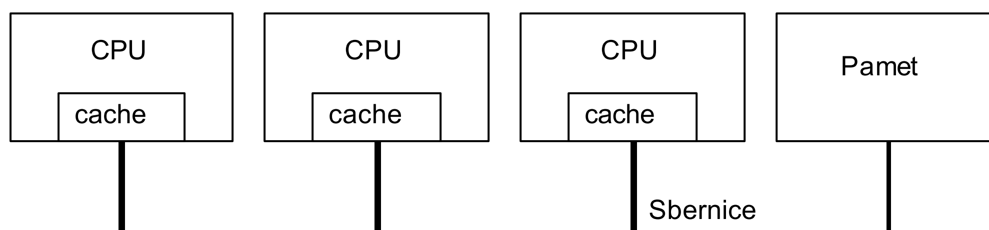
U multiprocesoru existuje jeden spolecný adresový prostor pro všechny propojené procesory. To znamená, že když jeden procesor zapíše na nejakou adresu urcitou hodnotu, druhý procesor z této adresy precte totéž císlo. Naproti tomu každý procesor multimulticomputeru má vlastní adresový prostor. Zmena na jakékoliv adrese nijak neovlivní adresový prostor ostatních procesoru.

Každá z techto dvou kategorií muže být dále rozdelena podle zpusobu propojení jednotlivých uzlu. **Sbernicová architektura** (bus architecture) využívá jednotné médium, napr. kabel, sbernici, ci jiné médium, které propojuje všechny uzly. Na podobném principu funguje napr. kabelová televize. **Prepínacová architektura** (switched architecture) naopak používá prímé propojení mezi jednotlivými uzly. Zprávy se zasílají mezi prímo propojenými uzly, pricemž v každém meziuzlu se provádí rozhodnutí o dalším postupném cíli zprávy. Na tomto principu funguje verejná telefonní sít.



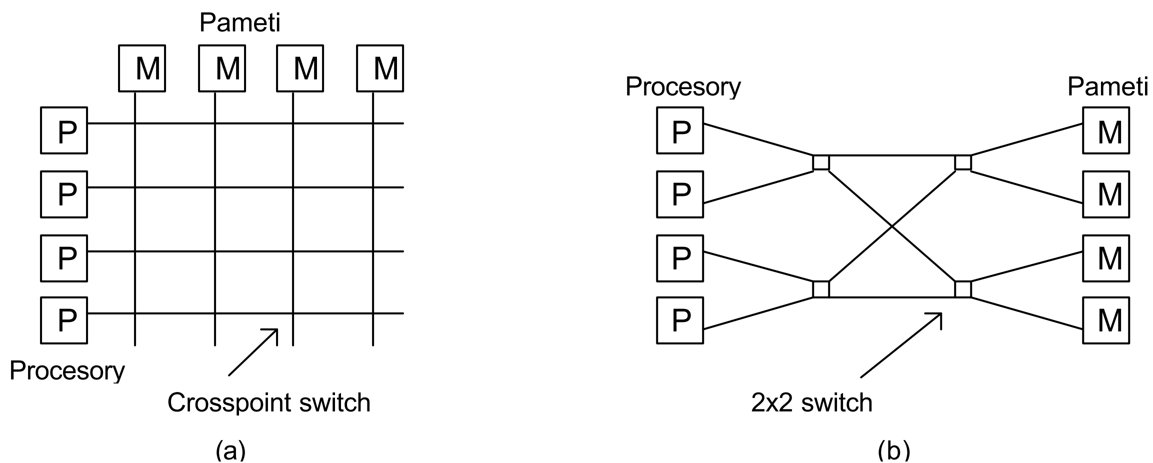
Obr. 1 - Taxonomie paralelních hardwarových architektur

Multiprocesory se sbernicovou architekturou jsou tvoreny nekolika procesory propojenými spolu s pametovým modulem spolecnou sbernicí. Tímto zpusobem je možné efektivne zapojit 32 až 64 procesoru.



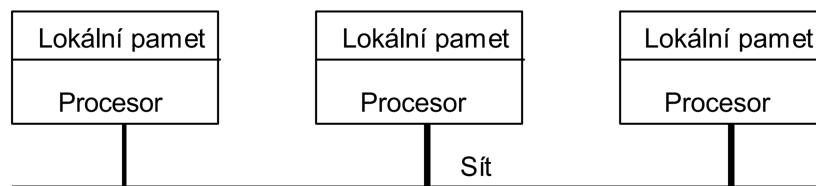
Obr. 2 - Multiprocesor se sbernicovou architekturou

Pro zapojení více než 64 procesoru je zapotrebí použít jinou architekturu. Jedno možné rešení je rozdelit pamet na nekolik pametových modulu a propojit je s procesory sbernicovou mrížkou. V prusecících mrížky jsou prepínace (**crosspoint switch**) urcující další cestu. Nevýhodou této architektury je kvadratický pocet prepínacu, což pri velkém poctu procesoru a pametových modulu muže být príliš nákladné. Druhou možností je použít sít typu omega (**omega network**), kde pocet prepínacu roste rychlostí n·log n. Nevýhodou tohoto usporádání je prodleva pri vetší hloubce prepínacu.



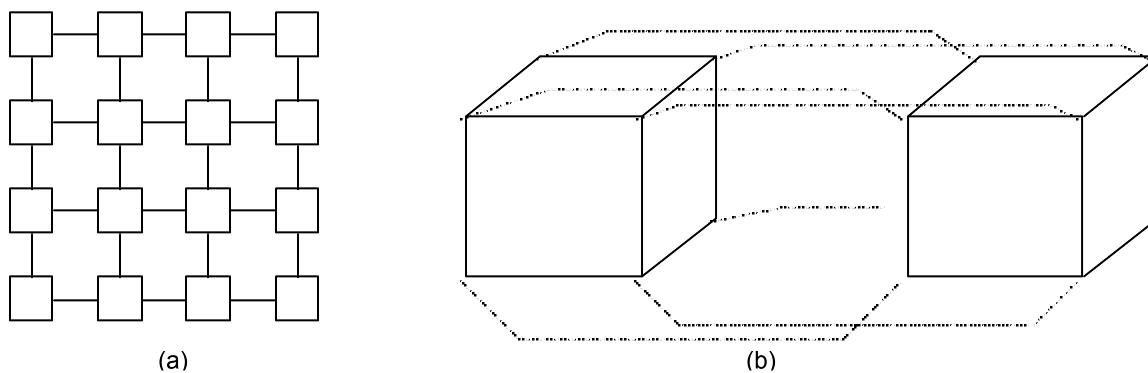
Obr. 3 - (a) Crossbar switch (b) Omega switch

Na rozdíl od rozsáhlých multiprocesoru je vytvorení multicomputeru mnohem méne nákladné. Každý procesor má prímý prístup do vlastní pameti. Vzhledem k tomu, že komunikují pouze procesory mezi sebou, je komunikace rádove nižší než u multiprocesoru. Na sbernici je proto možné umístit vetší množství procesoru (pracovních stanic).

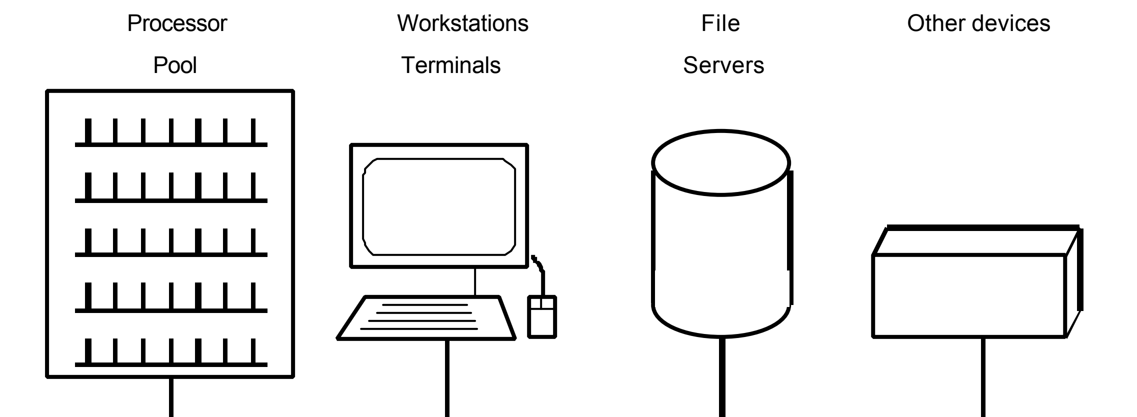


Obr. 4 - Multicomputer sestavený z pracovních stanic propojených sítí

Poslední možností jsou multicomputery s prepínacovou architekturou. Nejvíce používané topologie jsou **mrížka** a **hyperkrychle**. U mrížky je každý uzel propojen se ctyrmi svými sousedy. Jejich implementace je vzhledem k prirozené dvourozmernosti relativne jednoduchá. Multicomputery s mrížkovou topologií jsou vhodné zejména pro úlohy z oblasti teorie grafu, analýzu obrazu apod. Hyperkrychle je n-rozmerná krychle (podle této definice je mrížka 2-rozmerná hyperkrychle). Nejcasteji používané hyperkrychle jsou 4rozmerné, kde každý uzel obsahující procesor je propojen se dvema uzly v každém rozmeru, tj. celkem s osmi sousedními procesory. Pridáním každého dalšího rozmeru se pocet prímo propojených sousedu zvetší o 2. Bežne dostupné jsou hyperkrychle s 1024 procesory, v soucasné dobe se zacínají používat i hyperkrychle obsahující 16386 procesoru.



Obr. 5 - (a) Mrížka (b) Hyperkrychle



Obr. 6 - Komponenty distribuovaného systému

2. Meziprocesová komunikace

Asi nejvýraznejším rozdílem mezi jednoprocesorovým a distribuovanými systémy je meziprocesová komunikace (IPC). V klasickém jednoprocesorovém systému naprostá vetšina implementací IPC implicitne predpokládá existenci spolecné pameti. Typickým príkladem je problém producent - konzument, kde jeden proces zapisuje do spolecného bufferu a druhý proces z nej cte. Dokonce i nejzákladnejší forma synchronizace, semafor, vyžaduje pro svoji funkci jedno slovo spolecné pameti. Oproti tomu v distribuovaných systémech nelze predpokládat existenci spolecné pameti, tudíž veškeré formy meziprocesové komunikace a synchronizace musejí být založeny na jiném principu - na principu **zasílání zpráv**. Jestliže proces A chce komunikovat s procesem B, nejdríve vytvorí zprávu ve svém adresovém prostoru. Poté zavolá systémovou službu, která prevezme zprávu a pošle ji po síti pocítaci, na kterém beží proces B. Jakkoli je tato idea jednoduchá, je zapotrebí vyrešit mnoho problému, aby tato vzdálená komunikace fungovala a komunikující pocítace si vzájemne rozumely.

Pro usnadnení vzájemné komunikace mezi pocítaci ruzných výrobcu vytvorila Mezinárodní standardní organizace (ISO) sedmivrstevný referencní model, oznacovaný jako ISO/OSI (Open Systems Interconnection). Úlohou tohoto referencního modelu je vymezení jednotlivých vrstev a specifikace úkolu, které by mely tyto vrstvy rešit. Každá vrstva se zabývá jednou úrovní komunikace a poskytuje služby bezprostredne vyšší vrstve.

Aplikacní	Spolecné cásti sítových aplikací
Prezentacní	Konverze dat, komprese, šifrování
Relacní	Navazování, udržování a rušení relací
Transportní	Sestavování paketu, kvalitnejší služby
Sítová	Iluze všeobecné propojenosti
Linková	Prenos bloku dat mezi uzly
Fyzická	Prenos jednotlivých bitù

Tab. 6 - Referencní model ISO/OSI

Pro použití v distribuovaných systémech je však plná implementace protokolu na všech vrstvách príliš neefektivní. Vzhledem k tomu, že každá vrstva si pridává ke zpráve svoji vlastní hlavicku, je casová i prostorová režie pro každou zprávu príliš velká. Pri použití v rychlých lokálních sítích muže tato režie výrazne zpomalovat chod celého systému. Proto naprostá vetšina distribuovaných systému založených na lokálních sítích bud vubec nepoužívá takto detailne vrstvené protokoly, anebo implementuje pouze omezenou množinu protokolu.

2.1 Klient/server model

Komunikacní model v distribuovaných systémech by mel krome vlastního prenosu dat mezi jednotlivými pocítaci specifikovat i to, jakým zpusobem je celý systém strukturován. Toto reší v naprosté vetšine prípadu tzv. klient/server model, který je postaven na základní myšlence vytvorit celý operacní systém jako množinu spolupracujících procesu, nazývaných **servry**, které poskytují služby svým uživatelum, **klientum**. Pocítace, na kterých beží servrovské i klientské procesy, mají vetšinou stejné jádro systému (microkernel), pricemž jak klienti tak servry beží jako uživatelské procesy. V prípade, že je systém postaven na symetrickém klient/server modelu, mohou na libovolném pocítaci bežet zároven servrovské procesy i klienti.

2.1.1 Request/reply protokol

Aby se zabránilo príliš velké režii mnohovrstevných protokolu typu ISO/OSI, je vetšinou klient/server model založen na jednoduchém **request/reply protokolu** (žádost/odpoved). Klient pošle servru zprávu se žádostí o nejakou službu (request message). Server provede požadovanou cinnost a pošle zpet zprávu (reply message), která obsahuje prípadná požadovaná data a návratový kód, který specifikuje, zda požadovaná operace byla provedena a v prípade, že ne, i duvod neprovedení.

Hlavní výhodou tohoto protokolu je jeho jednoduchost. Klient pošle zprávu, dostane odpoved a to je vše. Není zapotrebí navazovat a rušit spojení, není treba vykonávat jiné vedlejší akce, nejsou zapotrebí vyšší vrstvy protokolu. Další výhodou je efektivita - soustava protokolu je kratší, typicky jsou implementovány 3 až 4 vrstvy. Fyzická a linková vrstva starající se o prenos paketu od klienta k servru a zpet je vetšinou implementována v sítové, napr. ethernetové karte. Na úrovni transportní vrstvy bývá request/reply protokol, který definuje množinu platných požadavku a množinu možných odpovedí. Další vrstvy vetšinou nejsou zapotrebí.

Díky této jednoduché strukture mohou být komunikacní služby poskytované jádrem systému redukovány na dve systémová volání - jedno pro posílání zpráv, druhé pro prijímání. Tyto systémové služby mohou být vyvolány prostrednictvím knihovních procedur `send` a `receive`. První procedura vezme pripravenou zprávu a pošle ji urcenému procesu, druhá procedura zpusobí zablokování procesu do té doby, než dorazí nejaká zpráva.

```
main()
{
    struct message m1, m2;

    for (;;)
    {
        receive (MY_SERVER_NUMBER, &m1);
        switch (m1.opcode)
        {
            case SERVICE1:do_something (&m1, &m2);      break;
            case SERVICE2:do_something_else (&m1, &m2);break;
            case SERVICE3:do_another_service (&m1, &m2);break;
            default:       m2.retval = ERR_BAD_OPCODE;
        }
        send (m1.client, &m2);
    }
}
```

Zjednodušený príklad servru

```
any_function()
{
      struct message m;

      ...
      m.opcode = SOME_OPCODE;
      m.arg1 = some_argument;
      m.arg2 = another_argument;
      send (MY_SERVER, &m);
      receive (MY_CLIENT_NUMBER, &m);
      if (m.retval == OK)
            read_data (&m.data);
      else
            error_handler (m.retval);
      ...
}
```

Zjednodušený príklad klienta

2.1.2 Struktura zpráv

Hlavicka, strukturovaná/nestrukturovaná data

Adresát
Odesilatel
Identifikace zprávy
Identifikace klienta
Císlo služby
Císlo objektu
Typ dat
Pocet elementu
Data
Typ dat
Pocet elementu
Data
...

Adresát
Odesilatel
Identifikace zprávy
Identifikace klienta
Císlo služby
Císlo objektu
Velikost dat
Data

Tab. 7 - Príklad formátu strukturované a nestrukturované zprávy

2.1.3 Synchronní a asynchronní komunikace

Výše popsaná komunikacní primitiva nazýváme **synchronní primitiva** (blocking primitives, synchronous primitives). Když proces volá proceduru `send`, specifikuje cílový proces a buffer, který má být poslán. Po dobu, co se prenáší zpráva, je proces blokován (suspendován). Instrukce následující po volání procedury nejsou vykonávány až do té doby, dokud zpráva nedorazí k príjemci. Podobne `receive` zablokuje proces do té doby, než dorazí ocekávaná zpráva.

Alternativní formou komunikace jsou **asynchronní primitiva** (nonblocking primitives, asynchronous primitives). V tomto prípade `send` vrátí kontrolu procesu ihned bez cekání na odeslání zprávy. Výhodou tohoto schématu je vyšší stupen paralelismu - proces muže pokracovat ve svých

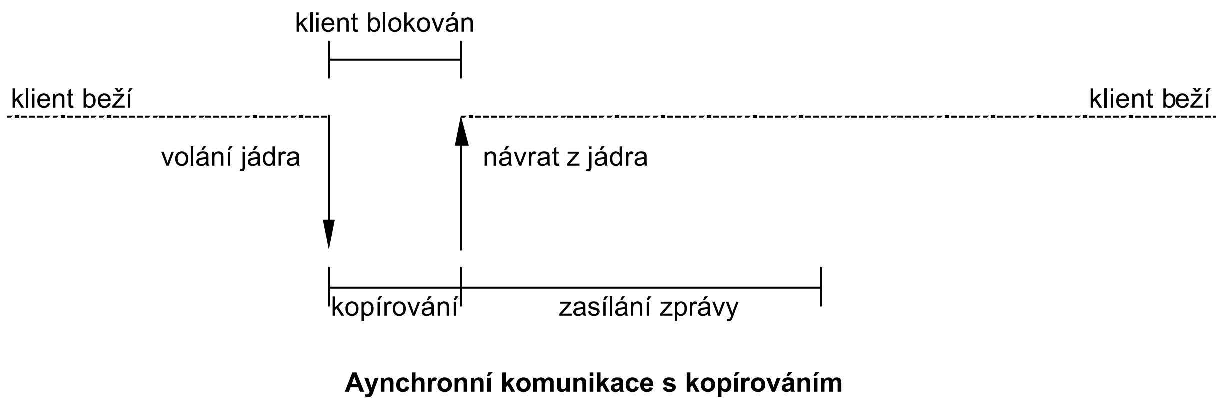
výpoctech po dobu prenosu zprávy. Avšak tato výhoda je vykoupená jednou vážnou nevýhodou - proces nemuže zmenit buffer se zprávou dokud zpráva není odeslána. Prepsání cásti zprávy behem prenosu by melo nepredvídatelné následky. Navíc proces nemá ani ponetí o tom, kdy byl prenos dokoncen, tudíž nikdy neví, kdy lze bezpecne použít buffer pro jiné úcely.

Existují dva zpusoby rešení. První zpusob spocívá v tom, že si jádro okopíruje zprávu do svých vnitrních bufferu, címž umožní procesu další bezpecný beh. Nevýhodou tohoto rešení je kopírování každé zprávy do vnitrních bufferu jádra, což pri velkém poctu zpráv muže zpusobit výrazné snížení výkonu celého systému.

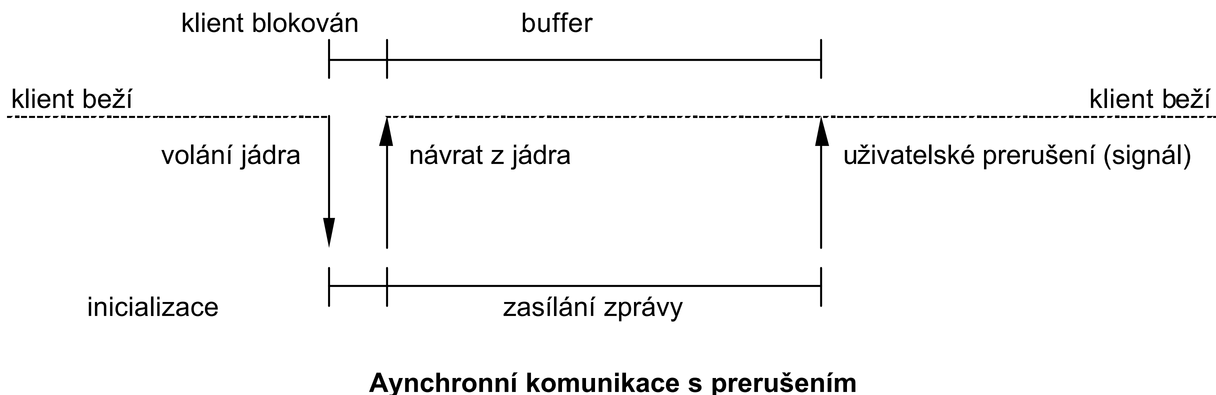
Druhé rešení je založeno na myšlence prerušení procesu poté, co byla zpráva úspešne odeslána. Toto rešení sice nevyžaduje žádné kopírování, nicméne programy založené na uživatelských prerušeních jsou ponekud neprehledné, obtížné na odladení a plné kritických sekcí, címž se stávají velice težko udržovatelné a rozširitelné.

Stejne jako `send` muže být i `receive` synchronní nebo asynchronní. Asynchronní `receive` pouze ríká jádru, kam má umístit došlou zprávu. Zde nastává stejný problém - jak se proces dozví, že nejaká zpráva došla. Možností je nekolik. První možností je zavedení explicitního primitiva `wait`, které pozastaví proces do té doby, než zpráva dorazí. Další možností je zavedení primitiva `test`, které otestuje, zda je nejaká zpráva pripravena, nebo `conditional_retrieve`, které bud vybere došlou zprávu nebo se vrátí s informací, že žádná zpráva není k dispozici. Poslední možností je opet zavedení uživatelského prerušení, které bude informovat proces o došlé zpráve.

Synchronní komunikace



Aynchronní komunikace s kopírováním



Aynchronní komunikace s prerušením

Obr. 7 - Synchronní a asynchronní komunikacní primitiva

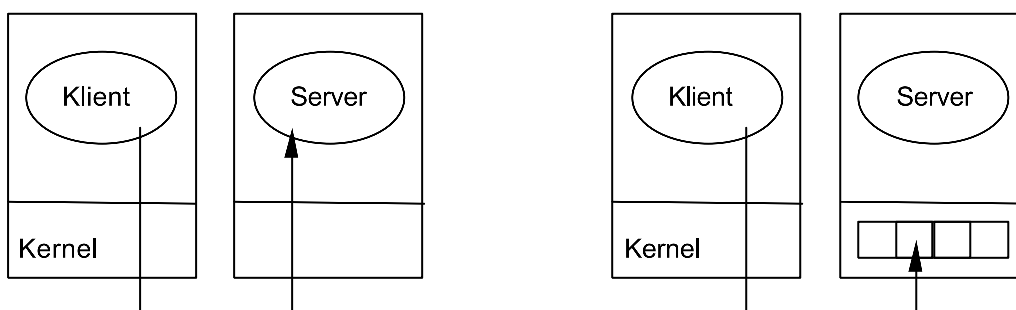
2.1.4 Prímé zasílání zpráv vs. zasílání zpráv pres schránku

Krome volby mezi synchronní a asynchronní komunikací mají tvurci distribuovaných systému možnost zvolit i zpusob, jak bude zpráva dorucena príjemci. Výše popsaná primitiva jsou príkladem prímého zasílání zpráv (unbuffered primitives). Pri tomto zpusobu komunikace prijímající proces sdelí systému, kam do adresového prostoru príjemce má došlou zprávu umístit. Tímto zpusobem funguje komunikace spolehlive tak dlouho, dokud prijímající proces stíhá zavolat `receive` pred tím,

než libovolný proces zavolá `send`. Problém nastane, když nejaký proces posílá zprávu, ale príjemce ješte nestihl zavolat `receive`. Jádro v tomto prípade neví, kam má došlou zprávu uložit.

Jedno možné rešení je došlou zprávu zahodit, informovat odesílatele o tom, že server ješte není pripraven a doufat, že pred tím, než dojde další zpráva, bude príjemce pripraven. Toto rešení je snadné na implementaci, avšak klient, resp. vysílající jádro, muže zprávu posílat nekolikrát, než je prijata. Dále je možné, že jádro po nekolika neúspešných pokusech o spojení oznací server chybne za nefunkcní nebo nedostupný.

Druhý prístup k tomuto problému je vytvorení bufferu v jádru systému - **schránky** (mailbox) - kam budou umistovány neocekávané zprávy. V prípade, že príjemce v dostatecne krátkém intervalu zavolá `receive`, obdrží zprávu ze schránky. Jestliže se príjemce neozve príliš dlouhou dobu, jádro zprávu znicí. Toto na první pohled elegantní rešení však problém nereší, pouze jej oddaluje. Schránky jsou totiž konecné a muže dojít k jejich preplnení. V tom prípade se jádro musí rozhodnout, zda novou zprávu prijmout a zahodit nejakou starší, anebo nechat schránku nedotcenou a novou zprávu odmítnout. V nekterých systémech je možné v prípade preplnení príjemcovy schránky zablokovat vysílající proces do té doby, než se schránka uvolní.



Obr. 8 - (a) Prímé zasílání zpráv (b) Zasílání zpráv pres schránku

2.1.5 Prímé a neprímé adresování - linky a porty

Meziprocesovou komunikaci je možné též delit podle toho, kam je zpráva zasílána. V prípade, že odesílatel specifikuje jako príjemce prímo identifikaci prijímajícího procesu, jde o komunikaci prímou. Druhá technika - neprímé adresování - využívá nejaké presne urcené místo, **port**, odkud príjemce zprávu dostane.

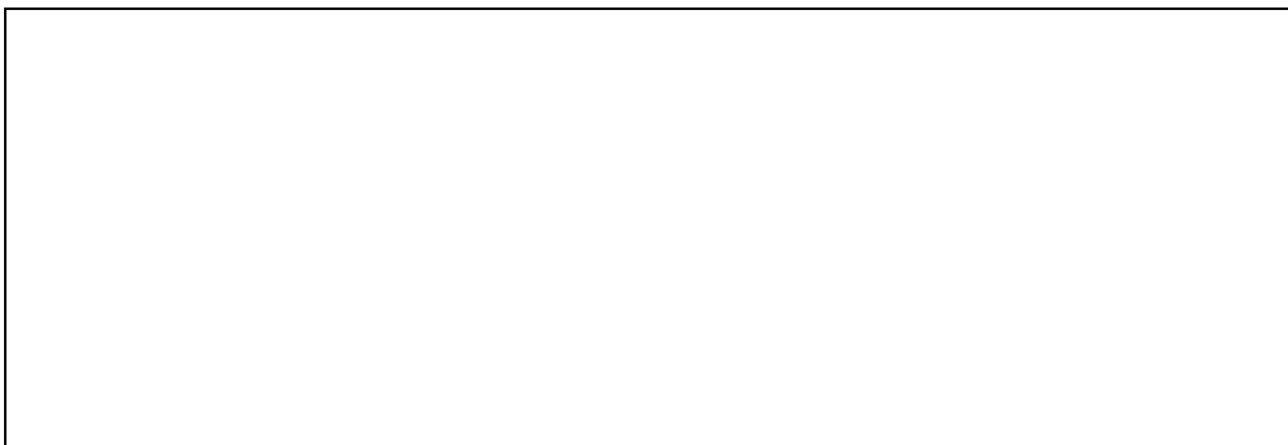
V prípade prímého adresování odesílající proces specifikuje jako príjemce proces, který zprávu dostane. Mezi procesy se vytvorí virtuální spojení, tzv. **link**. Spojení je automaticky vytvoreno mezi každou dvojicí komunikujících procesu, každé spojení je asociováno práve se dvema procesy.

Technika neprímého adresování využívá pro dorucení zprávy tzv. **porty**. Port je jednoznacne identifikovatelný objekt jádra, který umožnuje ukládání a vybírání zpráv. Vysílající proces zašle zprávu nejakému portu, prijímající proces zprávu z portu dostane. K jednomu portu muže být prihlášeno více príjemcu, mezi dvema komunikujícími procesy muže být navázáno spojení pres nekolik portu. Z implementacního hlediska je port fronta zpráv konecné délky, avšak nekteré "naléhavé" zprávy mohou frontu preskocit.

2.2 Spolehlivost komunikace

Doposud jsme predpokládali, že vlastní komunikace je spolehlivá, tj. když klient vyšle zprávu, tak ji server dostane, a to v nezmenené podobe. Bohužel tomu tak není - zprávy se mohou po ceste

ztrácet, muže se zmenit jejich poradí apod. Proto je nutné pro zabezpecení spolehlivé komunikace zajistit potvrzování príjmu zpráv nebo paketu.



Obr. 9 – Potvrzování zpráv

```
reply jako ack
potvrzování každého paketu / zprávy / dávky
plovoucí dávky
are you alive
vícenásobné pakety
timeouts
```

Kód	Typ zprávy	Odesílatel	Príjemce	Význam
REQ	Request	Klient	Server	Klient požaduje službu
REP	Reply	Server	Klient	Odpoved servru na žádost klienta
ACK	Ack	kdokoliv		Potvrzení príjmu paketu
NAK	Negative ack	kdokoliv		Oznámení o neprijmutí paketu
AYA	Are you alive?	Klient	Server	Žádost o potvrzení servru, zda je funkcní
IAA	I am alive	Server	Klient	Potvrzení funkcnosti servru
TRA	Try again	Server	Klient	Server nemá místo pro prijmutí zprávy

Tab. 8 - Základní typy paketu používané v klient/server modelu

2.2.1 Spolehlivost serveru

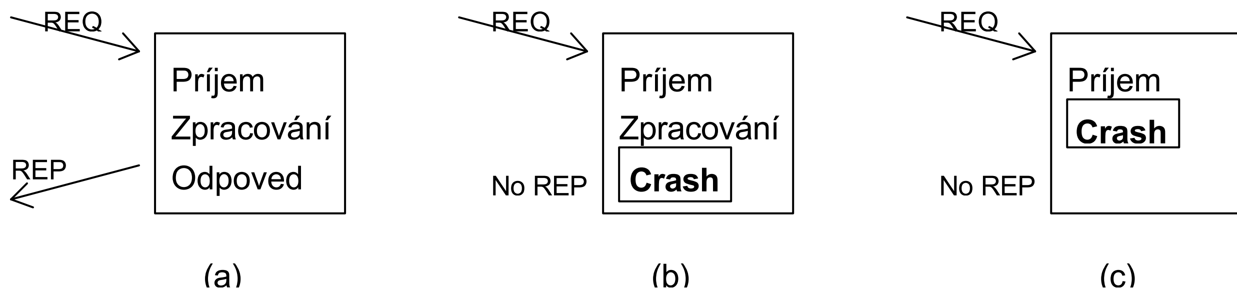
Další problém komunikace v distribuovaném prostredí je spolehlivost, resp. nespolehlivost samotných pocítacu, a to jak na strane servru, tak na strane klienta. Predstavme si situaci, kdy klient odešle žádost, ale odpoved delší dobu neprichází. Duvodu muže být nekolik:

? ztratila se zpráva se žádostí

? ztratila se zpráva s odpovedí

? server je príliš pomalý nebo zahlcen vetším poctem žádostí natolik, že ješte zpracovává žádost klienta

? server nefunguje



Obr. 10 – Havárie serveru

Nekteré operace mohou být bezpecne nekolikrát zopakovány, aniž by došlo knejaké škode. Napr. ctení prvních n bajtu daného souboru muže být provedeno nekolikrát, aniž by došlo k nejakým postranním efektum. Žádosti, resp. služby, splnující tuto vlastnost se nazývají **idempotentní**. Pokud se zasílá nejaká idempotentní žádost a nedošla odpoved, lze žádost opakovat. Ne všechny žádosti však mohou být idempotentní - napr. žádost o vybrání milionu dolaru z banky rozhodne idempotentní není. Proto je nutné komunikaci zabezpecit tak, aby i tyto žádosti byly vyrizovány korektne.

První metodou je prirazení jednotlivým zprávám jednoznacné identifikacní císlo. Server tím pádem muže rozeznat originální žádost od její kopie a v prípade prijmutí kopie zpracovávané žádosti tuto kopii ignorovat. Podobným zpusobem lze rešit i ztracené zprávy s odpovedí. Dokud nedojde od klienta potvrzení o prijmutí odpovedi, server si udržuje seznam vyrízených žádostí a odeslaných odpovedí a na prípadnou urgenci muže reagovat opetovným odesláním odpovedi.

Nejsložitejší situace nastane v prípade, že prestane fungovat server. V prípade, že nefungoval ani predtím, než zpráva došla, pak ho po nekolika pokusech o navázání komunikace prohlásí klient za nedostupný. Horší situace je, když server prestane fungovat poté, co zprávu prijal. Problém spocívá v tom, že obecne nelze spolehlive urcit, zda byl požadavek zpracován ci ne.

Existustují principiálne dva zpusoby rešení této situace. První technika, tzv. **at-least-once** sémantika ("alespon jednou"), zarucuje, že požadavek (pokud to bude možné) bude zpracován, tj. po znovunastartování servru bude požadavek opakován anebo bude predán jinému servru zajištujícímu stejné služby. Druhá technika, tzv. **at-most-once** sémantika ("nejvýše jednou"), zarucuje, že požadavek nebude zpracován více než jednou.

Žádná z techto možností není príliš pritažlivá, ideální by byla sémantika typu **exactly once** ("práve jednou"), to však obecne nelze zarucit. Predstavme si tiskový server, který práve zpracovává požadavek na vytištení nejakého textu a pritom zhavaruje. Havárie mohla nastat mikrosekundu pred vlastním tiskem i mikrosekundu po tisku, nikdo však nedokáže rozhodnout, který z techto prípadu nastal.

2.2.2 Spolehlivost klienta

```
Havárie klienta - orphani (sirotci)
 - exterminace - klient si pamatuje co rozbehl a zabije to
 - reinkarnace - klient po bootu nastartuje novou epochu
 - expirace - ex. kvantum, server si sám ríká o další kvantum
```

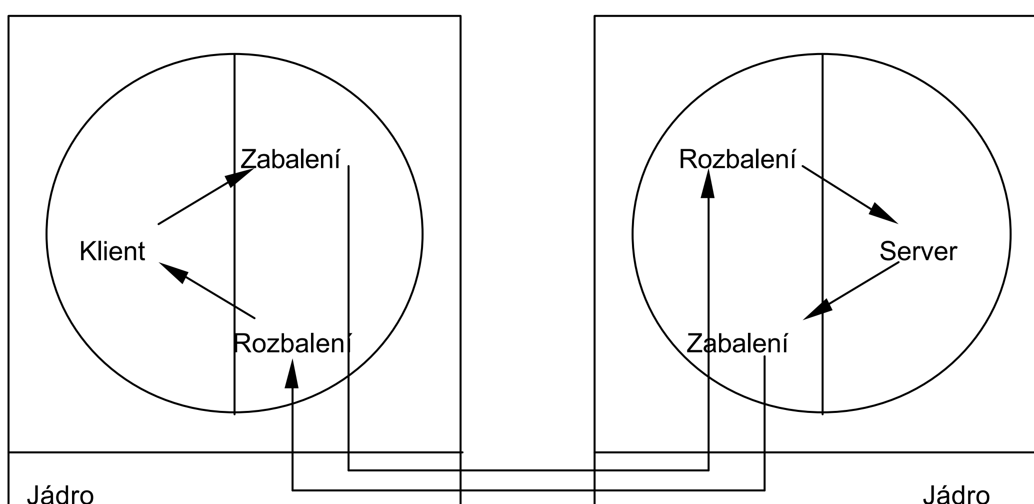
2.2.3 Vztah komunikace a virtuální pameti

```
Lokální zprávy:
 - krátké (<16B) - v registrech
 - strední (<4KB) - komunikacní stránka
 - dlouhé - mapování
```

```
Vzdálené zprávy:
- krátké a strední zprávy - OK
- dlouhé: problém efektivity: kopírování "prázdných" stránek, vzdálené
sdílení, ...
```

2.3 Remote procedure call

Vzdálené volání procedur, **RPC** (Remote Procedure Call), je prístup k meziprocesové komunikaci založený na základní myšlence lokálních programu - volání procedur. Idea RPC je založena na pozorování, že model *"klient zašle zprávu servru, poté se zablokuje a ceká na odpoved od servru"* velmi presne odpovídá dobre známému mechanismu volání procedur. Proto cílem RPC je umožnit, aby distribuované programy mohly být psány zcela stejným stylem jako konvencní programy pro centralizované operacní systémy.



Obr. 11 - Remote procedure call

1. Klientova funkce normálním zpusobem zavolá klientský stub
2. Stub vytvorí zprávu a zavolá jádro
3. Jádro pošle zprávu jádru pocítace, kde beží server
4. Vzdálené jádro predá zprávu servrovu stubu
5. Stub rozbalí parametry a zavolá server
6. Server zpracuje požadavky a normálním zpusobem se vrátí do stubu
7. Servruv stub zabalí výstupní parametry do zprávy a zavolá jádro
8. Jádro pošle zprávu klientovu jádru
9. Klientovo jádro predá zprávu stubu
10. Stub rozbalí výstupní parametry a vrátí se ke klientovi

Problémy RPC:

? reprezentace dat - indiáni, floaty, ...

