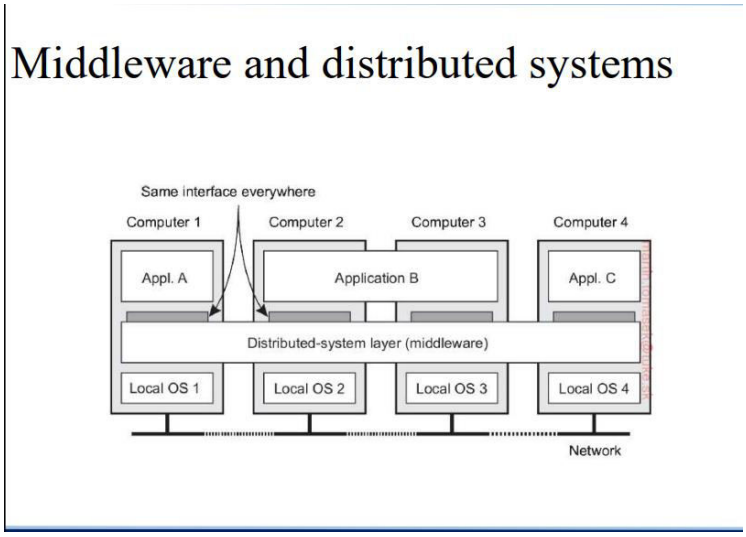
Prednáška 1

Distribuovaný systém

Kolekcia nezávislých výpočtových prvkov, ktoré sa ku koncovým používateľom javia ako jeden koherentný systém.

- **1 node (prvok)** môže byt softver/hardver



Všetky prvky distribuovaného systému sú prepojené na spoločné rozhranie, ktoré sa dá pomenovať ako **middleware**.

Typické služby middleware su:

- **Komunikácia** (Communication) sem patria napr. RPC cally
- **Transakcie** – middleware poskytuje support pre komunikáciu medzi viacerými aplikáciami v DS
- **Kompozícia služieb –** napríklad rozšírenie webových služieb ďalšími komponentami po sieti, celkové rozloženie prvkov
- **Spoľahlivosť**

Ciele Dizajnu

Dostupnosť –treba poskytnúť používateľom spoľahlivý prístup do systému spolu so zdieľaním zdrojov do systému

Transparentnosť - snaží sa zahaliť, že systém je distribuovaný, že sa správa ako 1 celok

> **Typy transparentnosti** - Access, location, relocation, migration, replication, concurrency, failure
> **Stupeň transparentnosti** - záleží od výkonu a porozumenia systému (tzn co vlastne chceme pouzivatelovi odhaliť)

Otvorenosť

- opisuje **sémantiku a syntax** systému (interfacy pomocou IDL)
- správne špecifikácie sú **kompletné a neutrálne**
- **Interoperabilita –** komunikácia medzi roznymi komponentami na základe štandardov
- **Portabilita –** znovupouzitelnost v roznych systémoch
- **Rozšíriteľnosť**
- **Policies** – dynamická konfigurácia komponentov skrz používateľov

Škálovateľnosť

Je použitá hlavne kvôli zvyšujúcemu sa počtu pripojení (requestov) na server.

Jednotky škálovateľnosti:

1. Škálovateľnosť veľkosti – pridanie používateľov, zdrojov
2. Geografická škálovateľnosť – napr. pridanie serverov do inej krajiny
3. Administratívna škálovateľnosť – systém, ktorý je ľahko spravovať aj keď zahŕňa mnoho nezávislých administratívnych organizácií tzn. bezpečné prepojenie medzi viacerými doménami.

Limity škálovateľnosti:

1. Centralizované služby – služby bežiace na jednom serveri
2. Centralizované dáta – dáta nachádzajúce sa na jednom serveri
3. Centralizované algoritmy – algoritmy nachládajúce sa na jednom serveri

Techniky škálovateľnosti:

1. Ukrytie latencie (Asynchrónna komunikácia)
2. Partitioning and distribution – rozkúskovanie komponentu na maličké časti a distribúcia tých časti medzi jednotlivé časti systému. Dobrým príkladom je DNS, kde pre jednotlivé domény sú rozdelené na neprekrývajúce sa zóny (Generické alebo krajiny). A pre každú zónu existuje server ktorý spracováva požiadavky (v jednoduchom ponínamní).
3. Replikácia a cachovanie

Bežné problémy pri návrhu DS

Nemali by sme počítať s tým že:

- sieť je spoľahlivá
- sieť je bezpečná
- sieť je homogénna (sieť ktorá používa architektúru jednej siete napr. LAN a jeden OS)
- topológia sa nemení
- Latencia je 0
- Šírka pásma (Bandwidth) je nekonečná
- Cena za prenos dát je nulová
- Existuje len 1 admin

Vysoko-výkonné výpočtové distribuované systémy

- Cluster – zoskupenie počítačov ktoré sa správajú ako "superpočítač", rovnaký OS, sieť
- Grid – skupina počítačových systémov v sieti z roznych druhov SW/HW a sieti, napr. Virtualna organizácia – workspace
- Cloud – sprístupnenie velkeho mnozstva služieb a dát skrz dátové centrá (Iaas, Paas, SaaS)

Distribuované informačné systémy

Je to súbor viacerých systémov, ktoré su fyzicky rozmiestnené na viacerých miestach a prepojené nejakým druhom komunikačnej siete

Distributed transaction processes – je to set akýchsi operácií, ktorý je definovaný pre komunikáciu medzi dvomi alebo viacerými dátovými zdrojmi (databázami). Dobrým príkladom je prevod peňazí medzi dvomi bankami ak sa zrube transakcia tak obe banky poslu request znova.

Enterprise Application Integration (EAI) – integrácia podnikových aplikácií je dôležitá pre veľké firmy, ktoré majú vytvorené nejaké to množstvo aplikácií a chcú medzi nimi mať spoľahlivú komunikáciu bez rôznych problémov a zbytočných nadbytočných autentifikácií. Integráciu vytvrára **middlewarový framework**, hlavným dôvodom je **efektívnosť.**

Service oriented Architecture (SOA) – metóda vývoja softvéru, ktorá využíva softvérové komponenty nazývané služby na vytváranie business aplikácií.

Enterprise service bus (ESB) – komunikačný systém medzi vzájomne sa interagujucími softvérovými aplikáciami v SOA architektúre

Pervasive systems

Viacmenej sú to prenosné zariadenia ako laptopy a mobilné telefóny

- Internet of Things (IoT)
- Ubiquitous computing system (AI, distribúcia, autonomia)
- Mobile computing systems
- Sensor networks

1 Prednáška 2 – Architektúry

Je velmi dôležité pri návrhu distribuovaného systému vytvoriť **softvérovú architektúru.** Vyžaduje si to menej úsilia s hardvérom a lepšie premyslenie pred nasadením do produkcie. Na správne architektúru potrebujeme **architektonický štýl,** tzn. Ako budú komponenty medzi sebou komunikovať, ako budú prepojené a ich konfigurácia do jedného celku.

Ďalším konceptom je **konektor**. Ten slúži najmä na komunikáciu a koordináciu medzi procesmi. Kontroluje riadenie toku medzi komponentami.

Konektory + komponenty -> konfigurácie / architektonické štýly

1.1 Najznámejšie architektonické stýly

- **Layered Architectures** - Vrstvené architektúry
- **Object-based architectures** - Objektové architektúry
- **Resource-based architectures** - Architektúry založené na zdrojoch
- **Event-based architectures** - Architektúry založené na udalostiach

1.2 Layered Architectures

Základnou ideou tejto architekrúry je zoradenie komponentov do **jednotlivých vrstiev**, kde komponenty môžu zasielať **downcally** do vrstiev pod nimi a posielať **upcally** vyšším vrstvám.

1.2.1 Downcall & Upcall

Downcall – request-response na nižšiu úroveň vrstvy

Upcalls – signalizovanie vyššej úrovne o výskytu udalosti (volanie **handlera**)

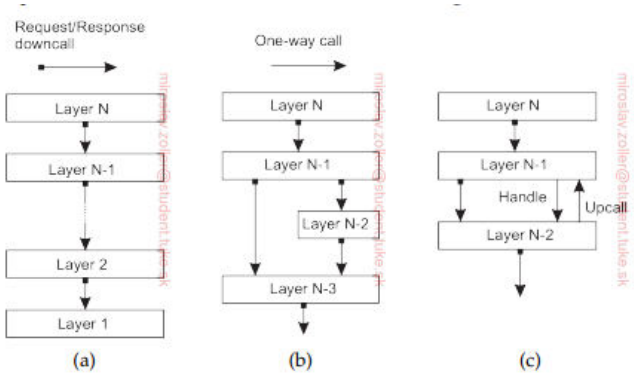


Figure 2.1: (a) Pure layered organization. (b) Mixed layered organization. (c) Layered organization with upcalls (adopted from [Krakowiak, 2009]).

1.2.2 Vrstvené komunikačné protokoly

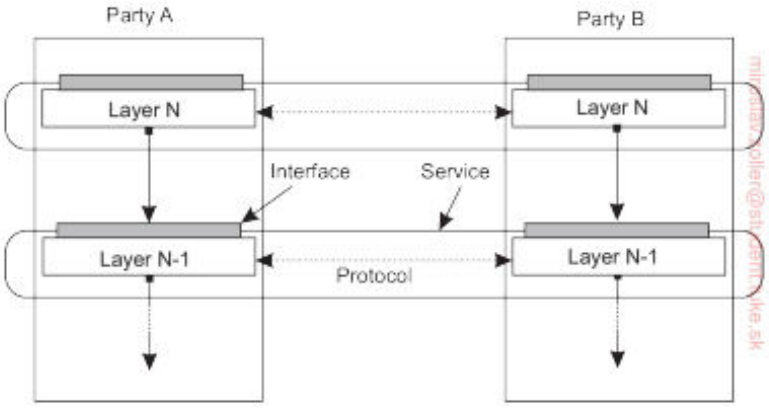


Figure 2.2: A layered communication-protocol stack, showing the difference between a service, its interface, and the protocol it deploys.

Každá vrstvá má **komunikačné služby,** ktoré umožnujú aby data boli rozoslané medzi jednotlivé ciele. Každá vrstva poskytuje **rozhranie (interface),** ktoré špecifikujú funkcie, ktoré budú volané. Čiže v jednoduchosti *interface zakryje implementáciu služby*. A komunikačný protokol **obsahuje pravidlá,** ktorými sa musia tieto služby riadiť aby mohli navzájom komunikovať napr. TCP protokol.

It is important to understand the difference between a service offered by a layer, the interface by which that service is made available, and the protocol that a layer implements to establish communication. This distinction is shown in Figure 2.2.

