Kapitola 22

Distribuované systémy

zdroje

- Middleware a Middleware notes
- Principy distribuovaných systémů a jejich prezentace
- `http://www.kiv.zcu.cz/~ledvina/Prednasky-DS-2007/`

22.1 Pozadavky

Komunikace, zasílání zpráv, RPC. Skupinová komunikace, virtuální synchronie, doručovací protokoly. Middleware (klasifikace, protokoly, RMI, EJB, CORBA, DCOM, SOAP, ...). Logické hodiny a jejich synchronizace. Distribuované synchronizační algoritmy. Distribuovaný konsensus. Distribuované sdílení paměti, konzistenční modely. Souborové a adresářové služby, distribuované souborové systémy (NFS, AFS, CODA, ...), replikace. Distribuovaná správa prostorů jmen, identifikace objektů a přístup k nim, služby (LDAP, JNDI, CORBA Namig/Trading). Procesy v distribuovaném prostředí, migrace procesů, vyvažování zátěže, zablokování.

22.2 Komunikace

Zasílání zpráv

V distribuovaném prostředí je hromada problémů se sdíleným adresním prostorem -> komunikace pomocí zpráv
nespolehlivý unicast – to co nám dává hardware, počítáme s best effort

- v IPv4 se pakety mohou na routerech fragmentovat po cestě, v ipv6 vyrazí s nějakou délkou a s tou i dorazí (pokud po cestě nějaká část sítě neumí danou délku přenést, tak je zahodí).

Spolehlivý unicast

chceme aby nám přišel každý paket a přišel nám jen jednou (exactly once sémantika)

- ochrana proti poškození (duplikace, checksums, parita, křížová parita, CRC)
  - forward error correction – když zjistím poškození, doplňuji další opravné informace
- ochrana proti ztrátě – potvrzování
- ochrana proti duplikaci – unikátní ID paketů (při TCP handshake se dohodne náhodné počáteční a pak roste)
- jiné problémy – když jedna strana spadne, zapomene jaká čísla paketů poslala atd

TCP – emuluje stream, pakety jsou číslované, mají dvoubajtový kontrolní součet

flow/congestion/etc

flow control ochrana proti ucpání příjemce; řeší se posíláním "flow control window" počet/velikost zpráv, které příjemce může dál sežvýkat, posílaný v ACK zprávách

congestion control ochrana proti ucpání sítě; "congestion control windows" si upravuje příjemce podle toho jak se mu ztrácí zprávy

Požadavky na real time/propustnost (soft – většinou splněny, hard – splněny vždy) musí být garantovány hardwarem, nad tím se pak dá zajistit rezervace

- **throughput** – propustnost (množství dat za jednotku času)
- **latency** – jak dlouho to trvá (doba na one way trip nebo roudtrip)
- **jitter** – rozptyl latence (výkyvy latence, dobré pro stanovení rozumné stanovení velikosti bufferů streamových aplikací)

Protokol **RSVP** (reservation protokol)

- odesílatel posílá Path zprávy, aby dal najevo uzlům po cestě že je tam nějaká session co něco chce
- příjemce posíla opačným směrem Resv zprávy, aby vyznačil cestu dalším směrem a dal najevo co se má rezervovat

RTP (Real Time Protocol) přenáší real-time data, k němu je RTCP (Real Time Control Protocol) se statistickými zprávami, ze kterých se vyhodnocují jitter, latency a asi i troughput

RTSP (Real Time Streaming Protocol) používá se k dohadování streamingu, nepřenáší data, ovládá třeba RTP, který ty data nese.

Multicast

Zpráva jde k více příjemcům, ale posílá se jen jednou. (Broadcast – ke všem uzlům v síti).

V TCP/IP se k ohlašování příslušnosti k multicast skupinám používá IGMP (Internet Group Management Protocol). Router s numericky nejnižší IP v každém segmentu periodicky posílá Membership Query, uzly odpovídají Membership report. Uzly, které opouští nebo přichází do skupiny posílají State Change Report Další protokoly řídí routování multicast zpráv.

spolehlivost:

- sender initiated protocols – odesílatel ví o všech příjemcích, ti mu posílají co přišlo; když je příjemnců moc máme ACK implosion problem
- receiver initiated protocols – příjemce ví, co má přijít, jinak posílá NAK; když má hodně příjemců problémy, máme NAK implosion problem
  - dá se řešit pomocí posílání NAK multicastem a čekáním náhodný interval, jestli už někdo neposlal NAK dřív
- stromově, pak node posílá lokální ACK (pro flow control) a agregované ACK (když už přide ACK od všech pod ním, kvůli zapomínání poslaných paketů)
- kruh s tokenem – uzel s tokenem posílá ACK odesílateli, uzly bez tokenu posílají NAK uzlu s tokenem
  - Např.**Reliable Multicast Protocol**. Příjmci v logikém kruhu, jeden uzel má token. Odesílatel pošle multicastem zprávu, uzel s tokenem pošle multicastem ACK spolu s globálním pořadím zprávy. Ostatní uzly mohou poslat NACK (s označením preferovaného příjmece s tokenem??). Každá zpráva obsahuje lokální pořadové číslo. Token se předává po poslání ACK, ten, kdo přebírá token ho nemůže převzít, dokud nedoručil všechny zprávy s nižším pořadovým číslem.

interfacy:

- **blokující** × **neblokující** (s callbackem nebo pollováním)
- **synchronní** (dokončení operace znamená že příjemce zprávu přijal) × **asynchronní** (operace se vrátí hned po odeslání)
- **Pozor**, může být např asynchronní a blokující akce (např, pokud jsou plné odesílací buffery), není to nesmysl... Stejně tak může být podle nějakého výkladu synchronní neblokující akce — např s callbackem, za dokončení se počítá až provedení callbacku.

RPC

Myšlenka je v tom, že na klientu se zavolá funkce, která se provede na serveru. Realizace je nakreslená ve skriptech na PDS,

- Nejprve se vygeneruje UUID rozhraní
- Programátor dodá definici rozhraní (interface definition file)
- IDL kompilátor udělá header, co se naincluduje do klienta i serveru
- IDL kompilátor taky udělá klientskou a serverovou část, která se stará o komunikaci (client stub, server skeleton)

- Programátor dodá implementaci serverové části
- Celé se to zkompiluje a slinkuje.
- Server si zaregistruje u nějakého directory serveru to, že dělá tenhle servis
- Klinet se rozběhne, když si chce zavolat RPC, tak
  - Normálně zavolá fci.
  - Vygenerovaný Stub si vytvoří zprávu (marshaling), předá jádru jádro, look-upne službu na directory serveru, předá jí jádru serveru, to ji dá skeletonu, ten si ji rozbalí (unmarshaling), a putuje to podobně zpět.
- marshalling / unmarshalling – balení a rozbalování parametrů funkcí do zpráv
- rozhraní může být i ze signatury v "normálním" jazyce, jen s nějakými doplňujícími informacemi
  - IDL – popisuje rozhraní funkcí, z něj se pak generují skeletony a stuby; pak mapování do jazyků
- varianta s objekty – máme proxy objekty u klienta a servanty na serveru. Implementace tohohle objektovýho cirkusu je třeba **RMI**.

Paralelizace na serveru

- single threaded
- thread per connection
- per request
- thread pool (šetří se overhead na vytváření vláken)

22.3 Skupinová komunikace

Posílání zpráv od jednoho odesílatele více příjemcům. Problémy

- Atomicita (všem nebo nikomu)
- Synchronizace (nějaké pořadí)
- Adresování (adresování skupin)
- Technické řešení (multicast, posloupnost unicastů, broadcast?)