? prístup ke globálním promenným
? predávání pointru, dynamické struktury
? copy/restore - velikosti polí
? variabilní parametry - printf
? group communication
? komunikacní chyby - odlišná sémantika

3. Skupinová komunikace

3.1 Unicast, broadcast & multicast

Doposud jsme predpokládali, že jeden proces posílá zprávu jinému procesu - tzv. **unicast**. V distribuovaných systémech však casto nastanou situace, kdy toto nejjednodušší schéma komunikace nepostacuje. Casto je zapotrebí zaslat nejakou zprávu potenciálne všem propojeným pocítacum - **broadcast**. V tomto prípade vysílající proces nemusí uvádet adresu príjemce - zpráva dojde všem uzlum zapojeným do systému. Zpusob komunikace pomocí broadcastu je používán napr. pri hledání nejakého servru nebo pro informování všech uzlu o nejaké výjimecné události.

V nekterých aplikacích je zapotrebí tzv. **multicastu**, který umožnuje príjem pouze vybrané podmnožine propojených uzlu. Na rozdíl od broadcastu zde vysílající proces je schopen nejakým zpusobem identifikovat množinu príjemcu. Existují principiálne tri možnosti ocekávání odpovedi:

1. **Bez odpovedi** - Vysílající proces neocekává žádnou odpoved, prijímající proces nemusí odpovídat.

2. **Alespon jedna odpoved** - Vysílající proces ocekává alespon jednu odpoved. Pro úspešné zakoncení komunikace stací jedna došlá a potvrzená zpráva.

3. **Všechny odpovedi** - Vysílající proces ocekává odpovedi od všech príjemcu. Tento zpusob komunikace je vhodný pro implementaci spolehlivé skupinové komunikace.

3.2 Pracovní skupiny

Pracovní skupina je množina procesu, které v nejakém systému spolu spolupracují. Klícová vlastnost skupiny je komunikace - jestliže je nejaká zpráva zaslána skupine jako celku, pak ji dostane každý clen této skupiny. Na rozdíl od jednoduché komunikace, kde spolu komunikují pouze dva procesy, skupinová komunikace je typu one-to-many, tj. jeden proces je vysílající a jeden nebo více procesu jsou príjemci.

Pracovní skupiny jsou dynamické struktury. Mohou být vytváreny nové skupiny, staré skupiny mohou zanikat, proces muže být zaclenen do již existující skupiny anebo ze skupiny odstranen. Proces muže být zároven i ve více skupinách. Duvodem pro vytvárení skupin je potreba zacházet s množinou procesu jako s nedelitelným celkem. Napr. libovolný proces muže komunikovat se skupinou servru aniž by znal jejich aktuální pocet nebo umístení.

Implementace skupinové komunikace závisí na použitém sítovém hardware. Nejjednodušší zpusob je použít multicasting. Bohužel ne všechen sítový hardware tuto službu poskytuje. Druhou možností je použít broadcast. Ty uzly, které do dané skupiny nepatrí, budou zprávu ignorovat. Nevýhodou rešení pomocí broadcastu je zbytecné zatežování uzlu, které nejsou ve skupine. Nejméne efektivní, ale v prípade absence multicastu ci broadcastu jedinou možnou, technikou je použití unicastu.

Skupiny mohou být bud uzavrené nebo otevrené. V systémech s uzavrenými skupinami mohou zasílat správy skupine jen ty procesy, které jsou jejími cleny. Procesy, které cleny skupiny nejsou, jí zasílat zprávy nemohou. V systémech s otevrenými skupinami mohou zprávy skupine zasílat jakékoliv procesy. Uzavrené skupiny bývají vetšinou používány pro paralelní výpocty. Napr. množina procesu rešící šachovou úlohu muže být zorganizována jako uzavrená skupina. Otevrené skupiny bývají vetšinou používány pro replikované nebo jinak spolupracující servry.

Dalším hlediskem, podle kterého mohou být pracovní skupiny rozdeleny, je vnitrní organizace skupiny. V nekterých skupinách si jsou všechny procesy rovnocenné. Každý proces muže zasílat zprávy ostatním, rozhodnutí jsou prijímána kolektivne. Jiný typ organizace skupiny je hierarchická organizace. V tomto modelu jsou všechny žádosti smerovány na koordinátora, který rozhodne, který proces se žádosti ujme. Tímto zpusobem lze vytváret hierarchické stromy servru.

Aby zprávy mohly být skupine skutecne zaslány, je zapotrebí, aby každá skupina byla nejakým zpusobem adresována. Jednou metodou adresování je priradit každé skupine jednoznacnou adresu. Jestliže použitá sít podporuje multicast, pak skupinová adresa muže být asociována s multicastovou adresou a každá zpráva skupine je pak tvorena jedním multicastem. V prípade implementace pomocí broadcastu každé jádro obdrží zprávu, zjistí z ní adresu skupiny a v prípade, že zpráva je urcena nejakému lokálnímu procesu, ji prijme. Konecne když neexistuje ani multicast ani broadcast, pak nezbývá nic jiného, než zaslat unicasty všem clenum skupiny. V tomto prípade bud musí vysílající jádro znát všechny cleny skupiny, anebo být schopno si tyto cleny zjistit.

Druhou možností adresování je uvádet prímo v hlavicce zprávy explicitní seznam príjemcu. Tento prístup má tu nevýhodu, že vysílající proces musí v každém prípade presne znát všechny cleny skupiny a jejich adresy. Dále, pokud se skupina nejakým zpusobem zmení, pak každý proces zasílající zprávu skupine, musí zmenit svuj seznam clenu.

Tretí možnou metodou adresování je tzv. **predikátová adresace**. Pri použití tohoto zpusobu adresace je zpráva adresovaná libovolným výše uvedeným zpusobem doplnena o predikát (logický výraz). Jádro po prijetí zprávy vyhodnotí tento výraz a zprávu prijme jen v prípade, že výraz má pravdivou hodnotu. Tímto zpusobem lze napr. zaslat zprávu jen tem pocítacum, které mají více než 16MB pameti apod.

3.3 Atomicita

Charakteristikou skupinové komunikace je vlastnost, která by se dala nazvat "**všechno nebo nic**". Vetšina systému, které podporují skupinovou komunikaci, je implementována tak, že když je skupine zaslána zpráva, pak bud všechny procesy korektne zprávu dostanou anebo zprávu nedostane žádný proces skupiny. Situace, kdy nekteré procesy zprávu dostanou a jiné ne, není prípustná. Tato vlastnost se nazývá **atomicita**, nebo také **atomický broadcast**.

Atomicita je velmi výhodná pro psaní distribuovaných programu. Když jakýkoliv proces zašle zprávu nejaké skupine, nemusí se již starat o to, kdo zprávu dostal a kdo ne. Napríklad v replikovaném databázovém systému by fakt, že nekteré procesy neobdržely zprávu o zmene stavu dat, mohl zpusobit nekonzistenci celé distribuované databáze.

Aby bylo zabezpeceno, že vysílaná zpráva dorazí ke všem príjemcum a ti ji akceptují, je zapotrebí zasílání potvrzení - jádro každého uzlu, který zprávu prijme, potvrdí vysílajícímu jádru príjem. V prípade pretížení nekterých uzlu nebo lokální výpadku nebo pretížení síte vysílající jádro zopakuje zprávu. Tato metoda funguje ovšem jen za predpokladu, že žádný z uzlu, at už vysílající nebo prijímající, nezhavaruje.

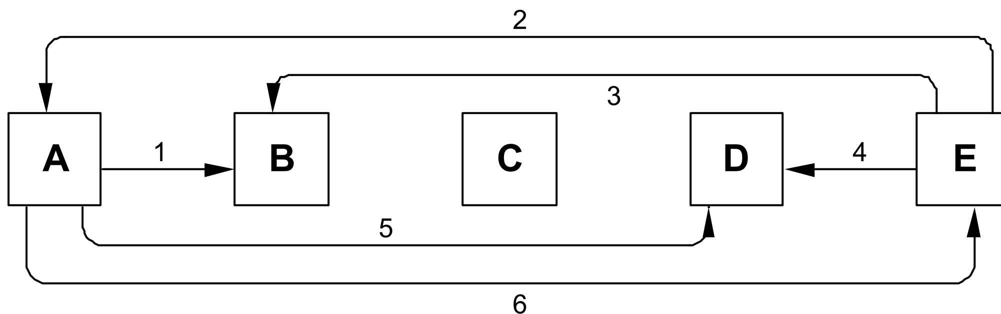
Ve vetšine distribuovaných systému je však zapotrebí, aby dodržely podmínku odolnosti proti poruchám (**fault tolerance**), tj. aby atomicita byla zajištena i v prípade výpadku nekterých uzlu. Výše uvedený mechanismus však tuto podmínku nezarucuje - v prípade odmítnutí zprávy príjemcem z duvodu pretížení a následné havárie odesílatele príjemce zprávu nedostane, což je presne nechtená situace - nekterí clenové skupiny zprávu dostali, zatímco jiní nikoliv.

Naštestí existuje algoritmus, který demonstruje prinejmenším možnost atomického broadcastu. Odesílatel zašle zprávu všem clenum skupiny, nastaví si casovace a v prípade nutnosti nekteré zprávy zopakuje. Když príjemce dostane zprávu, kterou ješte nedostal, opet ji rozešle všem clenum skupiny (s prípadným zopakováním nekterých zpráv). V prípade príjmu již prijaté zprávy príjemce nedelá nic a zpráva je ignorována. Tímto zpusobem se zabezpecí, že všichni clenové skupiny dostanou zprávu

bez ohledu na to, kolik uzlu mezitím zhavaruje a kolik zpráv se po síti ztratí. Existují efektivnejší algoritmy pro zabezpecení atomicity, ty budou uvedeny pozdeji.

3.4 Usporádání zpráv

Druhou podmínkou správného fungování skupinové komunikace je, aby zprávy docházely všem clenum skupiny ve stejném poradí. Na obrázku je znázornena situace, ke které by nemelo dojít. Procesy A a E se zhruba ve stejný okamžik rozhodly poslat zprávu skupine obsahující procesy A, B, D a E. Císla udávají poradí, ve kterém jsou jednotlivé zprávy dorucovány. Procesu B došla jako první zpráva od procesu A (c. 1) a jako druhá zpráva od procesu E (c. 3). Naopak procesu D došla jako první zpráva od procesu E (c. 4) a jako druhá zpráva od procesu A (c. 5).



Obr. 12 - Konkurencní zasílání zpráv s rozdílným usporádáním príjmu

Nejlepším zpusobem je zabezpecit, aby zprávy byly doruceny presne v tom poradí, v jakém byly vyslány. Jestliže se tedy proces rozhodne poslat zprávu nejaké skupine, pak zprávy od všech procesu, které se rozhodnou pozdeji, budou doruceny až po dorucení první zprávy. V tomto prípade by všichni príjemci obdrželi zprávy ve stejném poradí, ríkáme, že zprávy splnují **globální casové usporádání**.

Jelikož toto absolutní usporádání není vždy snadné nebo dokonce možné implementovat, vetšina systému nabízí slabší variantu - **konzistentní casové usporádání**. Jestliže dva procesy se v približne stejném case rozhodnou zaslat zprávu, systém vybere jednoho z nich jako prvního a dorucí jeho zprávu jako první. V tomto prípade se muže stát, že jako první bude zvolena zpráva, která ve skutecnosti nebyla první, avšak jelikož to stejne nikdo nemuže zjistit, chování systému by na tom nemelo záviset. Je tedy zaruceno, že zprávy budou doruceny všem clenum skupiny ve stejném poradí, avšak toto poradí nemusí být v nekterých prípadech shodné s reálným poradím, jak byly odeslány.

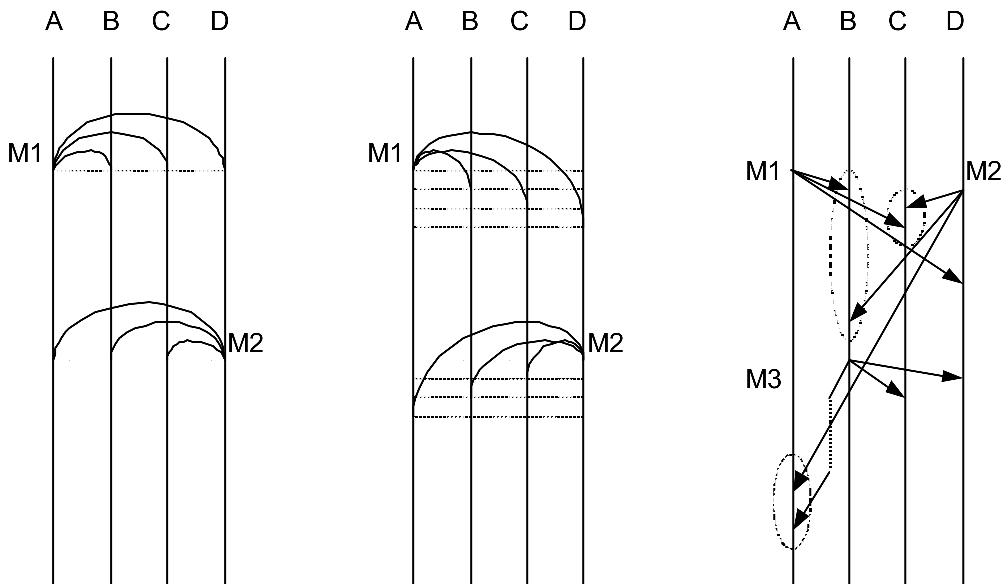
Skupinová komunikace muže být založena na pojmech kauzalita a synchronie. **Synchronní systém** je systém, ve kterém se události dejí striktne sekvencne, tj. jakákoliv událost (napr. zaslání zprávy) potrebuje ke svému dokoncení nulový cas. Zpráva tedy dorazí ke všem príjemcum ve stejný cas, kdy byla odeslána. Z hlediska vnejšího pozorovatele je systém jeví jako množina diskrétních událostí, které se v case neprekrývají.

Z technických duvodu je nemožné vytvorit plne synchronní systém, proto existují volnejší formy synchronie. **Slabe synchronní systém** je takový systém, ve kterém události trvají konecný predem neznámý cas, avšak všechny události se jeví všem procesum ve stejném poradí. Slabe synchronní systémy splnují podmínku konzistentního casového usporádání.

V nekterých systémech muže být z duvodu výkonnosti systému vhodnejší ješte volnejší typy synchronie. **Virtuálne synchronní systém** je systém, v kterém nekteré zprávy za urcitých presne definovaných podmínek nemusí dojít ruzným príjemcum ve stejném poradí. Ríkáme, že dve události jsou **kauzálne vázané**, když chování druhé události muže být jakýmkoliv zpusobem ovlivneno první událostí. Když proces A zašle zprávu procesu B, ten zprávu prijme a pak pošle zprávu procesu C, je druhá zpráva kauzálne vázaná s první zprávou, protože její obsah muže být ovlivnen obsahem první

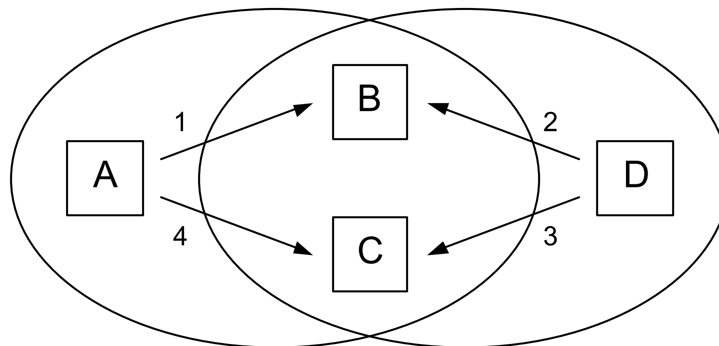
zprávy. Jestli je skutecne druhá zpráva ovlivnena nebo ne je irelevantní, podstatné je, že ovlivnena být mohla. Dve události, které nejsou vázané, se nazývají **konkurentní**. Když proces A zašle zprávu procesu B a zhruba v témže okamžiku proces C zašle zprávu procesu D, pak tyto zprávy jsou konkurentní, nebot se v žádném prípade nemohou ovlivnit.

Virtuální synchronie spocívá v tom, že kauzálne vázané zprávy jsou doruceny ve stejném (správném) poradí. Jestliže jsou zprávy konkurentní, pak jejich poradí dorucení muže být pro ruzné procesy ruzné.



Obr. 13 - (a) Synchronní systém (b) Slabá synchronie (c) Virtuální synchronie

Jak již bylo zmíneno dríve, jednotlivé procesy mohou být cleny více skupin. To muže vést k dalšímu druhu nekonzistence. Na obrázku je znázornena situace, kdy v jedné skupine jsou procesy A, B a C, v druhé skupine jsou procesy B, C a D. Jestliže zprávy dorazí v tom poradí, jak je znázorneno, pak prestože v rámci jednotlivých skupin jsou zprávy doruceny ve stejném poradí, procesy B a C dostanou zprávy urcené pro ruzné skupiny v odlišném poradí.



Obr. 14 - Rozdílné usporádání zpráv v prolínajících se skupinách

Globální i konzistentní casové usporádání dorucování zpráv v rámci jedné skupiny nezarucuje žádnou koordinaci mezi jednotlivými skupinami. Nekteré systémy umožnují dobre definované casové usporádání dorucování zpráv mezi prolínajícími se skupinami, zatímco jiné nikoliv.

3.5 Dorucovací protokoly

3.5.1 Kauzální závislost

Necht $e_1 \ ?\ ^p\ e_2$ je usporádání událostí v rámci procesu p, send(m) resp. rcv(m) je odeslání resp. príjem zprávy m. *Kauzální závislost* znacená ? je relace definovaná takto:

1. jestliže ? p: $e_1 \ ?\ ^p\ e_2$, potom e_1 ? e_2
2. ? m: send(m) ? rcv(m)
3. jestliže e_1 ? e_2 & e_2 ? e_3, potom e_1 ? e_3

Definice ríká, že událost e_3 kauzálne závisí na e_1 (e_1 ? e_3), jestliže existuje orientovaná cesta (složená ze šipek ? ^{p} nebo ?) od události k e_1 události e_3.

3.5.2 Kauzální usporádání dorucovaných zpráv

Kauzální usporádání dorucovaných zpráv definujeme takto:

Necht dest(m) je množina procesu ve skupine, do které je zaslána zpráva m, $deliver_p(m)$ je událost dorucení zprávy m procesu p. Pak

m_1 ? m_2 ? ? p ? (dest(m_1) ? dest(m_2)) : $deliver_p(m_1)$? ^{p} $deliver_p(m_2)$

Význam definice: Jestliže m_2 je zpráva kauzálne závislá na m_1, potom ve všech procesech, kterým budou doruceny obe tyto zprávy bude zachováno poradí tohoto dorucení.

3.5.3 Vektorové hodiny

Casová znacka (vektorové hodiny) procesu p_i znacená VT(p_i) je vektor délky n, kde n je pocet procesu ve skupine. Casová znacka zprávy m je znacena VT(m). Pro práci s casovými znackami platí následující pravidla:

1. pri startu p_i je VT(p_i) nulový
2. ? $send_{pi}(m)$: VT(p_i)[i]++, VT(m) = VT(p_i)
3. proces p_j, který dorucuje zprávu m obsahující VT(m), upraví VT(p_j) takto: ? k ? 1..n: VT(p_j)[k] = max(VT(p_j)[k], VT(m)[k])
4. ? m_i, m_j (i ? j): VT(m_i) ? VT(m_j)

Porovnávání casových znacek:

1. VT_1 <= VT_2 ? ? i: VT_1 [i] <= VT_2 [i]
2. VT_1 < VT_2 ? VT_1 <= VT_2 & ? i: VT_1 [i] < VT_2 [i]

3.5.4 Dorucovací protokol pro jednu skupinu

Protokol zarucuje kauzální usporádání dorucovaných zpráv. Ke každé vyslané zpráve je pridán aktualizovaný vektor VT(m), který ríká, kolik multicastu jednotlivých procesu kauzálne predchází zpráve m.

Protokol:

1. událost odeslání v procesu p_i

   $VT(p_i)[i]$++, oznacit tímto novým VT odesílanou zprávu

2. prijetí procesem p_j zprávy m vyslané procesem p_i

   proces p_j ruzný od p_i pozdrží dorucení m dokud neplatí:

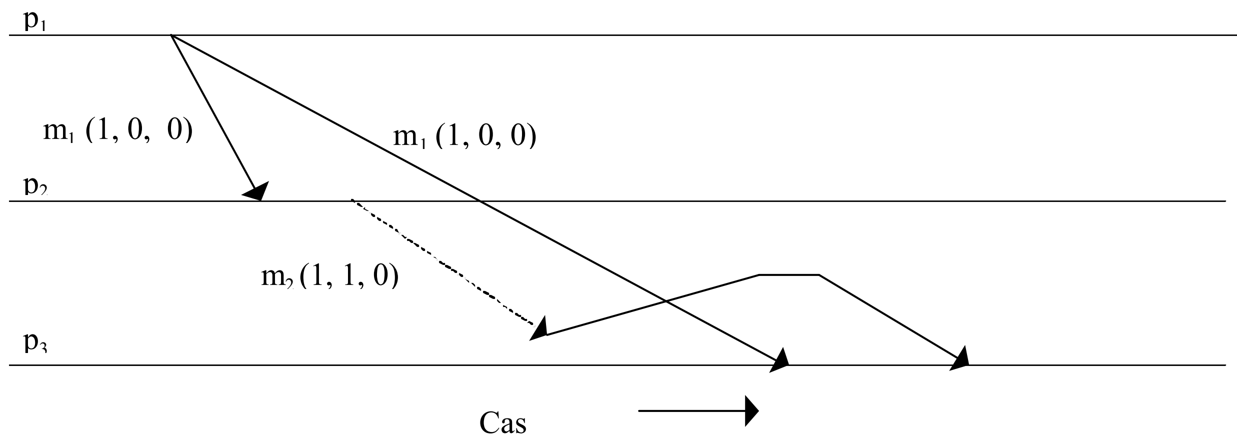
   a) $VT(m)[k] = VT(p_j)[k] + 1$ pro $k = i$

   b) $VT(m)[k] <= VT(p_j)[k]$ jinak

3. dorucení

   po dorucení si proces pj upraví VT podle pravidla 3 pro práci s VT

Bodem 1 protokol zarucuje, že každá odeslaná zpráva bude mít unikátní hodnotu VT. Bodem 2, který je zde klícový, je zaruceno, že libovolná zpráva kauzálne predcházející zpráve m, bude procesem pj dorucena pred zprávou. Bodem 3 je zarucena aktualizace VT.



Obr. 15 - Kauzálne usporádané dorucování

Na pocátku mají všechny procesy skupiny (p1, p2, p3) nastaven svoji lokální VT na hodnotu (0, 0, 0). Proces p1 vysílá zprávu m1, upraví tedy VT(p1) na (1, 0, 0) podle bodu 1 protokolu, dále pripojí tuto casovou znacku k vysílané zpráve a multicastuje zprávu ostatním procesum skupiny.

Proces p2 obdrží zprávu m1 a po overení podmínek z bodu 2 protokolu zprávu dorucí, pricemž upraví hodnotu lokálního VT(p2) na (1, 0, 0) (podle pravidla 3 pro práci s VT).

Nyní se proces p2 rozhodne vyslat skupine zprávu m2, což uciní podle bodu 1 protokolu.

Proces p3 však nejprve prijme zprávu m2, podle bodu 2b) protokolu. Tedy zprávu pozdrží. Mezitím prijme zprávu m1, kterou po overení platnosti podmínek bodu 2 dorucí a upraví si svoji

lokální VT(p3) podle pravidla 3 pro práci s VT. Nyní proces p3 muže dorucit i zprávu m2, protože jsou splneny podmínky bodu 2 protokolu 1.

3.5.5 Dorucovací protokol pro prekrývající se skupiny

Základem rozšírení dorucovacího protokolu pro prekrývající se skupiny je rozšírení zasílaných casových znacek. Casovou znacku spojenou se skupinou g_a znacíme VTa. VTa[i] tedy vyjadruje pocet multicastu procesu p_i do skupiny g_a. Se zprávou, kterou proces odesílá, posílá VT všech skupin, kterých je clenem.

1. odeslání procesem pi do skupiny g_a

   VTa(p_i)[i]++, oznacit odesílanou zprávu touto casovou znackou a casovými znackami ostatních skupin a odeslat ji.

2. Prijetí zprávy m procesem p_j (od p_i, VT(m), do g_a)

   Proces p_j ? p_i pozdrží m dokud neplatí:

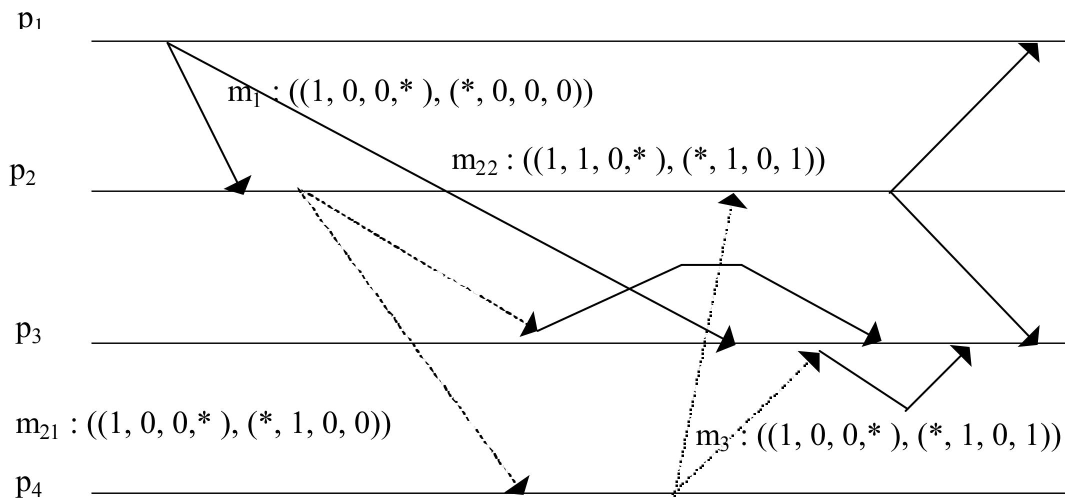
   a) VTa(m)[i] = VTa(p_j)[i] + 1

   b) ? k (p_k ? g_a & k ? i): VTa(m)[k] <= VT(p_j)[k]

   c) ? b (p_j ? g_b): VTb(m) <= VTb(p_j)

3. Dorucení

   Po dorucení si proces p_j upraví VT podle pravidla 3 pro práci s VT.



Obr. 16 - Kauzálne usporádané dorucování v prekrývajících se skupinách

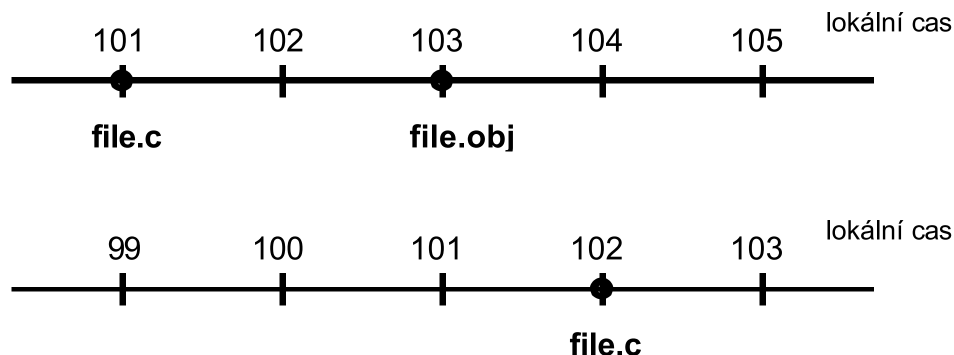
Skupinové komunikace se úcastní procesy p_1, p_2, p_3, p_4, pricemž $p_1..p_3$ jsou cleny skupiny g_1 a $p_2..p_4$ jsou cleny skupiny g_2. Proces p_1 pošle zprávu m_1 do skupiny g_1 doplnenou o casovou znacku. Proces p_2 zprávu m_1 prijme a následne pošle svuj vlastní multicast m_{21} do skupiny g_2, ke kterému pripojí aktualizovanou casovou znacku. Proces p_3 je proto nucen pozdržet dorucení m_{21} do doby, než dorucí m_1 i presto, že každý z multicastu smeroval do jiné skupiny. Stejne je pozdržena i zpráva m_3 procesu p_4, která smerovala do skupiny g_2. Zpráva m_{22} procesu p_2 je pak jen obycejná zpráva doplnená o aktualizovan casovou znacku.

4. Synchronizace

V centralizovaných systémech jsou všechny synchronizacní problémy, jako kritické oblasti, vzájemné vyloucení procesu apod., obecne rešeny metodami využívajícími semafory a monitory. Tyto metody nejsou pro použití v distribuovaných systémech príliš vhodné, nebot implicitne predpokládají nejakou formu sdílené pameti. Napr. dva procesy interagující pomocí semaforu musí mít k tomuto semaforu prístup. V distribuovaných systémech to z duvodu absence sdílené pameti není možné. Dalšími duvody pro použití zcela nových metod a synchronizacních algoritmu jsou obecné vlastnosti distribuovaných algoritmu:

- informace potrebná pro rozhodování je rozprostrena mezi nekolika uzly
- procesy se rozhodují na základe lokálních informací
- mely by být vylouceny takové komponenty, jejichž selhání zpusobí havárii systému
- neexistují spolecné hodiny ani presné merení globálního casu

4.1 Usporádání událostí a logické hodiny



Obr. 17 - Casový konflikt nesynchronizovaných hodin

L. Lamport ve svých pracech ukázal, že lze sesynchronizovat všechny hodiny tak, aby byl zarucen jednoznacný casový standard. Své rešení založil na myšlence, že synchronizace nemusí být absolutní. Jestliže dva procesy spolu nijak nekomunikují, pak není nutné, aby jejich hodiny byly synchronizovány, protože prípadné rozdíly stejne nejsou pozorovatelné a nezpusobují žádné problémy. Další jeho podstatnou ideou byl fakt, že není duležité, aby se procesy shodly na presném case, ale aby se shodly na poradí, v jakém se staly jednotlivé události.

Merení casu s temito vlastnostmi se nazývá logické hodiny. Logické hodiny zajištují konzistentní merení casu v celém systému, nezajištují však žádným zpusobem merení reálného casu.

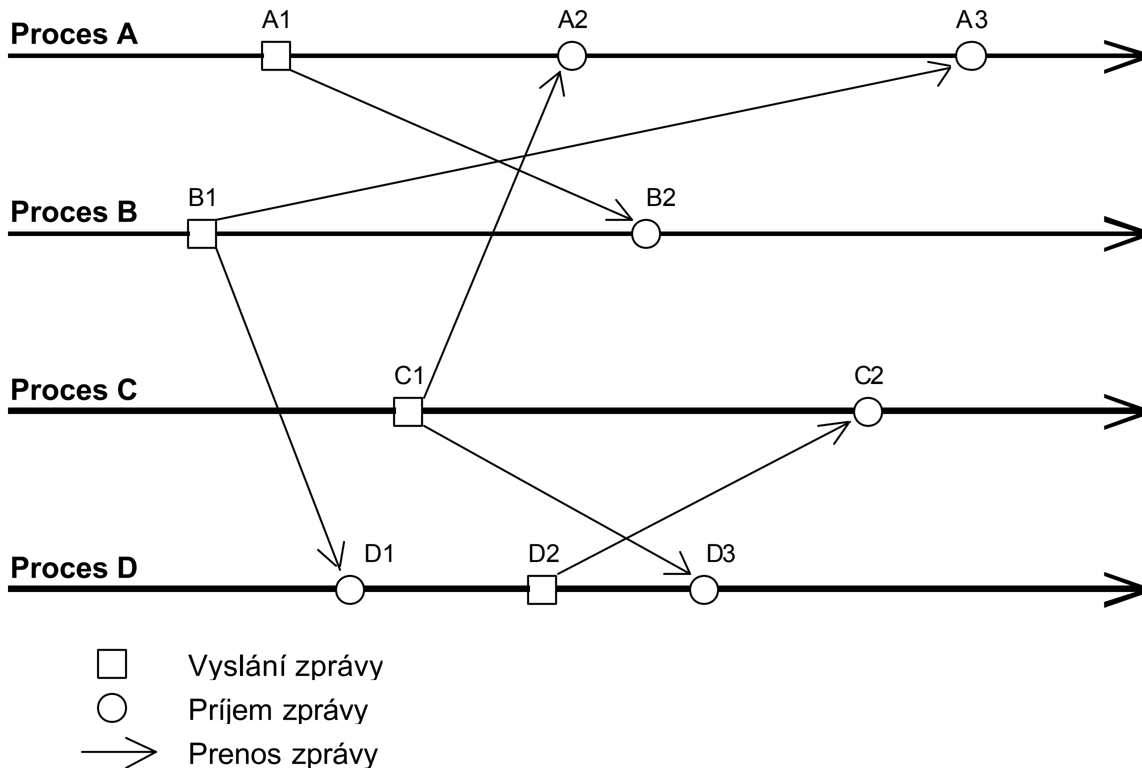
K zajištení synchronizace je definována relace "**predchází**", znacená *a* ? *b*. Tato relace znamená, že všechny procesy se shodují na tom, že událost *a* se udála pred událostí *b*. Relace *a*? *b* je definována temito axiomy:

? 1. Jestliže *a* a *b* jsou události v jednom procesu a *a* se udála pred *b*, pak *a* ? *b*

? 2. Jestliže *a* je událost vyslání zprávy a *b* je událost prijetí této zprávy, pak *a* ? *b*

? 3. Jestliže *a* ? *b* a *b* ? *c*, pak *a* ? *c*

Relace "predchází" je ostré částecné usporádání. Jestliže se události x a y staly ve dvou procesech, které spolu nekomunikují, pak neplatí ani x ? y ani y ? x; ríkáme, že události jsou konkurentní, tj. nedá nic ríci o tom, která se událost se stala první.

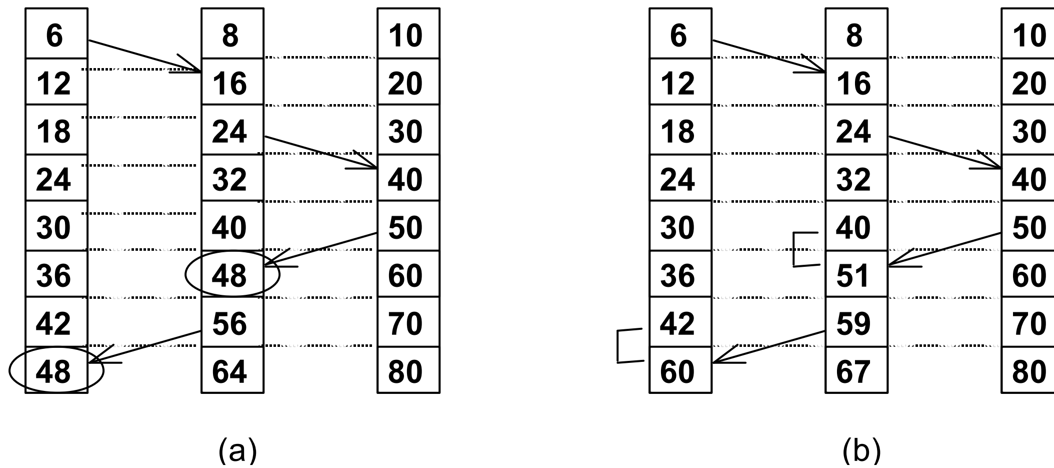


Obr. 18 - Události ve ctyrech konkurentních distribuovaných procesech

Necht cas, ve kterém se stala událost a a na kterém se shodly všechny procesy, je oznacen C(a), lokální cas procesu i $C_i(a)$. Potom jestliže a ? b, pak $C(a) < C(b)$.

Synchronizace podle prijímání zpráv - casová znacka zprávy m T_m:

- Proces i vysílá v case $C_i(a)$ zprávu m; $T_m = C_i(a)$
- Proces j prijme zprávu m v case $C_j(b)$. Pak $C_j = \max(C_j(b), T_m+1)$



Obr. 19 - (a) Procesy s lokálními hodinami (b) Korekce hodin dle Lamportova algoritmu

V nekterých situacích bývá nutné, aby se žádné dve události nestaly ve stejný okamžik. Proto se zavádí relace ? , která je definována temito axiomy:

Necht událost *a* se stala v procesu *i*, událost *b* v procesu *j* a P_n je libovolná jednoznacná ohodnocovací funkce procesu *n* (napr. interní císlo procesu). Potom

1. Jestliže C(a) < C(b), pak *a* ? *b*
2. Jestliže C(a) = C(b) a $P_i < P_j$, pak *a* ? *b*

Je zrejmé, že jestli *a* ? *b*, pak *a* ? *b*. Tedy relace ? je zúplnením relace ? .