1.2.3 Vrstvenie aplikácií

Typický 3-vrstvový model aplikácie sa delí na

- Úroveň aplikačného rozhrania (handluje interakciu s pouzivatelom alebo ext. zdrojom)
- Úroveň spracovania (obsahuje core aplikacie)
- Úroveň dát (databaza)

As a second example, consider a decision support system for stock brokerage. Analogous to a search engine, such a system can be divided into the following three layers:

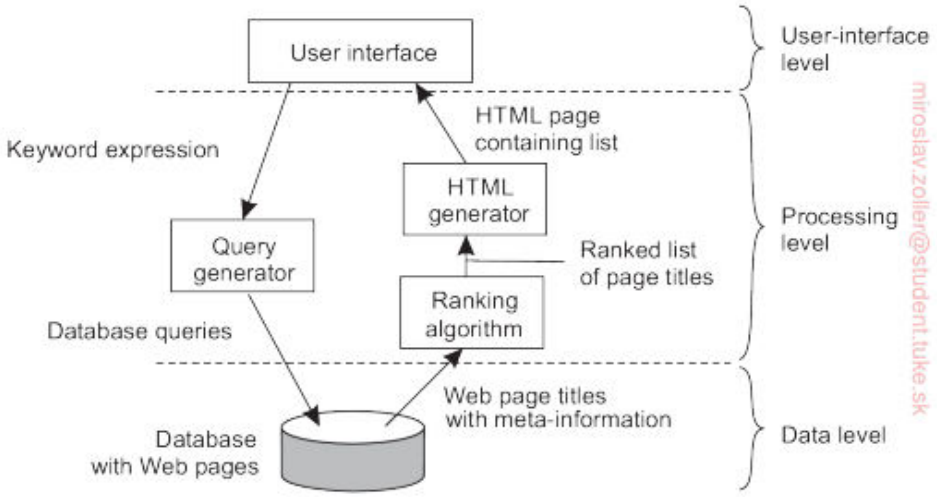


Figure 2.4: The simplified organization of an Internet search engine into three different layers.

- A front end implementing the user interface or offering a programming interface to external applications
- A back end for accessing a database with the financial data
- The analysis programs between these two.

1.3 Object based architectures

Každý **komponent** je vyjadrený ako **objekt**. A jednotlivé komponenty sú prepojené cez RPC (Remote Procedure Call) mechanizmy. V prípade Distribuovaných systémoch, volanie procedúr môže byť vykonávané aj po sieti, tzn. objekt sa môže nachádzať na inom zariadení.

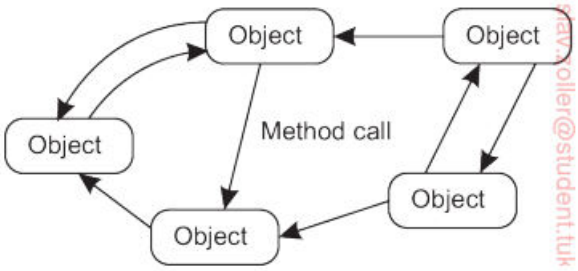


Figure 2.5: An object-based architectural style.

Objektová architektúra ma výhodu encapsulácie dát (object **state**) a operácie nad dátami sú vykonané pomocou **metód** v rámci jednej entity **Triedy**.

Interface (Rozhranie) pre takýto objekt zakrýva implementáciu, no umožňuje jednoduchú vymeniteľnosť komponentu ktorá tento interface používa.

1.3.1 Distribuovaný objekt

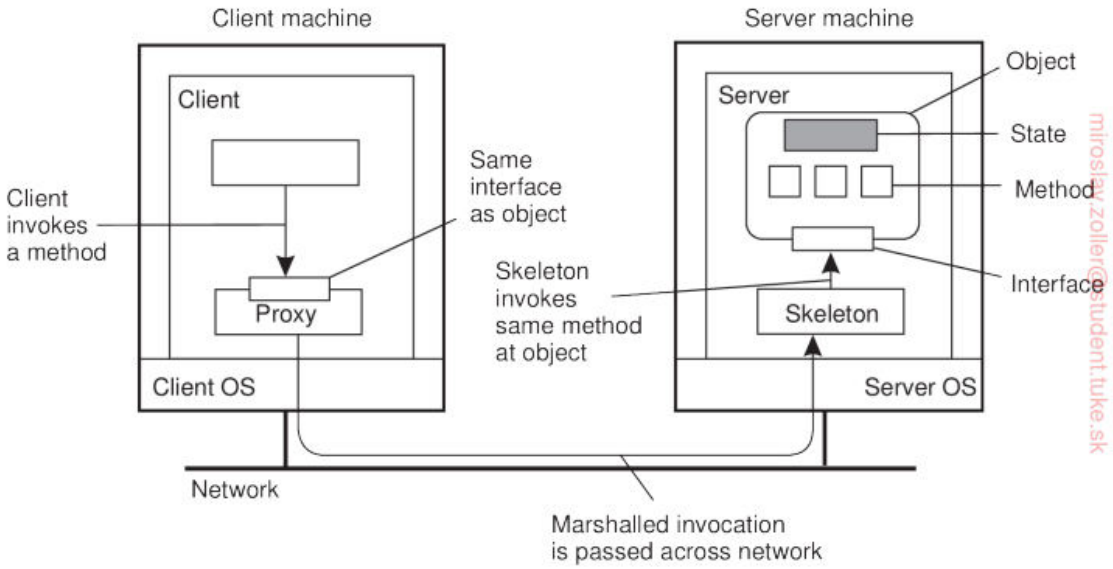


Figure 2.6: Common organization of a remote object with client-side proxy.

Na obrázku 2.6 je zobrazený **Distribuovaný objekt (vzdialený - remote object),** ktorý je uložený na Server Machine**.** V momente kedy sa klient pripojí **(bindne)** na vzdialený objekt, do klienta sa načíta rozhranie vzdialeného objektu - **Proxy** (v našom prípade zo zadaní - **STUB**). Proxy rieši volanie týchto procedúr po sieti (marshalling - proces transformácie pamäťovej reprezentácie objektu do dátového formátu vhodného na ukladanie alebo prenos.).

Distribuovaný objekt sa nachádza na serveri, kde ponúka ten istý interface, ktorý rovnako posiela klientovi. V momente kedy sa zavolá na stube servera - **Skeletone** metóda (unmarshall), tá sa vykoná a výsledok sa odošle späť klientovi cez sieť (marshalling).

1.3.2 Service-oriented architectures (SOA)

Dalo by sa tvrdiť, že objektovo založené architektúry tvoria základ zapuzdrenia služieb do nezávislých jednotiek.

- V architektúre orientovanej na služby je distribuovaná aplikácia alebo systém v podstate konštruovaný ako kompozícia mnohých rôznych služieb

- Kompozícia služieb – napr. integrácia podnikových aplikácií

1.4 Resource-based architectures

- Alternatívny prístup ku kompozícií služieb

Alternatívne je možné vidieť distribuovaný systém ako obrovskú zbierku zdrojov, ktoré sú individuálne riadené komponentmi. Zdroje môžu byť pridávané alebo odstraňované pomocou (vzdialených) aplikácií a podobne môžu byť získavané alebo upravované. Tento prístup je teraz široko prijatý na webe a je známy ako **Representational State Transfer**

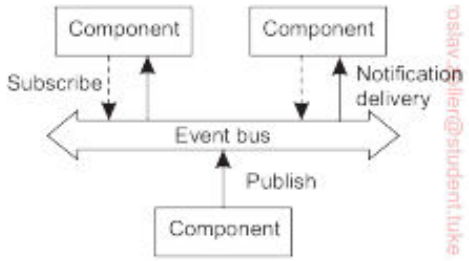
1.4.1 RESTful architektúry

- Jednotlivé zdroje sú identifikované skrz jedinú mennú schému (URL)
- Všetky služby ponúkajú jediný interface (HTTP operácie)
- Správy odosielané medzi jednotlivými službami sú **plne popísané** (XML,JSON)
- Po vykonaní operácie služba zabudne všetko o callerovi – **bezstavové vykonanie**

1.5 Publish-subscribe architectures

- Voľné prepojenie procesov (prcoes moze byt nezávisle pridaný alebo odstránený z Distribuovaného systému)

1.5.1 Event-based style



(a)

V jednoduchosti komponenty sa explicitne medzi sebou nepoznajú ale dokážu odosielať a prijímať určite správy a notifikácie a na základe nich vykonávať akcie. Mechanizmus cez ktorý sa spája odosielateľ s príjimateľom rieši **Event bus.**

1.6 Middleware organizacia:

Aby sme docielili **otvorenost (openness)** mame na vyber z dvoch „design patternov“:

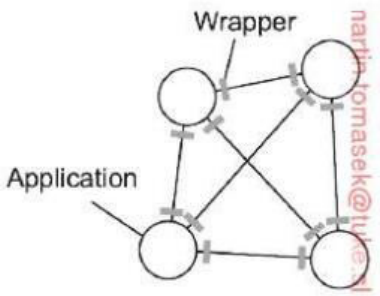
- **Wrappery**
- **Interceptory**

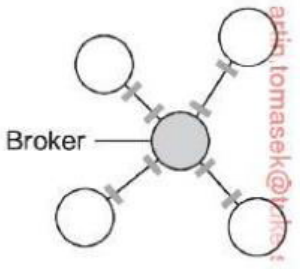
Na docielenie adaptívneho softverového vývoja používame

- **Komponentový dizajn => staticky,dynamicky**

1.6.1 Wrappery

Je to specialny komponent (objektovo orientovany adapter alebo broker) s akceptovatelnym rozhranim. Tzn riesi problem kde 2 interfacy su inkompatibilne.(Adapter sa pouziva viacmenej pre kazdu aplikaciu, broker redukuje pocet adapterov.)





1.6.2 Interceptory: (zachytavac)

Je to softvérový konštrukt, ktorý preruší zvyčajný tok riadenia a umožni spustenie iného (aplikacne specifickeho) kódu. Tzn Middleware sa adaptuje pre potreby aplikacie. To aby boli **interceptory generické** si vyžaduje vela úsilia.