Otevřenost skupin

- Uzavřené (posílat mohou jen členové)
- Otevřené, posílat může kdokoliv

Uspořádání skupiny

- Rovnocené
- Hierarchické
- S koordinátorem

Virtuální synchronie

Group view – množina uzlů ve skupině (též group membership, delivery list, etc). Značí se L (globální), L_i (lokální verze procesu i), L^x (verze pohledu x), ${L_i}^x$

Algoritmus spolehlivého doručování je virtuálně synchronní, když:

1. všechny uzly ve skupině udržují stejný L
2. pokud je zpráva m odeslána skupině s L^x před změnou na L^{x+1}
   - buď m doručí všechny uzly z L^x před provedením změny na L^{x+1}
   - nebo žádný uzel z L^x, který provede změnu na L^{x+1}, zprávu m nedoručí

Virtuální synchronie vlastně říká, že pokud někdo přibude nebo odejde ze skupiny, tak se všechny uzly shodnou na tom, které zprávy byly odeslány před touto změnou, a které po ní.

Přitom nemusí platit, že přijetí zprávy členem L implikuje doručení všem členům L (nespolehlivost, havárie odesílatele).

Když se uzly B a C dozvědí, že uzel A přestal být členem jejich skupiny, už od něj pak nemůžou přijímat skupinové zprávy.

doručovací protokoly

Rozlišením mezi přijetím a doručením zaručují nějaký typ uspořádání, nejčastěji kauzální uspořádání

- Source ordering — zprávy vyslané jedním uzlem dojdou v pořadí, vjakém byly odeslány
- Causal ordering — zprávy dojdou podle seřazené podle kauzální závislosti
- Total ordering — všichni účastníci komunikace vidí zprávy v nějakém, ale stejném pořadí pro všechny

Doručování zajišťující uspořádání zpráv

- Myšlenka je v rozdělení přijetí a doručení zprávy. Ikdyž zprávu fyzicky mám, logicky jí nedoručím, dokud není správný čas...

Kauzální doručování pro jednu skupinu

- Vektorové hodiny pro každou zprávu a každý proces (dékla je počet procesů ve skupině). U procesů nastaveny na začátku na samé nuly.
- Při odesíláni zprávy si svoji složku vektoru zvednu o jedna, a svůj stav vektorových hodin přibalím ke zprávě
- Po přijetí zprávu pozdržet, dokud značka ve zprávě nebude
  - V hodnotě náležeící odesílateli o jedna menší, než má přijímající proces (mám předešlé zprávy od něj)
  - V ostatních hodnotách menší nebo rovna než hodnoty co má k dispozici přijímající vektor (kauzální závislost na ostatních)

Kauzální doručování pro překrývající se skupiny

- Totéž, ale posílaji se vektory vektorových hodin (s každou zprávou včechny vektory skupin, ve kterých je odesilatel)
- Musí být zaručena kauzalita vzhedem ke skupině, **kam to je poslané**
- Pro všechny ostatní skupiny, jihž je příjemce členem, musí být také zaručena kauzalita.

Total Order protokol

- Odesílatel rozešle zprávu, všichni mu na ni odpovědí zprávou s timestamp, kdy ji dostali a odesilatel odešle finalizační zprávu s nejpozdější časovou známkou, (se známkou toho, kdo ji dostal nejpozději). Tento nejvyšší čas, který všechny procesy dostanou ve finalizační zprávě je čas, kdy mohou zprávu doručit.

Spolehlivé doručovací protokoly

záplavový algoritmus při každém přijetí zprávy, kterou uzel ještě neviděl, ji pošle všem ostatním – spolehlivé a neefektivní

algoritmus s potvrzováním

- p_s odesílatel, p_r (z L) je příjemce, p_x uzel co havaroval
- p_s odešle zprávu všem v L, schová ji dokud nedostal ACK od všech, nebo než nazná že zhavarovali
- p_r po příjmu odešle ACK p_s, zprávu si schová než zjistí že všichni ostatní přijali
- jestliže p_r zjistí, že p_s havaroval, odešle zprávu všem z L, o nichž neví, že ji dostali

Jak p_r zjistí které uzly zprávu přijaly?

- Třeba tím že se ACK posílají taky všem – neefektivní, pokud se to stejně nedělá broadcastem nebo multicastem
- Ack se dají nalepovat na jiné zprávy, a využitím kauzality – při korektním kauzálním doručování jsou potvrzení tranzitivní
  - D může z příjmu a, b+ACK(a), c+ACK(b) odvodit, že B přijal a, a C přijal a i b
  - z příjmu b+ACK(a), c+ACK(b) může D odvodit, že B přijal a, C přijal a i b, a taky že A poslal a (a vyžádat si jeho resend)

====== Trans algoritmus ====== Uzel si udržuje kauzální graf zpráv které přijal a ještě je nemají všichni, z došlých potvrzení si vypočítává další, přeposílá zprávy od kterých dostal NAK, detekuje stabilní zprávy. Na protokol se dá hledět tak, že posouvá stable line grafem kauzální závislosti. Každý uzel si udržuje jen graf zpráv, které přijal ale ještě nejsou stabilní. Z toho plyne, že když někdo zhavaruje a zpráva se nestane stabilní, může mít neomezenou paměťovou náročnost. Je tedy potřeba doplnit o členství ve skupinách — transis algoritnus

====== Transis algoritmus ====== spolehlivý kauzální multicast + členství ve skupinách. Při pomalé odpovědi vyhodí uzel ze skupiny, ten se pak musí vrátit explicitně.

Založeno na konzistentních změnách pohledů a doručování v rámci pohledů.

- Při detekci havárie procesu p zpráva FAULT(p)
- kauzální hranice pohledu – vynutí doručení kauzální předcházejících zpráv, pozdrží zprávy následující (doručí se až v L^{x+1})
- dvě havárie zároveň – společná hranice
- kauzální doručení zpráv havarovaného procesu vzhledem ke změně pohledu
  - zprávy odeslané kauzálně před zjištěním havárie se doručí v L^x
  - zprávy odeslané konkurentně se zjištěním havárie se zahodí
  - zprávy odeslané kauzálně po zjištěním havárie se zahodí

====== ISIS protokol ====== Maticové hodiny – každý proces si udržuje vektor s odeslanými zprávami (sem si uklada, pri prijeti nejake dalsi zpravy od prijemce cas odelsani posledni zpravy od nej, kterou prijemce odstal), a zároveň vektory co mu došly od všech ostatních. Z toho zjistí které zprávy už mají všichni (a jsou stabilní).

Zaručení synchronie: když se proces dozví o novém pohledu, rozešle všechny nestabilní zprávy a potom potvrzení instalace (flush message), když pak dostane flush od každého procesu, může nový pohled nainstalovat. Každý proces si udržuje seznam "havarovaných" procesů dle aktuálního pohledu, ten se posílá se zprávami a sjednocuje při příjmu. Zprávy od havarovaných se zahazují.

řazení zpráv

- source ordering – zprávy jednoho odesílatele dorazí v takovém pořadí v jakém je poslal
  - stačí sekvenční číslování lokálně odesílatelem
- causal ordering – zprávy o události dorazí před zprávami o jejích následcích
  - pomocí vektorových hodin (see #Vektorové hodiny)
- total ordering – všem příjemcům přijdou všechny zprávy ve stejném pořadí
  - sériová čísla zpráv vydává centrální autorita

22.4 Middleware

Midleware je software umožňující nebo usnadňující běh aplikací nějakým způsobem rozložných po více počítačích...