4.2 Synchronizace fyzických hodin

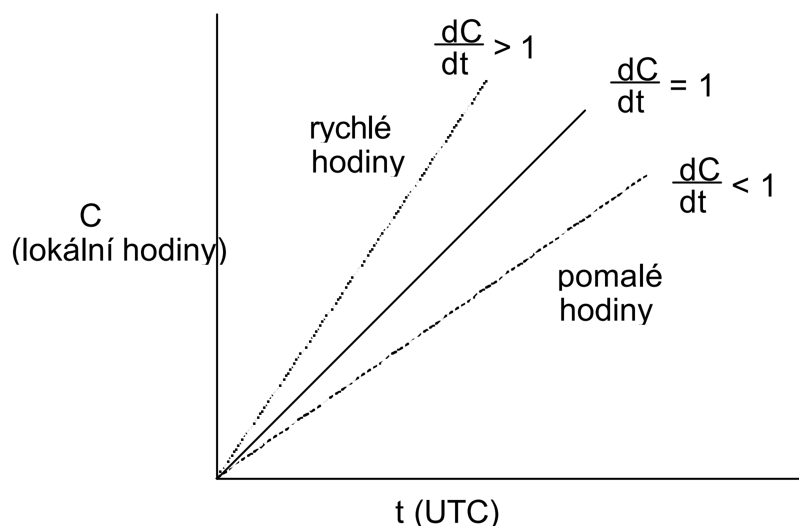
Prestože logické hodiny poskytují konzistentní usporádání událostí, casové hodnoty prirazené jednotlivým událostem nikterak neodpovídají reálnému casu. V systémech, kde je nutné, aby hodiny byly nejen vzájemne synchronizovány, ale aby se nelišily od reálného casu, je nutné zavést externí fyzické zdroje casu. Z duvodu výkonnosti a spolehlivosti je obvykle použito více casových generátoru, což zpusobuje další synchronizační problémy:

1. Jak sesynchronizovat lokální hodiny jednotlivých komponent systému
2. Jak sesynchronizovat jednotlivé hodiny mezi sebou

Na každém pocítaci beží vnitrní hodiny, jejich hodnotu oznacme C. Jestliže skutecný cas je *t*, pak hodnota hodin na pocítaci *p* je $C_p(t)$. U zcela presných hodin by platilo $C_p(t) = t$ pro všechna *t*, tedy dC/dt by ideálne melo být 1. Avšak vnitrní hodiny nemerí cas zcela presne; každé hodiny mají od výrobce zarucenou míru presnosti ***?***, pro kterou platí:

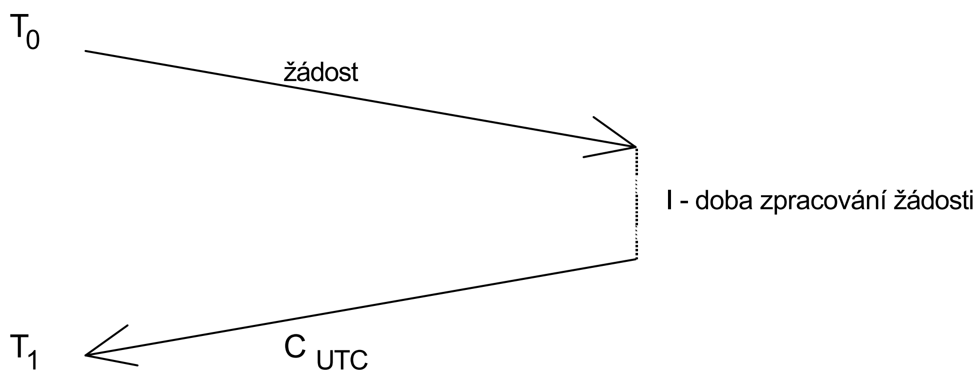
$$1 ? ? \; ? \; \frac{dC}{dt} \; ? \; 1 ? ?$$

Dvoje hodiny stejného typu se tedy mohou odlišovat od skutecného casu maximálne o hodnotu 2 ? ? t, kde ? t je doba od poslední synchronizace. Jestliže je tedy zapotrebí, aby se hodiny vzájemne nerozcházely o více než ?, pak musí být synchronizovány v intervalech ne vetších nežli ? /2? .



Obr. 20 - Rozdílné rychlosti lokálních hodin

4.2.1 Cristianuv algoritmus



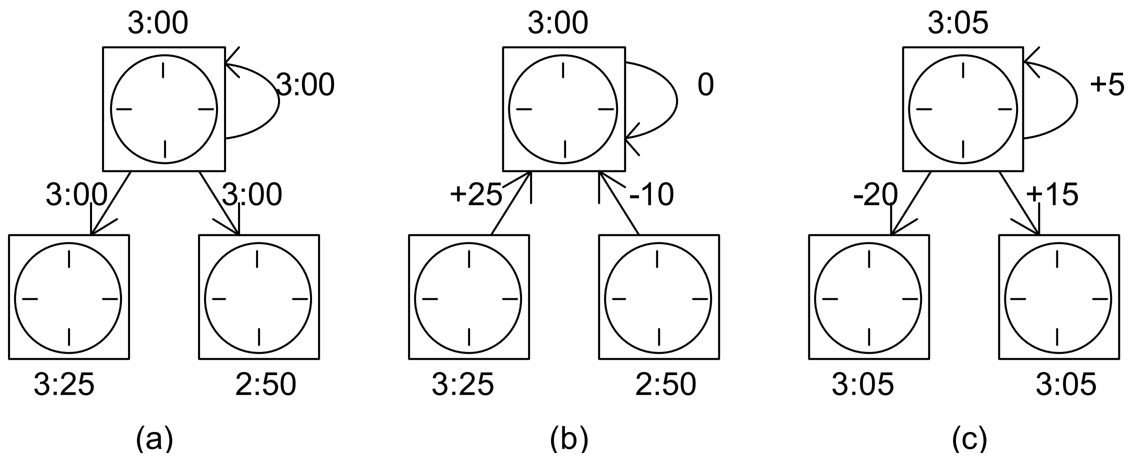
Obr. 21 - Žádost o presný cas

- jeden pasivní time server
- postupná zmena; zrychlování nebo zpomalování
- $T = T_{UTC} + (T_1 - T_0 - I)/2$

4.2.2 Berkeley algoritmus

(Berkeley UNIX)

- aktivní time server
- periodicky se ptá ostatních na rozdíl casu, pocítá prumer, vrátí rozdíl



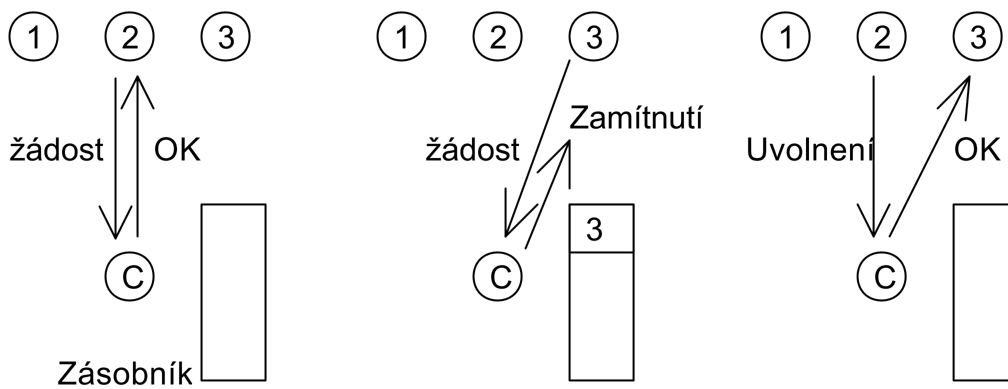
Obr. 22 - (a) Casový server zahájí synchronizaci hodin (b) Odpovedi obsahující casový rozdíl (c) Zprávy s urcením opravy lokálních hodin

4.2.3 Distribuovaný algoritmus

- resynchronizacní intervaly pevné délky
- i-tý interval: T_0+iR .. T_0+(i+1)R
- broadcast (ve fyzicky nestejný cas), spocítání prumeru, prípadné zahození extrému

4.3 Vzájemné vyloucení procesu

4.3.1 Centralizovaný algoritmus



Obr. 23 - (a) Žádost procesu o vstup do kritické sekce (b) Zamítnutí konkurentního procesu (c) Povolení vstupu do uvonené kritické sekce

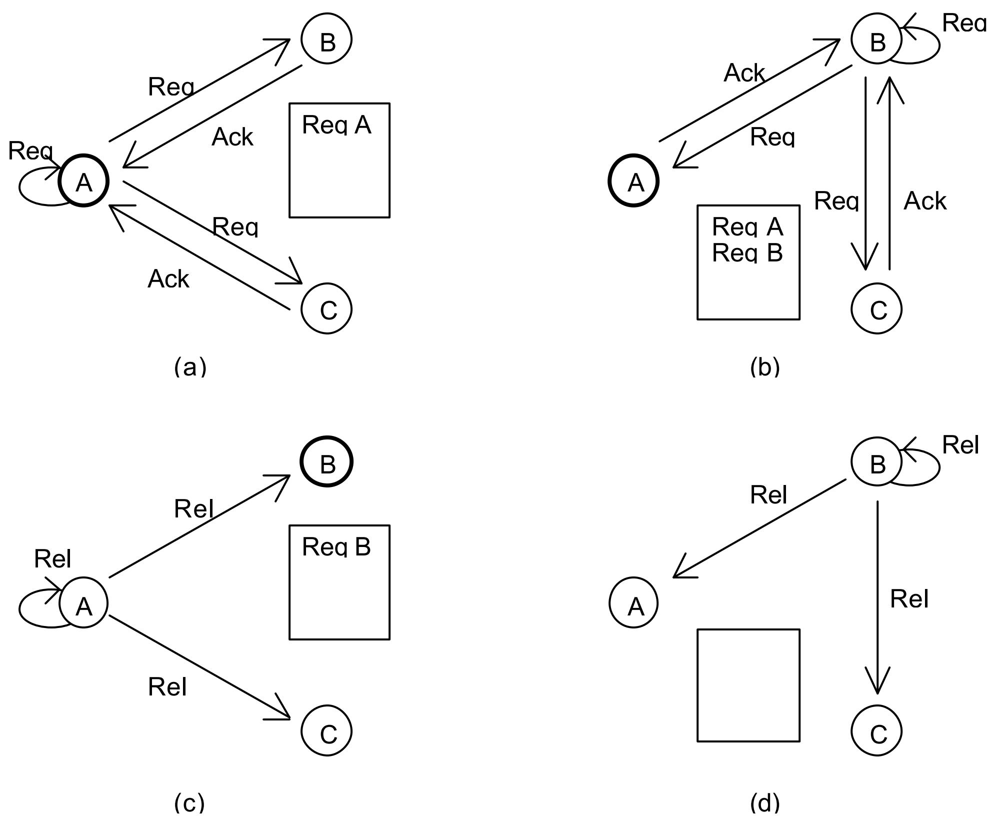
4.3.2 Lamport

Proces p, casová znacka odeslání jeho zprávy T_p zprávy typu req, ack, rel, akce se zprávami send, add, del.

Proces vyšle žádost a ceká až dorazí odpovedi od všech procesu a všechny žádosti v jeho fronte mají vetší casovou znacku.

Událost	Akce
Žádost M_p	`send` M_p={req,p,T_p}
Prijetí žádosti M_i	`add` M_i; `send` {ack,p,T_p}
Prijetí potvrzení A_i	`add` A_i
Podmínka vstupu M_p	? i? p ? {ack,i,T_i} : T_p<T_i & ? {req,i? p,T_i } : T_p<T_i
Uvolnení R_p	`send` {rel,p,T_p}
Prijetí uvolnení R_i	`del` {req,i,T_k } : T_k < T_i

Tab. 9 - Reakce na události v Lamportove algoritmu



Obr. 24(a) Vstup procesu A do kritické sekce (b) Žádost procesu B o vstup do kritické sekce (c) Uvolnení kritické sekce procesem A a vstup procesu B do kritické sekce (d) Uvolnení kritické sekce

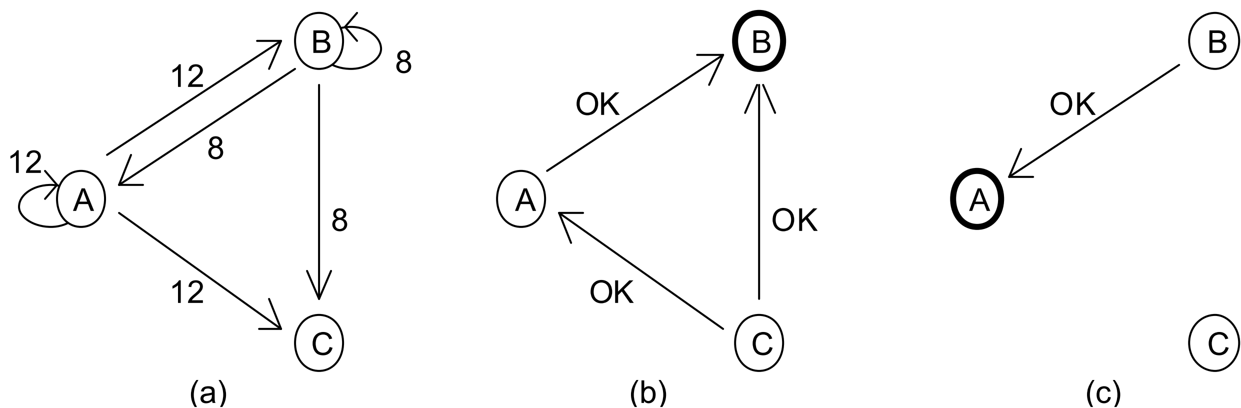
4.3.3 Ricart & Agrawala

Když chce proces vstoupit do kritické sekce, zašle žádost s casovou znackou ostatním procesum a ceká na všechny došlé odpovedi s potvrzením možnosti vstupu.

Když proces prijme zprávu se žádostí, pak:

1. Jestliže príjemce není v kritické sekci a ani do ní nechce vstoupit, pak pošle zpet zprávu s potvrzením.
2. Jestliže je príjemce v kritické sekci, pak neodpovídá, požadavek si zaradí do fronty.
3. Jestliže príjemce ješte není v kritické sekci, avšak chce do ní vstoupit, porovná casovou znacku prijaté žádosti s casovou znackou vlastní žádosti. Pokud vlastní žádost má nižší casovou znacku, tj. byla vyslána dríve, pak neodpovídá a zaradí žádost odesílatele do fronty. V opacném prípade, kdy byla žádost odesílatele odeslána dríve, pošle príjemce zpet zprávu s potvrzením.

Po opuštení kritické sekce proces pošle zprávu s potvrzením všem procesum, které má ve fronte.



Obr. 25 - (a) Procesy A a B ve stejný okamžik žádají o vstup do kritické sekce
(b) Proces B díky nemenší casové znacce vstupuje do kritické sekce, proces A ceká
(c) Po opuštení procesem B muže do kritické sekce vstoupit proces A

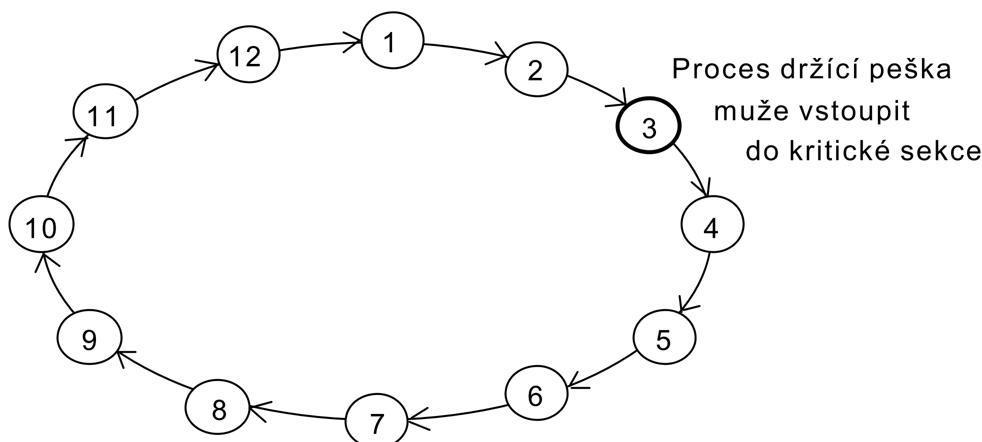
Problémy:

- n points of failure -> ack
- either group communication or selfmaintenance
- n points of overloading

4.3.4 Suzuki & Kasami

Prenos zprávy obsahující povolení ke vstupu do kritické sekce. Zpráva obsahuje frontu požadavku, která se updatuje lokálními došlými požadavky pri výstupu z kritické sekce. Komunikacní složitost: 2*(N-1)

4.3.5 Token ring



Obr. 26 - Softwarove konstruovaný token ring

4.3.6 Porovnání jednotlivých algoritmu

Algoritmus	Pocet zpráv	Prodleva	Problémy
Centralizovaný	3	2	Havárie koordinátora
Lamport	3(n-1)	3(n-1)	Výpadek libovolného procesu
Ricart & Agrawala	2(n-1)	2(n-1)	Výpadek libovolného procesu
Token ring	1 až ?	0 až n-1	Ztráta peška, výpadek procesu

Tab. 10 - Porovnání algoritmu pro vzájemné vyloucení procesu

4.3.7 Vyloucení procesu s prioritami

- Prostredí vyžadující priority
- Real-time systems w/ deadlines

Usporádaná fronta - podle priorit / podle casu, který zbývá na dokoncení akce

Predávání peška s frontou na žádost (není cirkulující)

Žádost - broadcast všem

Prijetí žádosti - zarazení do fronty

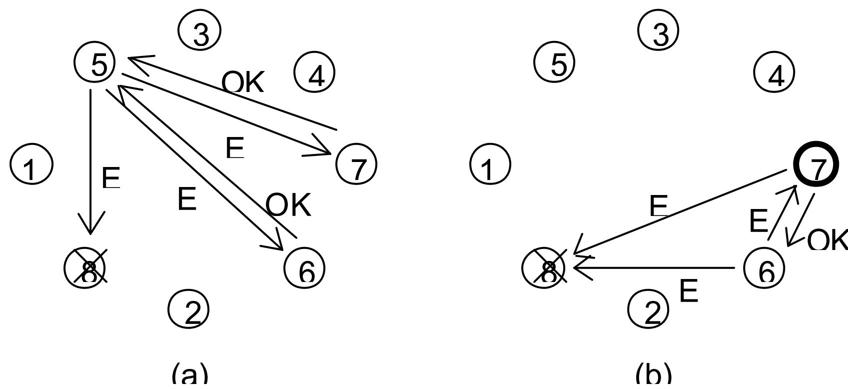
Prijetí peška - vstup do kritické sekce, merge fronty, poslání prvnímu

4.4 Volba koordinátora

4.4.1 Bully algoritmus

Když se proces rozhodne volit, zašle zprávu všem procesum s vyšší identifikací (císlo procesu apod.). Když prijde odpoved, proces koncí, když neprijde nic, proces vyhrál, je novým koordinátorem

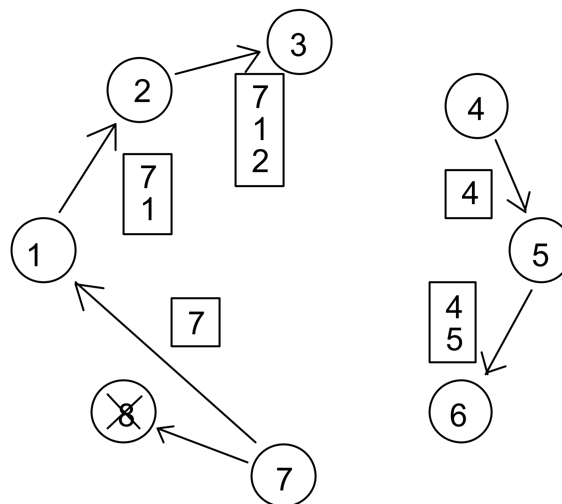
a pošle o tom zprávu všem ostatním. Když proces prijme zprávu o volbe, vrátí zpet odpoved a sám vyšle žádosti všem vyšším procesum. Volba se zabezpecí ve dvou kolech.



Obr. 27 - Bully algoritmus

4.4.2 Kruhový algoritmus

Když se proces rozhodne volit, pošle zprávu následníkovi, zpráva obsahuje císla procesu (stací odesílatel a nejvyšší živý). Až se zpráva dostane zpet k odesílateli, vyšle dokola zprávu s oznámením koordinátora. Stací znát následníky a zjistit následníka spadlého uzlu.



Obr. 28 - Kruhový algoritmus

4.4.3 Kruhový algoritmus - O(n log n)

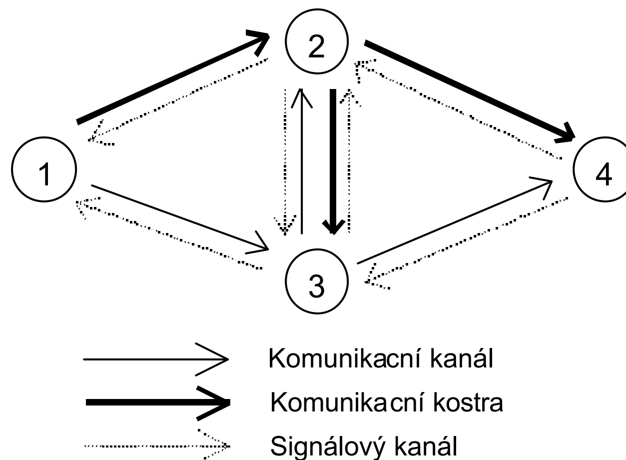
Klasický kruhový algoritmus má v nejhorším prípade složitost $O(n^2)$. Chytrejší algoritmus má složitost O(n log n). Využívá pojem okolí - k-okolí procesoru p je množina procesoru ve vzdálenosti max. k. Velikost této množiny je tedy 2k+1.

Algoritmus pracuje ve fázích, v n-té fázi se procesor pokoší být koordinátorem svého 2^n -okolí. Pouze procesory, které se stanou lokálními koordinátory, postupují do další fáze.

4.5 Synchronizace ukoncení a detekce globálního stavu

Sekvencní programy obvykle koncí provedením poslední instrukce. Konkurencní a distribuované procesy jsou však casto konstruovány metodou nekonecné smycky - proces ceká na zprávu, kterou zpracuje. V centralizovaných systémech je jednoduché urcit, že procesy již nejsou nikým využívány - fronta pripravených úkolu je prázdná. V distribuovaném systému však toto není možné, nebot neexistuje zpusob, jak zjistit v jeden casový moment celkový stav systému. Existuje však nekolik algoritmu, které problém ukoncení reší.

Predpokládejme množinu procesu propojených jednosmernými komunikacními kanály (obousmerná komunikace muže být implementována dvojicí jednosmerných kanálu). Kanály jsou spolehlivé a zachovávají poradí zpráv. Každý uzel tedy má množinu vstupních a množinu výstupních kanálu (hran). Ke každému komunikacnímu kanálu existuje signální kanál, který vede v opacném smeru. Signální kanál neslouží k prenosu dat, ale k zasílání signálu potrebných pro daný algoritmus.



Obr. 29 - Komunikacní a signálové kanály

Predpokládejme existenci iniciacního procesu, ze kterého je pres komunikacní kanály prístupný každý proces. Iniciacní proces tedy nemá žádné vstupní hrany. Výpocet zacíná vokamžiku, kdy iniciacní proces vyšle podél svých výstupních hran zprávu. Každý proces se pri príjmu zprávy dostane do stavu *výpocet*. V tomto stavu muže dále posílat zprávy podél výstupních hran a muže prijímat zprávy ze vstupních uzlu. Jakmile proces výpocet skoncí, dostane se do stavu *ukoncen*, žádné další zprávy už neposílá, muže však zprávy prijímat. Pri prijetí zprávy se proces zrestartuje a dostane se opet do stavu *výpocet*.

Algoritmus synchronizace ukoncení má za úkol zabezpecit následující: V prípade, že všechny procesy jsou ve stavu *ukoncen*, pak se v konecném case tuto skutecnost všechny procesy dozví. Z praktického hlediska postací, když se to dozví iniciacní uzel, nebot ten již muže tuto zprávu ostatním uzlum poslat.

4.5.1 Dijkstra-Scholten (DS) algoritmus

Jestliže graf procesu je strom, pak každý listový proces pri prechodu do stavu *ukoncen* pošle signál svému otci. Jakmile proces dostane signály od všech svých synu, pošle signál svému otci. Jestliže všechny signály dostane iniciacní proces, je distribuovaný výpocet ukoncen.

Tento algoritmus lze rozšírit na acyklický orientovaný graf. Ke každé hrane asociován cítac - tzv. *deficit*, který udává rozdíl mezi poctem zpráv došlých tímto datovým kanálem a poctem signálu poslaných signálním kanálem v opacném smeru. Koncící proces vyšle každým signálním kanálem tolik signálu, aby deficit byl nulový.

V prípade, že grafem procesu je obecný (orientovaný) graf, pak tento algoritmus nefunguje, nebot neexistují listy, které by mohly samy rozhodnout o ukoncení výpoctu. Tento problém se dá vyrešit vytvorením kostry grafu. Behem výpoctu se vytvorí kostra grafu tak, že uzel, od kterého prišla první zpráva, se oznací za otce. Algoritmus ukoncení pro jeden proces se pak uskutecní ve trech krocích:

1. Poslat signál podél všech vstupních hran krome hrany k otci.
2. Cekat na signál od všech výstupních hran.
3. Poslat signál otci.

Až dojde dostatecný signál korenu celého grafu, lze distribuovaný výpocet ukoncit - napr. rozesláním zprávy o ukoncení celému grafu.

4.5.2 Znackový (TM) algoritmus

Iniciacní uzel vyšle žádost o ukoncení (znacku, že je kanál prázdný) všem výstupním hranám.

Prijmutí první znacky:

- pripraven - propagace znacky výstupním hranám
- nepripraven - negativní signál (zamítnutí)

Prijmutí další znacky: signál / zamítnutí

Prijmutí zamítnutí: zamítnutí první žádosti

Prijmutí všech potvrzení: signál první žádosti

4.5.3 Detekce globálního stavu

Znackový algoritmus je speciálním prípadem mnohem obecnejšího algoritmu - zjištování globálního stavu systému (Chandy & Lamport, Distributed snapshot).

Množina událostí v systému $E = \{e\}$

Rez *c* je rozdelení *E* na P_c a F_c : P_c ? $F_c = E$ & P_c ? F_c = ?

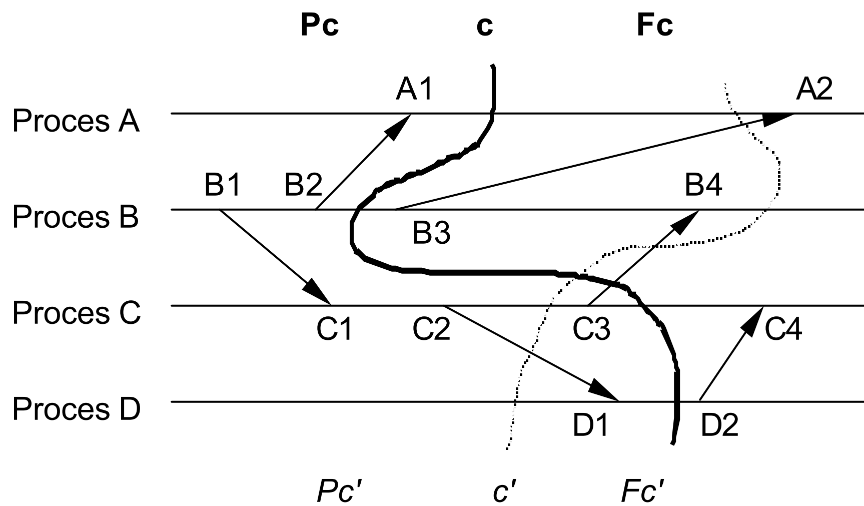
Konzistentní rez *c : a*? *b* & *a* ? F_c ? *b* ? F_c

Stav procesu (systému) je množina událostí, které se v procesu (systému) udály

Konzistentní stav systému $S = P_c$, kde *c* je konzistentní rez

Necht S je konzistentní stav a *e* taková událost, že S' = S ? *e* je konzistentní stav. Ríkáme, že S' je **dosažitelný** z S, znacíme S ? ^{e} S'.

Posloupnost událostí *s* = (e_1, e_2, ..., e_n) se nazývá **rozvrh (schedule)**, jestliže S ? ^{e_1} S_1 ? ^{e_2} S_2 . . . S_{n-1} ? ^{e_n} S_n. Znacíme S ? ^{s} S_n. Zrejme platí S ? ^{s} S_n ? S ? S_n.



Obr. 30 - Rez distribuovaného systému - konzistentní (c) a nekonzistentní (c')

Algoritmus

Stav uzlu je množina prijatých a odeslaných zpráv, stav kanálu je množina zpráv, které byly tímto kanálem odeslány, ale ješte nebyly doruceny.

Iniciátor vyšle všem výstupním uzlum znacku.

Príjem první znacky:

?? Uzel si zapamatuje poslední prijaté a odeslané zprávy = stav uzlu

?? Stav všech príchozích kanálu oznací za prázdný

?? Vyšle všem výstupním uzlum znacku

Príjem zpráv od uzlu, od kterých ješte neprišla znacka:

?? Zapamatuje si císla zpráv

Príjem znacek od dalšího uzlu:

?? Stav kanálu = zaznamenané zprávy došlé od uzlu mezi prijmutím první znacky a znacky od tohoto uzlu

Algoritmus koncí po prijmutí všech znacek. Zaznamenané stavy uzlu a kanálu definují konzistentní stav systému.

5. Transakcní zpracování

Všechny doposud zminované synchronizacní techniky byly nízkoúrovnové, vyžadující znalost problematiky semaforu, vzájemného vyloucení, kritických sekcí apod. Pro vývoj distribuovaných aplikací by však byla vhodnejší mnohem vyšší úroven abstrakce, tak, aby se programátor nemusel starat o technické detaily synchronizace. Takovýto mechanismus v distribuovaných systémech existuje, nazývá se **atomické transakce**, nebo také jen krátce transakce.

Pojem transakce pochází ze sveta obchodu. Dve strany nejdríve navazují kontakt, vyjednávají, sepisují smlouvy, mení podmínky, a teprve až po oboustranném podepsání je transakce uzavrena. Pokud kdykoliv do tohoto okamžiku libovolná strana z jednání odstoupí, transakce se neuskutecní a vše se vrátí do stavu, jaký byl pred zapocetím transakce. Jakmile je však smlouva podepsána, nelze se vrátit zpet a transakce musí být provedena.

Pocítacový model je obdobný. Jeden proces zacne transakci, behem jejího zpracování mohou být provádeny ruzné operace. Až se proces rozhodne transakci uzavrít, oznámí to spolupracujícím procesum. Jestliže všechny procesy souhlasí s uzavrením, výsledky transakce se projeví v celém systému, stanou se permanentními. Jestliže alespon jeden z procesu nesouhlasí s ukoncením transakce (nebo zhavaruje pred jejím dokoncením), stav systému se vrátí presne do té podoby, jakou mel pred zapocetím transakce a všechny provedené zmeny budou zrušeny. Tato vlastnost "všechno nebo nic" (all-or-nothing property) významne zjednodušuje vývoj distribuovaných aplikací.

Metodou transakcního zpracování fungovalo pocítacové zpracování dat v 60. letech. Na jedné pásce byl zaznamenán nejaký stav, na jinou pásku (nebo pásky) se postupne zaznamenávaly požadované zmeny. Ve vhodných intervalech (napr. jednou denne, každou hodinu apod.) se daly obe pásky zpracovat a pocítac na základe stavu jedné pásky a požadovaných zmen na druhé pásce vyrobil tretí pásku s novým stavem. Výhodou tohoto zpusobu zpracování byla možnost opakování celé akce v prípade havárie libovolné cásti systému.

5.1 Vlastnosti transakcí

Transakce je sekvence operací na jednom nebo více objektu, která transformuje jeden konzistentní stav na jiný konzistentní stav. Transakce je uzavrená mezi znackami **BeginTransaction** a **EndTransaction** a má následující vlastnosti:

?? Atomicita - pro vnejší svet je transakce nedelitelná

?? Konzistence - transakce neporušuje vnitrní konzistenci

?? Izolovanost (sekvencnost) - konkurentní transakce se navzájem neovlivnují

?? Trvanlivost - po úspešném ukoncení transakce jsou zmeny trvalé

Tyto vlastnosti se casto oznacují zkratkou ACID (Atomicity, Consistency, Isolation, Durability).

První vlastnost, **atomicita**, zajištuje, že každá transakce se bud provede kompletne celá, anebo se neprovede vubec. Dokud se transakce zpracovává, je pro ostatní svet neviditelná. Napríklad pokud by se provádela transakce pridání deseti bajtu k nejakému souboru, tak po celou dobu zpracování této transakce ostatní procesy vidí puvodní stav souboru a teprve po úspešném ukoncení transakce vidí soubor i s pridanými deseti bajty. Žádné mezistavy nejsou pro ostatní procesy viditelné bez ohledu na to, jak dlouho transakce trvala.

Konzistence zajištuje, že transakce zachovává systém konzistentní. Behem zpracovávání transakce muže být systém docasne nekonzistentní, avšak po ukoncení transakce musí být preveden do konzistentního stavu.

Další vlastnost, **izolovanost**, nebo také **sekvencnost**, zabezpecuje, že v prípade konkurentního zpracování dvou nebo více transakcí bude konecný výsledek takový, jako kdyby transakce byly provádeny sekvencne v nejakém (systémove závislém) poradí. Mejme tri konkurentne vyvolané transakce. Kdyby tyto transakce byly provedeny sekvencne, byl by výsledek 1, 2 nebo 3 podle toho, která transakce by se vykonala jako poslední. V tabulce jsou znázorneny tri možné rozvrhy. Rozvrh 1 je sekvencní. Rozvrh 2 sice není sekvencní, avšak splnuje podmínku sekvencnosti, nebot výsledek (v tomto prípade 3) je stejný, jako kdyby byly transakce provádeny sekvencne. Rozvrh 3 nesplnuje podmínku sekvencnosti, nebot výsledek (v tomto prípade 5) neodpovídá žádnému možnému sekvencnímu poradí transakcí.

```
BeginTransaction          BeginTransaction          BeginTransaction
x = 0;                    x = 0;                    x = 0;
x = x + 1;                x = x + 2;                x = x + 3;
EndTransaction            EndTransaction            EndTransaction
```

Rozvrh 1	x=0	x=x+1	x=0	x=x+2	x=0	x=x+3	OK
Rozvrh 2	x=0	x=0	x=x+1	x=x+2	x=0	x=x+3	OK
Rozvrh 3	x=0	x=0	x=x+1	x=0	x=x+2	x=x+3	ilegální

Tab. 11 - Možné rozvrhy trí transakcí

Trvanlivost znamená, že poté, co byla transakce úspešne ukoncena, jsou její výsledky trvalé. Libovolné pozdejší výpadky ci jiné havárie nemohou dokoncenou transakci zrušit.

5.1.1 Vnorené transakce

Transakce mohou obsahovat **vnorené transakce**. Pokud se vnorená transakce neprovede, neprovede se ani nadrízená transakce. Muže však nastat i opacný prípad - vnorená transakce se úspešne ukoncí, avšak nadrízená transakce je pozdeji zrušena. V tomto prípade musí být vnorená transakce zrušena a to i pres to, že je již úspešne ukoncena. Pravidlo permanence zde neplatí, permanence se vztahuje jen na transakce nejvyšší úrovne.

Vzhledem k tomu, že transakce mohou být vnorovány do libovolné úrovne, je zapotrebí jistá režie. Presto je však sémantika jednoznacná - pri startu libovolné (pod)transakce se vytvorí (logická) kopie všech objektu, ke kterým má transakce prístup. Behem zpracování transakce mohou být tyto lokální objekty libovolne meneny. V prípade zrušení transakce se zruší i tyto kopie. V prípade úspešného ukoncení transakce se nové verze lokálních objektu prenesou do prostoru svého otce. Tím se výsledky transakce stanou viditelné.

5.1.2 Zotavení z chyb

Transakcní systém by mel být odolný vuci ruzným chybám a poruchám. V prípade havárie libovolné cásti systému v okamžiku, kdy transakce zpusobila nekonzistenci stavu, je nutné uvést systém do konzistentního stavu. Existují dve možnosti:

1. Návrat do posledního konzistentního stavu, tj. všechny zmeny vyvolané transakcí se zruší. Tento zpusob se nazývá **zpetné zotavení** (backward recovery).
2. Nastavení výsledného konzistentního stavu, tj. v prípade dostatecného množství informací následné dokoncení transakce. Toto se nazývá **dopredné zotavení** (forward recovery). Dopredné zotavení je možné provést jen v prípade, že výpadek nastal po okamžiku dokoncení transakce, avšak pred tím, než mohly být zmeny provedeny.

5.2 Implementace transakcí

Transakcní systém musí poskytovat spolehlivé služby v systému složeném z libovolného poctu nezávislých procesu, z nichž libovolný pocet muže v náhodných okamžicích zhavarovat. Komunikace muže být obecne nespolehlivá, zprávy se mohou ztrácet, avšak predpokládá se spolehlivý protokol zabezpecený nižšími vrstvami, který zajištuje timeouty, potvrzování a prípadné opakování ztracených paketu.

5.2.1 Stabilní pamet

Pamet mužeme rozdelit podle její spolehlivosti do trí kategorií. První kategorií je obycejná pamet RAM, která je pri výpadku proudu nebo vypnutí pocítace vymazána. Druhou kategorií je disková pamet, která není citlivá na výpadky proudu ci procesoru, avšak data se mohou ztrácet vinou špatných sektoru, poruch ctecí a záznamové hlavy apod.