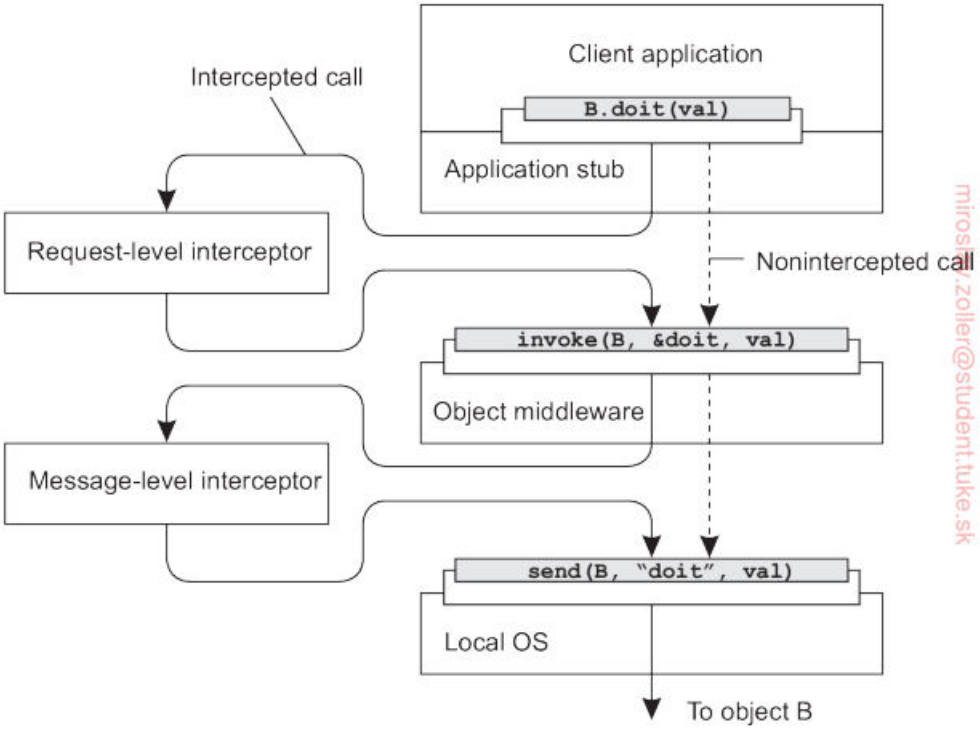


Figure 2.14: Using interceptors to handle remote-object invocations.

1.7 Architektúry systému

Reálna organizácia a umiestnenie komponentov v distribuovanom systéme.

- **Centralizovane (client-server modely)**
- **Decentralizovane (peer-to-peer model)**
- **Hybridne architektury**

1.7.1 Centralizované organizácie (client-server model)

- Jednoducha client-server architektura
- **Server** je process ktory ponuka sluzbu
- **Klient** je process ktory sa dotazuje na server a ocakava nejaku formu odpovede
- Na komunikaciu sa musi pouzit spravne prepojenie napr TCP/IP aby sa zarucila **idempotentnost** (odoslanie operacie viacnasobne neuskodi systemu)

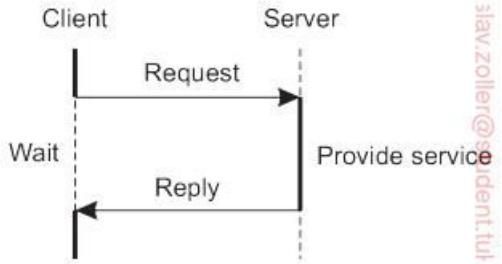


Figure 2.15: General interaction between a client and a server.

1.7.1.1 TWO TIERED ARCHITECTURE

Kedze mnozstvo distribuovancyh aplikacii su rozdelene do 3 vrstiev a to: UI,Processing layer a Data layer. Jeden z sposobov ako organizovat client-server architekturu je na nasledujucom obrazku. Nazyva sa to **two-tiered architecture.**

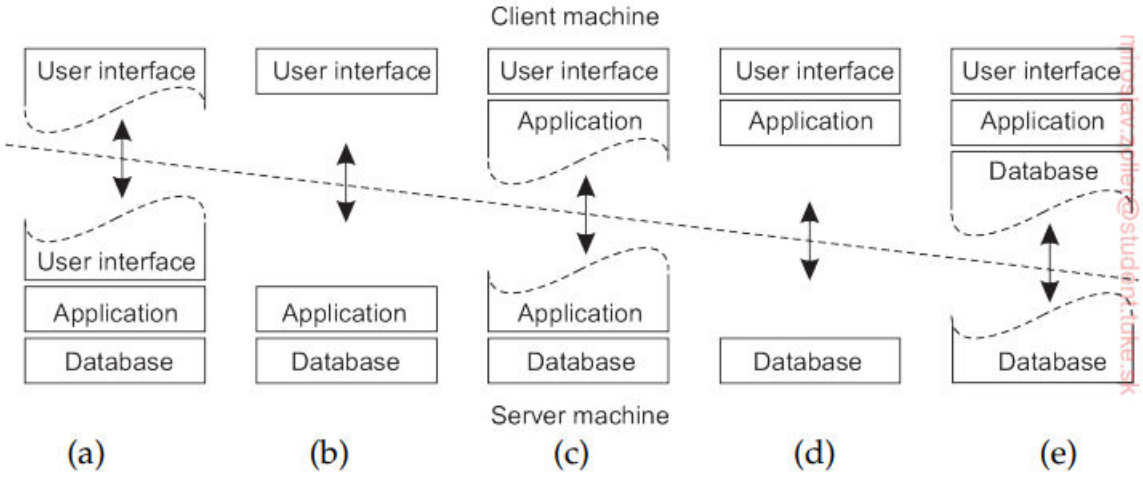
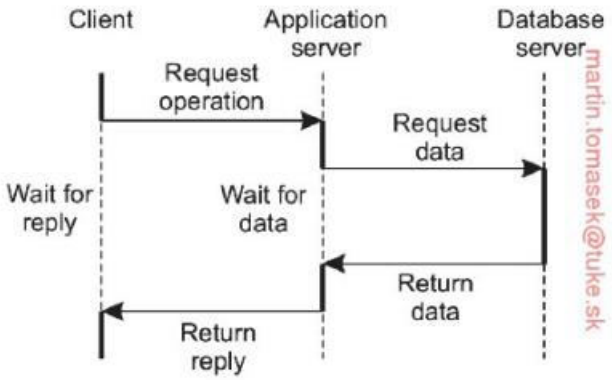


Figure 2.16: Client-server organizations in a two-tiered architecture.

- **A** je napr klasicka hlupa aplikacia co pouziva aj nejaky UI checky na serveri THIN CLIENT
- **B** je klasicke rozdelenie frontendu a backendu kde server spracuvave cele spravanie apky
- **C** je take hybridne, kde klient robi lokalne nejake checky (napr checkuje formu pred jej submitom, staticka kontrola gramatiky, cachovanie pamate atd)
- **D** vsetky operacie robi na strane klienta a len robi cally na databazu ak potrebuje (napr banking apky – po vykonani a overeni transakcie sa updatne databaza banky)
- **E** Napr webove apky si buldia OBROVSKU CACHE na lokalnom stroji FAT CLIENT

1.7.1.2 THREE TIERED ARCHITECTURE

- Server sa sprava ako klient



1.7.2 Decentralizované organizície: Peer-to-peer systemy

Interakcie na peer-to-peer systemoch su symetricke. Teda **nody (Node je jednotka p2p siete klient/server)** sa spravaju ako klienti a zaroven ako servery.

Vlastnosti:

- **Ziadna centralizovana koordinacia**
- **Ziadna centralizovana databaza**
- **Noda nema prehlad o systeme**
- **Spravanie je zalozene na interakcii medzi jednotlivymi nodami**
- **Nody su autonomne (nezavisle)**
- **Nody a ich prepojenia nemusia byt spolahlive**
- **Vsetky data v systeme musia byt dostupne**

1.7.2.1 Vertikálna distribúcia:

Pod ne spadaju **multi-tiered (**viacvrstvove**)** aplikacie. Ktore sa logicky rozdeluju komponenty na rozne stroje. Ale to je len client-server architektura.

1.7.2.2 Horizontálna distribúcia:

Pri tomto type distribúcie môže byť klient alebo server fyzicky rozdelený na logicky ekvivalentné časti, ale každá časť pracuje na svojom vlastnom podiele na kompletnom súbore údajov, čím sa rieši aj „load balancing“. Tzn kazdy node je aj klientom aj serverom.

1.7.2.3 Strukturovane peer-to-peer systemy

Tato topologia sa pouziva na efektivne vyhladavanie dat pomocou key value pairs. Obsahuje struktovane topologie ako ring,binary tree, grid etc.

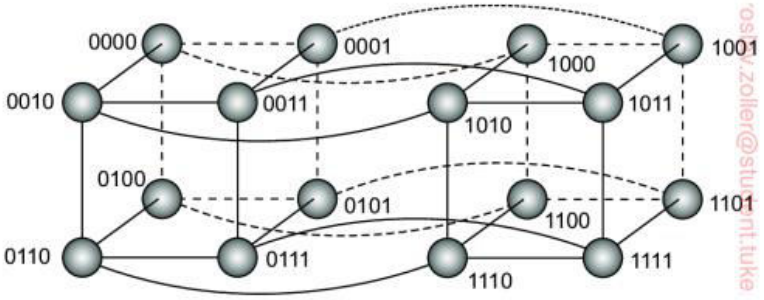


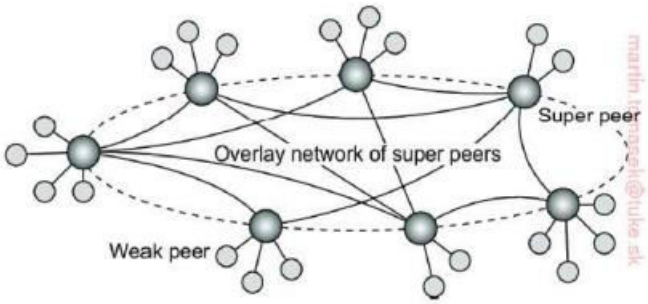
Figure 2.18: A simple peer-to-peer system organized as a four-dimensional hypercube.