Klasifikace

převzato z VŠE, berte s rezervou :-)

- communication middleware – zajišťuje přenos zpráv, vzdálené vykonávání kódu a tak podobně
  - synchronní – RPC, RMI (Remote Method Invocation) – SunRPC, DCOM, CORBA, Java RMI, SOAP
  - asynchronni – Message-Oriented Middleware
- data management middleware – přístup k datům
  - Remote Database Access – ODBC, JDBC, ADO.NET
  - Remote File Access
- platform middleware – běhové prostředí
  - Transaction (Processing) Monitor (TPM) – zajišťuje transakce – EJB — Java Transaction Service
  - Object Request Broker (ORB) – RPC v objektovém prostředí (+ life cycle services, naming services, ...)
  - Message Broker – zajišťuje doručení zpráv a tak (JMS – Java Message Service)

  - Application Server – kontejnery, které zajišťují standardizované služby pro aplikace (CORBA, .NET, J2EE EJB – perzistence, transakce, kontrola souběžných přístupů)

jiná struktura, podle lokiho

- messaging
- RPC
- data access
- kontejnery

Protokoly

Moc nevím, co tím myslely, možná třeba vědět co a k čemu jsou (už dřív popsané), které midleware může používat

- Reliable Multicast Protocol
- Resource Reservation Protocol
- Realtime Protocol
- Realtime Streaming Protocol
- RPC

Tohle tu bylo, ale podle mě je protokol jen SOAP. MPI je knihovna (rozhraní) a .NET Remoting je buhvico, ale asi ne protokol...

- SOAP – založené na XML, určené pro web services (zpráva obsahuje hlavičky pro routující systémy a tělo pro koncového příjemce)
- MPI (Message Passing Interface) – C/C++/Fortran knihovna (rozhraní) pro psaní paralelních aplikací komunikující pomocí zasílání zpráv, zprávy mají definovaný formát, jsou přenositelné, umí p2p, ale i skupinové metody broadcast, scatter, gather, reduce volitelnou funkcí). Komunikující procesy rozděluje do skupin...

.NET Remoting

Objektové RPC (podobné Java RMI). Hlavní idea je komunikace mezi doménami (různými aplikacemi), spojením se síťovými službami je pak jedno, jestli aplikace běží na jednom PC a povídají si přes paměť, nebo jestli běží na jiných strojích a posílají si zprávy.

- Marshaling používá tzv. data sink. Sink je interface, který zajišťuje zabalení zprávy (dostane objekt vyplivne stream dat). Jsou předimplementované 2 sinky (binární a HTTP). Binární kóduje data do vlastního formátu (klasická serializace objektů) a komunikuje přes TCP. HTTP balí data do SOAP zpráv a používá HTTP protokol, stejně jako třeba web services. Binární je efektivnější, HTTP zase lépe prolézá přes firewally.
- Při vytváření serveru se otevře zvolený port. Na něm poslouchá .NET a může na něm viset víc služeb. Služby se identifikují unikátním řetězcem. Každá služba může být
  - Singleton (jeden remote objekt sdílený přes všechna volání všech klientů)
  - Single Call (pro každé volání se vytvoří nový objekt, na kterém se volání provede)
  - Alternativně je možné používat Client Active, kdy si klient sám řeší vytváření instancí (ale to už je vyšší dívčí).
- Data se kopírují klasicky, objekty se předávají referencí (.NET automaticky generuje proxy objekty).
- Každý vzdálený objekt dostane time lease (dobu, jakou je platný) a sponsors. Když vyprší lease, ptáme se sponzorů jestli to ještě chtějí – stačí kladná odpověď od jednoho, takže je to lepší než pingování.

RMI – Remote Method Invocation

Jde vlastně o objektové RPC...

see also RMI tutorial

Objekty implementují Remote interface (kde navíc mohou házet RemoteException), implementace dědí z RemoteObject.

- Třída UnicastRemoteObject pro dočasné objekty
- Třída Activatable pro perzistentní objekty
- Klient si vyžádá referenci na objekt třeba v RMIregistry (naming udělátko).
- Lokálni stub (proxy), remote implementace
- Využití standardní serializace typů.

Lifecycle se vzdáleně řeší pomocí počítání referencí a keepalive zpráv – klient si při rozbalování příchozí reference na objekt vyžádá tzv. lease, ten pak periodicky obnovuje. Nakonci lease vrátí.

CORBA

wen:CORBA, CORBA EXPLAINED SIMPLY

- IDL, pak mapování do různých jazyků.
  - Problémy např s délkami integerů v C a C++, řeší se mapováním na třídy nebo typedefy.
  - Umí i složitější datové typy, např struktury, uniony, stringy, sekvence, pole, inteface typy (reprezentují objekty předávané referencí) nebo vyjímky — tam je zas problém jak je mapovat do C...
- O samotný přenos dat a komunikaci se stará ORB Core, která je k aplikacím přilinkovaná jako knihovna.
- Protokol GIOP (General InterORB Protocol), součást ORB (Object Request Broker) — definuje Common Data Representation – CDR a formáty zpráv, nadstavba IIOP (Internet InterORB Protocol), implementace GIOP pro internet (mapování GIOP na TCP/IP)
  - GIOP umí i location forward – zprávu, že požadavky mají teď jít na jiný server
- Messaging (hlavní dva typy zpráv, REQUEST a REPLY, celkem sedm druhů)
- stub/skeleton – jako u RPC
- proxy/servant – proxy objekt na klientovi, servant (implemntace objektu na serveru) na serveru – pro RMI
- POA – Portable Object Adapter – asociuje server s objekty — směruje volání buď do servantů (místních, nebo na jiné servery), demultiplexuje příchozí požadavky na server a spolupracuje s IDL
  - default servant – vyřizuje požadavky pro které není servant
  - servant activator – když není servant, vytvoří ho
  - thread pool – připravené thready k obsluze požadavků
  - servant retention policy – dá se úplně vypnout vedení info o servantech, všechny požadavky pak jdou na default nebo activator
- Naming Service – operace resolve a bind (viz Distribuované systémy#CORBA Naming.2FTrading)
- Trading Service – operace export, query Distribuované systémy#CORBA Naming.2FTrading
- operace se dají volat i neblokujícně, s callbackem nebo pollingem

DCOM

Microsoftí middleware pro RPC (s objekty). Místo IDL má MIDL (Microsoft Interface Definition Language), parametry jdou definovat jako in, out, několik typů pointerů (ref, unique, ptr podle tohle, jestli mohou být NULL a mohou být aliasované, tj jestli dva mohou ukazovat na tu samou paměť), taky pipe, který reprezentuje datový stream mezi klientem a serverrem.

Interface může být buď RPC nebo COM. COM verze musí dědit z IUnknown (metody QueryInterface, AddRef, Release) nebo IDispatch (GetTypeInfo, GetIDsOfNames, Invoke), a musí mít uuid atribut (unikátní id). Správa paměti pomocí počítání referencí, vzdáleně se to řeší před další IRemUnknown, které AddRef a Release před posláním serveru agreguje.

Kombinování objektů:

- Containment (nový objekt přeposílá volání vnitřním implementacím)
- Aggregation (export vnitřních interfaců; problém s QueryInterface, aby znal celý nový interface – nutná explicitní podpora agregace agregovaným rozhraním).

Když klient udělá nějaké volání, knihovna OLE32.dll se mrkne do System Reqisty a vyloví jak udělat stub a pomocí LPC (Local Procedure Call) nebo RPC zavolá skeleton a provede, co se po ní chce.

Reference na objekty se získávají buď přes factories, nebo nějakými metodami co vrací reference.

Servery mohou být buď in-process (linkované v DLL, adress space klienta, běží na tom samém stroji), nebo out-of-process (v samostatném procesu(v EXE), mohou běžet i na jiném stroji ve vlastním adresním prostoru).

Interfacy pro perzistenci, s metodami jako Load, Save a IsDirty.