Tretí kategorií je stabilní pamet, která je konstruována tak, aby prežila cokoliv krome kalamit typu zemetresení ci povoden. Stabilní pamet muže být implementována pomocí dvou spražených disku. Každý blok na druhém disku je presnou kopií bloku prvního disku. Když je nejaký blok zmenen, provede se zmena nejdríve na prvním disku a teprve po dokoncení a zkontrolování se zmena provede na druhém disku.

V prípade výpadku proudu jsou oba disky porovnány blok po bloku. Jestliže se bloky vzájemne liší, dá se predpokládat, že k výpadku došlo mezi zápisem na první a druhý disk. V tomto prípade se obsah bloku prvního disku prekopíruje na druhý disk.

Další potenciální problém je ztráta dat zpusobená špatným sektorem nebo poruchou hlavy. Data v bloku, kde nesouhlasí kontrolní soucet, mohou být prepsána daty z odpovídajícího bloku druhého disku.

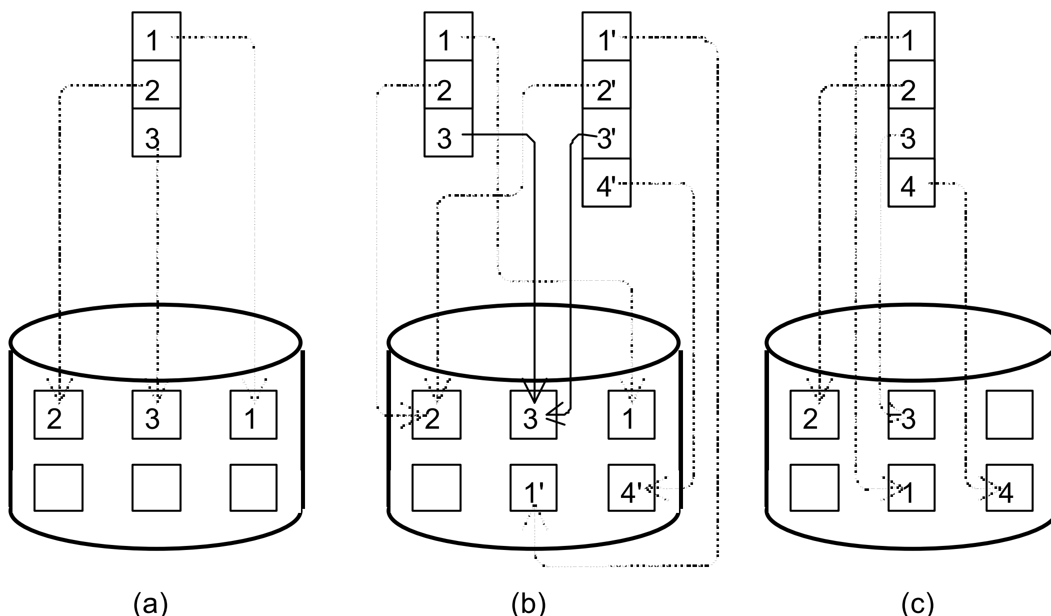
Stabilní pamet je vhodná pro aplikace vyžadující vysoký stupen spolehlivosti. Pokud jsou již data zapsána, pravdepodobnost, že budou ztracena, je extrémne nízká.

5.2.2 Lokální pracovní prostor

Jednou z metod implementace transakcí je tzv. lokální pracovní prostor. Když proces zacne provádet transakci, vytvorí si vlastní pracovní prostor obsahující všechny objekty, kterých se transakce bude týkat. Po dobu trvání transakce se zmeny provádejí pouze na techto lokálních objektech. Prímá implementace této metody je však príliš neefektivní. Transakce se casto provádejí na velkých databázích. Kopírovat pri každé transakci celou databázi je zbytecne náročné na cas i diskový prostor.

První optimalizace je založena na myšlence nekopírovat objekty, které nejsou transakcí meneny. Dokud není takový objekt menen, používá transakce reálnou verzi objektu. Pri startu transakce obdrží proces ukazatele na rodicovské objekty, v prípade transakce nejvyšší úrovne ukazatele na reálná data. V prípade kopie do jiného pracovního prostoru se všechny podrízené ukazatele presmerují na novou kopii.

Druhá optimalizace spocívá v kopírování jen tech cástí objektu, které jsou meneny (bloky souboru, databázové záznamy apod.). Pri startu transakce se vytvorí pro každý objekt blok indexu (ukazatelu), které urcují, kde je daná cást objektu umístena. Stejným zpusobem je rešeno i pridávání bloku ci záznamu k objektu.



Obr. 31 - (a) Index a diskové bloky tríblokového souboru
(b) Transakce menící obsah bloku 1 a pridávající blok 4 (c) Situace po ukoncení transakce

5.2.3 Intencní seznam

Další používanou metodou implementace je intencní seznam. Server behem provádení transakce zaznamenává požadované zmeny - vytvárí si tzv. intencní seznam. Do tohoto seznamu jsou zaznamenávány všechny zmeny, které požaduje klient provést na objektech, ke kterým má server prístup. Obsahuje identifikaci transakce, která zmenu požaduje, identifikaci objektu, který má být zmenen a jeho starou a novou hodnotu. Objekt muže být zmenen teprve až je záznam úspešne proveden.

V prípade, že je transakce úspešne ukoncena, je tato skutecnost zaznamenána do intencního seznamu. Vlastní data zustávají stejná, nebot již byla behem transakce zmenena. Jestliže je transakce zrušena, je seznam použit pro obnovení puvodního stavu. Všechny zaznamenané akce se provedou v obráceném poradí, címž se objekt dostane do stejného stavu, v jakém byl pred zapocetím transakce. Tato akce je nazývána **rollback**.

Intencní seznam muže být také použit pro obnovu stavu po nejakém výpadku. Jestliže server zhavaroval po zapsání do intencního seznamu, avšak pred zmenou vlastních dat, muže server po nastartování akci dokoncit a požadované zmeny provést.

5.3 Transakcní komunikacní primitiva

Komunikacní schéma používané v transakcním zpracování se nekdy nazývá **Commitment, Concurrency and Recovery** (potvrzování, konkurence a zotavení, CCR). CCR protokol je typu master/slave, kde master iniciuje transakce a slave je zodpovedný za provedení požadovaných zmen. Jelikož slave muže být master vzhledem k jiným procesum, muže se vytvorit exekucní strom.

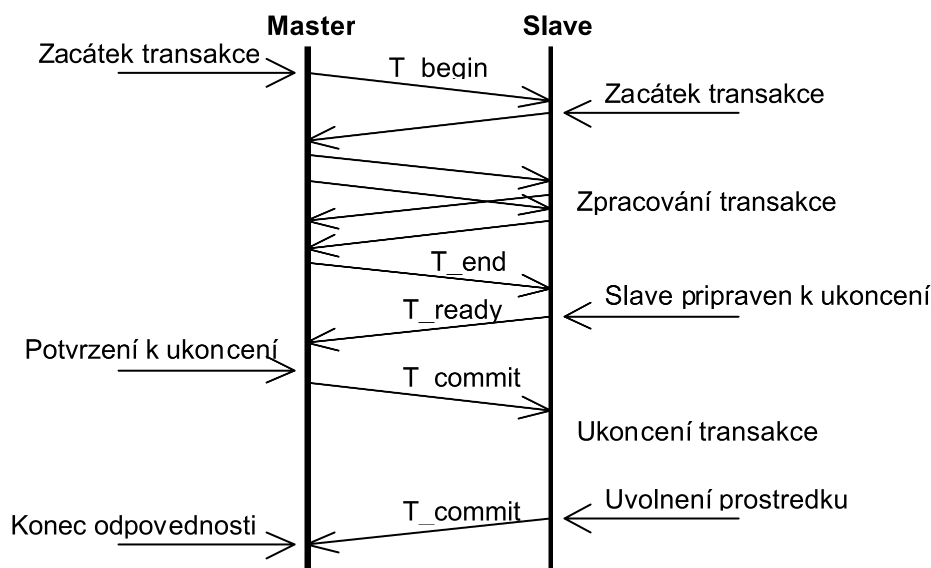
Události, které se vyskytují pri komunikaci mezi masterem a slavem:

1. Zacátek transakce - master vyšle príkaz **T_begin.**
2. Master a slave si vzájemne predávají transakcne specifické zprávy. Pokud se kdokoliv rozhodne transakci zrušit, pošle príkaz **T_abort**.

3. Konec transakce (pokud není zrejmý ze zasílaných zpráv) oznámí master zasláním príkazu **T_end**.
4. Slave, v prípade, že je pripraven transakci dokoncit, pošle zprávu **T_ready**.
5. Master pošle príkaz **T_commit**.
6. Slave provede požadované akce a potvrdí ukoncení transakce zprávou **T_commit**.

T_begin	M? S	Zacátek transakce
T_end	M? S	Explicitní konec transakce
T_ready	M? S	Pripravenost na commit
T_commit	M? S	Commit - potvrzení transakce
T_abort	M? S	Zrušení transakce

Tab. 12 - Základní transakcní komunikacní primitiva



Obr. 32 - Základní komunikacní schéma CCR protokolu (dvoufázový commit)

5.3.1 Dvoufázový commit (potvrzování)

Pro zabezpecení atomicity uzavrení transakcí byl vyvinut protokol nazývaný **dvoufázový commit** (dvoufázové potvrzování).

Necht se transakce vyvolané jedním procesem (master) úcastní n dalších procesu (slaves). V první fázi master posílá požadavky ostatním procesum, ty podle nich vykonávají požadované akce a prípadne vracejí odpovedi. Až se master rozhodne transakci ukoncit, pošle všem procesum zprávu o tomto zámeru a vycká na odpovedi. Každý slave po obdržení této žádosti zjistí, zda je schopen transakci úspešne uzavrít. V prípade, že ano, uzamkne prístup k modifikovaným objektum a vrátí potvrzení. V prípade, že není schopen transakci uzavrít, pošle zpet zprávu oznamující zrušení transakce. Jestliže master obdržel alespon jednu zápornou odpoved, transakci zruší.

Jestliže master obdržel všechny potvrzující odpovedi, zacíná druhá fáze. Vyšle všem slavum commit, ti provedou požadované zmeny a uvolní objekty. Po ukoncení všech potrebných akcí vyšlou potvrzení o ukoncení transakce. Poté, co master obdrží odpovedi od všech slavu, je transakce ukoncena.

5.4 Kontrola konkurence

Jestliže je více transakcí provádeno soucasne, je treba zabezpecit korektní chování transakcního systému. Takovýto algoritmus se nazývá kontrola (rízení) konkurence.

5.4.1 Zámky

Nejstarším a doposud nejrozšírenejším algoritmem je uzamykání (locking). Ve své nejjednodušší podobe, když proces chce císt nebo zapisovat nejaká data, nejprve objekt (soubor) uzamkne, címž k nemu znemožní ostatním procesum prístup. Zámky mohou být implementovány bud pomocí centralizovaného správce zámku anebo distribuovane, kdy každý server pristupující k transakcním objektum, zpracovává své lokální objekty. V obou prípadech si správce zámku udržuje seznam uzamcených objektu a nepovolí k nim prístup žádnému jinému procesu. Zámky jsou normálne používány transakcním systémem, vzhledem k uživatelum jsou transparentní.

Toto základní schéma je však príliš omezující, existuje nekolik vylepšení, která zvyšují pruchodnost transakcního systému. Zámky mohou být rozdeleny na zámky pro ctení a zámky pro zápis. Zámky pro ctení zabezpecují, že po dobu, co je objekt uzamcen, nedojde k jeho zmene. Jeden objekt muže být uzamcen zároven nekolika zámky pro ctení, protože po celou dobu uzamcení nebude zmenen, nemuže však být zároven uzamcen pro zápis. Objekt uzamcený pro zápis nemuže být zároven uzamcen jakýmkoliv jiným zámkem. Toto schéma je casto oznacováno termínem "many readers / single writer" (více ctenáru / jeden zapisovatel).

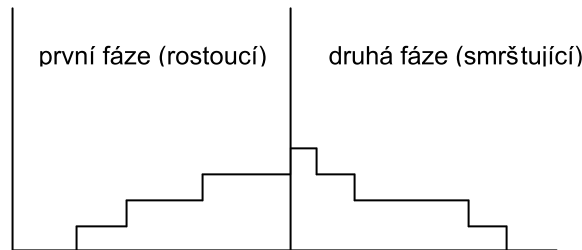
	Read	Write
Read	OK	-
Write	-	-

Tab. 13 - Kompatibilita zámku

Dalším hlediskem, které ovlivnuje celkový výkon systému je **granularita zámku**. Zámky mohou být vztaženy bud na celý soubor nebo databázi (hrubá granulace) anebo na jeden záznam ci blok souboru (jemná granulace). Cím je granulace jemnejší, tím je zámek presnejší a muže být dosaženo vetšího stupne paralelismu. Na druhé strane jemne granulované zámky vyžadují vetší režii a snadneji dochází k deadlockum.

5.4.2 Dvoufázové uzamykání

Uzamykání a odemykání objektu v okamžiku potreby muže vést k nekonzistencím a deadlockum. Proto je vetšina transakcních systému implementována pomocí dvoufázového uzamykání (two-phase locking, 2PL). V dvoufázovém uzamykání jsou v první fázi všechny požadované objekty uzamceny, poté jsou na nich provedeny požadované operace, a nakonec, v druhé fázi, jsou všechny zámky odstraneny. Druhá fáze se kvuli konzistenci dat casto provádí až pri ukoncení transakce, at již úspešném nebo neúspešném. Lze dokázat, že dvoufázove uzamykaná transakce vždy splnuje podmínku sekvencnosti.



Obr. 33 - Dvoufázové uzamykání

I dvoufázové uzamykání muže vést k deadlockum - když se dve transakce pokoušejí uzamknout stejný objekt, avšak v opacném poradí. Obvyklá technika v tomto prípade je uzamykání objektu v nejakém jednotném usporádání. Dalšími možnými metodami odstranení deadlocku jsou explicitní udržování grafu uzamykání a hledání cyklu nebo pomocí timeoutu.

5.4.3 Optimistická kontrola konkurence

Zcela jiný prístup ke kontrole konkurence je tzv. optimistická kontrola konkurence (Kung & Robinson). Idea této techniky je prostá - každá transakce provádí cokoliv bez ohledu na jakékoliv jiné transakce. Pouze v prípade, že dojde ke konfliktu, je transakce abortována.

Implementace této metody je nejvýhodnejší pomocí lokálního pracovního prostoru. Každá transakce ve svém lokálním prostoru provádí všechny požadované zmeny. V okamžiku commitu se zkontroluje, zda k nekterým objektum nepristupovala zároven jiná transakce. V prípade, že došlo ke konfliktu, se lokální pracovní prostor bez náhrady zruší a transakce se provede znovu.

Velkou výhodou optimistické kontroly konkurence je to, že nedochází k deadlockum. Zároven v prípade rídkých konfliktu umožnuje maximální stupen paralelismu. Nevýhodou této metody je skutecnost, že v prípade vetšího zatížení a tím pádem vetší pravdepodobnosti konfliktu se príliš mnoho transakcí provádí nekolikrát, dokonce pri velkém zatížení muže dojít k zahlcení systému.

5.4.4 Casové znacky

Dalším možným rešením problému konkurencního prístupu transakcí k spolecným datum je použití casových znacek. Každá transakce pri startu dostane svoji casovou znacku T. Každému souboru jsou prirazeny dve casové znacky TR a TW, které urcují transakci (resp. její casovou znacku), která ze souboru naposledy cetla, resp. do nej zapisovala.

Pred prístupem k souboru transakce porovná svoji casovou znacku s casovými znackami souboru. V prípade, že casová znacka transakce je vetší (tj. transakce je mladší), je vše v porádku. V opacném prípade mohlo dojít ke konfliktu.

	T<TR	T<TW
ctení	OK, novejší transakce cetla správná data	abort, data jsou prepsána novejší transakcí
zápis	abort, novejší transakce cetla stará data	OK, ale data nezapisovat, novejší transakce už zapsala nová data

Tab. 14 - Kombinace možných konfliktu casových znacek pro ctení a zápis

Kombinace možných konfliktu jsou uvedeny v tabulce. Nejzajímavejší je situace, kdy starší transakce chce zapisovat do souboru, který je již prepsán novejší transakcí. V tomto prípade se transakce abortovat nemusí a zápis do souboru se ignoruje, nebot soubor by byl stejne prepsán novejší transakcí.

Casové znacky mají oproti uzamykání jednu podstatnou výhodu - nedochází zde k deadlockum. Na druhou stranu mají ponekud odlišné chování - tam, kde pri zamykání stací pockat, dochází u casových znacek k abortum.

6. Distribuovaná sdílená pamet (DSM)

Page based / Shared variables / Object based DSM

6.1 Konzistencní modely

Znacení:

W(x)a - zápis hodnoty a do promenné x
R(x)a - pri ctení z promenné x je vrácena hodnota a
S - synchronizace
Acq - vstup do kritické sekce
Rel - výstup z kritické sekce
Pi – proces i

6.1.1 Striktní konzistence (strict consistency)

Jakékoliv ctení z pameti z adresy x vrátí hodnotu uloženou pri posledním zápisu na adresu x.

Nejsilnejší, na jednoprocesorových systémech je tradicne zajištena:

Pr. 1:

```
a=1; a=2; print(a);  // vytiskne vždy 2
```

Všechny zápisy jsou **okamžite** všude viditelné, musí existovat presný globální cas. Ideální pro programování, v distribuovaném systému však nedosažitelné.

Pr. 2:

P1:	W(x)1	P1:	W(x)1
P2:	R(x)1	P2:	R(x)0 R(x)1
Striktní konzistence		Pamet, která není striktne konzistentní	

6.1.2 Sekvencní konzistence (sequential consistency)

Výsledek jakéhokoliv výpoctu je stejný jako kdyby všechny operace všech procesoru byly vykonávány v nejakém sekvencním usporádání a operace každého jednotlivého procesoru jsou vykonávány v poradí specifikovaném programem.

Sekvencní konzistence je o neco slabší model, avšak je snadno implementovatelná a príjemná pro programování. Jestliže procesy beží na ruzných procesorech, je povoleno libovolné prokládání jejich instrukcí, avšak s podmínkou, že všechny procesy vidí stejné poradí zmen pameti. Zmeny nejsou propagovány okamžite, nic se nemluví o velikosti zpoždení (sec, min), je pouze zarucena kauzalita.

Pr. 3:

P1: W(x)1		P1: W(x)1
P2: R(x)1 R(x)1		P2: R(x)0 R(x)1

Dvakrát spuštený tentýž program

Pr. 4:

```
P1:              P2:              P3:
a=1;             b=1;             c=1;
print(b,c);      print(a,c);      print(a,b);
```

šest instrukcí - 6!=720 možných sekvencí, pouze 3*(5!/4)=90 nenarušuje konzistenci

```
a=1;             a=1;             b=1;             b=1;
print(b,c);      b=1;             c=1;             a=1;
b=1;             print(a,c);      print(a,b);      c=1;
print(a,c);      print(b,c);      print(a,c);      print(a,c);
c=1;             c=1;             a=1;             print(b,c);
print(a,b);      print(a,b);      print(b,c);      print(a,b);

Výstup:
001011           101011           010111           111111

Signatura:
001011           101011           110101           111111
```

Jestliže spojíme výstupy procesu P1, P2 a P3 (v tomto poradí), dostaneme **signaturu** - retezec charakterizující konkrétní proložení instrukcí jednotlivých procesu, a tím i poradí pametových referencí.

Celkem muže existovat 2^6=64 možných signatur, avšak ne všechny odpovídají sekvencní konzistenci - napr. 000000, 001001 neodpovídají žádnému sekvencnímu proložení instrukcí

Pr. 5:

P1: W(a)1 R(b)0 R(c)0

P2: W(b)1 R(a)1 R(c)0

P3: W(c)1 R(a)0 R(b)1

Signatura 001001 je nemožná v sekvencne konzistencním modelu

Sekvencní konzistence je príjemný model pro programování, avšak jeho výkonnost není príliš velká. Bylo dokázáno (Lipton & Sandberg, 1988) že je-li cas ctení *r*, cas zápisu *w* a cas prenosu zprávy (paketu) *t*, pak vždy platí $r+w?t$. Jinak receno - optimalizace protokolu pro ctení znamená delší cas pro zápis a naopak.

6.1.3 Kauzální konzistence (causal consistency)

Zápisy, které jsou potenciálne kauzálne vázané, musí být videny všemi procesy ve stejném poradí. Konkurencní zápisy mohou být videny v ruzném poradí.

Rozlišuje události, které jsou potenciálne závislé a které nikoliv. Pokud jeden proces W(x)a a druhý proces R(x) W(y)b pak W(y)b je kauzálne závislý na W(x)a. Operace, které nejsou kauzálne závislé jsou konkurentní.

Pr. 6:

P1:	W(x)1			W(x)3		
P2:		R(x)1	W(x)2			
P3:		R(x)1			R(x)3	R(x)2
P4:		R(x)1			R(x)2	R(x)3

Kauzální konzistence

Príklad 6 zobrazuje situaci, kdy W(x)2 a W(x)3 jsou konkurentní (tj. nejsou kauzálne závislé). Procesy P3 a P4 mohou potom zmenu x císt v opacném poradí. Tato sekvence splnuje podmínku kauzální konzistence, nesplnuje sekvencní a striktní.

Pr. 7:

P1:	W(x)1			
P2:		R(x)1 W(x)2		
P3:			R(x)2	R(x)1
P4:			R(x)1	R(x)2

PRAM konzistence

P1:	W(x)1			
P2:		W(x)2		
P3:			R(x)2	R(x)1
P4:			R(x)1	R(x)2

Kauzálne konzistentní sekvence

Na druhém podobrázku príkladu 7 není W(x)2 kauzálne závislý na W(x)1 a proto obrácené ctení R(x)1 a R(x)2 je vzhledem ke kauzální konzistenci korektní. Na prvním podobrázku je W(x)2 kauzálne závislé na W(x)1 (pres R(x)1) a proto obrácené poradí ctení porušuje kauzální konzistenci.

Implementace kauzálne konzistentní pameti vyžaduje udržování grafu závislostí zápisu na ctení.

6.1.4 PRAM konzistence (PRAM consistency)

Zápisy provádené jedním procesem jsou videny ostatními procesy v tom poradí, ve kterém byly provádeny, avšak zápisy ruzných procesu mohou být videny ruznými procesy ruzne.

PRAM (Pipelined RAM) konzistence je snadná na implementaci - nezáleží na poradí, v nemž ruzné procesy vidí prístupy k pameti. Jediné, co je treba dodržet, je poradí zápisu z jednoho zdroje. Jinak receno - všechny zápisy generované ruznými procesory jsou konkurentní.

Pr. 8:

```
P1:                     P2:
a=1;                    b=1;
if(b==0) kill(P2);      if(a==0) kill(P1);
```

V prípade PRAM konzistence je možný výsledek, že oba procesy budou zabity, a to v prípade, že P1 precte b dríve, než uvidí jeho zápis a P2 precte a dríve, než uvidí jeho zápis. Tento výsledek není možný pri libovolném usporádání instrukcí, neodpovídá tedy sekvencní konzistenci, avšak vzhledem k PRAM konzistenci je korektní.

6.1.5 Slabá konzistence (weak consistency)

Výše uvedené konzistencní modely jsou stále pro mnoho aplikací velmi restriktivní (a tedy málo efektivní), nebot vyžadují propagaci všech zápisu všem procesum. Ne všechny aplikace vyžadují sledování všech zápisu, natož pak nejaké jejich poradí. Napr. proces v kritické sekci muže v rychlé smycce císt a zapisovat hodnoty nejakých promenných. Ostatní procesy nemají duvod (nebo ani možnost) jednotlivé zápisy videt, proto není nutné, aby byly všechny propagovány. Pamet však nemá možnost vedet, že nejaký proces je v kritické sekci a tudíž musí propagovat všechny zápisy.

Rešením je nechat proces ukoncit kritickou sekci a poté zajistit rozeslání zmen všem ostatním procesum. Toho muže být dosaženo použitím speciálního druhu promenné - **synchronizacní promenné**, která bude použita pro synchronizacní úcely.

1. **Prístup k synchronizacním promenným je sekvencne konzistentní.**
2. **Prístup k synchronizacní promenné není povolen dokud neskoncí všechny predchozí zápisy.**
3. **Prístup k datum není povolen dokud nebyly dokonceny všechny predchozí prístupy k synchronizacním promenným.**

Bod 1 ríká, že všechny procesy vidí všechny prístupy k synchronizacní promenné ve stejném poradí. Bod 2 zarucuje, že pred prístupem k synchronizacní promenné budou dokonceny všechny predchozí zápisy. Bod 3 ríká, že pri prístupu k obycejným promenným jsou dokonceny všechny predchozí prístupy k synchronizacním promenným. Provedením synchronizace pred ctením dat proto zajistí jejich aktuální verzi.

Pr. 9:

P1: W(x)1 W(x)2 S		
P2:		R(x)2 R(x)1 S
P3:		R(x)1 R(x)2 S

Slabá konzistence

P1: W(x)1 W(x)2 S		
P2:		S R(x)1

Porušení slabé konzistence

Na prvním podobrázku prípladu 9 jsou události odpovídající slabé konzistenci - procesy P2 a P3 mohou pred synchronizací videt pamet v libovolném poradí. Druhý podobrázek ukazuje porušení slabé konzistence - proces P2 po synchronizaci vidí neaktuální hodnotu promenné x.

6.1.6 Uvolnovací konzistence (release consistency)

Slabá konzistence má tu nevýhodu, že pamet pri prístupu k synchronizacní promenné nepozná, zda se jedná o vstup nebo o výstup z kritické sekce. Pokaždé proto musí vykonat akce potrebné pro oba prípady. K rozlišení techto prípadu používá uvolnovací konzistence dve operace - acquire (požadavek) a release (uvolnení).

1. **Pred bežným prístupem ke sdílené promenné musí být úspešne ukonceny predchozí požadavky procesu.**
2. **Pred provedením uvolnení musí být ukonceny všechny predchozí zápisy i ctení provádené procesem.**
3. **Požadavky a uvolnení musí být PRAM konzistentní.**

Jestliže proces vydá požadavek, pak je zaruceno, že poté jsou všechny lokální kopie aktuální. Po uvolnení jsou propagovány všechny zmeny ostatním procesum. Jestliže jsou všechny podmínky splneny a požadavky a uvolnení se používají správe spárované, výsledek jakéhokoliv výpoctu je ekvivalentní sekvencne konzistentní pameti.

Pr. 10:

P1:	Acq W(x)1 W(x)2 Rel		
P2:		Acq R(x)2 Rel	
P3:			R(x)1

Uvolnovací konzistence

Eager release consistency - po release se propagují zmeny všem procesum.

Lazy release consistency - po release se nic nepropaguje, až pri acquire jiného procesu - casové znacky.

6.1.7 Prístupová konzistence (entry consistency)

1. **Požadavkový prístup procesu k synchronizacní promenné není povolen dokud nebyly provedeny všechny aktualizace chránených sdílených dat procesu.**
2. **Exkluzivní prístup procesu k synchronizacní promenné je povolen pouze v prípade, že žádný jiný proces nepristupuje k synchronizacní promenné, a to ani neexkluzivne.**
3. **Po exkluzivním prístupu k synchronizacní promenné si príští neexkluzivní prístup libovolného procesu k synchronizacní promenné musí vyžádat aktuální kopii dat od vlastníka synchronizacní promenné.**

Všechna sdílená data jsou vázána na synchronizacní promennou. Pri prístupu k synchronizacní promenné jsou synchronizována pouze ta data, která jsou vázána na synchronizacní promennou.

Každá synchronizacní promenná má v každém okamžiku vlastníka - proces, který k ní naposledy pristupoval. Vlastník muže opakovane vystupovat do a vystupovat z kritické sekce bez nutnosti komunikace. Proces, který není vlastníkem, musí požádat o vlastnictví. Pri prenosu vlastnictví se aktualizují hodnoty dat vázaných na synchronizacní promennou. Prístup k datum a synchronizacním promenným muže být exkluzivní (read & write) nebo neexkluzivní (read only).

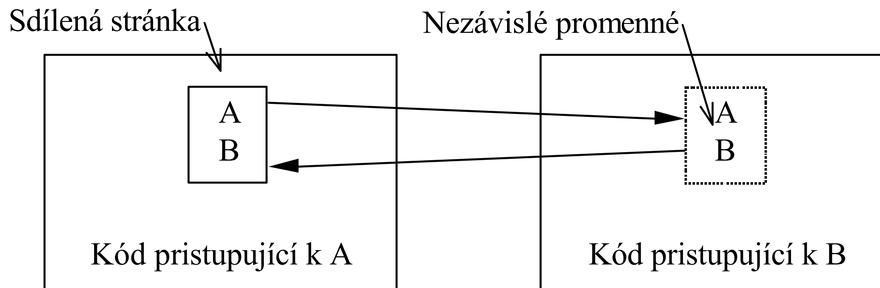
6.1.8 Shrnutí konzistencních modelu

	Konzistence	Vlastnosti
a	Striktní	Absolutní casové usporádání
	Sekvencní	Všechny události jsou videt ve stejném poradí
	Kauzální	Kauzálne vázané události jsou videt ve stejném poradí
	PRAM	Události jednoho procesu jsou videt ve stejném poradí
b	Slabá	Sdílená data jsou konzistentní po synchronizaci
	Uvolnovací	Sdílená data jsou konzistentní po opuštení kritické sekce
	Prístupová	Sdílená data vázaná na kritickou sekci jsou konzistentní pri vstupu do kritické sekce

Tab. 15 - Prehled základních konzistencních modelu
(a) bez synchronizacních operací (b) se synchronizacními operacemi

6.2 Distribuované stránkování

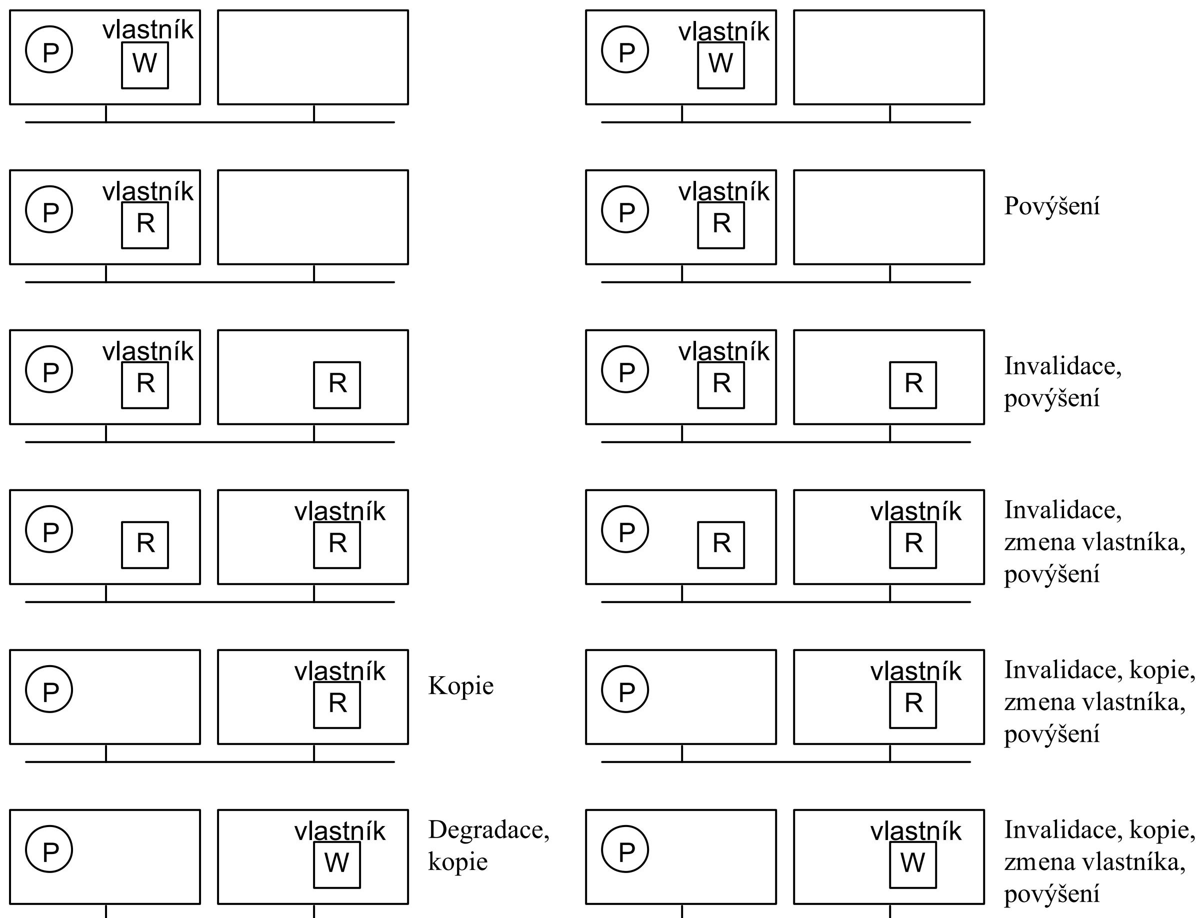
Základní schéma distribuovaného stránkování je obdobou virtuální pameti - je-li referencována stránka, která není na lokálním stroji, je vyvoláno prerušení a stránka musí být nactena ze stroje, kde se práve nalézá. Problémy: replikace -> konzistence, nalezení stránky, správa kopií, falešné sdílení.



Obr. 34 - Falešne sdílená stránka obsahující nezávislé promenné A a B

Jestliže jsou stránky (predevším z duvodu výkonnosti) replikovány, je treba udržovat konzistenci dat. Typická implementace je namapování všech stránek read-only a v prípade zápisu provádet potrebné synchronizacní akce. Teoreticky jsou možné dva zpusoby - invalidace nebo aktualizace. Distribuované stránkování v naprosté vetšine používá invalidaci - vzhledem k velikosti prenášených dat a dobe prenosu.

Na obrázku jsou rozebrány možnosti umístení a vlastnictví stránky pri ctení a zápisu a potrebné synchronizacní akce.



Obr. 35 - (a) ctení ze stránky (b) zápis do stránky

6.2.1 Nalezení vlastníka stránky:

- broadcast
- centralizovaný manager
- replikovaný manager - indexovaný spodními n bity adresy stránky / hash

6.2.2 Nalezení kopií:

- broadcast
- vlastník stránky udržuje copyset - množinu lokací kopií

6.2.3 Alokace stránek:

1. not owned read copy
2. owned replicated copy - transfer of ownership
3. local policy (LRU)

6.3 Distribuované sdílené promenné

Implementace na úrovni knihoven - potenciálne replikovaná distribuovaná databáze.

Výhody: potenciálne lepší výkonnost, eliminace falešného sdílení.

Nevýhody: Nepodporováno prímo operacním systémem, nutnost implementace pro ruzné jazyky, nutnost rekompilace.

6.3.1 Munin

ordinary / shared / synchronisation variables

Eager release consistency - pri opuštení kritické sekce se propagují zmeny.

sdílené promenné: read-only, migratory, write-shared, conventional

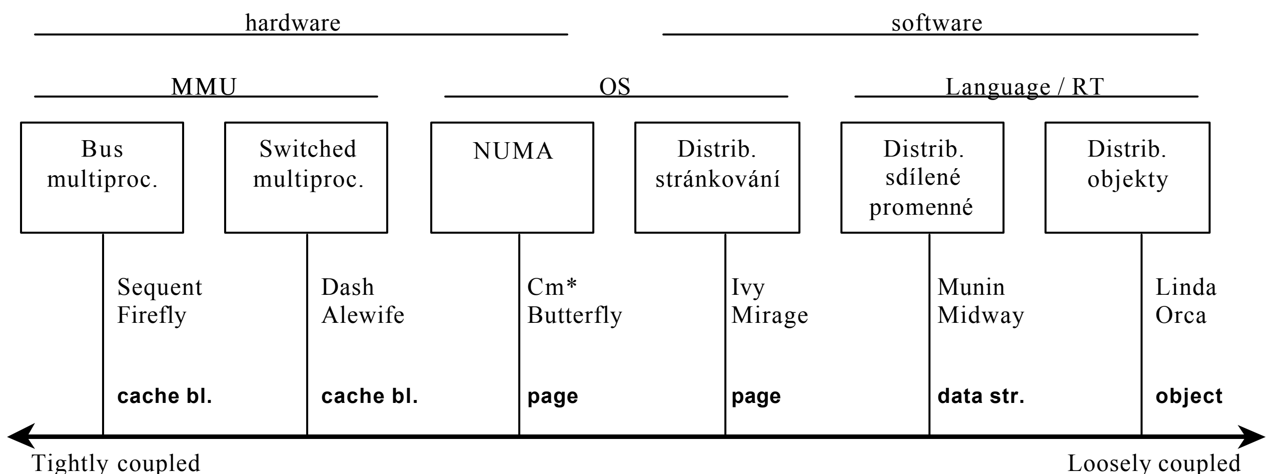
- read-only: když dojde k výpadku, v adresári promenných je nalezen vlastník, který je požádán o read-only kopii dat.
- migratory: acquire/release protokol implementující release consistency. Data chránená synchronizacními prístupy migrují na stroj, který je zrovna v kritické sekci.
- write-shared: všechny stránky jsou iniciálne read-only, pri zápisu stránka vytvorí kopii s puvodním obsahem a zmení se na rw a oznací se dirty. Po release se porovná stránka s puvodní a zmeny se propagují. Když prijde propagace zmen na ne-dirty stránku, zmeny se akceptují, v opacném prípade se word-po-wordu zkontroluje, zda nedošlo ke konfliktu. Když ne, data se sjednotí, když ano, dojde k runtime-erroru.
- konvencní sdílená data (nepatrící do žádné z výše uvedených kategorií): chovají se jako distribuované stránkování - single writer / many readers a sekvencní konzistence.