1.7.2.4 Nestrukturovane peer-to-peer systemy

- Random graf
- Lookup sa robi hladanim dat
- Potrebujeme zaviest TTL - time to live aby sme sa vyhli zahlteniu
- Random cesty – su uzitocne, ale taktiez potrebujeme TTL a mnoho takychto requestov
- Preferovanie ciest

1.7.2.5 Hierarchicke peer-to-peer orgganizaice

- Obsahuje super peer, ktory maintainuje nody pod sebou sprava sa ako broker
- Obsahuje Weak peer, je to noda ktora sa napaja na super peer

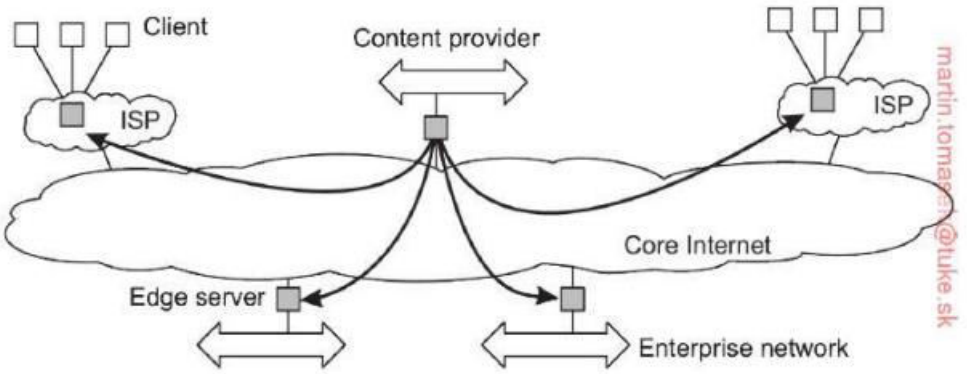


1.7.3 Hybridne architektúry

Kombinacia Client-Server architektur s Peer to peer architekturami.

1.7.3.1 Edge-server systemy

Server je ulozeny „na okraji“ siete. Napr hranica medzi ISP a podnikovou sietou.

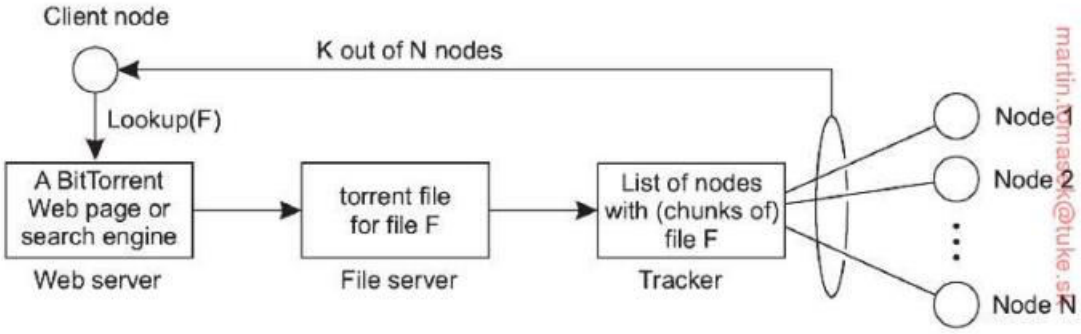


1.7.3.2 Colaborativne distribuovane systemy

2 fazy komunikacie:

- Join – client server komunikacia
- Operate – Decentralizovana scheeme kolaboracie

BitTorrent:



Základná myšlienka je taká… keď koncový používateľ hľadá súbor, stiahne si časti súboru z ostatných používateľov, kým sa stiahnuté časti neposkladajú kompletný súbor.

2 Prednáška 3 – Procesy

2.1 Proces

Operacny system vytvara mnozstvo **virtuálnych procesorov** aby spustili program

Jednotlive virtualne procesy su uložene v **procesnej tabulke**, ktora obsahuje hodnoty registrov, mapu pamate, otvorene subory a ine dolezite data.

Proces je program ktory je vykonavany na jednom z virtualnych procesorov. (Procesy nemôžu škodlivo alebo neúmyselne ovplyvniť správanie iných proceov – súbežne zdieľajú CPU a iné hardvérové zdroje)

Transparentnost subeznosti (concurrency transaprency) je nakladna:

- Vytvorenie nezavysleho adresneho priestoru
- Prepinanie CPU medzi procesmi
- Swapovanie procesov medzi diskami a pamatou

2.2 Vlakno

- **Proces obsahuje viacero nezavislych vlakien**. Su lepsie na vytvaranie distribuovanych aplikacii a zaroven zvysuju vykon.
- Nepokúšate sa dosiahnuť vysoký stupeň transparentnosť súbežnosti
- Mala by sa udrziavat jednoduchost pri pouzivani threadov
- **Multithreading** zvysuje vykon
- Thready nie su chranene medzi sebou

2.3 Implementácia Threadov

- User level threads (kniznice)

Lahko sa vytvaraju lahko sa nicia lahko sa medzi nimi prepina su easy. Pouzivatel ma nad nimi kontrolu.

- Many-to-one threading model

Tieto thready sa daju schedulovat pomocou entity, ktora sa o to stara. Daju sa blokovat procesy

- One-to-one threading model

Kazdy Thread je entita ktora sa da schedulnut. Vsetko riesi kernel. Prepinanie kontextu threadov je drahe takisto ako prepinanie procesov. Typicky sa pouzivaju v dnesnych operacnych systemoch.

2.4 Vyuzitie vlakien v nedistribuovanych systemoch

- Blokovanie systemovych volani v jedno-vlaknovej apke zastavi celu apku, preto ak pouzivame viacero vlakien, pri zastaveni jedneho ostatne mozu dalej vykonvavat.
- Vyuzitie paralelnosti a multiprocesnosti.
- Dobre vyuzitie vo velkych aplikaciach.

2.5 Multithreaded klienti

- Zvyčajný spôsob, ako skryť latenciu komunikácie, je spustit komunikáciu a pokračovať v niečom inom
- Napr webovy browser zobrazuje data medzitym co sa este tahaju zo servera
- Viacero pripojeni moze byt vytovrenych sucasne
- Viacero pripojeni na rozne repliky serverov umoznia odosielat data paralelne

2.6 Multithreaded servers

Dispatcher ocakava requesty a potom rozhoduje, ktory thread bude pracovat s poziadavkou.

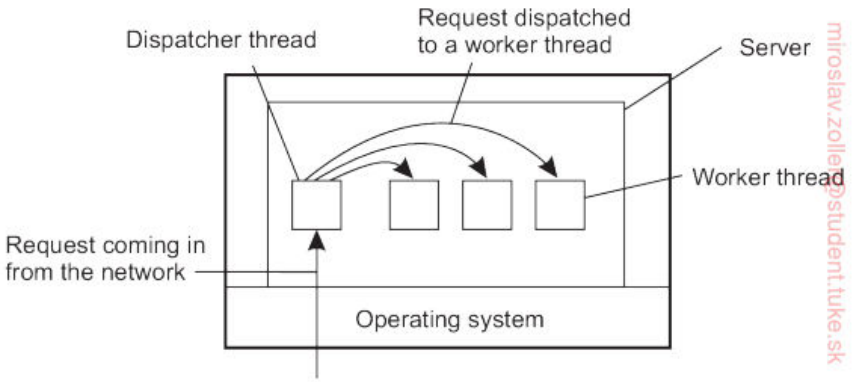


Figure 3.4: A multithreaded server organized in a dispatcher/worker model.

Modely ako vytvorit server:

- Multithreading server – paralelizmus, blokovanie systemovych volani
- Single-threaded process – ziadny paralelzimus, blokovanie systemovych volani
- Konecny automat – paralelizmus, neblokujuce sa systemove volania

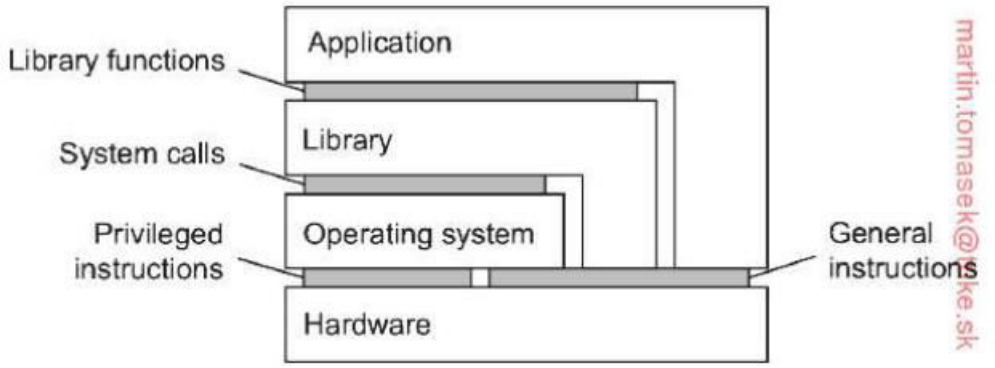
2.7 Virtualizacia

- **Multithreadovane** procesy vykonavaju simultanne vykonavanie
- **Resource virtualization** – proste pracujeme vo virtualizovanom systeme s aplikaciou teda, spravanie apky nezavisi od realneho softveru/hardwaru
- **Vyhody**: portabilita, flexibilita, portovanie softveru

2.8 Architektúry virtuálnych strojov

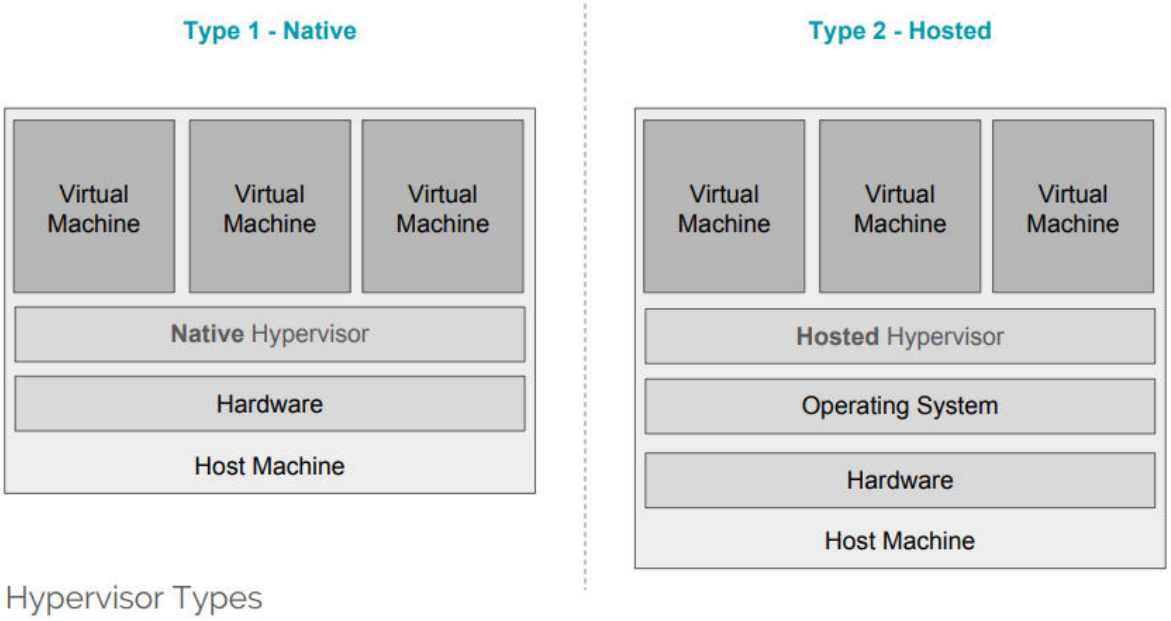
4 levely rozhrani:

- Vseobecne instrukcie
- Privilegovane instrukcie
- Systemove volania
- API



2.8.1 Virtualizacia:

- **Process virtual machine:**
  - virtuálna platforma vytvorená pre individuálny proces a zničená po ukončení procesu
- **Nativna virtualna machine:**
  - Natívne hypervízory bežia priamo na hostiteľskom počítači a zdieľajú zdroje (napríklad pamäť a zariadenia) medzi hosťovanými počítačmi.
  - e.g. XEN, Oracle VM Serve
- **Hostovana virtualna machine:**
  - Hostovany hypervizor spusteny ako aplikácia v rámci operačného a podporuje virtuálne stroje bežiace ako samostatné procesy.
  - e.g. VirtualBox, Parallels Desktop, QEMU



Hypervisor Types

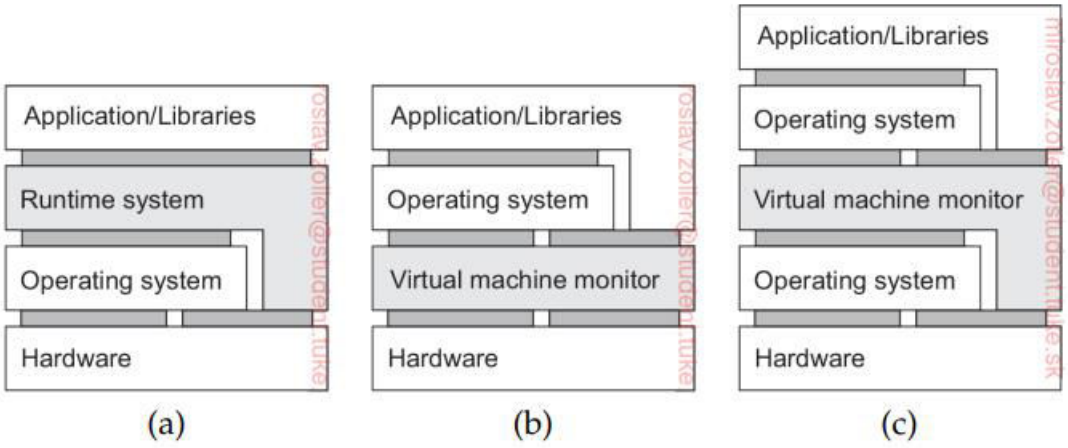


Figure 3.8: (a) A process virtual machine. (b) A native virtual machine monitor. (c) A hosted virtual machine monitor.

2.9 Klienti - Networked user interfaces

- Klient je v podstate zosietovany User interface aj sam server sa moze spravat ako klient
- Dva druhy syncrhonizacie podla obrazka:
    - Synchronizácia cez Aplikacnu vstvu rozhrania - A
    - Synchronizácia ktora nezavysi od aplikacie, tenky klient - B

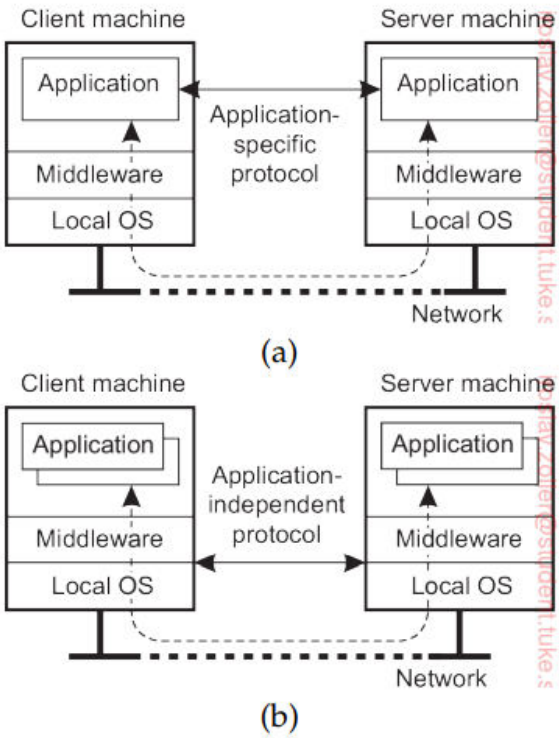


Figure 3.10: (a) A networked application with its own protocol. (b) A general solution to allow access to remote applications.

-

2.10 Thin-client network computing

Separacia logiky aplikacie od UI commandov.

Uplna kontrola vzdialeneho PC (VNC , RDP)

VNC - Virtual Network Computing je grafický program, který umožňuje vzdálené připojení ke grafickému uživatelskému rozhraní pomocí počítačové sítě.

2.11 Client-side software fro distribution transparency

- Procesy a data level je na strane klienta – POS systemy, ATM, barcode skenery, TV STB ...
- Vyuzivaju sa tu klientske stuby, ktore rovno komunikuju so serverom
- Requesty sa mozu replikovat

2.12 Server

Čaká na prichádzajúce požiadavky od klientov, stará sa o to a čaká na ďalšiu prichádzajúcu požiadavku

2.12.1 Druhy serverov

2.12.1.1 Interaktívny server

Caka, spracováva a odpovedá na request pomocou jedneho vlakna

2.12.1.2 Súbežný server

Subezne spracuvava requesty

2.12.2 Pripojenie na server

End point – port na ktorom server bezi, kde ho ma klient kontaktovat

No end point – port je dynamicky daný, klient si ho dohlada

Superserver – počúva na kazdom porte, vracia inštanciu servera každému klientovi

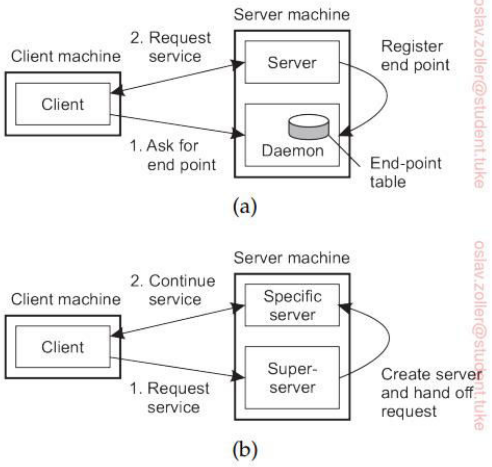


Figure 3.13: (a) Client-to-server binding using a daemon. (b) Client-to-server binding using a superserver.

2.13 Bezstavovy server - stateless

- Neudrziava informacie o aktualnom stave klientoch napr. Web service, RESTful apky
- Server uchováva informácie o svojich klientoch
- **Soft state -** V tomto prípade server sľubuje udržiavať stav v mene klienta, ale iba na obmedzený čas
- **Cookies –** ked server nemoze udrziavat stav klienta, web browser si pomocou cookies ulozi informacie pre tento server
- **Lahko skalovatelne a replikovatelne servery**

2.14 Stavovy server – stateful

- Persistuje data od klientov – napr subory na serveri
- Neslubuje konzistentne zotavenie po vypadku server, (stav sa musi vratit presne do toho bodu ako pred padom)
- Permanetne su informacie ulozene v databazach

Oba stavove aj bezstavove servery ponukaju sluzby, mozu pouzivat cookies a je dolezite pre nich aby boli skalovatelne.

2.15 Server cluster

Mnozina zariadeni prepojene cez siet, kde kazda masina rozbehava jeden alebo viac serverov.

3-vrstvova organizacia clusterov:

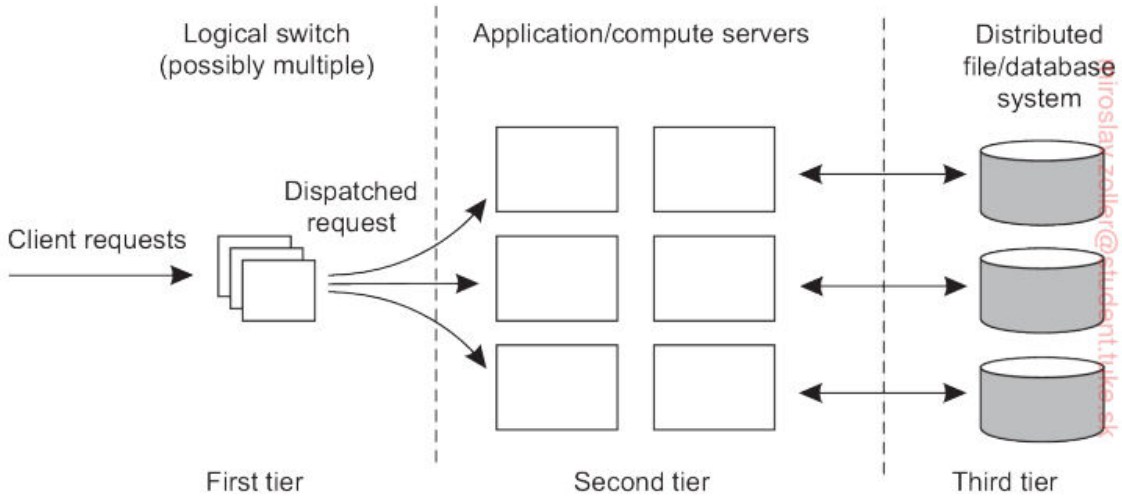


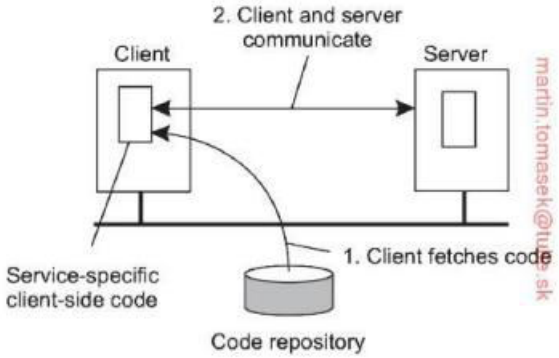
Figure 3.18: The general organization of a three-tiered server cluster.