JMS – Java Messaging Service

JAVA interface pro messaging middleware

- zprávy se vyměňují v rámci session se service providerem;
- předpokládá se existence messaging service providera, na něj se napojí přes JNDI (viz Distribuované systémy#JNDI)
- session je vázáno na jeden thread a stará se o pořadí a tak
- různé typy specifických obsahů zpráv (objekt, mapování, stream primitivních typů, text, stream bajtů)
- rozhraní Producer a Consumer, vytvářené volání metod ze Session; odesílání je blokující asynchronní; příjem je synchronní nebo asynchronní, dá se filtrovat co přijmout
- modely Point To Point (odesílatelé ukládají do fronty, příjemci vybírají; nejsou-li zůstavají zprávy ve frontě) a Publish Subscribe (kanál od odesílatelů příjemcům; když nejsou příjemci tak se zprávy zahazují – leda že by si někdo objednal durable subscription)

EJB

Prostředí pro komponentové aplikace. Beany obsahují logiku aplikace a bydlí v kontejnerech, které jim zajišťují přístup klientů, životní cyklus, perzistenci, transakce a tak. Volání metod u všech beanů je serializované.

- stateful session beans
  - Z pohledu klienta se objekt se vytváří když se na něj dodá reference, pak se stav inicializuje business metodou, další speciální metoda stav sundá;
  - Z pohledu kontejneru se bean aktivuje, pasivujce, stav se v kontejneru uchovává jako serializace tranzitivního uzávěru polí objektu.
- stateless session beans – jako stateful, ale odpadá aktivování/deaktivování
- message driven beans – požírají JMS zprávy, implementují JMS listener
- entities – reprezentují entity v databázi; vlastnosti instancí jsou perzistentní, odpovídají primitivním/serializovatelným typům a kolekcím. proměnná Id je primární klíč. Entity manažer poskytuje metody k vyhledávání. O perzistenci se stará kontejner.

Beany můžou mít definovaný stav vůči transakcím (neumíme, požadujeme v transakci, může být, pak taky co se má dělat když transakce je/není). Stav session beanu není v transakci a není ovlivněn commitem/rollbackem.

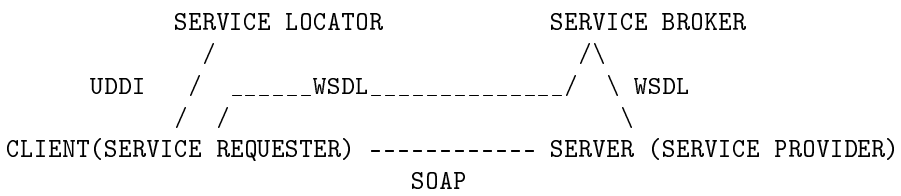
Transakce mohou být bean managed (pak commit nebo rolback startuje metoda beanu), nebo container managed, kde si bean nastavi nejake atributy a kontejner si podle toho nějak řídí commit a rollback, třeba atribut mandatory říká, že se metoda beanu musí vykonat ve volající transakci, a pokud taková neexistuje má se vyvolat vyjimka. Podobně never zas říká, že medora musí být vyvolána mimo transakci.

- business interface – samotné metody na beanu, dělají tu věc co se od beanu chce
- remote interface – proxy k beanu, třeba hází navíc výjimky typu "selhalo spojení"
- home interface – metody nejsou vázané na konkrétní instanci (třeba vytvoření nové instance, nebo vyhledání existující) (jen v EJB 2.1, ne v EJB 3.0... asi)

SOAP

- Simple Object Access Protocol — prototokol založený na XML, určený pro web services (přestože je to "protokol" definuje pouze formát zpráv)
- zpráva obsahuje hlavičky pro routující systémy a tělo pro koncového příjemce
- počítá se s přenosem přes HTTP
- umožňuje definovat datové typy, i složené (z toho plynou probémy s XML schema popisem SOAP souborů a tím s jejich validací)
- spolupracuje s WSDL, které popisuje webové služby v XML a s UDDI, které zas umí registraci, lokaci a klasifikaci webových služeb
- UDDI obsahuje:
  - White Pages — Naming Service
  - Yellow Pages — Tradin Service, používá globální specifikace properties (UNSPC, NAICS)
  - Green Pages — technické informace

Model spolupráce je přibližně



22.5 Logické hodiny a jejich synchronizace

Fyzické hodiny se nedají dostatečně přesně synchronizovat, takže se používají logické.

Lamportovy hodiny

Je důležité pořadí, nikoli přesný čas; nekomunikující procesy nemusí být synchronizovány.

Integer u každého uzlu, a:

1. Kdykoli proces zaznamená důležitou událost (generování zprávy), inkrementuje timestamp
2. Ke každé poslané zprávě přidá timestamp
3. Když proces p přijme zprávu m, aktualizuje si svůj timestamp: TS(p) = max(TS(p), TS(m))+1

Pak platí, že když událost A kauzálně předchází B, tak TS(A) $<$ TS(B). Opačná implikace ale neplatí. To řeší až vektorové hodiny.

Vektorové hodiny

Každý proces má svůj vektor hodin VT.

1. odeslání zprávy m procesem p_s
   - VT(p_s)[s]++; VT(m) = VT(p_s)
2. přijetí zprávy procesem p_r; proces pozdrží doručení, dokud
   - VT(m)[k] = VT(p_r)[k] + 1 pro k=s (tj. ve slotu odesílatele je zpráva o jednu napřed)
   - VT(m)[k] VT(p_r)[k] jinak
3. po doručení zprávy m si proces p_r upraví VT

  - VT(p_r)[k] = max(VT(p_r)[k], VT(m)[k])
- událost A kauzálně předchází B <=> timestamp A je menší než B
- událost A nemá kauzální vztah k B <=> timestamp A není porovnatelný s B

Maticové hodiny

Vektory vektorových hodin, proces si udržuje co ví o ostatních procesech (v jakém timestampu o nich naposledy slyšel) viz Distribuované systémy#ISIS protokol

22.6 Distribuované synchronizační algoritmy

- synchronizace fyzických hodin
- logické hodiny — lamport, vektorové, maticové
- vzájemné vyloučení
- detekce globálního stavu
- volba koordinátora
- členství ve skupinách

Synchronizace fyzických hodin

- V distibuovaném prostředí jsou fyzické hodiny celkem nanic (nedají se synchronizovat dostatečně přesně, aby k něčemu byli)
- Po nějaké době odchylka (x.t), jde určit po jaké době se je nutné synchronizovat k udržené nějaké maximální odchylky od správného času
- **Cristianův algoritmus** s jedním UTC serverem, který čas zná. Proces se na něj zeptá a nastaví si čas UTC s opravou danou odhadem chyby dané zpožděním komunikace a dobou zpracování požadavku serverem
  - Přenos po síti může mít klidně dobu obrátky v řádku 100vek ms, což přestavuje až stovky milionu instrukci na běžném procesoru (třeba geostacionární družice jsou ve výšce cca 36 000 km ... světlo tam letí přes 100ms, takže taková družice nemá dobrý ping :o))
- **Berkley algoritmus** sever se zeptá všech na čas, spočte průměr a pošle zprávy o kolik se má kdo opravit
- **Distribuovaný algoritmus** Bez koordinátora. Broadcast ve fyzicky nestejný čas, počítání průměru se zahozenými extrémy, oprava.

Vzájemné vyloučení

Není společná paměť, nutno synchronizovat přes zprávy.

- centralizované (s koordinátorem)
- princip soutěže (Lamport, Ricard-Agrawalla)
- volby (Maekawa)
- token-passing (Suzuki-Kasami)
- kruh, strom (Le Lann, Raymond)

Centralizovaný algoritmus

Jeden server s frontou, na žádost posílá potvrzení/zamítnutí/uvolnění

- ideově nevhodné, ale nejjednodušší a nejefektivnější
- výpadek serveru – ztráta informace
- výpadek klienta – vyhladovění

U všech následujících algoritmů je problém s výpadkem skoro libovolného procesu – řešit centralizovaný problém distribuovaným algoritmem nemá moc smysl.