6.4 Distribuované objekty

Díky enkapsulaci flexibilnejší - komunikace a synchronizace v metodách

Class Definition Language - automatické generování hlavicek a kódu

Základní "distribuovaná" trída, dedení vlastností, CORBA



Obr. 36 – Mechanismy sdílení pameti

7. Identifikace objektu

Komunikace mezi procesy a využívání vzdálených objektu vyžaduje, aby jádro systému zabezpecilo jednotné pojmenování - identifikaci objektu. Objektem je zde mínena libovolná entita, ke které má být zajišten prístup. Objekty mohou být aktivní (napr. procesy) nebo pasivní (napr. tiskárny). Pasivní objekt bývá spravován nejakým aktivním objektem.

K tomu, aby nejaký objekt mohl být prístupný, je zapotrebí vedet který objekt má být použit (identifikace), kde je tento objekt umísten (adresa) a jak je možné se k nemu dostat (cesta). Identifikacní systém musí být navržen efektivne, aby prodleva a pocet zpráv mezi zapocetím komunikace a vlastním prenosem zpráv byly co nejmenší. Vlastní komunikace muže zacít teprve poté, co je z jména nejakým zpusobem odvozena cesta k danému objektu. Adresa je casto chápána jako mezistav potrebný k urcení požadované cesty.

7.1 Identifikacní systém

Identifikacní systém má za úkol pridelovat jména, mapovat jména na adresy a starat se o nalezení nejefektivnejší cesty. Mel by podporovat alespon dve úrovne jmen - systémové a uživatelské identifikátory. Systémové identifikátory objektu typu `0C03A489F20` jsou obvykle pro bežné uživatele necitelné a naopak, uživatelský identifikátor typu '`Toto je dopis mému šéfovi`' je pro bežnou práci systému príliš nepohodlný. Systém tedy musí sám vhodným zpusobem prevádet uživatelská jména na systémová a naopak.

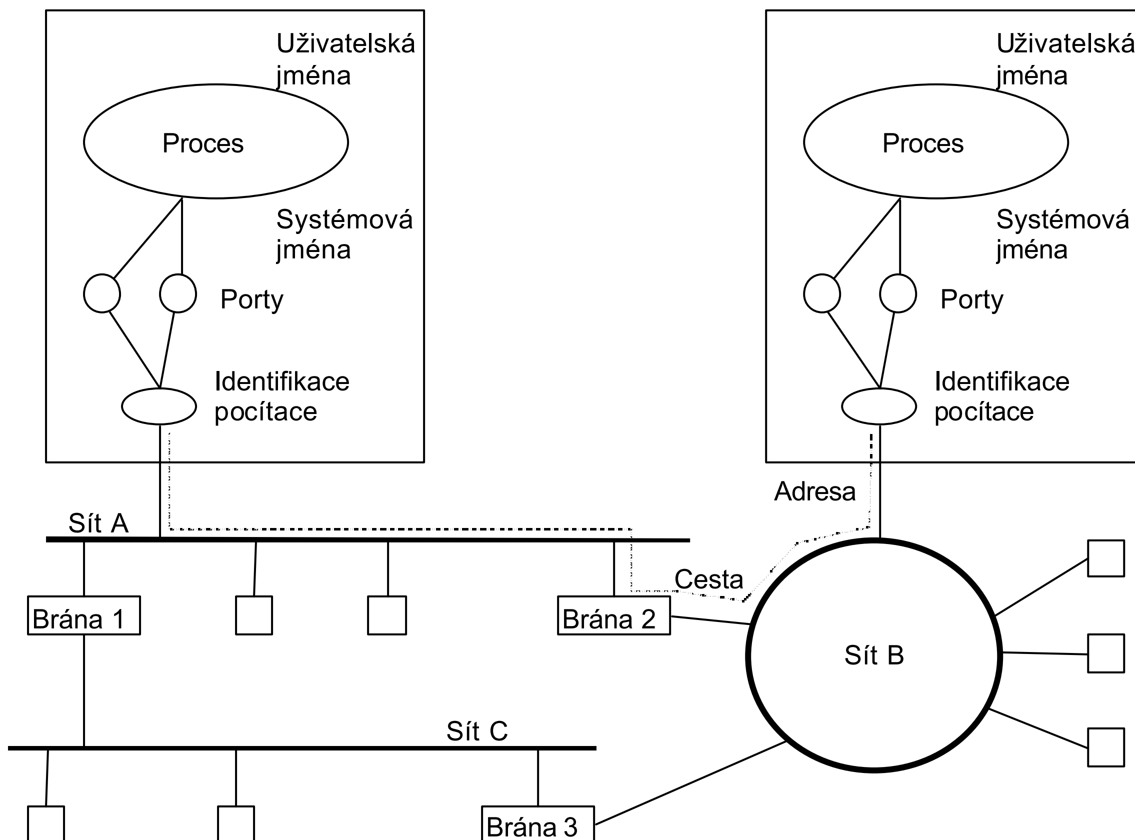
Prostor jmen musí být udržován decentralizovane, tj. identifikacní systém každého pocítace musí jména pridelovat jen na základe lokálních informací. Takto pridelovaná jména musí být jednoznacná, tj. žádné dva objekty nemohou mít stejné jméno. Proto jsou jména jednotlivými jádry pridelována bud podle nejakého pevne daného algoritmu, nebo jsou jednotlivým pocítacum pridelovány oblasti prostoru jmen, které muže využívat.

Prostor jmen muže být bud jednotný pro všechny objekty systému, nebo mohou být pro ruzné druhy objektu separátní prostory jmen. Napr. v klasických centralizovaných operacních systémech bývá prostor jmen souboru oddelený od identifikace ostatních objektu. Naopak v objektove orientovaných systémech bývá castejší jednotný prostor jmen.

Nekteré distribuované systémy podporují migraci objektu. U takových systému musí být jména objektu nezávislá na aktuálním umístení, resp. vazba mezi jménem a adresou musí být dynamická. V prípade presunu objektu na jiný pocítac musí být transparentne zjištena nová adresa a nalezena prístupová cesta k objektu.

Identifikacní systém by mel podporovat vícenásobné kopie (repliky) daného objektu. Vazba mezi uživatelským jménem a adresou tedy není 1:1, ale 1:N. V prípade nedostupnosti jedné kopie by mel systém transparentne namapovat jinou dostupnou kopii téhož objektu. Systém by také mel umožnovat ruzná lokální uživatelsky definovaná pojmenování pro stejné objekty a zároven nekolika ruzným objektum (skupine objektu) umožnit sdílet jedno uživatelské jméno. Celkove je tedy mapování mezi uživatelskými jmény a adresami objektu typu M:N.

Návrh a konkrétní implementace identifikacního systému je vždy kompromisem mezi výkonností, pružností, snadností mobility objektu, pametovou nárocností a snadností uchování a údržby konzistence.



Obr. 37 - Hierarchie identifikací v distribuovaném systému

7.1.1 Jména

Jméno jednoznacne identifikuje objekt nebo skupinu objektu. Muže to být bud textový retezec nebo nejaká binární hodnota. Binární jména jsou vhodnejší pro použití jako systémové identifikátory, retezcová jména jsou naopak vhodnejší pro uživatelskou identifikaci.

Jména mohou být rozdelena podle nekolika kritérií:

?? podle struktury - nestrukturovaná, strukturovaná, popisná

?? podle trvanlivosti - statická (trvalá), dynamická (docasná)

?? podle poctu referencovaných objektu - individuální, skupinová

?? podle rozsahu platnosti - globální, lokální pro uzel, lokální pro proces

Statická jména trvale oznacují nejaký objekt. Dynamická jména jsou naopak prirazována objektum jen (relativne k životnosti objektu) po krátký cas - napr. po dobu komunikace s jiným objektem. Dynamická jména jsou v naprosté vetšine systémová - systém je sám vytvárí a po ukoncení doby platnosti je ruší.

Individuální jména jsou jména, která jednoznacne identifikují jeden objekt; všechny objekty mají navzájem ruzná jména. Pro úcely skupinové komunikace, spolupráce replikovaných objektu apod. jsou casto využívána skupinová jména, která identifikují nejakou presne vymezenou skupinu.

7.1.2 Struktura jmen

Jména mohou být podle své struktury rozdelena na nestrukturovaná (flat), strukturovaná (partitioned) a popisná (descriptive).

Nestrukturovaná jména nemají, jak již název napovídá, žádnou vnitrní strukturu, avšak v rámci svého prostoru jmen jednoznacne identifikují objekty. Nestrukturovaná jména bud prideluje centralizovaný server jmen, nebo se vypocítávají podle nejakého algoritmu zajištujícího jednoznacnost.

Další, v poslední dobe casto využívanou, možností je pridelovat jména náhodne. V prípade, že šírka jména, tj. pocet bajtu, které jméno tvorí, je dostatecne velká a generátor náhodných císel spolehlivý, je pravdepodobnost, že dva objekty dostanou prideleno stejné jméno, velice nízká. Pri binárních jménech o šírce 64 bitu (8bajtu) se stejné jméno jedním uzlem vygeneruje jednou za 10^{19} cyklu. I pri rychlosti generování milion adres ze vterinu by to trvalo 10^{13} vterin, což je asi pul milionu let. I kdyby adresy generovalo neustále paralelne tisíc pocítacu, trvalo by zhruba 500 let, než by vygenerovaly stejnou adresu.

Strukturovaná jména jsou obvykle složena z posloupnosti jednoduchých jmen oddelených separátory. Jednotlivé cásti jména pak udávají oblast, podoblast, ... a tak dále až konecné (bud první nebo poslední) jméno udává identifikaci objektu v jeho lokálním prostoru. Separátory bývají speciální znaky jako napr. ‘@’, ‘.’, ‘/’, ‘%’ apod., tyto znaky pak nemohou být soucástí cástecného jména. Príkladem strukturovaného jména je internetová adresa, která muže být tvaru `uživatel@oddelení.organizace.oblast.stát`, kde cást jména pred znakem @ je identifikace koncového uživatele, cást jména za znakem @ hierarchicky urcuje adresu uživatele. Jiným príkladem strukturovaného jména je identifikace souboru v UNIX-like systémech. Jméno tvaru `/adresár1/adresár2/adresár3/soubor` urcuje cestu z korenového adresáre k souboru a identifikaci souboru ve svém adresári.

Strukturovaná jména mohou být absolutní a relativní. Absolutní jméno specifikuje celou cestu od nejakého korene až ke koncovému objektu. Absolutní jména by mela být stejná z libovolného uzlu systému. Relativní jména mohou nejvyšší cásti cesty vynechat, ty mohou být za urcitých okolností z kontextu zrejmé. Tímto zpusobem napr. funguje systém aktuálních adresáru - ty soubory, které leží v adresárovém strome pod aktuálním adresárem, mohou být adresovány relativne z tohoto adresáre; cesta od korene k aktuálnímu adresári muže být vypuštena. Relativní jména je však treba používat opatrne, nebot zmenou kontextu se jména mohou stát neplatnými anebo ukazovat na jiný objekt. Absolutní jméno je v podstate relativní jméno od korenového kontextu. V nekterých systémech však koren nemusí být jednoznacný, resp. korenu muže být nekolik. V tom prípade prostor jmen netvorí strom, ale les, v obecnejším prípade dokonce libovolný orientovaný graf.

Popisná jména jsou tvorena množinou atributu, která je pro každý objekt jednoznacná. Podle této definice jsou strukturovaná jména zvláštní formou popisných jmen s pevnou strukturou atributu.

Atributy objektu mají typ a hodnotu, kde typ oznacuje formát a význam atributu. Príkladem atributového jména muže být

`Jméno: Novák; Oddelení: KSI; Organizace: MFF UK; Místo: Praha`

Sjednocením nekolika atributu dostaneme jednoznacnou identifikaci objektu presto, že jednotlivé atributy mohou být nejednoznacné. Popisná jména mohou být snadno použita i jako skupinová jména.

7.1.3 Cesty

Distribuovaný systém bývá vetšinou složen z nekolika propojených sítí. K tomu, aby jednotlivé objekty spolu mohly komunikovat, je treba znát cestu, kudy zpráva musí projít. Cestou rozumíme posloupnost jmen (adres), které urcují jednotlivé úseky mezi pocátecním a koncovým uzlem. Algoritmu, který mapuje jména na cesty ríkáme smerování (routing). Smerovací algoritmy mužeme rozdelit podle nekolika hledisek:

?? podle místa, kde se provádí rozhodování o dalším úseku cesty

?? podle frekvence zmen dat potrebných pro smerování

?? podle zpusobu spolupráce jednotlivých uzlu

Jsou tri možné zpusoby, kde se muže provádet rozhodování o dalším úseku cesty:

?? zdrojové smerování

?? postupné smerování

?? záplavové smerování

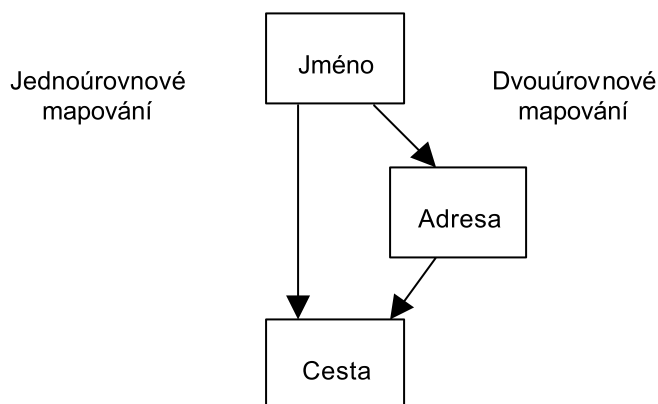
Podle frekvence zmen potrebných dat lze smerovací algoritmy rozdelit na statické (pevné) a dynamické (adaptivní).

Centralizované / lokální / distribuované smerování

7.1.4 Adresy a mapování

Adresa muže být považována za jakýsi mezistav mezi jménem a cestou. Na rozdíl od jména je adresa pevne svázána s umístením objektu. Jméno jednoznacne identifikuje objekt, kdežto adresa jednoznacne identifikuje nejaké umístení. Podobne jako jména, i adresy mohou být strukturované nebo nestrukturované - napr. trojice `<sít, uzel, soket>` jednoznacne urcuje adresu.

Mapování jmen na cesty muže být bud prímé (jednoúrovnové), kdy je cesta vygenerována bezprostredne ze jména, nebo zprostredkované (dvouúrovnové), kdy je nejprve jméno namapováno na adresu a poté adresa namapována na cestu.



Obr. 38 - Jednoúrovnové a dvouúrovnové mapování jmen na adresy a cesty

7.2 Systémová jména

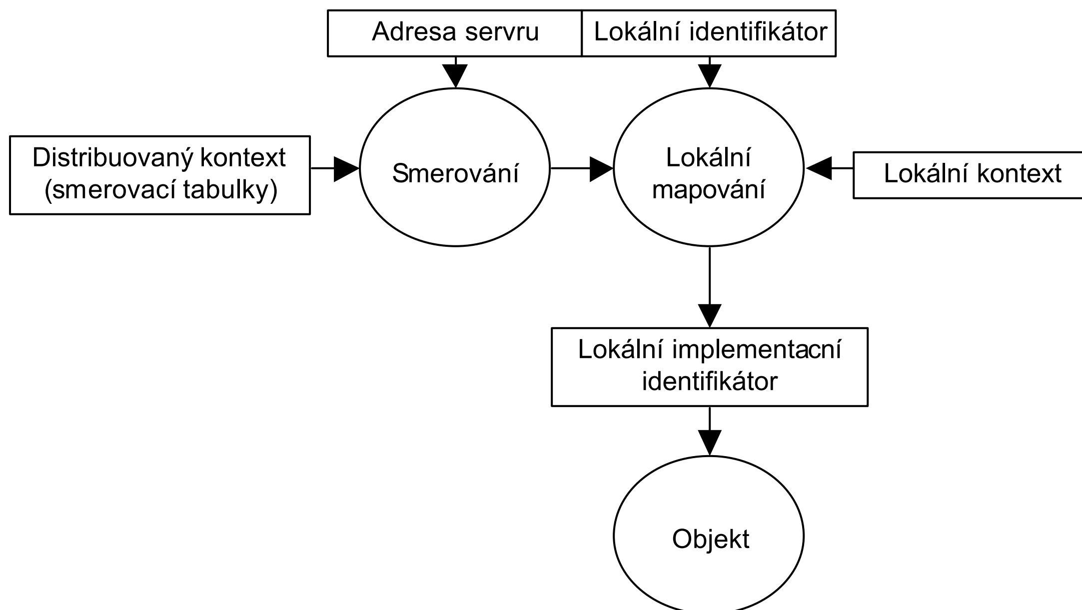
Systémová jména jsou generována a využívána systémem k vnitrní identifikaci objektu. Vetšinou to jsou binární retezce pevné délky, které jsou pro uživatele necitelné. Systémová jména jsou založena bud na jednoznacných interních identifikátorech nebo na speciálních druzích jmen, které poskytují podporu ochrany objektu, tzv. kapabilitách.

7.2.1 Interní identifikátory

Jednoznacné interní identifikátory jsou v distribuovaných systémech casto využívány. Typicky jsou tyto identifikátory nestrukturované, mohou však být i strukturované.

Nezávažnejším problémem nestrukturovaných interních identifikátoru je správa prostoru jmen - vytvorení globálního identifikátoru a zarucení jeho jednoznacnosti.

7.2.2 Lokálne implementacní identifikátory



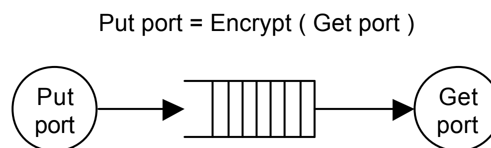
Obr. 39 - Mapování jednoznacného globálního identifikátoru

7.2.3 Mapování systémových jmen

Identifikace portu

?? globální - nutnost ochrany

?? lokální pro proces - ochrana vetšinou jádrem



Obr. 40 - Vstupní a výstupní port

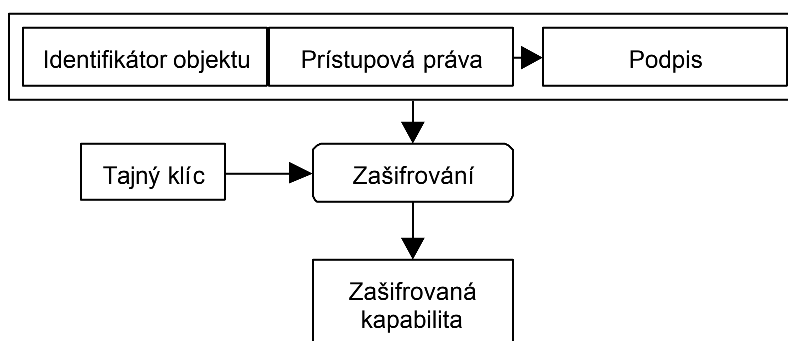
7.3 Kapability

Speciálním druhem systémových jmen jsou tzv. kapability (capabilities). Kapabilita je zvláštní "lístek" (bumážka) umožnující jednoznacnou identifikaci objektu a zároven obsahující množinu prístupových práv, která umožnují držiteli kapability prístup k objektu a operace nad ním.

Z duvodu bezpecnosti musí být uživatelským procesum znemožneno vlastní generování kapabilit jakož i zmeny prístupových práv. Nejcasteji používaný zpusob implementace je založen na rídkosti kapabilit a kódování.

7.3.1 Kapability s podpisem

Platná kapabilita je sestavena z dvou cástí - identifikace objektu a prístupových práv. Platná kapabilita je rozšírena o podpis, který je vypocítán z obsahu hlavní cásti. Trojice <objekt, práva, podpis> je platná jen tehdy, pokud podpis odpovídá hlavní cásti kapability. Aby se zabránilo uživatelským procesum odvodit podpisovou funkci, je celá kapabilita zašifrována tajným klícem. Rídkost je zde chápána v tom smyslu, že ze všech binárních císel délky velikosti kapability jsou platné jen ty, jejichž podpis odpovídá zbytku kapability.

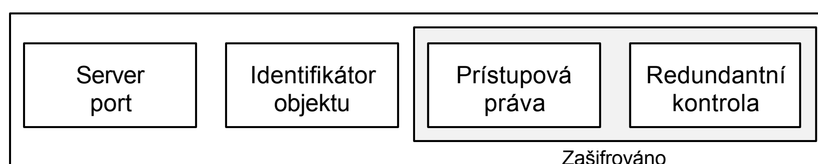


Obr. 41 – Kapabilita

7.3.2 Kapability s redundancí

Jiným zpusobem ochrany kapabilit je redundantní kontrola. Platná kapabilita je složena ze ctyr cástí. První dve cásti, servrovský port a identifikátor objektu, jsou volne prístupné. Další pole obsahující prístupová práva, je spolu s náhodne vygenerovaným binárním císlem pevné délky zakódováno.

Ochrana je zde složena ze dvou cástí - ochrany serveru a ochrany prístupových práv. Ochrana serveru je dána portem, což je dostatecne velké náhodne generované císlo. Proces, který toto císlo nezná, nemuže služeb servru využít. Ochrana prístupových práv je zajištena zašifrováním pole s prístupovými právy spolu s redundantní kontrolou. Tím je uživatelskému procesu znemožnena zmena prístupových práv - proces nemuže svá práva zmenit, nebot nezná dešifrovací funkci, ani ji nemuže (kvuli náhodne generované redundanci) z niceho odvodit.



Obr. 42 - Kapabilita v Amoebe

7.4 Uživatelská jména

7.4.1 Servry jmen

Mapování uživatelských jmen na systémová je typicky provádeno servry jmen (name servers). Lokální name server udržuje veškeré informace o dostupných objektech na každém uzlu systému. Komunikacní složitost je minimální, avšak údržba prostoru jmen a udržování konzistence je znacne nárocné. Proto jsou lokální name servery vhodné pouze pro malé a relativne statické prostory jmen.

Zcela opacným extrémem je centralizovaný name server. V tomto prípade je celý prostor jmen spravován jedním name serverem. Údržba prostoru jmen a udržování konzistence je triviální, ale v prípade výpadku name serveru je celý systém nefunkcní. Zároven pro vetší systémy muže být name server úzkým místem omezujícím výkon systému - v prípade vetšího poctu požadavku je centrální server nemusí být schopen dostatecne rychle zpracovat. Proto se centrální name servery dají rozumne použít pouze u malých systému.

Vylepšením centralizovaného name serveru je hierarchická struktura name serveru. V prípade, že požadavek nemuže být nejakým name serverem vyrešen, je predán name serveru vyšší úrovne. Ten vetšinou došlé požadavky sám nereší, ale zasílá je k vyrešení name serveru nižší úrovne, který je tento požadavek schopen vyrešit.

Distribuované name servery žádnou pevnou organizaci vetšinou neudržují. Každý lokální name server vetšinou udržuje informace o lokálních objektech a pro vzdálené objekty, resp. pro jejich podstromy, udržuje odkaz na vzdálený name server.

Pro minimalizaci komunikace bývají nejcasteji užívaná jména lokálními servery cachována. Jestliže prijde požadavek na rezoluci nejakého jména, pak lokální server prohledá cache, a v prípade nalezení jména ihned odpoví. K predání požadavku vzdálenému name serveru pak dochází jen u jmen, které se do cache již nevešly, nebo které ješte nebyly resolvovány. Jiným zpusobem minimalizace komunikace je replikace údaju o duležitých nebo casto používaných jménech. Takové jméno muže být resolvováno libovolným name serverem, který o nem udržuje údaje. V tomto prípade je však správa prostoru jmen podstatne nárocnejší.

7.4.2 Správa prostoru jmen

name distribution - pridelování autority spravovat cást prostoru jmen

name resolution - výber atributu objektu zadaného jménem

jména lokální / doménová / globální

jména absolutní / relativní / kontextová

7.4.3 Rozklad jmen

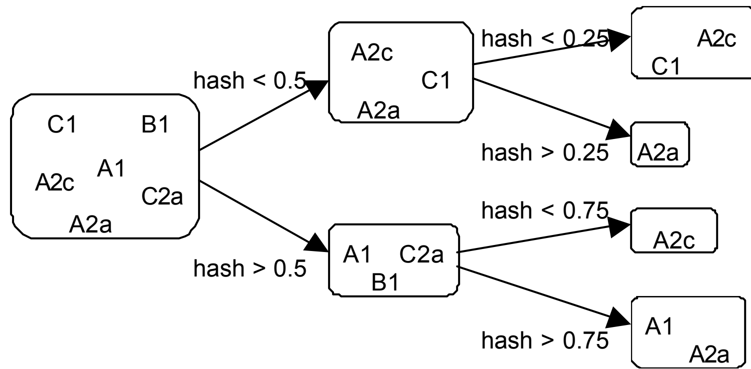
Zpusoby rozkladu:

?? hierarchický podle posloupnosti cástecných jmen

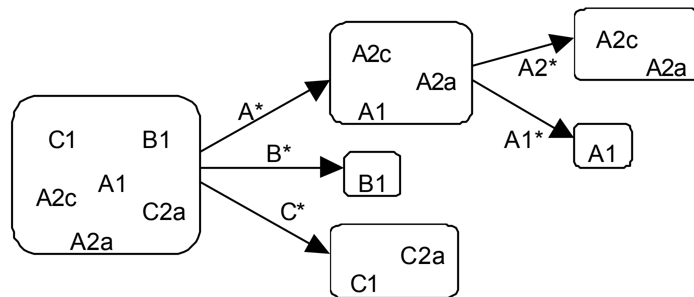
?? aritmetický podle výsledku ohodnocovací funkce

?? syntaktický podle syntaktického rozkladu jména

?? atributový pro popisná (atributová) jména



Obr. 43 - Aritmetický rozklad

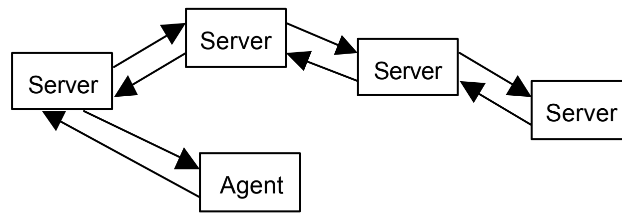


Obr. 44 - Syntaktický rozklad

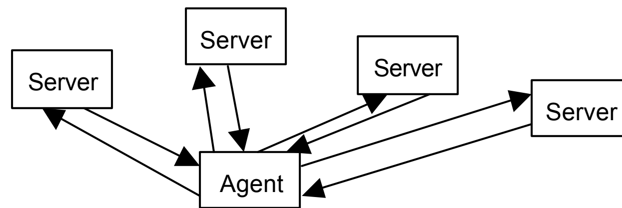
7.4.4 Agenti

lokální name server / exkluzivní (prilinkovaná knihovna) / sdílená knihovna

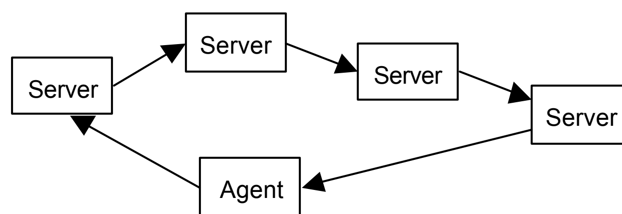
7.4.5 Vyhledávání jmen



(a)



(b)



(c)

Obr. 45 - Styly vyhledávání jmen

8. Procesy

Cílem správy procesu v distribuovaných systémech je, krome úkolu bežných z jednoprocesorových systému, sdílet výpocetní sílu systému, rozdelovat zátež na jednotlivé procesory, provádet operace na vzdálených procesech, synchronizovat procesy a vést evidenci jejich stavu.

8.1 Vlákna, nite a thready

Per proces
Adresový prostor
globální promenné
deskriptory (soubory, ...)
signály, semafory, ...
práva, úcty, ...

Per thread
Cítac instrukcí
Zásobník
Registry
Stav
(globální promenné threadu)

Tab. 16 - Entity spravované procesem a threadem

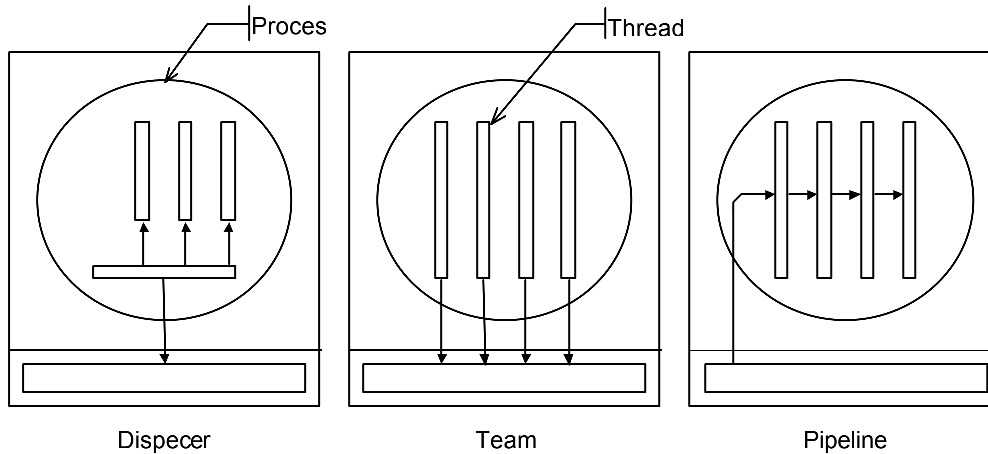
globální data threadu	globální data threadu	globální data threadu
Zásobník	zásobník	zásobník
sdílená dynamická data		
sdílená statická data		
kód		

Tab. 17 - Organizace pameti vícethreadového procesu

Vzhledem k tomu, že všechny thready jednoho procesu sdílejí adresový prostor, mají prístup ke všem globálním promenným. Casto by však bylo treba, aby každý thread mel své vlastní globální promenné, avšak lokální v daném threadu (tj. neprístupné z ostatních threadu). Príkladem možného využití muže být chybová promenná `errno`. To je globální promenná, do které je pri jakékoliv chybe uložen kód této chyby. Ve vícethreadovém prostredí však muže dojít ke kolizi - thread A vyvolá chybu 1, do `errno` se uloží 1. Bezprostredne poté thread B vyvolá chybu 2, `errno` se prepíše hodnotou 2. Thread A pak pri prístupu k `errno` precte nesprávnou hodnotu.

Vzhledem k tomu, že snad žádný programovací jazyk nemá konstrukci umožnující práci s globálními daty threadu, musí být prístup k takovým datum implementován na uživatelské (knihovní) úrovni - napr. funkcemi `create_global`, `write_global` a `read_global`.

8.1.1 Použití vláken



Obr. 46 - Organizace vláken v procesu

8.1.2 Implementace vláken

Existují v zásade dve možnosti, jak vlákna implementovat, a to bud prímo v jádre, anebo jako knihovní nadstavbu na uživatelské úrovni.

Knihovna na uživatelské úrovni

Na uživatelské úrovni se systém threadu implementuje jako kolekce knihovních funkcí. Pri pokusu threadu o zavolání systému se zavolá knihovní funkce. Ta krome toho, že zavolá již prímo príslušnou systémovou službu, navíc provede prípadnou zmenu kontextu (tj. v podstate pouze registru) threadu.

Výhody:

?? nadstavba nad již existujícím softwarem (UNIX)

?? rychlé prepínání kontextu

?? možnost vlastního plánování

Nevýhody:

?? nutnost obalení

- ?? blokovaná (synchronní) systémová volání - ostatní thready jsou blokovány
- ?? neblokovaná (asynchronní) systémová volání - nutnost zmen v jádru

?? výpadky stránek - ostatní thready jsou blokovány

?? není preemptivní

?? globální promenné

?? reakce na signály

?? nereentrance standardních knihoven

Implementace vláken v jádru

Jaderná implementace threadu nevyžaduje žádnou behovou podporu - veškerá rozhodování, plánování, prepínání kontextu, atd. se dejí prímo v jádru.

Výhody:

?? preemptivní plánování

?? jednodušší interakce s jádrem

Nevýhody:

?? vetší jádro

?? pomalejší zpracování - vytvárení/rušení threadu, ...

8.1.3 Vlákna a zprávy

V nekterých distribuovaných systémech je implementována velmi tesná vazba mezi príjmem zpráv a thready - tzv. implicitní receive, nebo také pop-up thread. Myšlenka spocívá v tom, že místo toho, aby zpráva cekala ve fronte, než si ji nekdo vyzvedne, tak se automaticky vytvorí nový (resp. vezme se nejaký volný) thread, z dat zprávy se mu vytvorí zásobník a automaticky se spustí. Po skoncení zpracování zprávy a odeslání odpovedi thread opet automaticky zanikne.

8.2 Systémové modely

Procesy (resp. jejich thready) beží na procesorech. V tradicních jednoprocesorových systémech není problém urcit procesor, na kterém daný proces pobeží. Avšak v multiprocesorových a distribuovaných systémech je metoda alokace procesoru jednou ze základních otázek designu celého systému. Existují dva zásadne odlišné modely alokace procesoru - workstation model a procesor pool model. Krome techto dvou základních forem existují ješte ruzné hybridní formy jako mezistavy slucující výhody (a casto i nevýhody) jednotlivých systému.

8.2.1 Workstation model

Workstation model je založen na myšlence, že každý uživatel má “svuj” pocítac, který je mu plne k dispozici. Krome uživatelských uzlu bývají v takovýchto systémech ješte uzly dedikované pro nejaké servery - napr. fileserver.

Vzhledem k tomu, že naprostá vetšina distribuovaných systému využívá jakýsi fileserver (at už centralizovaný, distribuovaný, apod.), je otázkou, co s lokálními disky jednotlivých uživatelských stanic (pokud ovšem nejsou bezdiskové). Možností využití lokálních disku je nekolik, každá má své výhody i nevýhody.

	Výhody	Nevýhody
Bezdiskové stanice	Levné, snadná údržba, flexibilita	Vytížení síte, zátež fileserveru
Stránkování, docasné soubory	Snižuje vytížení síte	Vetší náklady (na disky)
Stránkování, docasné soubory, systémové programy	Ješte více snižuje vytížení síte	Vetší náklady, údržba systémových programu
Stránkování, docasné soubory, systémové programy, file caching	Redukce záteže fileserveru	Udržování konzistence cache
Soucást distribuovaného filesystemu	Výrazná redukce záteže ostatních fileserveru	Udržování konzistence celého filesystemu
Kompletní lokální filesystem	Eliminace potreby fileserveru	Ztráta transparentnosti

8.2.2 Procesor pool model

Druhou možností je abstrahovat od pojmu “uživateluv pocítac/procesor” a pristupovat ke všem procesorum dostupným systému jednotne. Hardwarove je tento prístup realizován vetšinou polem procesoru, ke kterým jsou lokální sítí pripojeny uživatelské terminály (vetšinou grafické X-terminály). Tímto zpusobem se snáze a rovnomerneji rozloží výpoctová zátež mezi dostupné procesory, avšak je zapotrebí speciální, ne vždy dostupný, hardware.

8.3 Vzdálené spouštení procesu

Aby mohla být efektivne využita veškerá výpocetní síla celého systému, je obcas treba spustit nový proces na jiném procesoru.

Problémy:

1. Jak nalézt volný pocítac
2. Jak proces vzdálene spustit (tak aby to proces sám nepoznal)
3. Co se stane, když pocítac prestane být ‘volný’

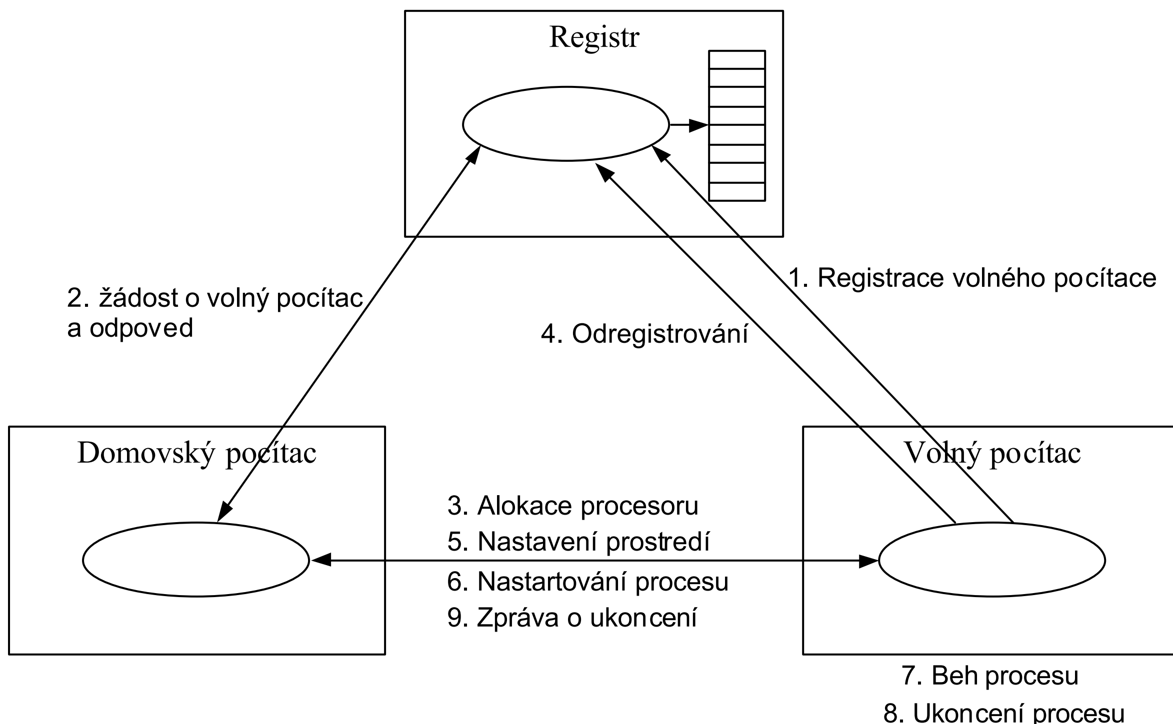
První zajímavou otázkou je, co to je vlastne volný procesor. Pro první priblížení to muže být procesor v pocítaci, na kterém není nikdo prihlášen. To, že na daném pocítaci není nikdo prihlášen, však nemusí nutne znamenat, že procesor je volný a naopak, muže být volný procesor v pocítaci s prihlášeným uživatelem. Prvnímu prípadu odpovídají ruzné démony (mail daemon, clock daemon, atd.), které dostatecne vytežují procesor, naopak druhému prípadu odpovídá (v nekterých institucích zcela bežná) situace, že uživatel prijde k pocítaci, prihlásí se, precte si noviny, dojde si na obed apod. a procesor nemá dlouhou dobu co na práci.