2.16 Migracia kodu

Komunikácia nie je obmedzená na odovzdávanie dát – odovzdávanie programov a vzdialené spúšťanie môže zjednodušiť distribuovaný systém

2.16.1 Dovody preco pouzivat migraciu:

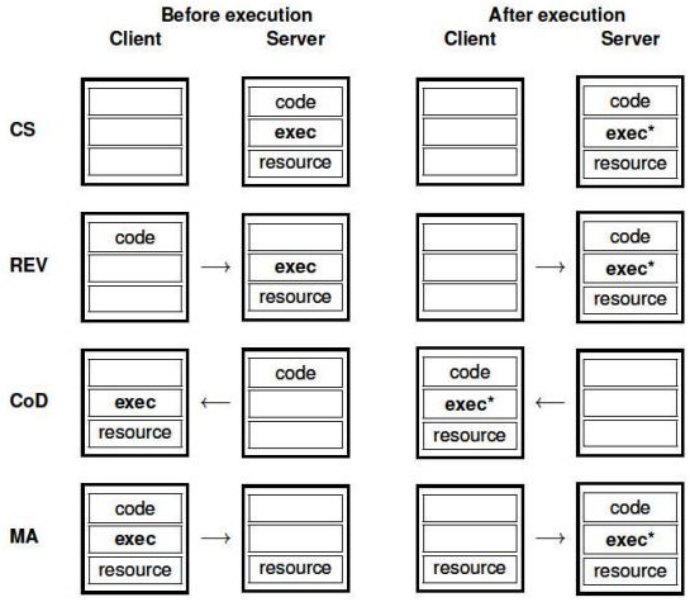
- Mobilne agenty
- V prípade potreby si stiahnite implementáciu proprietárneho protokolu
- Migrácia procesov pre rozloženie záťaže



-

Models for code migration

- Framework of a process
  - Code segment
  - Resource segment
  - Execution segment
- Models
  - Client-server (CS)
  - Remote evaluation (RE)
  - Code-on-demand (CoD)
  - Mobile agent (MA)

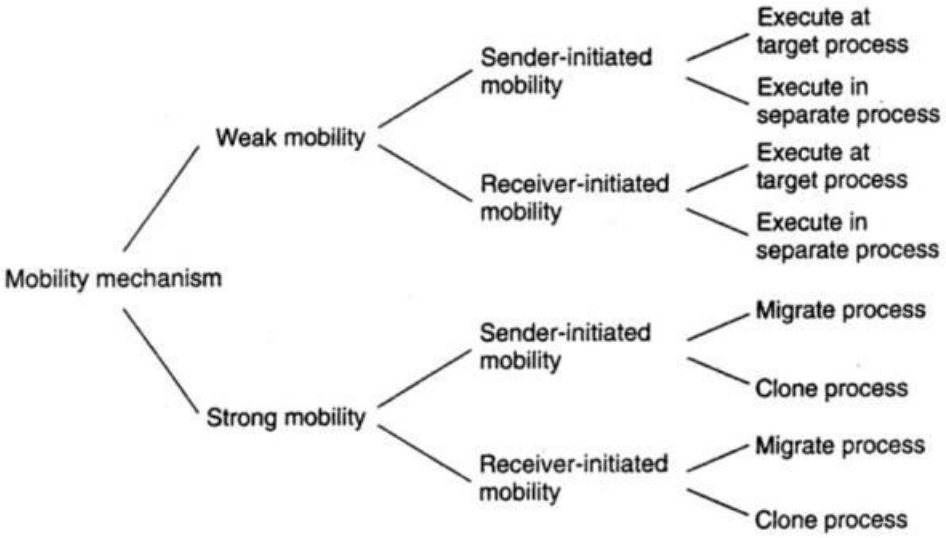


SLABA MOBILITA

Iba cast kodu je migrovana. Napr Triedy.

SILNA MOBILITA

Cast procesu je migrovana.



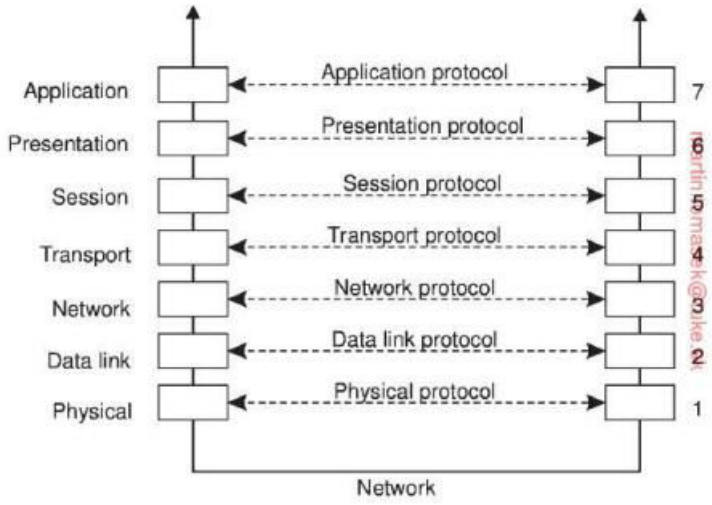
2.17 Migracia lokalnych zdrojov:

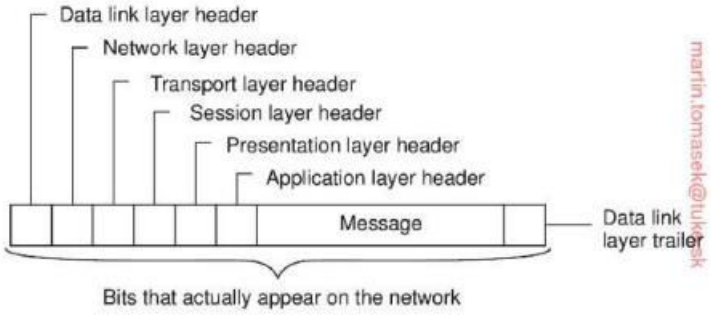
- Vazby (bindings) :
  - Identifier – URL
  - Value – locally accesible libraries
  - Type – references to local devices

- Resource relocation – nepripojene zdroje, databzy, data, fixne zdroje
- Migracia celeho prostredia
- Migracia VM

3 4. Prednaska – Komunikacia

3.1 OSI model a klasická message cez siet:





3.2 Low level protokoly:

Fyzicka vrstva – posiela bity na zaklade elektrickych, mechanickych a signalizacnych rozhrani

Data link vrstva – zoskupuje bity do ramcov, kontroluje cez checksum spravnost prenosu

Sietova vrstva (network) – Posiela spravu od odosielatela k prijimatelovi

3.3 Transportne protokoly – transportna vrstva

- Sprava z aplikacnej vrstvy je rozbitá na packety, ktore su vhodne na prenos medzi low-level protokolmi
- Konstruuje spravu z aplikacnej vrsty pomocou ziskanych paketov
- Spolahlivy transport:
    - **connection-oriented network (packet pride v korektnej sequencii) - TCP**
    - **Connectionless network (packet pride v inej sequencii) - UDP**
    - **Vrstva je zodpovedna za vyskladanie spravy**
- TCP,UDP,SRTP,SCTP

3.4 Protokoly vyššej vrstvy – Higher level protocols

Session layer (relačná vrstva):

- udržiava komunikačný kanál
- synchronizuje
- dôležitý na **vývoj middleware riešení**
- sleduje ktorá strana práve komunikuje

Prezenčná vrstva:

- Strukturovane zaznamy spravy
- Encoding-decoding sprav
- Prijimatel pozna specificky format spravy

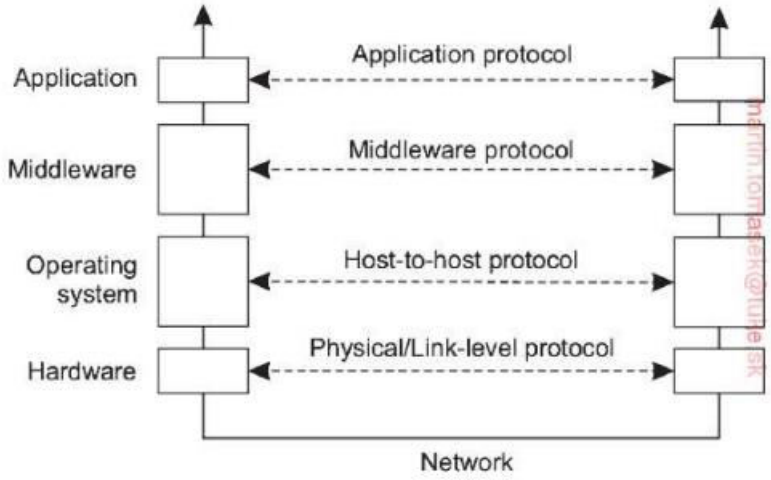
Aplikačná vrstva:

- Štandardne protokoly : SMTP,FTP,HTTP,Telnet..
- Všetky ostatne aplikacie a protokoly
- Všetky distribuovane systémy sú len aplikacie

3.5 Middleware protokoly

Middleware je aplikácia, (nachadzajuca sa na aplikacnej vrstve), ktora obsahuje univerzalne protokoly.

- Napr. RPC protokoly, autentifikacne protokoly, autorizacne protokoly, distribuovane lockovacie protokoly a mnohe..



3.6 Typy komunikácie

Middleware ako medzislužba na aplikačnej úrovni komunikácie. Typickym prikladom moze byt elektronicka mailova sluzba, kde cele jadro je povazovane za middleware.

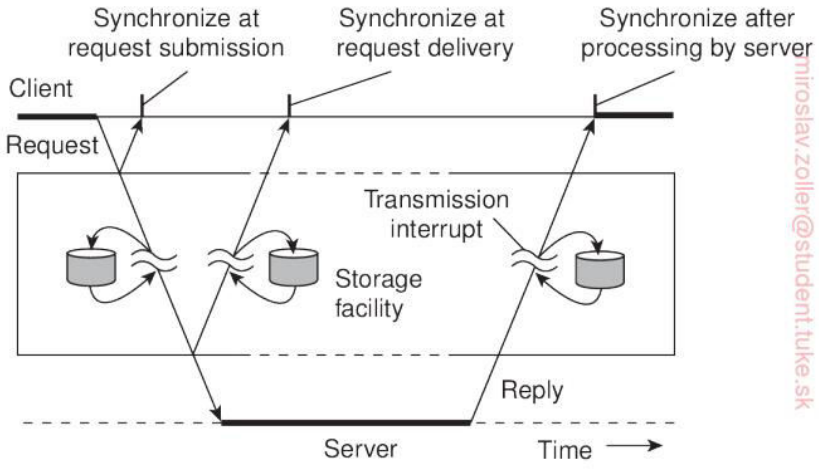


Figure 4.4: Viewing middleware as an intermediate (distributed) service in application-level communication.