Lamportův algoritmus

Proces vyšle žádost, a čeká až dorazí odpovědi od všech ostatních, a všechny žádosti v jeho frontě mají vyšší časovou značku.

- p posílá žádost Mp se svým timestampem
- přijetí žádosti od i: zapamatuje si žádost, pošle ACK s vlastním timestampem
- když dostane od někoho ACK, přidá si ho ke svému požadavku
- do kritické sekce proces vstoupí, když
  1. od všech ostatních dostal ACK
  2. a zároveň neví o žádném starším požadavku
- když skončí s kritickou sekcí, pošle ostatním release
- po přijetí release si proces vymaže k němu patřící žádost (a někdo další pak na základě toho může vlézt do kritické sekce)

Ricart & Agrawala

Proces chce vstoupit do kritické sekce

- zašle žádost ostatním a čeká na došlé odpovědi s potvrzením

Proces přijme žádost:

- jestliže není v kritické sekci a ani nechce, pošle potvrzení
- jestliže je v kritické sekci, neodpovídá a požadavek si zařadí do fronty
- jestliže do kritické sekce chce, porovná čas příchozí žádosti se s časem své vlastní
  - pokud je vlastní dřív, neodpovídá a zařadí do fronty
  - pokud je příchozí dřív, pošle potvrzení
- po opuštení kritické sekce pošle potvrzení všem procesům co má ve frontě

Princip voleb

Proces se snaží zíkat hlasy ostatních, kdo má nejvíc může do kritické sekce. Proces může v jeden okamžik hlasovat jen pro jednoho. Problém: jak počítat výsledky, kdy už proces ví že vyhrál. Při stejném počtu hlasů může nastat deadlock.

Maekawa – optimalizace komunikační složitosti pomocí volebních okrsků, pro vstup do kritické sekce je potřeba získat všechny hlasy z vlastního okrsku (podmínky: každé dva okrsky mají společného člena, velikost okrsků je konstatní, každý proces ve stejném počtu okrsků). Komunikační složitost odpovídá velikosti okrsků.

Prevence deadlocku – logické hodiny (proces ruší původní hlas, pokud ještě pak dostane žádost s nižším TS, realizace pomocí zprávy reject a přidělení hlasu procesu s nižším timestampem)

Optimalni rozděleni pro $M = K * (K-1) + 1$(M počet procesů, K velikost okrsku), reálně se rozdělují na pruniky vždy jednoho sloupce a řádku ve čtverci procesů.

Token based

Kdo má peška může do kritické sekce. Problém se ztrátou peška.

Detekce globálního stavu

- Množina událostí v systému $E = \{e\}$
- Řez c je rozdělení E na Pc a Fc : $P_c \cup F_c = E \; AND \; P_c \cap F_c = \emptyset$
- Konzistentní řez c :$a \to b \; AND \; a \in F_c \to b \in F_c$
- Konzistentní stav je Pc, tak, že c konzistentní řez.

Algoritmus detekce GS:

- Jeden iniciátor pošle značku, která říká že se jde detekovat globální stav.

- Po přijetí první značky si zapamatuju svůj stav (poslední odeslanou a přijatou zprávu/zprávy), a stav kanálů označím za prázdný a přepošlu značku
- Pak si pamatuju zprávy co mi chodí od uzlů, od kterých mi ještě nepřišla značka.
- Když mi od nějakého uzlu přijde značka (ale už jsem predtím od někoho značku dostal a detekuju), tak uzavřu stav kanálu od toho uzlu.
- Po přijetí všech značek konec, zapamatované stavy uzlů a kanálů definují konzistentní stav.

Volba koordinátora

Bully algoritmus

Předpokládá se omezená doba přenosu a zpracování zprávy kvůli detekci havárií.

- Když proces usoudi ze stavajici koordinator zhavaroval, rozhodne volit. Zašle zprávu všem procesům s vyšší identifikací (ProcessID)
  - Když přijde odpověď od nejakeho procesu z vyssim ID, process vycka nejaky zvoleny casovy interval jestli ten proces s vyssim ID se prohlasi za koordinatora. Pokud ne, zacne volby zas.
  - Když nepřijde nic, proces vyhrál, je novým koordinátorem a pošle o tom zprávu všem ostatním.
- Když proces přijme zprávu o volbě, vrátí svou odpověď a pošle žádosti všem vyšším procesům.
- Kdyz proces prijme zpravu od noveho koordinatora a zjisti ze ma vyssi Process ID nez ten koordinator tak se naštve :) a zacne volbu. Proto se tomu algoritmu se rika "bully" (z angl. je to žák týrající spolužáky)

Volba se provede ve dvou kolech (Proc?).
Asi protoze prijde vic odpovedi od procesu z vyssim PID a ty pak maji mezi sebou vyresit kdo je koordinator ve druhem kole.

Invitation algoritmus

Procesor může zhučet, při selhání komunikace se může síť rozdělit na izolované segmenty, zprávy se můžou ztratit – nelze spolehlivě detekovat havárii.
Idea: koordinátor je vázán na skupinu (všichni členové skupiny vidí stejného koordinátora), skupiny lze štěpit.
pravidelná výzva AreYouCoordinator

- příjem koordinátorem – sjednocení skupin pod vyššího koordinátora
- pokud člen skupiny nějakou dobu neobdrží AYC svého koordinátora
  1. prohlásí se za koordinátora nové vlastní skupiny
  2. rozešle AYC ostatním

Konzistence je relativní vzhledem ke skupině. Procesy se shodují na členství ve skupině a na nějaké hodnotě. Separované uzly jsou konzistentní samy se sebou.

Kruhový algoritmus

1. Proces se rozhodne volit, pošle zprávu následníkovi.
   - zpráva obsahuje čísla procesů (odesílatel a nejvyšší živý)
   - po návratu obsahuje zpráva nového koordinátora
2. následuje fáze oznámení

Stačí znát následníky a mít možnost zjistit následníka nedostupného uzlu. Složitost je $O(n^2)$

členství ve skupinách

- Procesy si udržují informaci, kdo je v daný okamžik členem skupiny
- viz TRANSIS a ISIS protokoly

22.7 Distribuovaný konsensus

Byzanstký konsensus (1->1) iniciátor zvolí hodnotu rozešle ji všem; všechny loajální uzly se musí shodnout na stejné hodnotě, je-li iniciátor loajální, musí se shoudnout na jeho

Konsensus (n->1) každý uzel má iniciální hodnotu, všechny loajální uzly se musí shodnout na společné hodnotě, pokud je iniciální hodnota všech loajálních uzlů stejná, musí se shodnout na té

Interaktivní konzistence (n->n) každý uzel má inicální hotnotu, všechny loajální uzly se musí shodnout na společném vektoru, hodnota položek vektoru odpovídající loajálním uzlům se musí shodovat s jejich init hodnotou

Problém dvou armád

- Početnější armáda je rozdělena, uspěje jen při synchronizovaném útoku.
- Obě části musí mít jistotu, že druhá část začne útok také.
- Komunikace pouze nespolehlivým kurýrem.

Řešení neexistuje – pošlu-li "Útok v pět", nevím jestli to dostala druhá strana, když mi ona pošle ACK, tak neví jestli jsem ho dostal.

Problém Byzanstkých generálů

- Někteří generálové jsou zrádci.
- Všichni loajální generálové musí rozhodnout shodně.
- Každý generál se rozhoduje na základě informací od ostatních generálů (mají spolehlivou komunikaci).

BÚNO: 1 generál, ostatní důstojníci. Generál vydá rozkaz, důstojníci předají dál dolů – rozkaz bude vydán na základě většiny. Cíle:

1. všichni loajální důstojníci vydají stejný rozkaz
2. nebo, je-li generál loajální, každý loajální důstojník vydá rozkaz generála.