Jiným možným kritériem “volnosti” procesoru je stav, kdy nikdo dostatecne dlouho nevytvárel uživatelské signály, tj. nepsal nic na klávesnici ani nepohyboval myší, a zároven aktivne nebeží žádný uživatelský proces.

Další krok, který se musí provést, je spuštení procesu. Prenesení kódu a statických dat procesu je pomerne jednoduché (snad s výjimkou heterogenních procesoru, kde je treba jistých konverzí). Zajímavejší je nastavení prostredí na hostitelském pocítaci tak, aby výpocet probíhal stejne, jako kdyby proces byl spušten na domovském pocítaci. Je treba prenést množinu kontextu (tj. aktuální adresáre filesystemu a nejen jeho), ruzné promenné domovského prostredí (environmental variables), pokud takové existují, apod. Dalším problémem jsou systémová volání - nekterá systémová volání by se mela prováden na hostitelském pocítaci (napr. `sbrk`, `nice`, `profil` apod.), zatímco jiná by mela být presmerována na domovský pocítac (napr. ctení z klávesnice, zápis na terminál apod.). Jiná volání mohou být podle kontextu volaná lokálne anebo presmerována na domovský pocítac (prístup

k souborum apod.). Proto je zapotrebí vytvorit kanály zpet na domovský pocítac a nekterá systémová volání presmerovat na tyto kanály.

Zajímavá je otázka, co se má stát, když hostitelský pocítac prestane být volný (napr. uživatel se vrátil z obeda). První možností je nechat všechny cizí procesy dobehnout s tím, že uživatel má proste smulu a výkonnost "jeho" pocítace se patrne docasne o neco sníží. To však v nekterých prípadech muže být neprijatelné. Druhým extrémem je nekompromisní ukoncení procesu; to však má dve nevýhody: jednak muže být ztraceno znacné množství práce, jednak muže být systém v nekonzistentním stavu. Proto jako jedna z možností je nechat procesu jistý cas na dokoncení potrebných akcí a dovedení systému do konzistentního stavu. To však vyžaduje od takto vzdáleného procesu jistou spolupráci. Poslední možností je migrace procesu v rozbehnutém stavu na jiný pocítac, to je však o nekolik rádu složitejší akce než vzdálené spouštení.



Obr. 47 - Nalezení a využití volného pocítace

8.4 Alokace procesoru

Každý distribuovaný systém je složen z množiny procesoru, at již jsou organizovány jako pole procesoru, uživatelské stanice nebo nejak jinak. V každém prípade je zapotrebí v okamžiku vzniku procesu urcit, na kterém procesoru pobeží. Algoritmy rešící tento problém se nazývají algoritmy alokace procesoru.

8.4.1 Klasifikace alokacních algoritmu

Algoritmy alokace procesoru lze hodnotit podle mnoha ruzných hledisek (podtržené varianty jsou bežnejší nebo považovány za prakticky použitelnejší):

?? zda berou v úvahu rozdíly mezi jednotlivými procesory a speciální požadavky procesu na vlastnosti procesoru - homogenní / heterogenní

?? zda umožnují premístení již rozbehnutého procesu - migracní / nemigracní

?? co se snaží optimalizovat - využití CPU / cas odpovedi / spravedlivé pridelování prostredku / komunikacní složitost, ...

?? zda jsou predem známy údaje o procesech, podle kterých se algoritmus rozhoduje - deterministické / heuristické

?? podle charakteru algoritmu - centralizované / distribuované

?? podle toho, do jaké míry se provádí optimalizace - optimální / suboptimální

?? podle jakých informací se provádí rozhodnutí o vzdáleném spuštení procesu - lokální / globální

?? podle toho, kdo iniciuje vzdálené spuštení procesu - odesilatel / príjemce / symetrický

8.4.2 Implementace alokacních algoritmu

Základem pro jakýkoliv alokacní algoritmus je znalost vlastní záteže. To však není tak jednoduché, jak se na první pohled zdá. Pri prvním priblížení by to mohl být pocet spuštených procesu. To však o vytíženosti procesoru príliš nevypovídá - jeden výpocetne nárocný proces muže vytížit procesor daleko více nežli treba deset procesu, které však vetšinu casu cekají na nejaký signál.

Druhou možností je pocet aktivne bežících procesu. Tento pocet se však velmi rychle mení - ruzné démony jsou periodicky oživovány a po vykonání nekolika instrukcí jsou opet suspendovány, procesy interaktive komunikující s uživatelem typicky cekají na nejaký signál. Presnejší výpocet záteže muže být založen na periodickém merení míry vytíženosti procesoru. To muže být implementováno periodicky vyvolávanou prázdnou smyckou, kdy cas provedení urcitého poctu cyklu je prímo úmerný záteži procesoru ostatními procesy. Problém tohoto rešení je v tom, že když jádro provádí nejakou duležitejší akci, je zakázáno prerušení, tudíž i uživatelské casové signály. Proto nekdy merení záteže nedojde a vytížení procesoru muže být podhodnoceno.

Dalším problémem alokacních algoritmu je režie vlastního algoritmu. V prípade, že by nejaký alokacní algoritmus mel výrazne snižovat výkon procesoru (rádove o nekolik až desítky procent), pak je casto lepší zvolit jednodušší algoritmus, který sice nedosahuje teoreticky optimálních výsledku, avšak v dusledku podstatne menší režie je celkove efektivnejší.

8.4.3 Deterministický grafový algoritmus

Príkladem centralizovaného deterministického optimálního alokacního algoritmu je algoritmus minimalizující vzdálenou komunikaci. Každou dvojici vzájemne komunikujících procesu mejme ohodnocenu velikostí vzájemné komunikace. Pak celý systém muže být reprezentován grafem s ohodnocenými hranami. Úkolem algoritmu je nalézt takové vzájemné prirazení procesu a procesoru, aby komunikace mezi jednotlivými procesory byla minimální. Úloha se pak redukuje na rozdelení grafu na podgrafy takové, aby soucet ohodnocení hran mezi podgrafy byl minimální. Nevýhoda tohoto algoritmu je, že potrebuje znát predem pro každou dvojici procesu jejich komunikacních složitost, proto je (krome teoretického zkoumání) použitelný jen pro velmi speciální prípady.

8.4.4 Up-down algoritmus (Mutka-Livny)

Jiný typ alokacního algoritmu, nazývaného up-down algoritmus (cesky snad schodovitý), publikovali v roce 1987 Mutka & Livny. Pro každou doménu existuje jeden koordinátor, který udržuje tabulku se záznamem pro každý uživatelský pocítac. Pri každé významné akci, tj. vytvorení procesu, ukoncení procesu a tik hodin (presneji každý n-tý tik), se pošle zpráva koordinátoru, který provede zmeny v príslušném záznamu.

Algoritmus je optimalizován na stejnomerné sdílení výkonu všemi uživateli. K tomuto úcelu tabulka pro každý procesor obsahuje “trestné body”. Kladné trestné body znamenají, že daný procesor vyslal nejaký proces na jiný procesor, zatímco záporné trestné body znamenají, že existují neuspokojené požadavky na beh vzdáleného procesu.

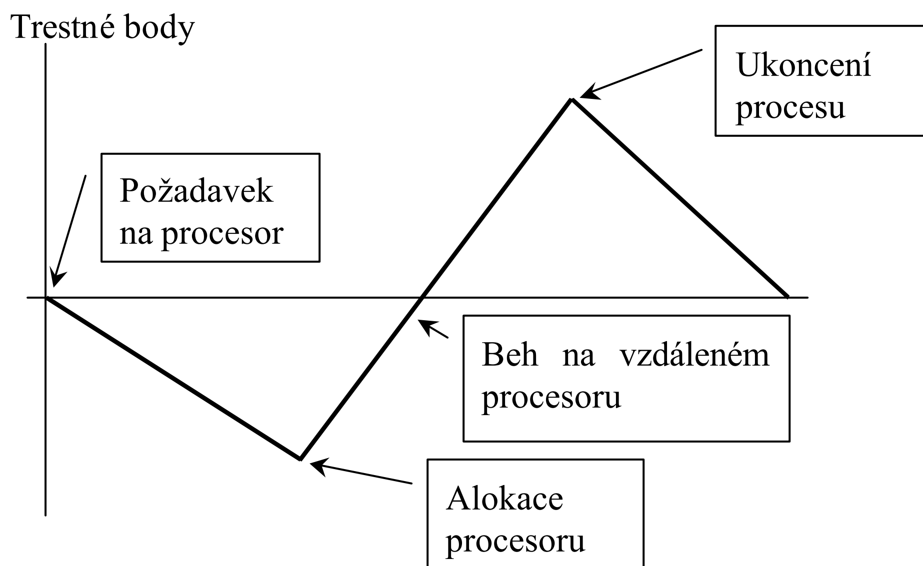
Pri každé významné události se tabulka mení následujícím zpusobem:

?? za každý proces bežící na jiném pocítaci - plus trestné body

?? za každý zatím neuspokojený požadavek na vzdálený procesor - mínus trestné body

?? jestliže nic z tohoto - smerem k nule

Pri pokusu o beh procesu na vzdáleném pocítaci se pošle zpráva koordinátoru. V prípade, že je nekterý procesor volný, alokuje se pro tento proces. V opacném prípade se požadavek pozdrží a uloží do fronty. V okamžiku uvolnení nekterého procesoru se vezme ten proces z fronty neuspokojených požadavku, jehož vysílající procesor má nejméne trestných bodu.



Obr. 48 - Nalezení a využití volného pocítace

Podle výše uvedených klasifikací lze tento algoritmus charakterizovat jako homogenní, nemigracní, spravedlivý, heuristický, centralizovaný, suboptimální, globální a symetrický.

8.4.5 Hierarchický algoritmus

Centralizované algoritmy jsou vetšinou špatne použitelné v rozsáhlých systémech, kde jsou výhodnejší hierarchické nebo distribuované algoritmy. Príkladem hierarchického algoritmu je alokacní algoritmus použitý v systému MICROS. Je založen na logické hierarchii procesoru nezávislé na fyzickém rozmístení jednotlivých pocítacu. Procesory na nejnižší úrovni vykonávají vlastní práci, procesory na vyšších úrovních jsou manažeri. V prípade, že manažer dostane požadavek na pridelení procesoru, zjistí, zda-li nekterý z jemu podrízených procesoru je volný. V prípade, že ano, pak ho alokuje, v prípade, že ne, pošle požadavek vyššímu manažeru.

Zajímavý je problém, co se stane, když nejaký manažer vypadne. Musí se nahradit nekým z jeho podrízených. Který z nich to bude si mohou urcit podrízení sami (napr. nejakým elekcním algoritmem), nebo ho muže urcit manažer vypadlého procesoru. Pro zabezpecení vetší stability mohou být manažeri nejvyšší úrovne replikovaní, tj. manažer nejvyšší úrovne nemusí existovat; je nekolik vysokých manažeru se stejnými pravomocemi.

8.4.6 Distribuovaný heuristický algoritmus

k náhodných výberu cíle

server initiated / receiver initiated / kombinovaný

8.4.7 Bidding (obchodní) algoritmus

Procesy kupují výpocetní sílu, procesory ji nabízejí

9. Migrace procesu

Migrací procesu rozumíme korektní a transparentní prenesení již spušteného procesu kdykoliv behem jeho výpoctu ze zdrojového na cílový pocítac. “Korektnost” znamená, že ostatní procesy nejsou touto migrací nijak ovlivneny (krome neprímých vlivu jako napr. zmena dostupného výkonu procesoru apod.) a že po ukoncení migrace stav systému odpovídá stavu, jako kdyby byl migrovaný proces spušten na cílovém pocítaci. Vzhledem k migrovanému procesu “transparentnost” znamená, že proces o migraci ani o tom, že beží na novém pocítaci, nemusí vubec vedet a nemusí ani pri migraci spolupracovat (tj. nemusí obsahovat kód podporující nebo umožnující migraci). Vzhledem k procesum komunikujícím s migrovaným procesem to znamená, že zustanou zachovány všechny vazby na migrovaný proces a že budou doruceny všechny zprávy, a to i ty, které byly vyslány behem vlastní migrace.

Pro zavedení mechanismu migrace procesu existuje nekolik duvodu:

?? vyvažování záteže

?? optimalizace - I/O, komunikacní, ...

?? premístení servru

?? shutdown

Pred tím, než dojde k vlastní migraci, je treba urcit který proces má být migrován a kam má být prenesen. To muže být provedeno bud rucne, napr. premístení nejakého serveru pred vypnutím pocítace, nebo automaticky, napr. nejakým algoritmem vyvažování záteže.

9.1 Mechanismus migrace

Vlastní mechanismus migrace je prováden v nekolika krocích. Jelikož v ruzných systémech mohou být migracní mechanismy implementovány znacne rozdílne, je treba se na tyto kroky dívat spíše jako na logické události, než na výkonné akce.

1. Zmražení procesu - všechny thready migrovaného procesu musí být vyjmuty z behových front. Casto bývá takto vyjmutý proces oznacen v systémových tabulkách "v migraci".
2. Inicializace cílového pocítace - cílový pocítac je informován, že mu pribude proces. Jádro pripraví pro nový proces potrebné struktury, tj. vytvorí prázdný proces.
3. Prenesení stavu procesu - prenesení kódu, zásobníku threadu, datových segmentu, vazeb na jádro a ostatní procesy apod. na cílový pocítac.
4. Reinicializace komunikacních kanálu - presmerování komunikacních kanálu na cílový pocítac a dorucení zpráv odeslaných behem migrace.
5. Vycistení zdrojového pocítace - jsou zrušeny všechny jaderné struktury použité pro migrovaný proces.
6. Spuštení procesu na cílovém pocítaci - od tohoto okamžiku proces zcela normálne beží dál, v ideálním prípade ani nezaregistruje, že byl prenesen.

9.1.1 Dorucování zpráv

Je treba implementovat mechanismus, který behem migrace a krátce po ní dorucí ve správném poradí všechny zprávy urcené pro migrovaný proces.

První možností je zaslat všem potenciálním odesilatelum zprávu, že proces je migrován. Procesy, které by chtely zasílat zprávu migrovanému procesu, budou suspendovány až do ukoncení migrace a navázání nových spojení. Problém tohoto rešení je v tom, že potenciální odesilatelé nemusí být zdrojovému jádru známi. Navíc takovýchto odesilatelu muže být velké množství, casto v podstate jakýkoliv proces celého systému. Pak by taková metoda byla komunikacne velmi nárocná, a to i v prípade, že behem migrace s migrovaným procesem nikdo nekomunikuje.

Druhou možností je nechat ve zdrojovém jádre odkaz na cílový pocítac s tím, že všechny zprávy budou presmerovány. To však vytvárí reziduální dependence se všemi nepríjemnými dusledky.

Další možností je všechny zprávy došlé behem migrace a po ní vracet zpet odesilateli s poznámkou “adresát odmigrován” a s adresou cílového pocítace. Tento zpusob však také vytvárí reziduální dependence, proto je možné implementovat slabší variantu, že adresa odmigrovaného procesu je známa pouze urcitý casový interval a po uplynutí tohoto intervalu je prípadným odesílatelum zpráva vrácena s poznámkou “adresát neznámý”. Je pak na odesílateli, aby si zjistil novou adresu príjemce.

copy on reference - imaginary segment

prískoky - viskozita

9.1.2 Stav procesu

Každý proces je charakterizován jednak svým vnitrním stavem (napr. obsah registru), ale i stavem spojeným s ostatními procesy nebo cástmi systému (komunikacní kanály, ...). Problémy, které se musí rešit pri návrhu migrace jsou zejména jak

1. zjistis stav procesu - nejedná se jenom o stav procesu samotného (napr. obsah registru), ale i stav týkající se komunikace s ostatními procesy, eventuální otevrené soubory, ...
2. vyjmout proces ze zdrojového pocítace
3. prenést proces na cílový pocítac
4. vložit proces na cílový pocítac

Stav procesu jako takový se skládá z velkého množství soucástí. Pri migrování procesu z jedné stanice na druhou je potreba u každé cásti stavu rozhodnout, jakým zpusobem zajistit, aby daná cást stavu byla procesu k dispozici i na cílové stanici (z tohoto hlediska by bylo presnejší v bode 3 výše uvedeného výctu uvést: „prenést proces a zajistit, aby mel na cílové stanici všechny prostredky, které mel na stanici zdrojové.“

Možnosti jak toho dosáhnout jsou tri:

1. Presunout danou cást stavu spolu s procesem.
   Neodiskutovatelným zástupcem této skupiny je obsah virtuální pameti.
2. Pripravit cást stavu pro forwardování požadavku, které se jí týkají
   Zde je možné si vymyslet napríklad komunikaci s konzolí systému. Stejne tak se forwardovací prístup volí tam, kde je nemožné stav bezpecne vyextrahovat.
3. Použít odpovídající prostredek na cílové stanici.
   Napríklad fyzická pamet.

9.1.3 Komunikace s okolím behem migrace a po migraci

Protože okolní procesy netuší, že se s migrovaným procesem „neco deje“ (a po pravde receno ani není vhodné, aby se to mohly dozvedet), nemohou po dobu migrace daného procesu prestat s posíláním zpráv pro migrovaný proces. Migrovaný proces je ovšem z poloviny na starém pocítaci (kam jsou mu

posílány zprávy) a z poloviny již na novém (kam by mu zprávy mely být posílány po migraci). Je nekolik možností, jak se s tímto problémem vyrovnat, budeme je podrobneji rozebírat.

9.1.4 Implementace migracní strategie a vlastní migrace

Soucástí migracní strategie (tj. procesu, behem nehož se rozhoduje co a kam odmigrovat) je zjištovat stav záteže jednotlivých pocítacu, sledovat, které pocítace nejsou jejich uživateli využívány a na základe tohoto pozorování rozhodnout o migraci.

Druhá vec je vlastní realizace migrace, což je proces, behem kterého dochází k presunu stavu procesu na cílový pocítac. Je rozumné tyto dve cásti migracního mechanizmu od sebe oddelit jednak z duvodu modularity, jednak z duvodu bezpecnosti.

Migracní strategie se typicky ponechává na nekterém specializovaném procesu, který beží na uživatelské priorite, zatímco migrace samotná je casto (alespon z cásti) implementovaná v jádre systému - odtud bezpecnostní duvody pro rozdelení.

Co se realizace migrace týce, je nekolik možností, jak ji implementovat.

1. V jádre systému
2. Vne jádra, tj. jako uživatelský proces - prináší komplikace, protože ne vše si muže uživatelský proces dovolit
3. Jako soucást procesu, v run-time knihovne - jde prímo proti požadavkum na transparenci

9.1.5 Vícenásobná migrace

Bežný pohled na migraci ríká: „Vezmi proces, odmigruj jej.“ Problém se týká skeptického pohledu na cást výroku „vezmi proces“. Proc jeden, proc ne treba dva ci nekolik najednou?

Vícenásobná migrace se snaží rešit problémy spojené s migrováním nekolika procesu zároven. Tyto problémy se týkají zejména meziprocesové komunikace, protože je potreba speciálne ošetrit, zda náhodou nemigrují dva procesy, které spolu komunikují. Jinak se totiž reší, když migruje jeden z komunikující dvojice a druhý zustává na zdrojovém pocítaci.

9.1.6 Transparence

Transparence migrace znamená, že „nikdo o migraci neví a nemuže zjistit, že k ní dochází“. Význam slova „vedet o migraci“ se liší s tím, koho se migrace týká.

Migrovaný proces - neobsahuje kód, který se týká migrace, nemusí ošetrovat situace, zda došlo k jeho migraci, ci nikoliv, ...

Ostatní procesy - stejne jako migrovaný proces neobsahují kód rešící otázky migrace

Uživatel - Program nesmí kvuli migraci zmenit chování a musí vydat takové výsledky, jako kdyby celou dobu bežel na zdrojovém pocítaci.

Casto zminovaným tématem souvisejícím s otázkami transparence je dilema „systémová volání“ a „posílání zpráv“ jako základní kameny operacního systému. Panuje bežný názor, že pri použití meziprocesové komunikace založené na posílání zpráv se zjednoduší otázky transparence. Do jaké míry jsou tyto názory pravdivé, je diskutabilní.

Bežný pohled totiž ríká, že u zpráv stací pouze presmerovat komunikacní kanál a je vše v poradku. Narozdíl od toho se príliš nedá mluvit o presmerování systémových volání.

Presmerování komunikacního kanálu nelze ovšem v žádném prípade provést slepe, aniž vím, co se nachází na druhém konci. Príkladem muže být správce virtuální pameti. Po presunu procesu na novou

stanici je na míste ocekávat, že o pamet bude proces žádat toho správce, který beží na nové stanici a nikoliv toho starého. Slepým presmerováním komunikacního kanálu by požadavek došel na puvodní stanici a to evidentne nebylo cílem.

Naproti tomu není zase tak veliký problém implementovat „presmerování" volání jádra. Znamená to v extrémním prípade všechna volání jádra forwardovat na domácí stanici. Dokonce jeden z prvních distribuovaných operacních systému Remote UNIX tuto variantu používal.

9.1.7 Reziduální dependence

Jestliže proces, který odmigroval, stále ke své cinnosti potrebuje spolupráci zdrojového pocítace, jedná se o reziduální dependenci. Problém, který prinášejí, je práve závislost procesu na jiném pocítaci, než na kterém proces momentálne beží. V prípade, že se preruší spojení obou zainteresovaných pocítacu, muže to mít pro proces fatální dusledky. To komplikuje i situaci, kdy se snaží systém odsunout procesy z pocítace, který bude v blízké budoucnosti od síte odpojen, popr. vypnut. Další vada reziduální dependence je ta, že se musí jiný pocítac zabývat existencí procesu na jiném srtroji. V techto okamžicích je snaha reziduálních dependencí se vyvarovat.

V jiné situaci se reziduální dependence hodí, zejména tehdy, když uživatel zjištuje, které procesy spustil. Tehdy by mu systém mel ukázat i procesy, které odmigrovaly, tj. systém musí o nich mít informace, které jsou ovšem reziduálními dependencemi.

9.1.8 Virtuální pamet

Virtuální pamet tvorí casto nejvetší cást stavu procesu. Na tom, jakým zpusobem se bude provádet presun virtuální pameti, velmi závisí celková doba migrace procesu. Tvurci systému postupne vymysleli nekolik možností, jak presunout obsah pameti pokud možno rychle a efektivne.

?? Prvotní prímocarý prístup k presunu virtuální pameti byl presunout veškerou pamet najednou jako soucást presunu stavu procesu.
Výhody - po provedení presunu stavu muže zdrojová stanice zapomenout, že na ní kdy proces bežel, tedy eliminace reziduálních dependencí.
Nevýhody - prodloužení doby, po kterou je proces zmražen (behem presunu stavu proces typicky nebeží). Navíc je mnohdy zbytecné presouvat veškerý obsah virtuálního adresového prostoru (viz napríklad veliký proces, který okamžite po odmigrování volá `exit()`).

?? Pre-copying. Tento postup spocívá v tom, že proces beží behem doby, po kterou se presouvá obsah jeho virtuální pameti. Tesne pred koncem kopírování se proces zmrazí, provede se dokopírování zbytku adresového prostoru (ty cásti, které proces ješte stacil zmenit) a presun stavu (obsah registru, ...).
Výhoda - proces je zmražen pouze po dobu presunu malého množství informací.
Nevýhoda - nekteré cásti adresového prostoru se kopírují vícekrát, což prodlužuje celkovou dobu migrace.

?? Copy-on-reference.
Copy on reference spocívá v tom, že se nejprve prenese ten stav procesu, který je potrebný pro beh procesu - obsah registru, komunikacní kanály, ... Presun adresového prostoru se odloží na pozdeji. Stránky adresového prostoru na cílové stanici jsou oznaceny jako neprezentní (stejne jako by byly odswapovány na disk), ale je u nich poznámka, že se mají prenést ze zdrojové stanice. Pri prístupu na takovou stránku (on reference) se její obsah prenese (copy) a na zdrojové stanici se smaže.

?? Další možností je kombinace výše uvedených variant. Jedna z nich bude popsána v odstavci venovanému operacnímu systému Sprite.

9.1.9 Cíle návrhu migrace

1. Oddelení migracní strategie od realizace migrace
2. Co možná nejvetší nezávislost implementace migrace na ostatních cástech systému
3. Transparence
4. Autonomie - systém sám rozhoduje o tom, co a kam migrovat
5. Možnost evikce procesu z pocítace, jehož majitel se vrátil
6. Vyvarovat se reziduálních dependencí, kde je to jenom možné
7. Spolehlivost
8. Efektivita
9. Jednoduchost
10. Fault-tolerance

9.1.10 Kdy migrovat

Možnosti jsou vesmes dve. Pri spouštení procesu nebo pri jeho behu.

První varianta je méne nárocná na cas, protože napríklad nemusí docházet k presunu virtuální pameti mezi pocítaci, nicméne není to postacující varianta pro implementaci migrace z duvodu obcasné potreby provést migraci na již bežícím procesu (pr. evikce).

Druhá možnost je obecnejší, ale vyžaduje složitejší mechanizmus.

9.1.11 Výber cíle migrace - migracní strategie

Pro rozhodování co a kam se bude migrovat je potreba provádet sber velkého množství nejruznejších dat. Tato data se porizují ruznými zpusoby podle toho, jakou mají povahu:

?? Pri nejaké události - event sampling - napríklad sber informací o posílání zpráv, využívání výstupu na konzoli, ...

?? V intervalech - zátež CPU, zátež síte, ...

?? Periodická statistika - slouží k vypocítávání dlouhodobejších prumeru

Je pravda, že na rozhodnutí jaký proces kam odmigrovat má vliv celá rada faktoru. Krome tech, které se týkají samotného procesu a jeho cinnosti (využití procesoru, síte, cetnost vytvárení potomku) jsou to také faktory týkající se hardwarové kofigurace, stavu diskových keší, doby, po kterou je stanice uživatelem nepoužita, ...

Tyto informace je potreba sbírat, sledovat a vycházet z nich pri rozhodování o migraci.

Návrh funkce migracního politika, tj. té cásti migracního mechanizmu, který toto rozhodování provádí, musí zohlednit následující požadavky:

?? Výkonnost - nelze dlouho premýšlet o volbe procesu a jeho cíle pro migraci, nebot potom muže být pozde.

?? Rozširitelnost - mechanizmus by mel pocítat s velkým poctem stanic, které bude obsluhovat. Bežným požadavkem je obsloužení rádove stovek stanic pripojených do síte.

?? Fault-tolerance - výpadek jedné komponenty by nemel ohrozit rozhodování ve zbylé funkcní cásti síte.

?? Spravedlivost - rozhodovací systém by nemel dovolit, aby procesy, jejichž zdrojem je jedna “ciperná stanice“ ovládly sít do té míry, že by jiné stanice nemohly rozumne provádet svoji cinnost.

?? Jednoduchost

Je potreba vzít v úvahu i možnost, ve které jsou data potrebná pro rozhodování uložena na jiném míste, než rozhodovací „mozek“.

Pro návrh implementace rozhodovacího mechanizmu je tedy nekolik možností.

Sdílený soubor

V souboru, který je uložený na nejakém centrálním fileserveru, se ukládají záznamy obsahující identifikátor stanice, její prumernou zátež v posledních nekolika (5-15) minutách, doba od posledního interaktivního použití stanice (signalizace idle stanic), príznak, zda stanice prijímá cizí procesy, pocet cizích procesu na stanici, ...

Každá stanice aktualizuje v souboru ty záznamy, které se jí týkají. Pri rozhodování o cíli migrace se soubor použije jako zdroj informací.

Toto je príklad situace, kdy jsou data relevantní pro migraci uložena na jiném míste než rozhodovací mechanizmus.

+ používají se již existující vyvinuté mechanismy (synchronizace akcí nad souborem)

- pád fileserveru, pomalost prístupu k datum

Centrální server

Modifikací predchozího modelu je varianta centrálního serveru. Ta prichází s tou zmenou, že informace potrebné pro rozhodování jsou uloženy v pameti rozhodovacího serveru a server sám o sobe vykazuje jistou inteligenci. Mechanizmus rozhodování tedy muže být implementován jako soucást serveru.

+ Komunikace je orientována spojove, inteligence, vyšší rychlost.

Distribuované servery

Každý server shromažduje cást informace a tyto servery mezi sebou komunikují.

- komunikacne nárocné rešení, nebo na úkor komunikace zaostává informovanost jednotlivých serveru, nárocnejší udržování konzistence, výpocetní overhead

+ vyšší míra spolehlivosti

Je možné použít pravdepodobnostní model, který je implementován v systému MOSIX. Ten využívá stárnutí dat.

9.2 Prehled nekterých systému

V následující cásti probereme nekteré zajímavé aspekty implemetace migrace procesu v operacních systémech DEMOS/MP, Charlotte, V, MOSIX a Sprite. Zmíníme se také o objektovém systému Emerald, který implementuje migraci objektu a nekteré zajímavé operace nad nimi.

9.2.1 DEMOS/MP

V systému DEMOS/MP, vyvinutém na univerzite v Berkeley v r. 1983, je implementována meziprocesová komunikace pomocí linku spojených s procesy. Tyto linky spravuje jádro systému. Systém nabízí plnou transparenci, k cemuž mu pomáhá uniformní komunikacní interface, který je nezávislý na poloze komunikujících procesu. Jádro samo muže posílat právy jako každý proces, což zjednodušuje prístup ke komunikaci ze strany jádra.

Prístup k predávání zpráv urcených pro migrovaný proces behem migrace, který byl autory zvolen, je forwarding.

Vlastní migracní síla je implementována v jádre systému.

Vzhled identifikace procesu:

Poslední známý pocítac, kde proces bežel	Globální unikátní PID	Vytvárející pocítac	Lokální unikátní PID

Fáze migrace

1) Negotiation - domluva
   Zdrojová stanice se dotazuje cílové, zda pristupuje na migraci za daných podmínek uvedených ve zpráve. Odpoví-li cíl ano, je vše v poradku, jinak se vybírá jiná cílová stanice.
2) Presun
   U presunu je zajímavé sledovat, která stanice je v dané fázi presunu aktivní. Tyto informace budou uvedeny vždy v závorce.

   ?? Vyjmutí procesu (*Zdroj*)
   Proces je oznacen jako „v migraci" a vyjmut z run-queue.

   ?? Podrobnejší informace pro cíl (*Zdroj*)
   Posílá se napríklad velikost procesu a další informace o alokovaných prostredcích.

   ?? Alokace stavu na cíli (*Cíl*)
   Vytvorí se nové datové struktury pro proces na cílové stanici a alokují se potrebné prostredky. Nový proces vytvorený na cílové stanici má PID migrovaného procesu.

   ?? Presun stavu (*Cíl*)
   Tohle je zajímavé, protože v tomto prípade si k sobe cílová stanice proces „tahá", prestože je bežnejší postup, kdy je cílová stanice pasívní a zdrojová naopak data „tlací".

   ?? Presun adresového prostoru a pameti (*Cíl*)

   ?? Forward cekajících zpráv na cílovou stanici (*Zdroj*)
   Posílají se jednak ty zprávy, které procesu prišly pred zapocetím migrace a které už si nestacil vyzvednout a dále ty zprávy, které procesu prišly behem migrace.
   Pred posláním techto zpráv zmení jádro adresu procesu ve strukture identifikující proces na novou adresu.
   (Prepíše se adresa posledního známého pusobište procesu)

   ?? Vycištení stavu na zdroji (*Zdroj*)
   Zdrojová stanice smaže veškeré struktury týkající se odmigrovaného procesu krome informace o jeho novém pusobišti. Tato je pozdeji využívána pro forwarding zpráv.

   ?? Restart procesu (*Cíl*)

Forwardování zpráv

Z hlediska forwardování jsou 3 druhy zpráv podle toho, jak (od-) migrovanému adresátovi pricházely.

1. Odeslané pred migrací, ale neprijaté pred migrací.
   - Preneseny v rámci bodu 6 presunu pri migraci
2. Odeslané po migraci za použití staré adresy
   - Jádro „starého“ pocítace, tj. nekdejšího zdrojového pocítace, tyto zprávy prubežne forwarduje na pozici, kterou má poznamenanou ve své informaci o odmigrovaném procesu (viz bod 6 migrace). Prestože se nabízejí jiné možnosti rešení (vracet tyto zprávy zpet - „adresát nenalezen“, nebo obnovit adresy u všech procesu komunikujících s procesem migrovaným), zvolil DEMOS/MP forwardovací strategii.
3. Odeslané po migraci s využitím nové adresy
   - Tyto zprávy neciní problémy.

9.2.2 Charlotte

Systém Charlotte teží ze svého návrhu meziprocesorové komunikace - zasílání zpráv. Komunikace probíhá pomocí linku a je naprosto nezávislá na umístení procesu. Dalsími vlastnostmi IPC jsou možnost komunikovat pres více kanálu zároven, možnost prerušit zprávu (message cancelation) a možnost presouvat linky.

Této vlastnosti Charlotte využije tím, že po odmigrování procesu nezustanou na zdrojové stanici žádné reziduální dependence (*v systému DEMOS/MP zbyla nová adresa procesu kvuli forwardování zpráv*).

Návrh migrace v Charlotte

Návrhári systému se snažili o dosažení následujících cílu:

1. Migracní politiku implementovat jako uživatelský proces
   migracní sílu zahrnout do jádra systému
2. Dosáhnout maximálního stupne transparence
3. Umožnit preempci procesu, který byl odmigrován na stanici, jejíž uživatel se poté vrátil
4. Snaha o maximální fault-toleranci.

Migracní strategie - sber statistiky

Uživatelský proces poverený vykonáváním migracní strategie za tímto úcelem sbírá jisté informace

O pocítaci:

?? Pocet procesu

?? Pocet komunikacních linku na tomto pocítaci

?? Množina nejpoužívanejších komunikacních linku

?? Celkové využití CPU

?? Celkové využití sítové komunikace

O každém procesu:

?? Doba, po kterou proces již bežel

?? Stav procesu

?? Využití CPU

?? Využití sítové komunikace

Perioda sberu informací je 50-80 ms, informace se okolním pocítacum oznamují jednou za 100 intervalu sberu (5-8 s).

Podpora migrace

V jádre systému jsou vyclenena 3 vlákna, která se o migraci starají.

1. Vlákno obhospodarující „migracní“ volání jádra
2. Vlákno vykonávající jednotlivé fáze migrace
3. Vlákno podporující sber statistiky.

Interface jádra týkající se migrace obsahuje ctyri služby:

1. Zacíná/prestává se sbírat statistika
2. Migrate Out
3. Migrate In
4. Zruš migraci, je-li to možné

Prubeh migrace

Vlastní migrace probíhá ve trech fázích.

Proces je po celou dobu migrace suspendován, zamražen (frozen), interakce procesu s externím prostredím jsou odloženy.

1) Negotiation - domluva
   Tato fáze se skládá z výmeny informací mezi politickými procesy na zainteresovaných stanicích.
   Když se tyto dohodnou, politický proces na zdrojové stanici zavolá jádro „Migrate Out“.

   ?? Jaderné vlákno pošle na cílovou stanici podrobnejší informace o procesu (rozvržení pametových segmentu, tabulka komunikacních linku, množina aktivních linku, využití CPU a síte).

   ?? Jsou-li na cílové stanici prostredky pro nový proces, pošle jádro na cílové stanici podrobné informace migracnímu politikovi cílové stanice.

   ?? Politik cílové stanice zavolá „Migrate In“

   ?? Probehne alokace prostredku

   ?? Kernel cílové stanice provede fork interního procesu urceného pro príjem migrovaných procesu

   ?? Poté se na zdrojovou stanici posílá odpoved. Bud commit (migrace akceptována) nebo refused.
   Pred odesláním commitu je možné migraci prerušit.