3.6.1 Perzistentna komunikacia

- Správa na prenos je uložená v komunikačnom middleware, kým nie je doručená
- Odosielateľ nemusí po odoslaní pokračovať v komunikácii
- Prijímacia aplikácia nemusí byt spustená aby si správu mohol prijímateľ precitať

- Napr. Mail server

3.6.2 Prechodná komunikácia (Transient communication)

- Správa sa nachádza v komunikačnom kanáli, len ak odosielateľ a prijímateľ sú spustení
- Keď nastane prerušenie spojenia, správa sa vyradí/znici/zrusi (discard)
- Napr. store-and-forward routers a vsetky transport level komunikacie su transient

3.6.3 Asynchronna komunikácia

- Odosielatel pokracuje v inych ulohach po tom co je sprava odoslana
- Sprava je dočasne ulozena v middlewari

3.6.4 Syncrhonna komunikácia

- Odosielatel je blokovany, kym jeho poziadavka nie je akpectovana
- Odoslielatel moze byt blokovany kym ho middleware nenotifikuje ze prebera poziadavku za neho
- Odosielatel moze byt synchronizovany, kým sprava nebude doručená príjemcovi
- Odosielatel moze pockat kym poziadavka nebude plne spracovana a odpoved od servera prijata

3.7 Remote Procedure call – RPC

Explicitná výmena správ medzi procesmi a nezakrýva komunikáciu. Nie je možné dosiahnuť transparentnosť prístupu v distribuovaných systémoch.

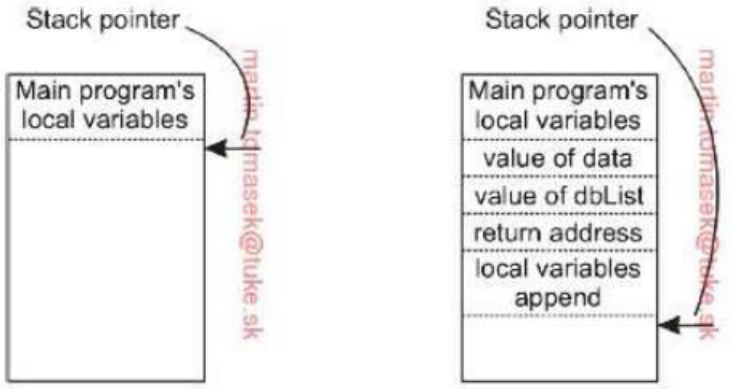
Je to v podstate technologia, ktora dovoluje programom vykonat kod na inom mieste, nez je skutocne umiestneny program. Dobrym prikladom je vykonanie narocnejsej operacie na vykonnejsom pocitaci.

IDEA pacol som ju z wikipedie myslim ze je dost korektna:

1. Proběhne zabalení identifikátoru procedury a vstupních parametrů do formy vhodné pro přenos. (Jedná se o tzv marshalling.)
2. Balíček se odešle.
3. Entita určená k vykonání procedury balíček rozbalí a seznámí se s jeho obsahem. (Jde o tzv. unmarshalling.)
4. Dojde k provedení procedury.
5. Proběhne další zabalení, tentokrát výstupu procedury.
6. Data se odešlou zpět volající entitě.
7. Dojde k rozbalení.
8. Proběhne předání nadřazenému podprogramu.

3.8 Convencional procedure call (single machine call) konvenčne volanie procedury

```
newList = append(data, dbList);
```



Call by value – v nasej funkcii su to data, ktore obsahuju jednoduchu zlozku dat

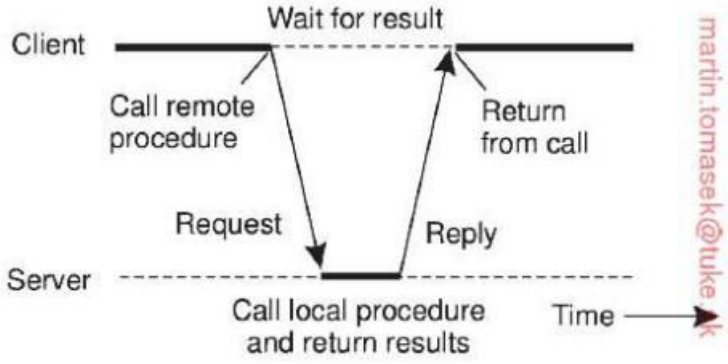
Call by reference – v nasej funkcii je to pointer na DbList

Call by copy – zastarala, v podstate kopiruje hodnotu na stack a potom ju kopiruje spat vo vysledku

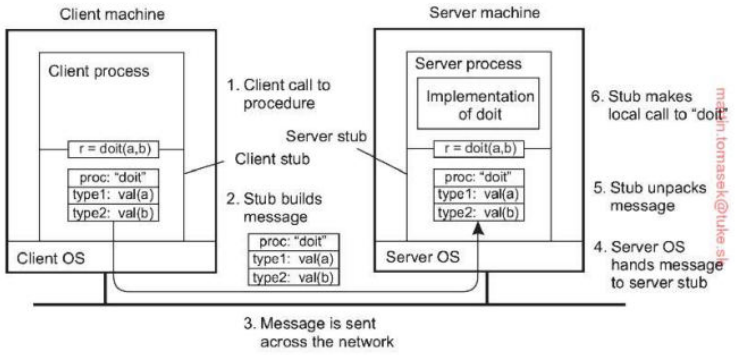
3.9 Klient a server stuby

Predstavme si ze server ma implementovanu tu Append funkciu z predosleho prikladu.

- Klient implementuje **stub** proceduru append – ktorou zabali parametre a posle ich ako spravu na server
- Server implementuje **stub** proceduru append – ktorou ziska spravu, preparsuje parametre a spusti funkciu append na strane servera, potom vrati vysledok ako spravu do klienta



Steps of remote procedure call



1. The client procedure calls the stub in the normal way
2. The client stub builds the message and calls the local client's RPC middleware
3. The client's RPC middleware sends the message to the remote server's RPC middleware
4. The server's RPC middleware gives the message to the server stub
5. The server stub unpacks the parameters and calls the server
6. The server does the work and returns the result to the stub
7. The server stub packs it in a message and calls its local server's RPC middleware
8. The server's RPC middleware sends the message to the client's RPC middleware
9. The client's RPC middleware gives the message to the client stub
10. The stub unpacks the results and returns to the client

3.10 Posielanie parametrov

Marshalling – serializacia, prenos dat cez siet

Input message (vstupna sprava) – obsahuje:

1. Nazov procedury
2. Data pre parametre procedury

Output message (vystupna sprava) – obsahuje:

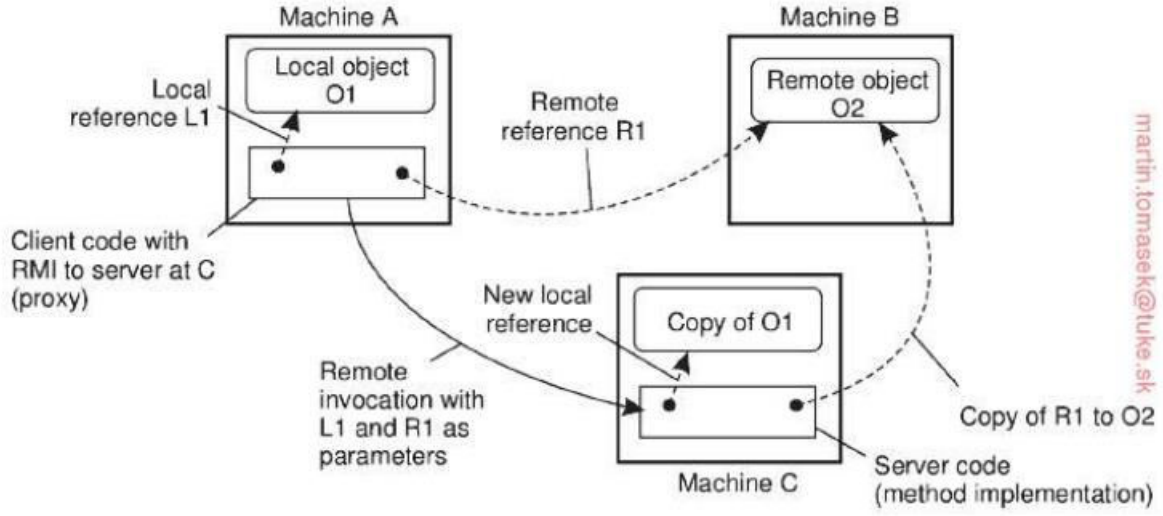
- Data ako vysledok volania funkcie

Odosielanie dat (bytov) – Little endian, big endian

Odosielanie referencii:

- Obsah (content) pamäte sa odošle spolu so vsetkymi udajmi
- Jednoduche polia a struktury su jednoduche
- Pri dynamickych datovych typoch ako su grafy si server ziska od klienta vsetky hodnoty z referencie vsetky VALUES

3.11 Odosielanie parametrov v Objektovo orientovanych systemoch



A vola C s referenciou do lokalneho O1 a referenciou do vzdialenej O2 ktora sa nachadza v B

O1 je kopirovana z A do C

O2 je referencovana z B do C

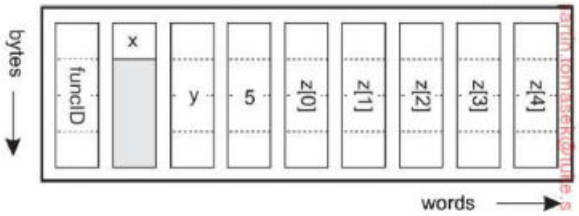
Cize ked Referencia je volana z toho isteho Objektu/klienta tak sa na druhy server referencia odosle ako kopia. O1.

Vzdialene referencie sa vzdy budu odosielat ako referencie.