Pro tři uzly s jedním zrádcem nejde.
Pro 4 uzly:

- zrádce generál: alespoň 2 stejné rozkazy: důstojníci si vzájemně přepošlou většinový rozkaz (C1), pro tři různé se shodnou že je generál zrádce (C2)
- zrádce důstojník: v nejhorším případě pošle všem ostatním falešný rozkaz, loajální ale dostanout většinu správných (C2)

Existuje řešení pro 4 uzly s jedním zrádcem, obecně pro m zrádců existuje řešení pro n3m+1 uzlů

Konsensus s nezfalšovatelnými zprávami

Pokud nelze předávanou zprávu zfalšovat, pak libovolný počet zrádců neznemožní konsensus.
Idea algoritmu:

- každý přepošle vše, co dostal, v nezměněné podobě
- každý uzel nakonec sám uvidí co kdo komu poslal
- loajální uzly se shodnou buď na majoritní nebo default hodnotě

22.8 Distribuované sdílení paměti

- Konzistenční modely
- Distribuované stránkování
- Distribuované sdílené proměnné a objekty

Různé mechanismy, v HW/SW, od SMP s busem/switchem přes NUMA, distribuované stránkování po distribuované sdílené promenné a objekty.

Konzistenční modely

Modely bez synchronizační proměnné

implementace možná na úrovní virtuální paměti, procesy o tom nemusí vůbec vědět

striktní konzistence jakékoli čtení z adresy x vrátí hodnotu uloženou při posledním zápisu do x – absolutní časové uspořádání, všechny zápisy okamžitě všude viditelné; musí existovat přesný globální čas

sekvenční kozistence výsledek výpočtu je stejný, jako kdyby všechny operace všech CPU byly vykonávány v nějakém sekvenčním uspořádání, a operace každého CPU jsou v pořadí specifikovaném programem – snadno implementovatelné, povoleno libovolné prokládání instrukcí na různých CPU, všechny procesy vidí stejné pořadí změn; platí že čas čtení a zápisu dohromady musí trvat alespoň tak jako přenos jednoho paketu -> pomalé

kauzální konzistence kauzálně vázané zápisy musí být viděny všemi procesy ve stejném pořadí; vyžaduje udržování grafu závislostí zápisu na čtení

PRAM (pipelined RAM) konzistence zápisy prováděné jedním procesem jsou viděny ostatními procesy v tom pořadí, ve kterém byly prováděny; neexistuje jednotný pohled na rozvrh, snadná na implementaci

slow memory zápisy jedním procesem do jednoho místa musí být viděny ve stejném pořadí; lokální zápis, pomalá nesynchronizovaná propagace, neposkytuje žádnou synchronizaci

Všechny modely vyžadují propagaci všech zápisů všem procesům, přitom ne všechny aplikace potřebují vidět všechny zápisy a jejich pořadí.

Modely se synchronizační proměnnou

Operace Synchronize, Acquire a Release určují kdy je proces v kritické sekci. (Po jejím skončení se data propagují ostatním). Procesy musí o SP vědět, ale výkonnost je vyšší. Rozlišení Acq() (vstup do kritické sekce) a Rel() (výstup z kritické sekce)

slabá konzistence

1. přístup k SP je sekvenčně konzistentní (všechny procesy ho vidí ve stejném pořadí)
2. před přístupem k SP musí být dokončeny všechny předchozí zápisy
3. před přistupem k obyčejným proměnným musí být dokončeny všechny předchozí přístupy k SP

Sáhnutím na SP před čtením se zajistí aktuální verze dat.

Výstupní konzistence

1. Před přístupem ke datům musí být úspěšně dokončeny všechny předchozí Acq() procesu.
2. Před provedením Rel() musí být dokončeny všechny předchozí zápisy a čtení prováděné procesem.
3. Acq() a Rel() musí být PRAM konzistentní.

- eager release consistency – změny se všem propagují po Rel(); optimalizace přístupové doby
- lazy release consistency – změny se propagují až po Acq() jiného procesu; menší nároky na síť

Vstupní konzistence

1. Před Acq() k SP se aktualizují chráněná sdílená data procesu
2. Exkluzivní přístup k SP je povolen jen když k ní nepřistupuje jiný proces (ani neexkluzivně)
3. Pro exkluzivní přístup si musí proces vyžádat aktuální kopii dat od posledního vlastníka (kdo to měl exkluzivně)

Distribuované stránkování

obdoba virtuální paměti (problémy: replikace, nalezení stránky, správa kopií, uvolňování stránek, falešné sdílení — v jedné sdílené stránce jsou dvě proměnné, co spolu nesouvisí)

- sekvenčně konzistentní – stránky mají vlastníka co na ně může psát, ostatní mají kopie pro čtení
- kauzálně konzistentní – vektorové hodiny u stránek i procesů, velká prostorová režie
  - Udržuje se defakto graf
  - U stránek se udržuje vektor udávající na kterých stránkách závisí obrah
  - U procesů vektor, ze kterých stránek znám data
  - Při čtení si proces aktualizuje svůj vektor, pokud je nižší než stránky, případně se invalidují staré stránky
  - Při zápisu se aktualizuje vektor stránky, jako inc(vektor_procesu).

Distribuované sdílené proměnné

- implementováno v knihovnách (např. Munin – sdílení read-only; migratory s eager release konzistencí; write-shared – do lokální kopie se dá psát, po release se propagují změny, případný merge, při konfliku runtime error; normální sdílená data se sekvenční konzistencí).
- odpadá problém s falešným sdílením.
- nutnost rekompilace pro různé jazyky
- distribuované objekty – flexibilnější díky zapouzdření (CORBA, RMI, etc)

22.9 Souborové a adresářové služby

see #Identifikace objektů a přístup k nim

Distribuované souborové systémy

- distribuovaný FS vs. jednotný přístup k síťovým FS
- monolit vs. oddělené souborové a adresářové služby (které mapují uživatelská jména na systémová)
- stavové vs. bezestavové servery
- replikace, cache

sémantika přístupu k souborům

- centralizovaná (každá změna hned vidět)
- relační (změny jsou vidět až po zavření – AFS)
- imutabilní soubory
- transakce

NFS

wen: Network File System (protocol)

- postaveno nad RPC
- vyvinul v 80tých letech Sun
- XDR — eXternal Data Representation – popis datových struktur které NFS používá
- Klient si pošle mount na server, ten mu pošle file handle na mounted directory.
- verze 2, původní, první vypuštěná ven ze Sunu
- verze 3
  - bezstavová, nemá open a close, jen operace READ, WRITE, LOOKUP, REMOVE, MKDIR, RMDIR. Z toho ale plynou problémy, např s ověřováním práv (normálně se to řeší v open). To jde řešit třeba sdílením UID a GID a ověřováním na klientovi.

  - některé podpůrné protokoly běží na různých portech
  - jedna akce nad NFS se typicky skládá z většího množství RPC callů
  - zavádí NLM protokol na managování zámků.
- verze 4
  - stavová (!)
  - složené operace – umožňují omezit počet nutných RPC callů
    * odhaduje se až pětinásobná úspora potřebných client-server interakcí
  - Podpora replikací, bezpečnosti...