2) Vlastní presun procesu
   Odehrává se ve trech krocích:

   ?? Presun obrazu procesu
   obraz se na cílové stanici uskladní ve strukturách pripravených v bode 1) migrace

?? Nastavení komunikacních linku
Kernely na druhých koncích všech linku jsou obeznámeny s novou pozicí migrovaného konce spoje. Dojdou-li potvrzení na všechny zprávy, muže se stehovat.
Všechny zprávy, které až doposud procesu prišly na zdrojovou stanici, se tam ukládaly do bufferu. S procesem se na cílovou stanici presunou pouze hlavicky techto zpráv bez vlastního datového obsahu. Až bude chtít v budoucnu tyto zprávy proces prijmout, jádro nasimuluje buffer-cache-miss a obsah zprávy se prenese.
Od této chvíle muže nová stanice prijímat zprávy pro migrovaný proces, který ovšem ješte nebyl prenesen. Tyto zprávy jsou na cílové stanici bufferovány.
Behem nastavování linku se objevuje jejich nekonzistence, protože nekteré linky jsou již nastaveny a jiné ješte ne. V dusledku toho chodí zprávy urcené procesu jak na zdrojovou stanici tak i na stanici cílovou.

?? Presun stavu
Stehování provádí zdrojová stanice (narozdíl od DEMOS/MP)
Presunují se deskriptory procesu, komunikacních linku, událostí a zpráv. Tyto se odesílají v paketech.

3) Vycištení zdrojové stanice - Clean-up

?? Odesílatel smaže všechny datové struktury týkající se odmigrovaného procesu (až na nekolik zpráv, které se smažou až po jejich úplném prenesení na cílovou stanici).

?? Príjemce rozbalí prijaté informace a zaradí proces do run-queue.

9.2.3 V

V je systém, který byl vyvinut na univerzite ve Stanfordu.

Základní vlastnost systému, duležitá z hlediska migrace, je sítove transparentní exekucní prostredí a meziprocesová komunikace (mírné komplikace pusobí hardwarové device drivery).

Návrh migrace

V systému V není predmetem migrace proces, ale „logical host“, který proces obsahuje. Logical host je adresový prostor, kde muže být nekolik V-procesu.

V systému V se návrhári snažili minimalizovat dobu, po kterou je migrovaný proces zamražen. Tuto dobu je výhodné minimalizovat kvuli možným timeoutum procesu cekajících na odpoved od migrovaného procesu.

Dobu migrace se podarilo minimalizovat na úkor celkové doby migrace. Mechanismus, který byl použit, se nazývá „precopying“.

Pri migraci dochází k atomickému presunu procesu. Význam je ten, že proces není nikdy viditelný v obou instancích (nové i staré)

Meziprocesová komunikace má schopnost zotavení ze ztráty zprávy. Této vlastnosti se pri migraci využívá.

Mechanika migrace

Systém V implementuje migraci ve ctyrech krocích.

1) Inicializace cíle

?? V cílové stanici se vytvorí nový logical host (s jiným identifikátorem, což umožnuje paralelní existenci starého i nového logického hostu. (na konci fáze 3 se nové ID prepíše na puvodní).

2) Precopying stavu

?? V této fázi proces na zdrojové stanici, který beží s vysokou prioritou (ve smyslu castejšího prístupu k CPU), provádí kopírování stavu migrovaného procesu, zejména jeho adresového prostoru. Migrovaný proces stále beží, takže se muže stát, že po zkopírování nejaké oblasti pameti dojde k její zmene. Tato oblast se musí prenést znovu. Proces probíhá do té doby, než zbyde malé množství pameti, kterou je treba presunout.

3) Dokoncení kopírování

?? Migrovaný logický host je zamražen

?? Dokoncí se kopírování posledních zmodifikovaných oblastí pameti

?? Ošetrí se problémy se zprávami od ostatních procesu. Zde se rozlišuje podle toho, zda byla zpráva žádostí ci odpovedí.

?? žádosti - schovávají se u adresáta (migrovaného procesu) ve fronte. Ta se ovšem po migraci maže, tedy odesílatel musí poslat svoji žádost znovu na novou stanici (po migraci je nová adresa brodcastem oznámena všem).

?? odpovedi - jsou okamžite zahozeny

?? Zahazování zpráv je umožneno díky vlastnostem IPC.

?? Kopírování stavu z jádra zdrojové stanice a z manažera programu na zdrojové stanici

?? Zmena identifikátoru logického hostu, tady vzniknou dve identické kopie log. hostu.

4) Odmražení, presmerování odkazu

?? Smaže se stará kopie log. hostu, nová se odmrazí.

?? PID se naváže na novou kopii log. hostu.

?? Provede se navázání na nový pocítac (zejména cache)

?? Broadcast nové adresy log. hostu.

Spice a Accent

Tyto systémy jsou zajímavé tím, že (podobne jako systém V) presouvají adresový prostor v situaci, kdy migrovaný proces beží. Systém V používá precopying, systémy Spice a Accent si vybraly opacný prístup - copy on reference.

Tento prístup podle merení autoru systému Spice a Accent ušetrí presun 21% až 96% adresového prostoru, ale za cenu velkého množství reziduálních dependencí, mnohdy rozprostrených po více stanicích (napríklad když proces „obehne“ celou sít, na každé stanici zanechá cást svého virtuálního adresového prostoru).

9.2.4 MOSIX

Operacní systém MOSIX je systém UNIX adaptovaný na distribuované prostredí. Migrace procesu je v systému rozsáhle využívána, protože je to základní prostredek pro vyvažování záteže.

Protože není v systému UNIX migrace procesu standardne implementována, museli tvurci systému MOSIX provést nekteré zmeny oproti standardní implementaci UNIXu. Zajímavé jsou zejména ty zmeny, které se týkají procesu samotného, tedy zmeny ve strukture, v níž si systém uchovává informace o procesu.

Ve strukture pribyly následující položky:

?? Cas, který uplynul od posledního pokusu odmigrovat daný proces

?? Cas strávený na soucasném procesoru.

?? Prícina migrace

?? explicitní žádost voláním `migrate();`

?? vyvažování záteže - load balancing

?? intenzivní vzdálené I/O operace

?? rozsáhlé vytvárení nových procesu voláním `fork();`

?? shutdown stanice, na které proces beží

?? Nové flagy procesu

?? SMOVE - v migraci

?? SHERE - nesmí odmigrovat

?? Pocet vytvorených procesu - detí a informace o posledních operacích `fork()`

?? Statistika meziprocesové komunikace

Návrh migrace

Každý proces má svuj domácí procesor, tj. procesor, na kterém byl proces vytvoren. Není možnost vzdáleného `forku`. Pri vytvárení procesu se nový vytvorí vždy na procesoru, na kterém zrovna beží otec.

Migrace procesu je implementována voláním jádra `migrate(destination, lock)`.
`destination` - cílový procesor
`lock` - provést uzamcení procesu na cílovém procesoru?

Toto volání má dva speciální prípady. Prvním z nich je zmena stavu uzamcení na soucasném procesoru. Tehdy nedojde k migraci, proces se pouze uzamkne/odemkne. Druhý speciální prípad je, když o migraci rozhodne samotné jádro systému.

Stejný postup je používán jak pro migraci mezi stanicemi v rámci síte, tak i pro migraci procesu mezi procesory jedné stanice.

Vlastní migrace

Zdrojová stanice volá rutinu `passto()`, která prenese proces na cíl. Tato rutina provede následující kroky:

(Rutiny, jejichž názvy zacínají S znamenají, že jde o komunikaci RPC mezi stanicemi)

1) Tesuje se, je-li možné proces odmigrovat

2) Volání rutiny `Smakeproc()` ze zdrojové stanice na cílovou stanici.

?? Cílová stanice zjistí, jestli si muže dovolit príjem procesu

?? Zajistí opravu komunikacních kanálu

?? Kontrola, zda na cílové stanici nebeží proces, který má stejné PID jako proces migrovaný (to by znemožnilo migraci, ale není to tak castý prípad, muže nastat v dusledku pádu nejaké stanice)

?? Nastaví se pametové oblasti

?? Ceká na dorucní procesu. Až bude proces dorucen, rutina jej probudí a vrátí se.

3) Jestliže se migrace provádí v dusledku vyvažování zátěže, bylo vyvažování po dobu provádení prvních dvou kroku vypnuto. Ted se znovu zapíná.
4) Presun dat procesu - presun adresového prostoru.
5) Volání `Spassproc()` na cílovou stanici. Toto volání nastaví dosud nenastavené položky ve strukture procesu.
6) Zdrojová stanice volá na cílovou stanici `Sactivate()`. Toto volání rozbehne odmigrovaný proces.
7) Na zdrojové stanici se smažou lokální data týkající se procesu a uvolní se pamet, kterou proces zabíral.

9.2.5 Sprite

Operacní systém Sprite je vybudován na jádre podobném 4.3 BSD UNIXu. Jádra systému na jednotlivých stanicích spolu komunikují pomocí RPC.

Predpoklady, ze kterých vyšli tvurci systému Sprite, když zacínali navrhovat migraci procesu, byly zejména tyto:

?? V síti, na které pracovali, bylo po vetšinu casu mnoho „idle“ stanic, jejichž výpocetní výkon ležel ladem.

?? Prestože je hezké využít výkon „idle“ stanic, jakmile se vrátí vlastník stanice, je potreba uvolnit výpocetní sílu stanice pro nej, tj. odmigrovat všechny cizí procesy. Stejne tak jako cíl migrace je možné použít pouze stanici, o které se ví, že je „idle“.

?? Velká vetšina programu beží velmi krátce, tedy migrace procesu musí být provedena velmi rychle.

?? Sprite je systém založený na UNIXu a na systémových voláních (narozdíl od systému Charlotte, který je založen na komunikaci pomocí posílání zpráv.

?? Protože již existovala jistá podpora využití zdroju systému po síti, snažili se tvurci migrace maximálne tuto podporu využít (prístup k souborum, zarízením, „svetové“ identifikace procesu, ...)

Cíle, které si tvurci vytycili, byly hlavne rychlost (už byla zminovaná výše) a dosažení maximální míry transparence.

Pro každý proces je definována domácí stanice, což je ta stanice, na které by proces bežel, kdyby migrace v systému neexistovala. Je to stanice, která o procesu stále ví, tj. porád má informaci o momentální pozici procesu.

Návrh migrace

Stejne jako u predchozích systému, i v systému Sprite se návrhári rozhodli implementovat maximální množství funkcí mimo jádro systému. Z toho vyplynulo již standardní rozložení funkcí.

Vlastní migraci provádí jádro systému, sber statistiky, sledování cinnosti uživatele na stanici a rozhodování provádí uživatelská aplikace - load average daemon.

Migrace procesu je cást systému, která zasahuje svým vlivem témer do všech cástí systému. S tímto problémem se setkali i návrhári systému Sprite. Zjistili, že zmeny, ke kterém docházelo v prubehu vývoje migracního mechanizmu, zpusobují nekonzistenci jednotlivých verzí jader (už jsme mluvili o tom, že cást implementace migrace je obsažena v jádre). V dusledku toho se objevily problémy s komunikací odlišných verzí migracních mechanizmu. Návrhári zavedli císlování verzí jader podle

zmen v migracním mechanizmu. Pouze jádra se stejným císlem verze mohou mezi sebou vymenovat procesy.

Puvodní idea návrháru byla taková, že se budou všechna volání jádra, která proces provede na nové stanici, forwardovat na domácí stanici, kde budou vykonána. To zajištuje vysokou míru transparence, ale za cenu reziduálních dependencí. Pozdeji od této ideje Sprite ustoupil a zvolil kombinaci všech trí možností pro prístup k prostredkum procesu po migraci (prenos, forward, podobný prostredek na cílové stanici).

Presun virtuální pameti

Nekteré možnosti presunu virtuální pameti byly rozebrány dríve. Operacní systém Sprite zvolil variaci nekolika výše uvededných zpusobu. Uvedená kombinace je spojení presunu veškeré pameti najednou a „copy-on-reference“.

V první fázi presunu virtuální pameti se všechny modifikované stránky uloží na fileserver. Potom se presune stav procesu na cílovou stanici. Pamet se potom zpusobem copy-on-reference presouvá na cílovou stanici z fileserveru tak, jak ji proces potrebuje.

Nevýhodou tohoto postupu je, že se pamet presouvá po síti dvakrát. Podle autoru systému Sprite však k migraci dochází vetšinou pri volání funkce exec, takže efekt dvojího presunu pameti je v této souvislosti irelevantní.

Problémy v systému Sprite nastávají tehdy, když procesy sdílejí úsek pameti pro zápis. Procesy, které sdílejí pamet pro zápis, by se mohly presouvat všechny zároven, ale vzhledem k tomu, že mohou tvorit rozsáhlou a složitou skupinu, není tato možnost využita a platí zákaz migrace procesu sdílejících pamet pro zápis. (Tuto situaci by mohlo vyrešit použití distribuované sdílené pameti, ale toho Sprite nevyužívá.)

Reziduální dependence a domácí stanice

Otázku reziduálních dependencí vyrešili návrhári systému Sprite tak, že pro každý proces je definována jeho domácí stanice, která má neustále o procesu prehled. Po odmigrování procesu z jeho domácí stanice na této zustane jistá informace, ale po odmigrování procesu ze stanice jiné (napr. v dusledku evikce) po sobe proces žádné reziduální dependence nezanechá.

Soucástí stavu procesu je v systému Sprite i Process Control Block - PCB. Otázkou je, jak zarucit, aby proces mel svuj PCB na cílové stanici, protože na té stanici beží. Ale i na domácí stanici je potreba udržovat informace o momentální poloze procesu. Za tímto úcelem je PCB ve dvou kopiích. Jedna kopie je na stanici, kde proces momentálne beží, druhá je na jeho domácí stanici, pricemž domácí stanice má vetšinu položek v PCB nevyužitou.

Migracní rutiny

Stav procesu se skládá z mnoha ruznorodých soucástí, z nichž každá vyžaduje specifický prístup. Napríklad presun obsahu registru vyžaduje jiné zacházení než presun informace o otevreném komunikacním kanálu.

Tvurci migrace pro operacní systém Sprite premýšleli nad tím, jak v této ruznorodosti najít spolecné rysy a tím i spolecný prístup k zacházení s cástmi stavu migrovaného procesu. Výsledkem jejich snažení se stal koncept **migracních rutin**.

Každá soucást prenášeného stavu má definovány práve 4 migracní rutiny.

1. Predmigracní rutinu - **pre-migration routine**
   Jejím úkolem je pripravit danou cást stavu na migraci. Návratovou hodnotou je velikost dat, která se budou presouvat mezi stanicemi. (Pro virtuální pamet je to napríklad príprava modifikovaných stránek procesu k zápisu na disk.)

2. Zabalovací rutinu - **encapsulation routine**
   Tato rutina zabalí svojí cást stavu procesu do pripraveného bufferu. Jejími vedlejšími efekty jsou napríklad komunikace s file-serverem v prípade zabalování informací o otevrených souborech nebo zápis modifikovaných stránek adresového prostoru migrovaného procesu v prípade zabalování virtuální pameti.
3. Rozbalovací rutinu - **de-encapsulation routine**
   Je volána na cílové stanici, slouží k „nainstalování" stavu primigrovaného procesu.
4. Pomigracní rutinu - **post-migration routine**
   Zdrojová stanice ji volá po ukoncení prenosu odmigrovaného procesu. Tato rutina vycistí danou cást stavu procesu, uvolní alokovanou pamet, ...

Vlastní prubeh migrace

1. Procesu, o kterém se rozhodlo, že se bude migrovat, je poslán **signál**. V dusledku toho proces musí provést trap do jádra a navíc je v tomto prípade v dobre definovaném stavu (není uprostred nejakého choulostivého jaderného volání).
2. Jedná-li se o migraci z domácí stanice ven na jinou stanici, dotazuje se domácí stanice cílové, zda tato žije a zda je možné na ni poslat migrovaný proces. Pri této príležitosti si také vyžádá verzi implementace migracního mechanizmu. Dopadnou-li odpovedi na **otázky** dobre (je možné migrovat), alokuje se na cílové stanici struktura PCB (Process Control Block) a na zdrojovou stanici se vrátí identifikátor nove vytvoreného procesu.
   Migruje-li se z nekteré stanice síte na domácí stanici (napríklad v dusledku evikce), odpadá vetšina otázek a také alokace PCB. Jako identifikátor procesu na cílové stanici (tedy na stanici domácí - proces migruje domu) se použije jeho normální PID.
3. Pro každou cást presunovaného stavu procesu se zavolá **pre-migracní rutina**. Tato rutina inicializuje jednotlivé cásti stavu pro budoucí prenos. Výsledkem jejího volání je velikost dat, která se budou prenášet.
4. Podle výsledku volání pre-migracní rutiny se na zdrojové stanici naalokuje **buffer**, do kterého se bude proces zabalovat. Velikost bufferu závisí zejména na velikosti pameti používané procesem - na velikosti jeho tabulek stránek.
5. Pro každou cást stavu (modul) je volána príslušná **encapsulacní rutina**. Tato rutina do bufferu alokovaného v kroku 4 zabalí príslušný modul stavu procesu.
6. Presun stavu je realizován jedním RPC. V tomto okamžiku se pro každý modul na cílové stanici volá jeho **deencapsulacní rutina**.
7. Na zdojové stanici se opet pro každý modul volá **post-migracní rutina** (je-li pro modul definována). Tato rutina vycistí danou cást stavu.
8. Na zdrojové stanici je provedena **dealokace** bufferu a cílová stanice se informuje o tom, že proces muže být spušten.

Transparence

Možností, jak dosáhnout transparence, využili návrhári Spritu nekolik.

1. Kde to bylo možné, tam zmenili volání jádra takovým zpusobem, aby byla nezávislá na tom, zda je proces provádí na domácí stanici ci nekde jinde. K tomu pomohl jednotný prostor jmen, jednotný filesystém, „svetové" identifikátory procesu, ... Odpovídá to vlastne použití podobného prostredku na cílové stanici.
2. Kde nebylo možné použít první možnost, snažili se presunout danou cást stavu na cílovou stanici.
3. V prípade, že selhaly snahy o obe predchozí varianty, nezbylo, než provádet forwardování akcí na domácí stanici - slepé forwardování.

Tímto zpusobem je kuriózne ošetreno napríklad volání `gettimeofday()`, nebot v systému Sprite neexistuje rozumná synchronizace hodin jednotlivých stanic.
Forwardování se neprovádí pouze ze stanice, kde proces beží, na stanici domácí, jak je tomu v prípade `gettimeofday()`, ale i ze stanice domácí na stanici, kde se proces nachází (napríklad signály, ...).

4. Jistá skupina voláních musela být ošetrena vzájemnou kooperací obou stanic. Jak té domácí, tak té, na které proces beží. Príkladem budiž volání `fork()`, `exit()`.

Ze 106 volání jádra systému Sprite je 91 volání ošetreno zpusobem 1) nebo 2), 11 volání je slepe forwardováno a zbylá ctyri se reší komplikovanejším zpusobem kooperace obou zainteresovaných stanic.

9.2.6 Migrace objektu - systém Emerald

Systém Emerald byl vyvíjen zároven s programovacím jazykem Emerald. Jednotkou manipulace je objekt. Objekt se skládá ze ctyr cástí:

1. „Svetove" (tj. v rámci síte) unikátní jméno.
2. Reprezentace - data. Data jsou bud primární (skutecný datový obsah objektu), nebo reference na jiné objekty.
3. Operace, které je možné na objektu provádet.
4. Proces, který je k objektu prirazen. Je-li prirazen, jedná se o aktivní objekt, v opacném prípade je objekt pasívní.

Koncept mobility je obsažen v programovacím jazyku Emerald a v behové podpore. Jazyk dává k dipozici následující primitiva:

?? `LOCATE objekt ..... nalezne momentální polohu objektu`

?? `MOVE objekt TO uzel ..... presun objektu`

?? `FIX objekt AT uzel ..... upevní objekt na uzlu`

?? `UNFIX objekt ..... uvolní objekt`

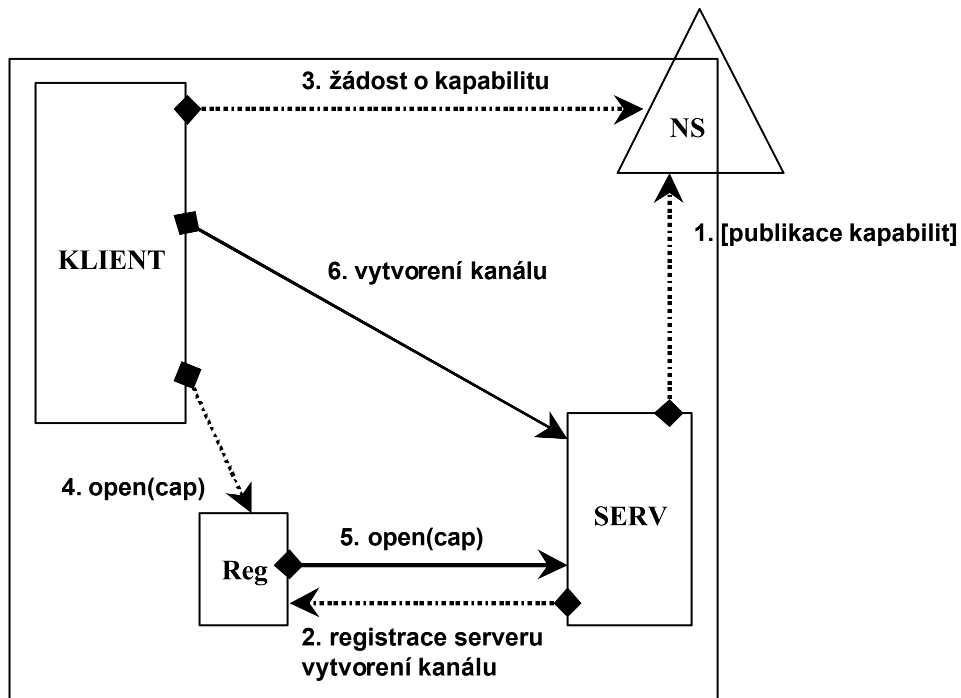
?? `REFIX objekt AT uzel ..... atomické provedení`
```
UNFIX objekt;
MOVE objekt TO uzel;
FIX objekt AT uzel;
```

Lze explicitne ríci, co všechno se má presunout. Stací spolecne svázat objekty, které se potom presunují najednou.

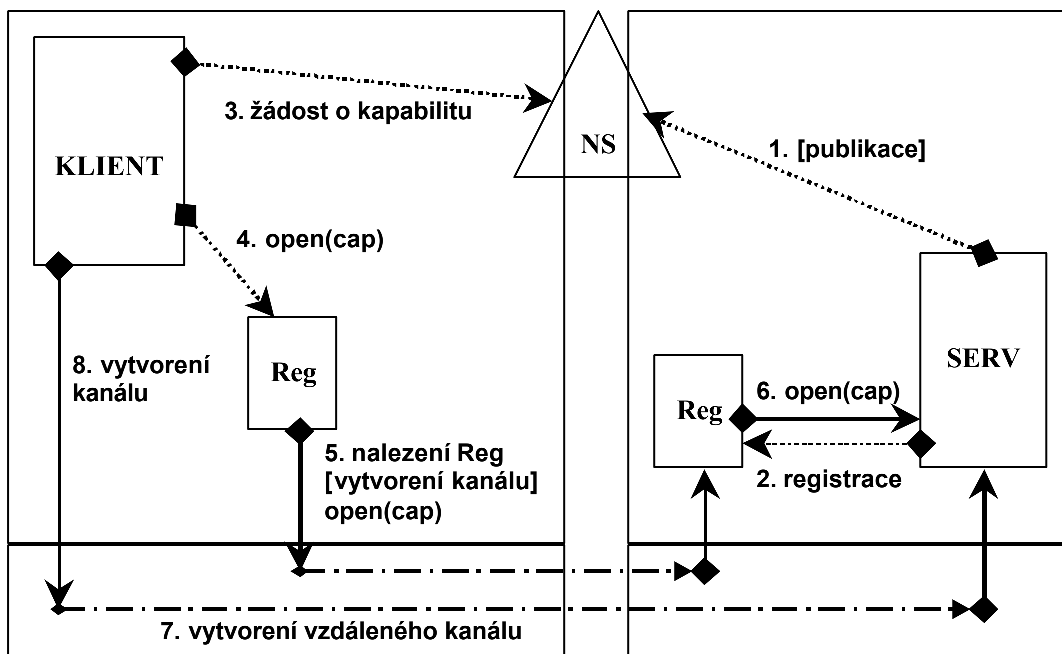
Presun objektu

1. Zdroj vyrobí zprávu, která obsahuje objekt a tabulky pro preklad adres závislých na poloze objektu.
2. Zpráva se odešle na cíl.
3. Cíl alokuje pamet, zkopíruje data a vyrobí vlastní translacní tabulku podle té, co obdržel.

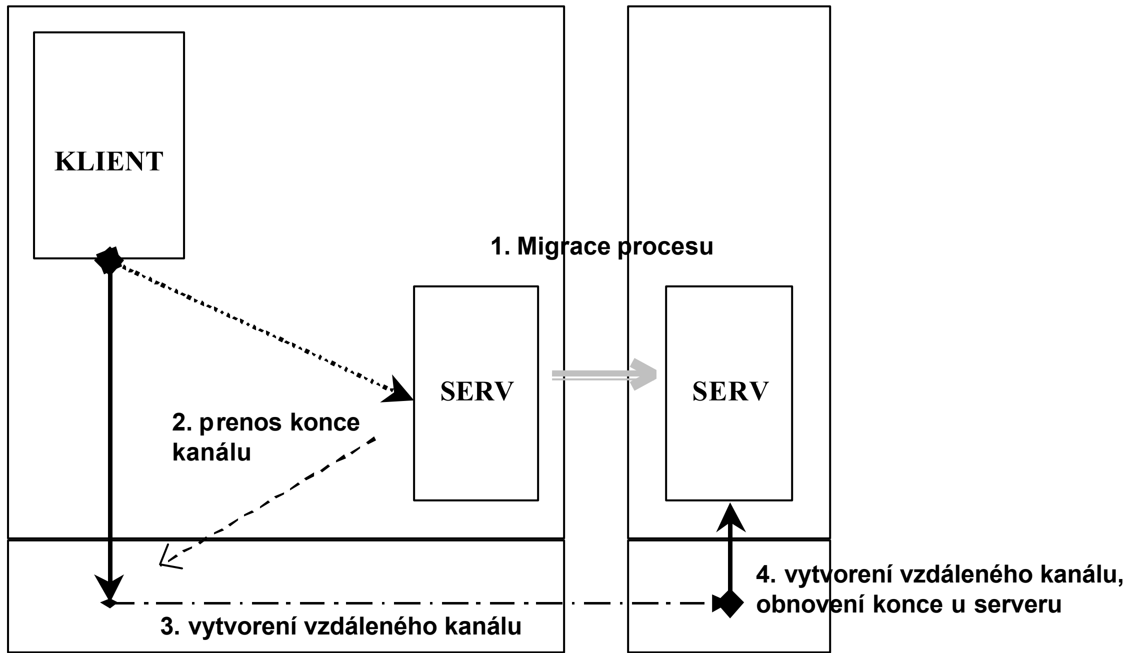
9.3 Komunikace pri migraci



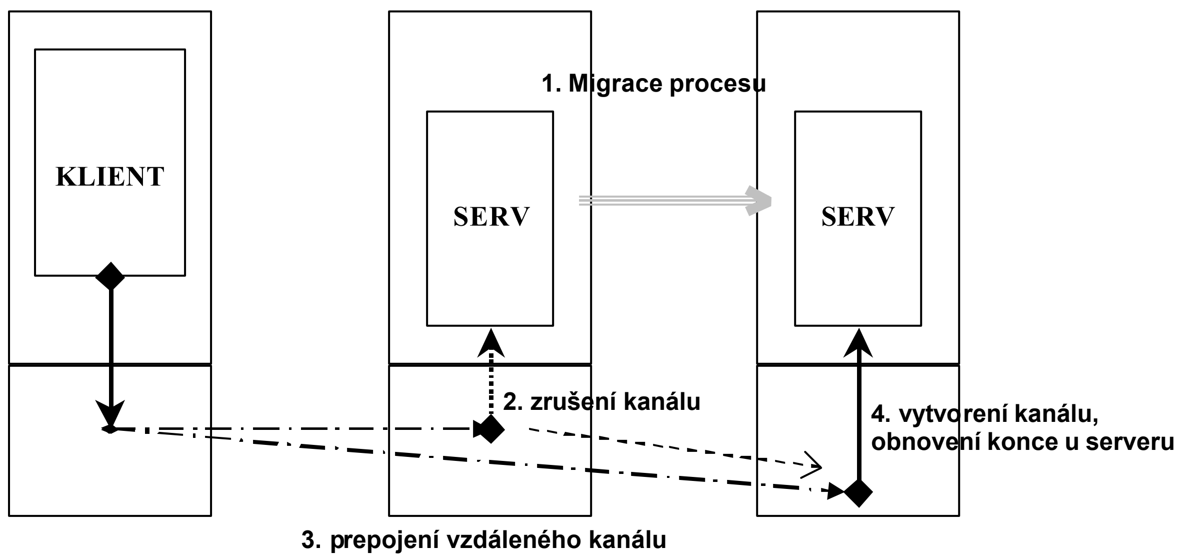
Obr. 49 - Lokální komunikace klient - server



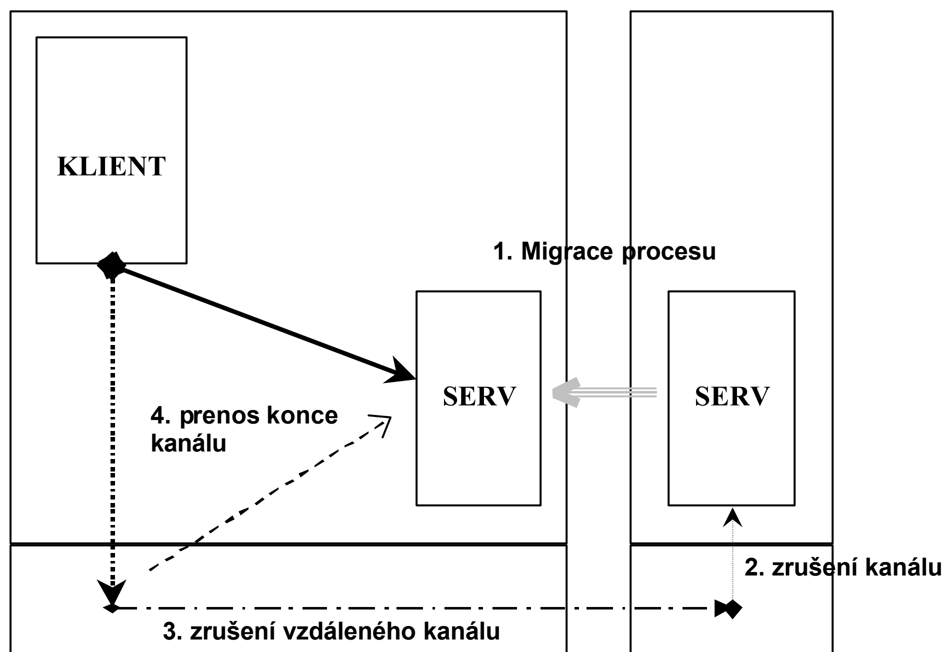
Obr. 50 - Vzdálená komunikace klient - server



Obr. 51 - Migrace serveru vzhledem k lokálnímu klientu



Obr. 52 - Migrace serveru vzhledem k vzdálenému klientu



Obr. 53 - Migrace serveru ke klientu

9.4 Vyvažování záteže

Kontinuální prerozdelování práce mezi procesory, dlouhodobé distribuované plánování, load balancing.

Otázky k vyrešení:

1. Rozhodnutí o okamžiku migrace
2. Jak porovnávat zatížení procesoru
3. Udržování konzistence údaju
4. Volba migrujícího procesu
5. Volba príjemce
6. Prenos procesu
7. Vzdálený beh procesu

9.4.1 Párový algoritmus (Bryant & Finkel)

Dynamicky se vytvárejí páry, které se vzájemne vyvažují

1. Pocítac A pošle nejakému sousedovi B žádost o vytvorení páru. Žádost obsahuje výpis bežících procesu a jejich základní charakteristiky.
2. B po prijetí zprávy:

(a) odmítne žádost; A musí zkoušet jinde

(b) vytvorí s A pár; dokud nebude tento pár zrušen, A i B odmítají další žádosti

(c) oznámí pocítaci A, že práve migruje; A ceká, až se B rozhodne

3. Po vytvorení páru zatíženejší procesor (dejme tomu A) vybere migrující proces podle míry vylepšení $k_i = T_{Ai} / (T_{Bi} + T_{ABi})$, kde

   T_{Ai} je ocekávaný cas odpovedi procesu *i* na pocítaci A

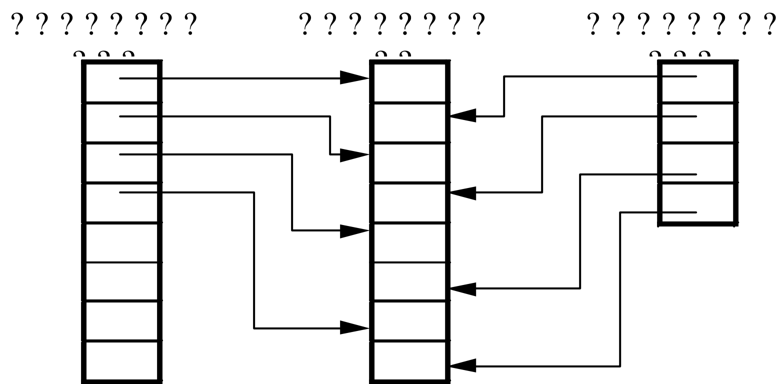
   T_{Bi} je ocekávaný cas odpovedi procesu *i* na pocítaci B

   T_{ABi} je cas prenosu z A na B
4. V prípade, že migrací nelze dosáhnou (významného) zlepšení, pár je zrušen. Jinak je vybraný proces prenesen a pokracuje se od bodu 3.

9.4.2 Vektorový algoritmus (Mosix)

Každý pocítac má pevný malý vektor záteže L ostatních pocítacu, každý pocítac sám periodicky merí vlastní zátež (L_0 = vlastní zátež). Každý casový interval každý pocítac provede:

1. Zjistí vlastní zátež
2. Zvolí náhodne jiný pocítac a pošle mu první polovinu svého vektoru
3. Pri príjmu cásti vektoru záteže (L’) každý pocítac provede $L_{2i} = L_i$, $L_{2i+1} = L'_i$
4. Pri požadavku na migraci se k záteži jednotlivých procesoru pricte komunikacní režie a vybere se nejvýhodnejší procesor



Obr. 54 - Výpocet nového vektoru záteže

9.4.3 Bidding algoritmus (Stankovic & Sidhu)

Využívá McCulloch-Pittsovu vyhodnocovací proceduru. Tato metoda je založena na tzv. vyhodnocovací bunce, která má excitátory, inhibitory a jednohodnotový výstup. Výstup je soucet hodnot excitátoru anebo nula v prípade nastavení libovolného inhibitoru.

1. Všechny procesy jsou pravidelne vyhodnocovány, vstupy jsou vlastnosti procesu a síte.
2. Jestliže výstup je vyšší než prahová hodnota, proces je v porádku. Jestliže je výstup nižší, ale nenulový, mel by být prenesen. Jestliže je výstup nula, je nekterý z inhibitoru nastaven a proces nemuže být prenesen.
3. Jestliže má být prenesen alespon jeden proces, pocítac vyšle až do vzdálenosti *d* broadcast typu ‘žádost o nabídku’ (RFB) obsahující charakteristiky nabízeného procesu. Každý pocítac, který obdrží RFB, tento proces ohodnotí a v prípadne nadprahové hodnoty vrátí odesilateli odpoved nabídku.

4. Po urcité dobe t se všechny odpovedi zkorigují o cenu prenosu a nejlepší nabídka je považována za potenciálního príjemce procesu. Jestliže po dobu t žádná nabídka nedošla, zvetší se vzdálenost d a opakuje se bod 3.

Problém: obtížná kvantifikace vlastností procesu

9.4.4 SLA algoritmus (Stankovic)

Stochastic learning automata - zpetné ucení

9.4.5 BDT algoritmus (Stankovic)

Bayesian Decision Theory - posílání globálních stavu

9.4.6 Centralizovaný algoritmus

Jeden centrální server, zná zátež všech pocítacu, velí

9.4.7 Lokální algoritmus

Každý pocítac zjišťuje pouze lokální zátež. Jestliže ta prekrocí prahovou hodnotu, náhodne se vybere n pocítacu, tem se zašle žádost o migraci a první / nejlepší odpoved se akceptuje.

9.4.8 Porovnání distribuovaných vyvažovacích algoritmu

Z hlediska efektivity a režie vlastního algoritmu je pro reálné distribuované systémy nejzajímavejší vektorový algoritmus, který umožnuje získat globálnejší predstavu o záteži systému s dostatecne malou režií. Ostatní algoritmy se ukazují být pro reálné použití príliš nárocné.