AFS

wen: Andrew File System

- vznikl v rámci projektů Andrew Project na Carnegie Mellon University (jméno podle Andrew Carnegie a Andrew Mellon)
- Soubory jsou organizovány pod jednotný globální namespace, rozdělený na cels (administrativní jednotky uzlů) (dejte si ls na /afs na unixu na MS)
- Jednotlivé servery udržují podstromy ve svazcích, servery se na se navzájem replikují.
- poskytuje lepší možnosti scalability a security
- pro security využíván Kerberos, implementována ACL adresářů
- soubory jsou cacheované na klientovi => možnost omezeně fungovat i po pádu serveru/sítě
  - změny jdou do cache a jsou na server propagovány až při zavření souboru
  - pokud je soubor na serveru změněn, klienti kteří ho mají v cache jsou informování
- svazek – strom souborů, adresářů a mountpointů. Sestavuje jej admin.
  - uživatel s ním může pracovat jakoby byl lokální
  - může mít nastaveny kvóty
  - admin ho může přesunout na úplně jiný server bez toho aby se to uživatel dozvěděl
  - může mít několik read-only kopií, AFS zajistí že budou obsahovat správná data, pokud mám jednu připojenou a server kde je spadne => nic se neděje, začnu seamlessly pracovat s jinou
- hodně se jím inspirovalo NFS verze 4, z AFS 2 vychází CODA

CODA

wen:Coda (file system), základy přímo na CMU

- začal vznikat na Carnegie Mellone University v 1987
- vychází přímo z AFS 2
- client-side caching souborů, adresářů a atributů
- čte se z jednoho serveru, píše se na všechny, případně se řeší kolize
- write-back cache
- kerberos-like autentizace
- ACLka
- reintegrace dat na čas odpojených klientů
  - Jde pracovat v connected a diconected mode, tj stahnout si soubor a pak na něm offline dělat.
  - Ve strongly connected modu jsou změny změny zapisovány synchronně
  - Ve weakly connected modu jsou zapsány dodatečně a ručně se zamergují konflikty
- možnost replikace serverů (read/write)

Replikace

viz http://www.kiv.zcu.cz/~ledvina/Prednasky-DS-2007/DS-07-Replikace.pdf

Udržování kopií na více fileserverech. Důvody: spolehlivost, dostupnost, výkon.

- explicitní (uživatel se stará sám)
- odložená (aktualizuje se primární replika, sekundární pak)
- skupinová komunikace (zápisy se simultánně posílají všem replikám)

Aktualizace kopií:

- primární kopie vítězí
- většinové hlasování
- vážené hlasování (různý důraz na čtoucí a zapisující procesy)
- hlasování s duchy (bezdatový server – ghost, obsahuje pouze verze, účastní se hlasování o zápisu ale neučastní se hlasování o čtení)
- dynamická kvóra

klientocentrické konzistenční modely

see also `http://www.cc.gatech.edu/classes/AY2005/cs4210_spring/Lectures/22-Concurrency-2.ppt`

Replikovaná databáze (www+cache, zápisy málo časté), když se klient přesune jinam, musí vidět stejná data.

Implementace: klient si udržuje read-set a write-set (množiny čtení a zápisů co už viděl), posílá s požadavky, podle toho se vynucuje aktualizace replik, jde i vektorovými hodinami (podle replik?). Protože tohle má neomezenou paměťovou náročnost, lepší implementace se sdružováním do sessions vázaných na aplikaci/výpočet/modul a následné mazání z write a read sets. Viz hnusinka zelená :-)

eventuální konzistence po ukončení všech zápisů budou všechny repliky v konečném čase aktualizovány; problém je, že jeden proces může koukat na data jiných replik a vidět něco jiného

monotonní čtení po přečtení hodnoty x všechna další čtení vrátí stejnou nebo novější hodnotu (při připojení k jiné replice uživatel vidí všechny zprávy co už si přečetl dřív – dá se řešit třeba logem updatů, co už klient viděl – replika si pak si může ověřit, že je dost aktuání případně si sehat aktalizaci a poskytnout správná data.

monotonní zápis zápis do proměnné je proveden před každým následním zápisem do ní; než do repliky zapíšu, musí si aplikace přijmout aktuální změny od ostatních. Implementace: podle klientského write-setu si replika ověří, jestli něco nemá dozapsat, po zápisu si klient aktializuje write set.

read your writes procesy při následném čtení vidí svoje zápisy (po aktualizaci wiki nekoukám na kopie z cache). Implementace: buď forward čtení na aktuální repliku, nebo replika ověřuje podle write-setu svoji aktuálnost.

writes follow reads zápis se provede do kopie proměnné, která je alespoň tak aktuální jako ta, která se předtím přečetla. Implementace: aktualizace podle read setu. Po zápisu se aktualizuje read-set i write set klienta.

epidemické protokoly

Eventuální konzistence, optimalizuje pro hodně velké systémy, neřeší konflikty.

- servery jsou: infected (rozšiřují "epidemii"), removed (data mají ale nerozšiřují), susceptible (data nemají)
- antientropie: každý server jednou za čas zkontaktuje náhodný jiný, vymění si co ještě nemají (nebo jen push nebo pull)
- gossiping: pokud byl P aktualizování, kontaktuje nějaký další server Q, aby šířil update; jestliže Q už update má, P se s pravděpodobností 1/k nastaví jako removed
  - nezaručuje že update budou mít všechny servery

22.10 Distribuovaná správa prostorů jmen

Identifikace objektů a přístup k nim

Problémy k řešení

- Který objekt má být použit
- Kde je umístěn
- Jak se k němu jde dostat
- identifiace a přístup k objektům (který, kde je, jak se k němu dostat)
- struktura jmen, trvanlivost, distribuovaná správa jmen, řešení systémových jmen
- kapability a jejich ochrana
- přístup k objektům, distribuovaná správa prostředků, prostředky mohou být replikované a přístup tím komplikovaný
- objekty: aktivní (kód) / pasivní (přes správce, nebo prostředky jako třeba paměť)

Pojmenování, indetifikace, struktura jmen

- prostory jmen mohou být separátní (file system, registry, URL) nebo může být jednotný (distribuovaný name server).
- nestruktorovaná jména (UUID) versus strukturovaná (ms.mff.cuni.cz)
- jména: uživatelská (human-readable) / systémová (interní čílesné kódy)
  - mapování jmen na replikované objekty
  - jména mohou být plochá, strukturovaná (hierarchická), nebo popisná (více atributů)

Kapabilita je datová struktura umožňující jednoznačnou identifikaci objektu, obsahuje i přístupová práva pro držitele – k jednomu objektu typicky patří víc různých kapabilit. Uživatelským procesům je znemožněno vlastní generování kapabilit i změny práv. Kapability se publikují na nameserveru, zároveň si server registruje u Reg serveru. Klient si najde kapability na NS, otevře si ji u Reg serveru, ten ověří a předá serveru, ke kterému si pak klient otevře kanál.

- snadný test oprávněnosti, každý správce si může nadefinovat vlastní druhy práv
- problematické kontrola propagace, potřeba definovat oprávněné uživatele, nebo revokovat
- kapability mohou být buď podepsané, nebo je jejich část s přístupovými právy zašifrována
- Uživatel, který kapabilitu má ji nemůže měnit nebo replikovat.
- Možné jiné řešení pomocí Access Control Listu — prostředek a k němu seznam oprávněných uživatelů. V distribuovaném prostředí nepříliš vhodné.

Adresáře jsou množina položek (jméno,hodnota), hodnota může být:

- primitivní (čísl, řetězce, binární data)
- perzistentní reference (trvalé odkazy na objekty, kapability)
- tranzientní reference (na živé objekty, porty, kanály)
- odkazy na jiné adresáře
- operace jako SET, LOOKUP(jmeno) — změna kontextu, LOOKUP(složené jméno)
- Vyhledávání typicky přes server, ten publikuje jména a vrací reference na objekty (to může být třeba nějaká kapabilita nebo něco jako handle do FS)

Služby

LDAP

see also wen:Lightweight Directory Access Protocol

"Odlehčená" verze X.500 DAP pro použití v TCP/IP sítích.

- adresář je strom položek, každý má sadu atributů
- atribut má jméno a jednu nebo více hodnot (podle definovaného schématu)
- každý položka má DN (distringuished name) – skládá se z RDN (relative DN, vyrobeného z nějaké položky) a DN rodiče
  - DN se může v průběhu živata měnit, někdy se jim přidává i UUID

LDAP poskytuje autentizaci přístupu, služby čtení a vyhledávání v položkách, ověření jestli má položka nějakou hodnotu atributu, aktualizace dat a tak.

JNDI

see also wen:Java Naming and Directory Interface

Jde o nástroj jak unifikovaně z JAVY přistupovat k adresářově organizovaným datům.
Hledání objektů pro Java RMI a Java EE, poskytuje:

- bind objektu ke jménu
- hledání v adresáři (directory lookup iface pro obecné dotazy)
- event interface, dovolující klientům zjistit když se položky změnily
- Service Provider Interface (SPI) pro napojení libovolných adresářových služeb (LDAP, CORBA naming service,...)
- hledá se v kontextu, root je initial context

CORBA Naming/Trading

Naming service:

- naming context: sada vazeb jméno objekt (cosi jako adresář)
- resolve: nalezení objektu podle jména v kontextu
- bind: vytvoření vazby v kontextu (kontext je něco, jako nadřazený adresář, tj. na začátku se dostanu do root kontextu a pak až něco můžu)
- v kontextu může být pod jménem i jiný kontext – složené cesty (jako ve stromovém fs)

viz http://www.iona.com/support/docs/orbix/gen3/33/html/orbixnames33_pguide/Introduction.html
Trading Object Service:

- podobný jako naming service, připomíná zlaté stránky telefonního seznamu
- nabízí služby spolu s referencí (IOR) a popisem, organizované do kategorií (service offer types)
- kategorie jsou definovány pomocí rozhraní `ServiceTypeRepository`
- aplikace exportují reference pomocí rozhraní `Register`, operace `Export`, objekt dá traderovi kapabilitu (popis služby, a interface kde je)
- rozhraní `Lookup` definuje operaci `query`, která umožňuje vyhledat službu podle nějaké podmínky. Někdo se zeptá tradera na službu s danými vlastnostmi, trader dodá umístění
- podobně jako naming service je možné trading services propojovat (tradery se dají navzájem linkovat)

viz http://www.ciaranmchale.com/corba-explained-simply/trading-service.html

22.11 Procesy v distribuovaném prostředí

- sdílení výpočetní síly systému
- vzájemná synchronizace
- vzdálené spouštění, alokace procesorů, migrace, load balancing

Vzdálené spouštění by mělo být transparentní, a vytvořit prostředí odpovídající domácímu

1. registr volných počítačů, tam se nějaký najde
2. vytvoření prostředí pro proces, ten se pustí, po ukončení zpráva jeho domovskému systému

Pokud hostitel přestane být volný, tak se proces zabije, nechá doběhnout, dostane čas na uložení stavu, nebo přemigruje.

Alokace procesorů:

- up-down alogoritmus (koordinátor má tabulku procesorů, ty mu hlásí co dělají; dostávají trestné body za proces jinde, odebírají se jim za neuspokojené požadavky, jinak jdou směrem k nule; při uvolnění procesoru ho dostane proces z fronty neuspokojených požadavků, jehož vysílající procesor má nejméně trestných bodů)
- deterministický grafový – minimalizuje komunikaci, nutno vědět jak co bude komunikovat. Optimální deterministický algoritmus – tok v sítích
- hierarchický – manažeři skupin, při neúspěchu žádost nahoru
- distribuovaný heuristický – několik náhodných výběrů cíle
- bidding – procesy kupují výpočetní sílu

Migrace procesů

- vyvažování zátěže, shutdown, optimalizace
- korektnost – ostatní procesy nejsou migrací ovlivněny, přenesený proces potom je ve stejném stavu
- transparentnost – proces o migraci neví a nemusí spolupracovat

problémy:

- přenesení stavu a adresového prostoru
- komunikace mezi procesy (neztrácet zprávy a tak)
- reziduálni dependence (nechat něco na původním místě)
- vícenásobná migrace

Postup přenosu

1. zmrazení
2. oznámení příjemci, alokace místa tam
3. přenos stavu (registry, zásobník) a kódu / adresového prostoru
4. přesměrování / doručení zpráv
5. dealokace, vyčištění původního místa
6. vazby na nové jádro, nastartování přeneseného procesu (přesunutí částí stavu spolu s procesem, jiné požadavky forwardovat – konzole, některé se používají z nového místa – alokace paměti)
7. dokončení přenosu vazeb

jak kopírovat paměť

- celou při migraci – eliminuje reziduální dependence, ale je pomalé
- pre-copying – proces zmražen jen krátkou dobu, ale ty věci co změní od kopírování se přenášejí víckrát
- copy on reference – stránka se přenese až když je vyžadována, na zdrojové stanici se smaže

zprávy:

- dočasné nebo trvalé přesměrování
- upozornit kamarády předem (ale které?)
- neposílat ACK, on si je zdroj pošle znovu
- migrace kanálu

Vyvažování zátěže

Rozhodnutí o okamžiku migrace – nutno porovnávat zatížení procesorů (nějak konzistentně), pak vybrat co bude migrovat a kam.

- párový algoritmus – vytvoří se páry které se vzájemně vyvažují, zatíženější procesor vybere proces podle míry vylepšení stavu
- vektorový algoritmus (MOSIX) – první vždy vlastní zátěž, pak pošle první půlku nahodnému uzlu, došlá se proloží s vlastní půlkou
- bidding algoritmus: procesy pravidelně vyhodnocovány, pod určitý prah se migruje:
  1. broadcastem žádost o nabídku do vzdálenosti d
  2. adresát proces ohodnotí, případně vrátí nabídku
  3. odpověďi se zkorigují o cenu přenosu, bere se nejlepší; když nedorazí žádná zvýšíme d
- centralizované/hierarchické vyvažovací algoritmy – koordinátor zná zátěže svých procesorů
- lokální (prahová hodnota, když přelezu, ptám se po volných n počítačů, vyberu nejlepší odpověď)

Zablokování

- oblíbené řešení – pštrosí algoritmus
- detekce horší než lokálně: wait-for-graph
- chceme: Každý existující deadlock je v konečném čase detekován, detekovaný deadlock musí existovat

modely deadlocků

Kdy už je deadlock?

- single
- AND model – všechny požadované prostředky musí být přiděleny, aby se proces odblokoval – na deadlock stačí cyklus
- OR model – výpočet může pokračovat, pokud proces dostane alespoň jeden požadovaný prostředek – cyklus je nutná podmínka deadlocku, na postačující je nutný nějaký horší uzel
- k of m
- AND-OR

metody kontrukce wait-for grafu

- centralizovaně (prenos informací po každé změně, v intervalech, nebo na požádání)
  - kauzální doručování proti falešnému uváznutí kvůli zpoždění zpráv
  - hierarchický (každý řeší deadlocky podřízených)
- path-pushing (uzly spravují lokální kusy WFG, sousedním uzlům zasílání externí žádosti, potřeba rozlišovat různé procesy uvnitř jiných uzlů)
- edge-chasing (pošlu zprávu všem, na které čekám, pokud se mi vrátí, jsem v háji; mezitím se to ale mohlo odblokovat – řešením je aging; overkill – zpráva zároveň hledá kandidáta na zahubení)
- diffusing computation – těm na které čekám se posílají pingy, oni je vrací pokud jsou taky zablokováni. pokud dostanu všechny své pingy zpět, mám deadlock
- detekce globalniho stavu – existuje-li deadlock, pak existuje i v konzistentním řezu; při příjmu značku (okamžik řezu) uzel zaznamená lokální WFG, externí závislosti jsou zaslány iniciátorovi