Duležitým hlediskem je i komunikacní složitost jednotlivých algoritmu. U párového a bidding algoritmu pocítace posílají detailní popisy jednotlivých procesu. To zvyšuje zátež síte a pri vetším množství pocítacu sít zahlcuje. Naopak ostatní algoritmy (vektorový, SLA, BDT) prenášejí pouze globální stavovou informaci, címž jsou z hlediska komunikacní složitosti efektivnejší.

Dalším problémem je aktualizace údaju. Pouze párový algoritmus zajišťuje, že rozhodnutí jsou provádena na základe aktuálních informací - pocítace komunikují po párech. Zajišťuje to ovšem pouze lokální optimalizaci.

10. Správa prostredku

Prostredkem (resource) rozumíme jakoukoliv relativne stabilní hardwarovou nebo softwarovou cást systému využitelnou uživateli, resp. jejich procesy. Prostredky mužeme rozdelit na:

1. **fyzické prostredky**: permanentní fyzické cásti systému - procesor, pamet, disky, I/O zarízení, interní zarízení (hodiny, casovace), apod.
2. **logické prostredky**: softwarové entity - procesy, soubory, sdílená data, apod.

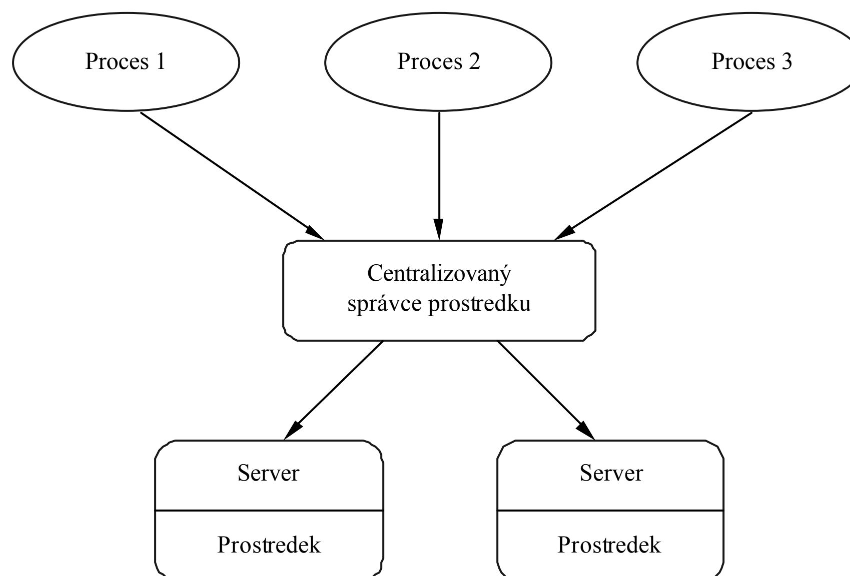
V distribuovaných systémech jsou prostredky fyzicky decentralizované, stavová informace je rozprostrena mezi jednotlivými uzly. Nekteré prostredky mohou být také z duvodu vyššího výkonu systému replikovány na více uzlech. To vše má za následek vyšší režii správy prostredku v distribuovaných systémech oproti centralizovaným systémum.

10.1 Správci prostredku

Prostredky jsou vetšinou pasivní, jsou spravovány servry - správci prostredku (resource managers). Správce prostredku má za úkol:

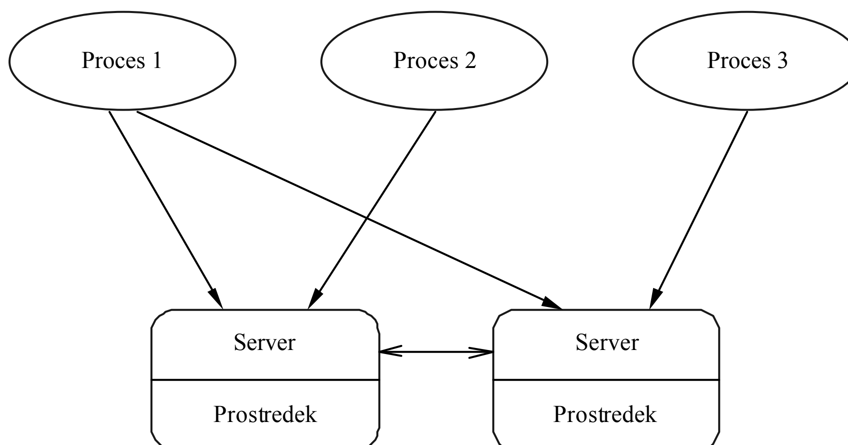
1. Udržovat spravované prostredky - lokaci, konzistenci, . . .
2. Pridelovat prostredky uživatelským procesum na jejich žádost
3. Dbát o ochranu prostredku pred nepovolaným zásahem

10.1.1 Centralizovaná správa prostredku



Obr. 55 - Centralizovaná správa prostredku

10.1.2 Distribuovaná správa prostredku



Obr. 56 -Distribuovaná správa prostredku

10.1.3 Správa prostredku pomocí agentu

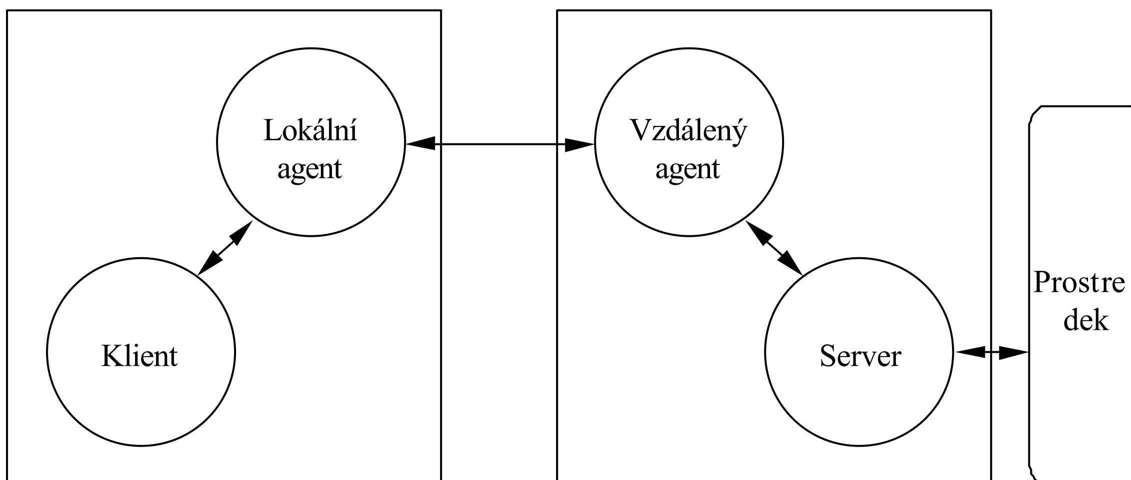
Uživatelské procesy komunikují se servry prostrednictvím agentu. Lokální agent komunikuje prímo s uživatelským procesem (klientem), vzdálený (remote) agent komunikuje s vlastním servrem poskytujícím služby nejakého prostredku.

Úkoly lokálního agenta:

1. Prijímat od klienta požadavky na prostredky
2. Lokalizovat vhodného vzdáleného agenta
3. Vyjednávat s vzdáleným agentem o zapujcení prostredku
4. Vyvolat služby požadované klientem
5. Spravovat vyjímecné stavy - odmítnutí požadavku, migraci prostredku, havárie apod.

Úkoly vzdáleného agenta:

1. Prijímat žádosti klientských agentu
2. Rozhodovat o vyhovení žádosti
3. Vytvorit prípadné exekucní prostredí pro server - nastavení práv, navázání spojení s klientem apod.
4. Spravovat vyjímecné stavy



Obr. 57 - Správa prostredku pomocí agentu

10.2 Zablokování (deadlock)

Zablokování (deadlock) v distribuovaných systémech je podobný problém jako v centralizovaných systémech; ba ješte vetší. Hure se jim predchází, hure se odstranují, hure se dokonce i detekují, protože potrebná informace je rozprostrena mezi nekolika pocítaci.

V literature jsou nekdy rozlišeny dva druhy deadlocku - komunikacní deadlock a deadlock prostredku. Komunikacní deadlock nastává napr. v prípade, že proces A vyšle zprávu procesu B a ceká na odpoved, B pošle zprávu procesu C a C pošle zprávu zpet procesu A. Protože však proces A ceká na odpoved od procesu B, zprávu od procesu C neprijímá, tudíž všechny procesy jsou zablokovány. Vzhledem k tomu, že oba druhy deadlocku mají stejné chování a navíc procesy lze považovat za speciální druh prostredku, nebudeme dále tyto dva druhy rozlišovat.

Podobne jako v klasických nedistribuovaných systémech je i v distribuovaných systémech nejpopulárnejší a nejpoužívanejší tzv. pštrosí algoritmus, tj. problém ignorovat a prípadné rešení deadlocku nechat na uživateli, resp. na uživatelské aplikaci (napr. databáze), která si vše hlídá sama.

Dalšími zpusoby vyporádání se s deadlocky jsou detekce a zotavení (vznik deadlocku je pripušten, po jeho zjištení je rešen násilím) nebo prevence (vznik deadlocku je vyloucen).

10.2.1 Modely deadlocku

Graf prostredku (WFG, wait-for-graph)

?? Model s jedním prostredkem - proces ceká pouze na jeden prostredek

?? Konjunkcní (and) model - musí být uspokojeny všechny požadavky

?? Alternacní (or) model - musí být uspokojen alespon jeden požadavek

?? Model vícenásobných prostredku (multiple resources) - musí být uspokojeno *n* požadavku z *k* dostupných prostredku

10.2.2 Algoritmy detekce deadlocku

Bylo vyvinuto mnoho algoritmu, avšak mnohé z nich byly špatne, a to i pres to, že byla autory "dokázána" jejich správnost. Chyby v dukazech byly v naprosté vetšine zpusobeny absencí spolecné pameti a komunikacními prodlevami.

Centralizovaný algoritmus

Existuje centralizovaný server, který udržuje graf prostredku. Jednotlivé pocítace informují server:

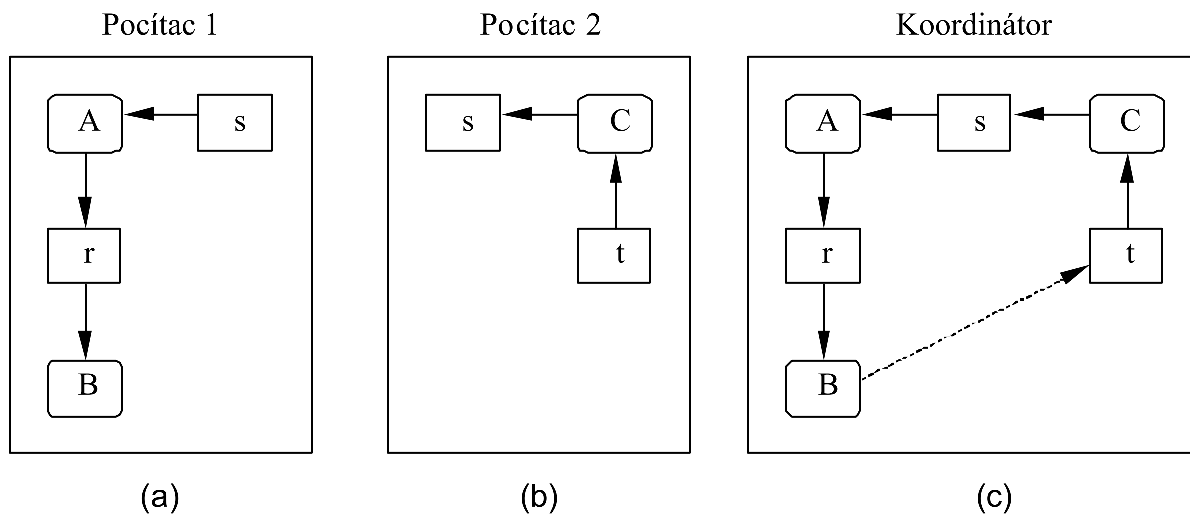
?? Po každé zmene lokálního stavu

?? V pravidelných intervalech

?? Na požádání servru

Problém - mohou vznikat tzv. falešné deadlocky kvuli zpoždení zpráv. Príkladem muže být situace na obrázku. Na pocítaci 1 proces B drží prostredek r, proces A drží prostredek s a ceká na r. Na pocítaci 2 proces C drží prostredek t a ceká na s. Proces B muže provést uvolnení r a požadavek na t. Zpráva od prostredku t (na pocítaci 2) však muže koordinátoru dojít dríve než zpráva o uvolnení prostredku r (na pocítaci 1) a koordinátor muže zjistit deadlock, i když ve skutecnosti k deadlocku nedošlo.

Rešení - globální timeordering, server se pri deadlocku zeptá, jestli nekdo nevyslal zprávu, která by predcházela zpráve o požadavku, který zavinil deadlock.



Obr. 58 - Centralizovaný algoritmus
(a) Lokální stav na pocitaci 1 (b) Lokální stav na pocitaci 2
(c) Možný stav koordinátora po akcích B.release(r), B.alloc(t) - falešný deadlock

Hierarchický algoritmus

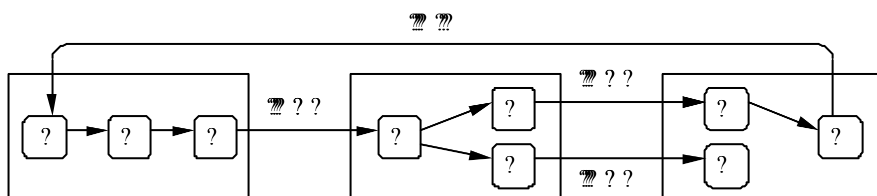
Centralizované rešení s jedním serverem není príliš vhodné (z mnohokrát opakovaných duvodu) pro rozsáhlejší systémy. Existuje proto trída algoritmu, které detekují deadlocky pomocí hierarchické struktury koordinátoru. Každý uzel reší deadlocky lokálne, koordinátor nejnižší úrovne reší deadlocky

jemu podrízených uzlu, a teprve když deadlock mírí do jemu nepodrízených uzlu, predá rešení koordinátoru vyšší úrovne. Príkladem takového algoritmu je Menasce & Muntz ’79.

Distribuovaný algoritmus - CMH

Príkladem distribuovaného algoritmu detekce deadlocku je algoritmus Chandy-Misra-Haas (CMH). Iniciátor pošle všem procesum, kterými je blokován, zprávu, která obsahuje identifikaci iniciátora, identifikaci odesílajícího procesu a identifikaci príjemce. Každý proces, který takovou zprávu obdrží, ji vyšle opet všem procesum, kterými je blokován, anebo zprávu zruší jestliže takové nejsou. Jestliže zpráva dojde zpet k iniciátoru, pak graf obsahuje kružnice a procesy jsou v deadlocku a treba cyklus násilne prerušit.

Jestliže se nekolik procesu spontánne rozhodne být zároven iniciátorem, algoritmus bude stále fungovat, avšak kdyby mel být pri zjištení deadlocku zabit proces, který algoritmus inicioval, docházelo by zbytecne k zabití více procesu - overkill. Rešením tohoto problému mohou být jednoznacné casové znacky pridelené procesum. Zabit potom nebude iniciátor, ale proces s nejvetší casovou znackou, kterému iniciátor zašle zprávu o deadlocku.



Obr. 59 - Distribuovaný detekcní algoritmus

Detekce globálního stavu

Detekce deadlocku muže být založena na detekci globálního stavu systému. Algoritmus (Chandy & Lamport) nezjištuje aktuální stav systému, ale pouze konzistentní stav. Graf prostredku (WFG) má však jednu podstatnou vlastnost: pokud vyloucíme násilné odebírání prostredku ci ukoncování procesu, pak

Jestliže WFG ? WFG’ a proces *p* je v deadlocku v WFG, pak je v deadlocku i v WFG’

Proto jestliže je zjištena kružnice v WFG’, který byl zjišten algoritmem detekce globálního stavu, pak došlo k deadlocku a je treba nekterý proces zabít.

10.2.3 Prevence deadlocku

Prevence deadlocku v distribuovaném systému je založena na stejném principu jako u centralizovaných systému - vytvorit takový systém alokace a držení prostredku, aby k deadlockum vubec nemohlo dojít.

První použitelnou (alespon v nekterých systémech) metodou je jednoznacné usporádání všech prostredku. Každý proces potom muže alokovat prostredky pouze v rostoucí posloupnosti, tj. mohou být alokovány pouze ty prostredky, které jsou v daném usporádání ohodnoceny výše, nežli všechny dosud naalokované prostredky daného procesu. Tímto zpusobem nemohou vznikat cykly, tedy ani deadlocky.

Další dva zpusoby prevence deadlocku jsou založeny na tom, že každé transakci je pri startu pridelena jednoznacná casová znacka. V okamžiku alokace prostredku drženého jinou transakcí jsou porovnány casové znacky obou transakcí. Zablokování je povoleno pouze starší transakci (tj. s menší

casovou znackou), mladší transakce je zrušena. Potom budou zretezeny transakce s rostoucí casovou znackou, nemuže tudíž dojít k deadlocku. Tento algoritmus bývá casto nazýván wait-die (starší ceká, mladší umre). Druhým zpusobem rešení konfliktu je ukoncení transakce, která drží konfliktní prostredek. V prípade, že prostredek drží starší transakce, mladší transakce ceká, v prípade že prostredek drží mladší transakce, je tato ukoncena a prostredek je pridelen starší transakci. Tento algoritmus je nazýván wound-wait.

	starší ? mladší	mladší ? starší
wait-die	starší ceká	mladší umre
wound-wait	mladší umre	mladší ceká

Tab. 18 - Prevence deadlocku

10.3 Ochrana prostredku

Prístupová matice - kapability / Access Control List (prístupový seznam)

Zabezpecení - kryptografie - verejné tajné klíce, ...

10.3.1 Klasifikace bezpecnosti DoD - the Orange Book

- ? **Trída D** - Nezabezpecené
- ? **Trída C** - Volitelná ochrana
  - ? **C1** - Zabezpecení objektu pred nepovolaným prístupem
  - ? **C2** - Login, izolace bezpecnostních mechanismu
- ? **Trída B** - Povinná ochrana - všechny objekty musí být nutne zabezpeceny
  - ? **B1** - Úrovnová ochrana - každý objekt má prirazenu klasifikaci úrovne, uživatel má prístup k objektum klasifikovaným menší úrovní ochrany, než jakou má uživatel
  - ? **B2** - Celý systém musí být napsán s využitím formálního bezpecnostního modelu, který musí být zdokumentován. Musí být identifikován každý kanál umožnující potenciálne únik informací a tento musí být zabezpecen
  - ? **B3** - Musí být implementován referencní monitor a bezpecnostní domény - seznamy uživatelu a skupin s prístupovými právy k objektu a bez jakýchkoliv práv prístupu
- ? **Trída A** - Dokazatelná ochrana - vyžaduje kompletní formální dukaz bezpecnosti

11. Správa souboru

Správa souboru v distribuovaných systémech je implementována množinou file servru, což jsou typicky bežné uživatelské procesy. V nekterých systémech jsou file servry nebo nekteré jejich cásti implementovány prímo v jádru systému. Jednotlivé file servry mohou poskytovat ruzné služby - v jednom systému mohou být vedle sebe jak server poskytující bežné souborové služby, tak i server poskytující bezpecné transakcní služby. Podobne muže být v jednom systému server umožnující prístup k UNIXovým souborum a zároven server poskytující služby filesystému MS-DOSu.

Identifikacní a adresárové služby

Adresárový modul		Kontrola prístupu
Umistovací modul	Lokacní modul	Relokacní modul

Replikacní služby

Konzistence dat	Multiple copy update

Transakcní služby

Transakcní operace	Modul zotavení	Kontrola konkurence

Souborové služby

Prístup k souborum	Prístup k atributum

Blokové a diskové služby

Modul správy bloku	Ovladace zarízení	Caching

Tab. 19 - Možná struktura distribuovaného filesystemu

servry dedikované / uživatelské

jeden fileserver / oddelené souborové a adresárové služby

servry stavové / bezstavové (stateles)

více servru - soubory disjunktní / replikace

11.1 Diskové služby

11.1.1 Bloková struktura souboru

11.1.2 Systémová identifikace

UFID - identifikace serveru ? interní císlo souboru (? zabezpecení)

UFID ? císlo bloku ? lokální / vzdálený blok

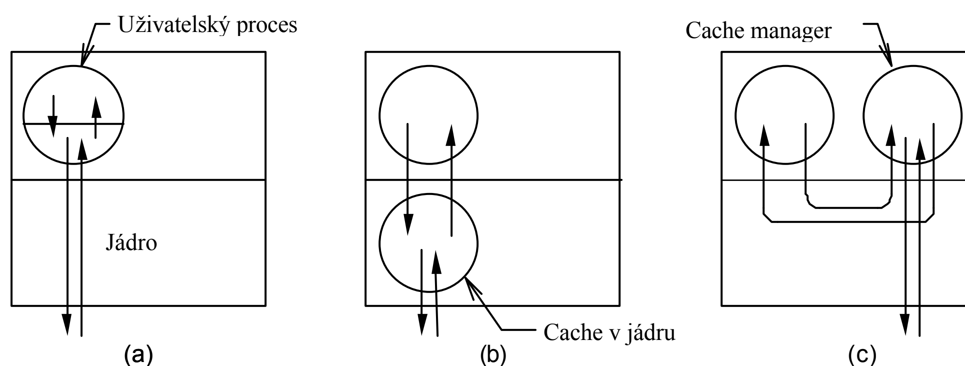
11.1.3 Vyrovnávací pamet (cache)

Cache muže být v zásade umístena na nekolika ruzných místech: v pameti serveru, v pameti klienta nebo na disku klienta.

Cache v pameti serveru snižuje pocet prístupu na disk. Vzhledem k tomu, že pamet fileserveru bývá až rádove menší nežli kapacita disku, je treba rozhodnout, co bude umísteno v cache. Duležitým rozhodnutím musí být, jak velké kusy dat bude cache udržovat. Jednou možností je udržovat celé soubory. V tomto prípade muže být prístup na disk prováden najednou, anebo po velmi velkých kusech, a tudíž (relativne) velmi rychle. Druhou možností je cachovat soubory po blocích nebo stránkách. Toto rešení má naopak výhodu v tom, že pamet dostupná pro cache je daleko lépe využita. Cache v pameti serveru je pro klienty zcela transparentní - server si sám serializuje požadavky a provádí prístup na disk, klient se o kopii souboru v pameti vubec nedozví.

Aby se snížila zátež síte, musí být cache na klientské strane. Zde jsou principálne dve možnosti, kam cache umístit - do pameti anebo na disk. Cachování souboru na lokálním disku je výrazne pomalejší a je proto efektivní pouze u prístupu po pomalých sítích (napr. WAN). Na druhou stranu má tu výhodu, že umožnuje cachovat relativne velký pocet velkých souboru.

Pro umístení cache v pameti klienta jsou v zásade tri možnosti: v adresovém prostoru procesu, v jádru, nebo jako specializovaný cache manager.



Obr. 60 - Caching v pameti klienta
(a) Cache v adresovém prostoru procesu (b) Cache v jádre (c) Cache manager

Protože cache zpusobují docasnou nekonzistenci mezi daty udržovanými lokálne a daty na fileserveru, prípadne daty u jiných klientu, je treba nejakým zpusobem konzistenci zajistit. Jedním rešením je použít write-through cache - v prípade zápisu do cachovaného souboru se zápis pošle zároven i fileserveru. To však nereší problém více klientu pristupujících ke stejnému souboru. Navíc tato metoda nijak nešetrí provoz síte pri zápisu do souboru.

Jinou metodou je posílat zápisy serveru ne pri každé zmene dat, ale až po urcité dobe - tzv. delayed-write. Tento zpusob mimo jiné efektivne reší problém docasných souboru. Soubor, který je treba jen krátkou dobu na uchování mezivýsledku, tak nemusí být vubec na fileserver prenášen. Odkládání zápisu má však jednu nevýhodu - zvláštní sémantiku. To, co jiné procesy vidí, silne závisí na case. Dalším rešením konzistence cachí je write-on-close, tj. zápis až pri zavrení souboru.

Zcela jiným zpusobem rešení konzistence je centralizovaná kontrola. Fileserver má odkazy na všechny cachované soubory a konzistenci reší sám.

11.2 Souborové služby

struktura souboru

?? posloupnost bajtu

?? záznamy

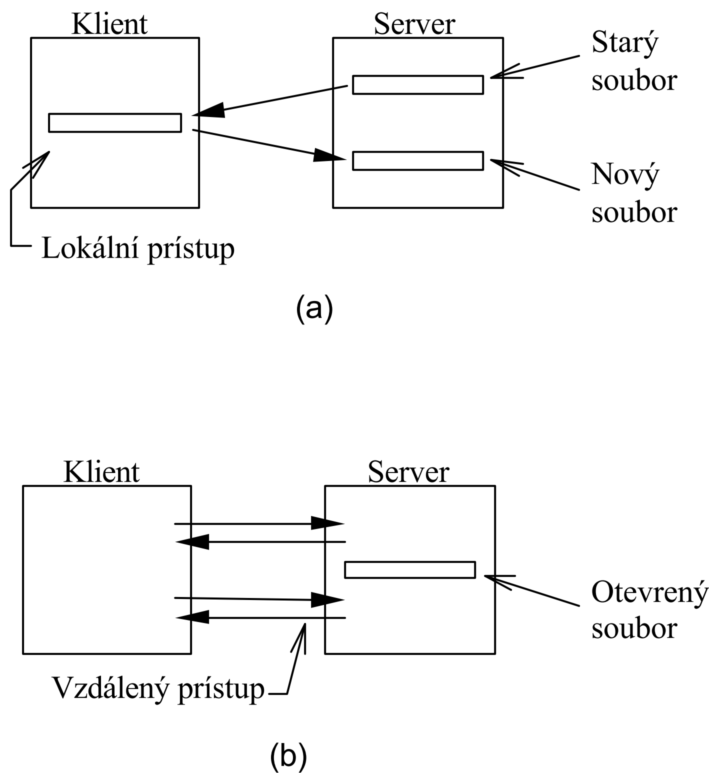
?? typovaný
?? virtuální adresový prostor

atributy - per soubor, ne per adresár entry

mutabilní / imutabilní

kapability / prístupové seznamy (Access Control List)

upload model / remote access model



Obr. 61 - (a) Upload/download model (b) Vzdálený prístup

11.2.1 Sémantika sdílení souboru

Jestliže více procesu soucasne pristupuje k nekterým souborum, je treba presne definovat vzájemné chování ctení a zápisu. V centralizovaných systémech bývá sémantika sdílení jednoznacná - každé ctení vrátí hodnotu uloženou pri posledním zápisu. Dosažení této sémantiky však v distribuovaných systémech není jednoduché. Prvním problémem je neexistence presného globálního usporádání událostí. Proto ctení na jednom procesoru, které se událo nekolik mikrosekund po zápisu na druhém procesoru, nemusí vrátit naposledy zapsanou hodnotu. Druhým problémem je cachování. Každý z konkurentních procesu si sám cachuje soubor a zmeny provádí lokálne. Možné rešení by bylo implementovat write-through cache, tj. všechny zmeny by se ihned promítly do souboru spravovaném fileserverem. To však ješte nezarucuje plné sdílení - procesy, které již mají soubor nacachovaný, by se stejne zmenu dat nedozvedely.

Jiným rešením tohoto problému je uvolnení sémantiky. Výše popsané chování zustává, avšak je prohlášeno za korektní. Presneji receno veškeré zmeny otevreného souboru jsou viditelné pouze lokálne, zviditelní se až pri zavrení souboru. Takováto sémantika bývá nazývána relacní (session semantics).

Zcela odlišným zpusobem je rešeno sdílení imutabilních souboru. Zde již neexistují akce "otevrení souboru na zápis" apod., ale pouze CREATE a READ. Jestliže nejaký proces vytvorí nový soubor a uloží ho pod již existujícím jménem, pak stará kopie již není dostupná (alespon ne pod starým jménem). Otázkou je, co se stane, jestliže jiný proces cte soubor, který byl prepsán nove vytvoreným. Bežným rešením je nechat procesu otevrený starý soubor, a to i když na nej byl odstranen odkaz ze všech adresáru. Jiným rešením je po zjištení, že soubor byl zmenen, další pokus o ctení vrátit s chybou.

Pokud systém podporuje transakce, je možno sémantiku sdílení souboru prenechat transakcnímu manažeru, který sám implementuje potrebné mechanismy (zamykání, casové znacky apod.).

Centralizovaná sémantika	Každá operace je ihned viditelná
Relacní sémantika	Zmeny jsou viditelné až po zavrení souboru
Imutabilní soubory	Soubory nelze menit
Transakce	Zabudované transakcní mechanismy

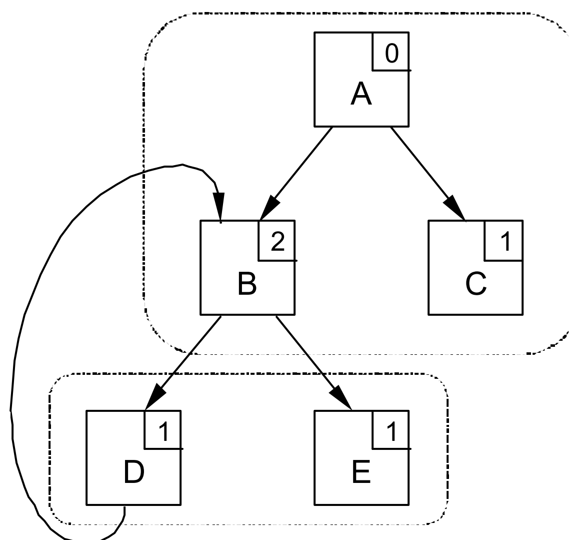
Tab. 20 - Zpusoby sdílení souboru

11.3 Adresárové služby

mapování uživatelských jmen na systémová

hierarchický systém souboru

linky, graf adresáru - ! mazání linku



Obr. 62 - Graf adresáru

11.3.1 Prostor jmen adresáru

jednoduchá jména - napr. uzel : adresár/jméno

montování do lokálního prostoru

jednotný celosvetový prostor jmen / kontexty

11.4 Replikace

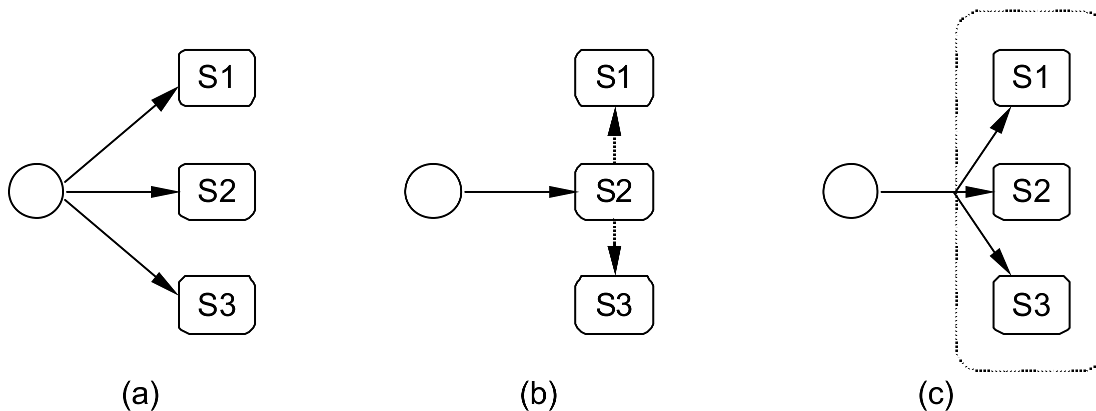
Distribuované filesystemy casto umožnují replikaci souboru, tj. udržování více kopií všech nebo jen nekterých souboru na více fileserverech. To má nekolik dobrých duvodu:

1. Spolehlivost (reliability) - pri výpadku jednoho serveru nejsou ztracena data
2. Dostupnost (availability) - k souborum lze pristupovat i v prípade výpadku jednoho nebo nekolika serveru
3. Výkon (performance) - lze pristupovat k nejbližším datum, výkon fileserveru je rozdelen mezi nekolik serveru

?? **explicitní replikace** - uživatel se sám stará o udržování konzistence

?? **odložená replikace** - zápis do primární repliky, aktualizace sekundárních replik pozdeji

?? **skupinová komunikace** - zápisy jsou simultánne zasílány všem dostupným replikám



Obr. 63 - (a) Explicitní replikace (b) Odložená replikace (c) Skupinová komunikace

11.4.1 Aktualizacní protokoly

Pri zápisu replikovaného objektu je treba zajištení konzistence replik. Pokud by všechny servry spravující repliky byly vždy dostupné, pak by dosažení konzistence nebyl žádný problém. To je však prakticky nesplnitelný požadavek - vždy je možnost, že nekterý server je nefunkcní nebo nedostupný. Proto je nutné použít protokol umožnující zápisy i v neideálních podmínkách, kdy je dostupná pouze cást potrebných serveru.

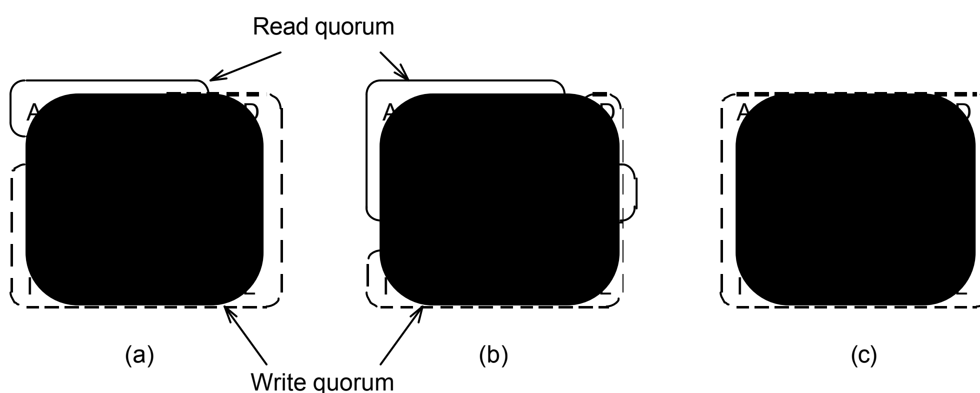
Nejjednodušší metodou je tzv. **primární replika**. Všechny zmeny se provádejí na jedné vybrané replice a tento server se sám stará o aktualizaci ostatních. Toto rešení má jednu vážnou nevýhodu - v prípade nedostupnosti primární repliky nelze s objektem pracovat.

Rešení nabízí tzv. hlasování (voting). Základní myšlenkou tohoto prístupu je získání dostatecného poctu hlasu pro povolení prístupu k objektu. Klient pred prístupem k objektu pošle žádosti všem dostupným serverum a ceká na povolení. Jakmile obdrží dostatecný pocet hlasu (potvrzení prístupu), muže provést požadovanou operaci. Jestliže dostatecný pocet hlasu nezíská, nelze operaci provést. Podle toho, co presne znamená "dostatecný pocet hlasu", rozeznáváme nekolik druhu protokolu.

Nejjednodušším aktualizacním protokolem založeným na hlasování je tzv. **vetšinové hlasování** (majority voting). Je založeno na tom, že pro jakoukoliv operaci (ctení nebo zápis) je treba získat vetšinu (nadpolovicní množství) z celkového poctu serveru. Pri zápisu se všem zapsaným replikám priradí nové císlo verze (všem replikám stejné). Pri ctení klient zažádá o hlasy, odpovedi obsahují císla verze príslušné repliky. Jelikož jak pro ctení, tak pro zápis bylo treba nadpolovicního poctu hlasu, je zaruceno, že v odpovedi je alespon jedna replika s aktuálním císlem verze. Klient si pak vybere ze serveru, jejichž odpoved obsahovala aktuální císlo verze.

Zobecnením vetšinového hlasování je **vážené hlasování** (weighted voting). Pro zápis je treba tzv. **write quorum** (pocet hlasu), pro ctení **read quorum**. Necht celkový pocet replik (serveru) je N, read qourum N_r, write quorum N_W. Pak musí platit $N_W + N_r > N$. (Z tohoto pohledu je vetšinové hlasování pouze speciálním prípadem váženého, kdy $N_W = N_r$). To, že ctecí a zápisové quorum mohou být ruzné, se dá využít pro optimalizaci prístupu. Vetšinou totiž platí, že ctení bývá mnohem castejší než zápis. Proto když ctecí quorum bude menší nežli zápisové, bude režie pro ctení, jakožto castejší operaci, menší.

Singulárním prípadem je read quorum = 1. Pak pro ctení stací prístup k jednomu libovolnému serveru, zápis však potrebuje prístup ke všem serverum. V prípade, že by nekterý server nebyl dostupný, pak by zápis nešel provést. Tento problém reší tzv. **hlasování s duchy** (voting with ghosts). Krome normálních (datových) serveru je vytvoren jeden bezdatový (dummy) server, který slouží k náhrade nedostupných datových serveru. Tento server (duch) se neúcastní hlasování o ctení (protože neobsahuje žádná data), ale muže se úcastnit hlasování o zápisu, resp. jeho hlas je pocítán do write quora. Klient pak pri pokusu o zápis uspeje, jestliže je alespon jeden server reálný.



Obr. 64 - Hlasovací algoritmus pro 12 serveru
(a) N_r=3, N_w=10 (b) N_r=7, N_w=6 (c) N_r=1, N_w=12