3.11.1 Specifikacia parametrov a vytvorenie stubov

- Zainteresovane strany musia suhlasit na **formate spravy**

```
void someFunction(char x, float y, int z[5])
```



- Zainteresovane strany musia suhlasit na **reprezentacii dat** (String,Bool,Int…)

- Zainteresovane strany musia suhlasit na **sposob transport dat** (UDP/TCP)
- **Interface Definition/Description Language** – jazyk popisujuci rozhranie, je všeobecný výraz pre jazyk, ktorý umožňuje programu alebo objektu napísanému v jednom jazyku komunikovať s iným programom napísaným v neznámom jazyku
- **Supprt na zaklade jazykov** – JAVA(RMI), RPyC(Python), gRPC(Java,Go,Python,C++,C#)

3.12 Asynchronne RPC

- Volanie RPC ako v konvenčnej procedúre sa zablokuje, kým sa nevráti odpoveď
- Niekedy nepotrebujeme cakat na vysledok, (napr. Pridanie do databazy, start sluzby, money transfer..)
- **Assyncrhonne RPC:**
    - server okamzite odosle odpoved po tom co ziska request od klienta, teda mu odpovie ze “ano ziskal som poziadavku idem vykonat zmenu na serveri”
    - Odpoved (reply) je len ACK pre klienta

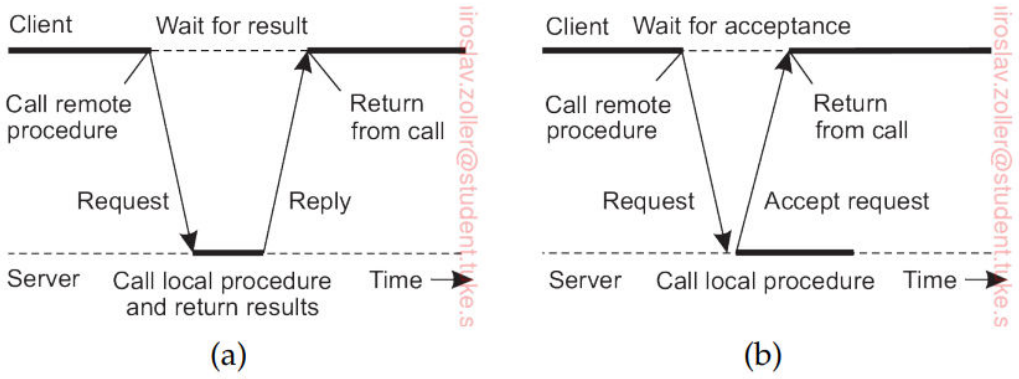


Figure 4.13: (a) The interaction between client and server in a traditional RPC. (b) The interaction using asynchronous RPC.

3.13 Deferred (postponed) synchronous RPC – odložené synchronne RPC

Niekedy klient potrebuje robit ine operacie kym caka na odpoved z RPC. Preto potrebujeme nejaky callback.

- Najprv, klient zavola server a bude pokracovat po tom co server mu posle ACK, ze zacal spracovavat poziadavku
- Podruhé, server zavola klientovi ze ma data, Klient je preruseny a prebera vysledok.

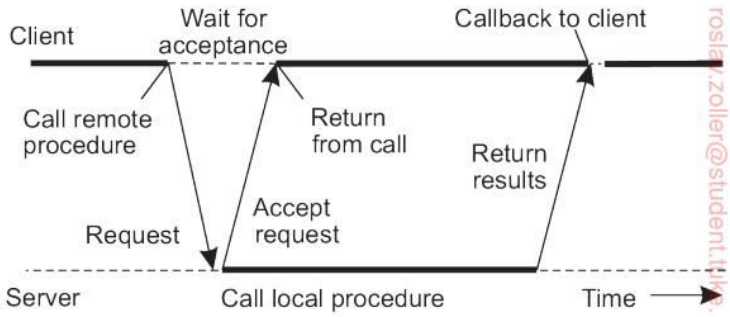


Figure 4.14: A client and server interacting through asynchronous RPCs.

3.14 One way RPC – jednosmerne RPC

- Klient odosle poziadavku a necaka na ACK od servera.
- Spolahlivost nie je garantovana, pretoze klient nevie ci poziadavka bola spracovana
- Klient moze poziadat server, ci su vysledky dostupne
- **Velmi sa vyuziva v CLOUDOVYCH VYPOCTOCH**, pretoze sa to lepsie skaluje

3.15 Multicast RPC

- One way RPC sa moze odoslat poziadavku skupine serverov, vysledok sa vrati vo forme callbacku do klienta
- Klient nevie kolko serverov je requestnutych
- Klient sa moze rozhodnut ci chce pockat na jeden alebo vsetky vysledky

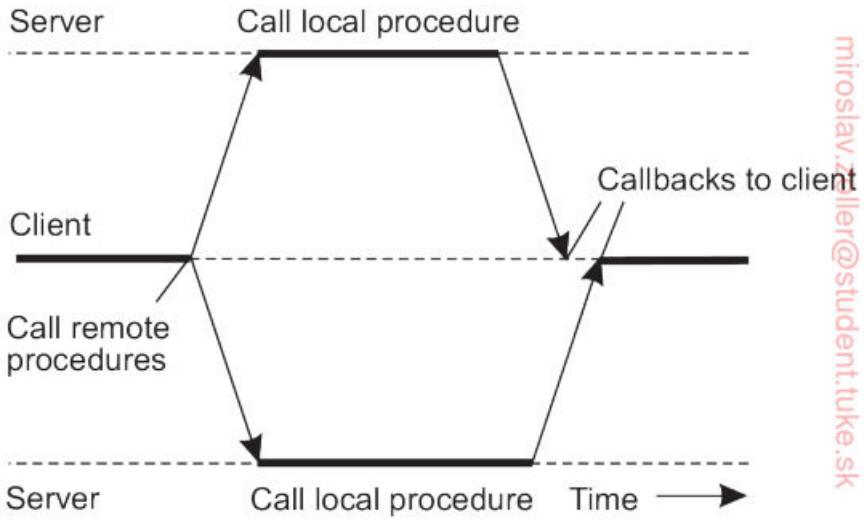


Figure 4.15: The principle of a multicast RPC.

4 5 Prednaska – komunikacia #2

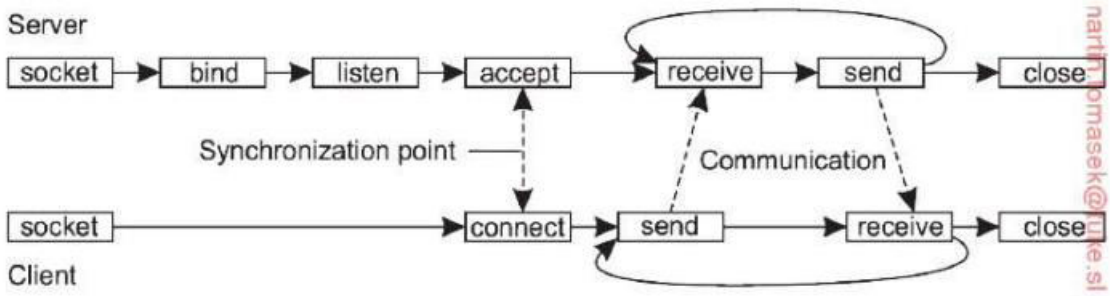
4.1 Komunikacia orientovana na spravy (message oriented communication)

- RPC zahaľuje komunikáciu v distribuovanych systemoch, transparentnosť
- RPC je možné použiť iba vtedy, keď **prijímacia strana je spustená** súčasne s odoslaním požiadavky
- Nahradou za RPC je **messaging** (message oriented communication). Mozene pouzit klasicky synchrony message exchange alebo message-queueing ktore umoznuju odosielat spravy aj ked druha strana momentalne nie je spustena

4.2 BSD sockety (Berkeleyho)

Socket je komunikačný koncový bod, na ktorý si aplikácie môžu zapisovať dáta a z ktorých možno čítať prichádzajúce data.

Sockety sú vhodné pre počítače pripojené do počítačových sietí



4.3 Message-Passing Interface (MPI)

Message Passing Interface (MPI) (rozhranie na výmenu správ) poskytuje prenosný a silný medziprocesorový komunikačný mechanizmus, ktorý zjednodušuje niektoré úskalia komunikácie medzi stovkami a až tisícami paralelne pracujúcich procesorov. Používa sa zvačša pri počítačových blokoch – **CLUSTEROCH**

- MPI bol navrhnutý pre paralelné aplikácie a priamo využíva základnú siet
- Je navhrnuty na riesenie paralelnych problemov s vysokym vykonom

- Pocita s chybami v systeme a neriesi ziadne zotavenie
- Group of processes – groupID, processID

4.4 Message oriented persistent communication

Systémy radenia správ poskytujú rozsiahlu podporu **pre trvalú asynchrónnu komunikáciu**. Podstatou týchto systémov je, že **ponúkajú strednodobú pamäťovú kapacitu** pre **správy bez toho, aby vyžadovali, aby bol odosielateľ alebo príjemca aktívny počas prenosu správy**.

Dôležitým rozdielom medzi soketmi a MPI je to, že **systémy radenia správ sú zvyčajne zamerané na podporu prenosov správ**, ktoré **môžu trvať minúty** namiesto sekúnd alebo milisekúnd.

Sú to **Message-queuing systems or message-oriented middleware (MOM).** Teda dokážu urobiť jednoduchú frontu a riešia poziadavky skrz frontu.

4.5 Model pre message-queueing

- Spravy sa vkladaju do **front**
- Kazda aplikacia ma svoju **privatnu frontu**, z ktorej aplikacia moze len **citat** a ine aplikacie don mozu posielat spravy, je mozne samozrejme **zdielat frontu medzi viacerymi aplikaciami**
- System garantuje ze spravu bude **eventualne vlozena** do prijimatelovej fronty
- KOMBINACIE pre fronty:

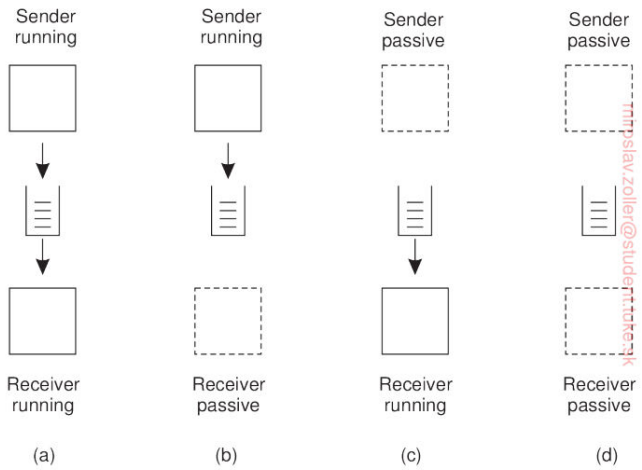


Figure 4.26: Four combinations for loosely-coupled communication using queues.

- Správa može obsahovať dáta
- Správy sú správne adresované

- Velkosť správy može byť limitovaná
- Interface obsahuje pár funkcii:
    - PUT – vloz spravu do fronty
    - GET – blokuj a ziskaj prvu spravu z fronty
    - POLL – neblokuj a ziskaj prvu spravu z fronty
    - NOTIFY – ked je sprava vlozena do fronty, priprav handler

4.6 Architektúra message-queue systemu

SOURCE QUEUE –

- Správy je možné zaradiť iba do lokálneho frontu, t. j. do frontu na rovnakom počítači alebo lokálnej sieti
- Správy je možné čítať iba z lokálnych frontov

DESTINATION QUEUE –

- Správy obsahujú špecifikáciu cieľa
- Systém radenia správ je zodpovedný za prenos zo zdrojových do cieľových radov

Distributed database of queue names

- Maps queue names to network location

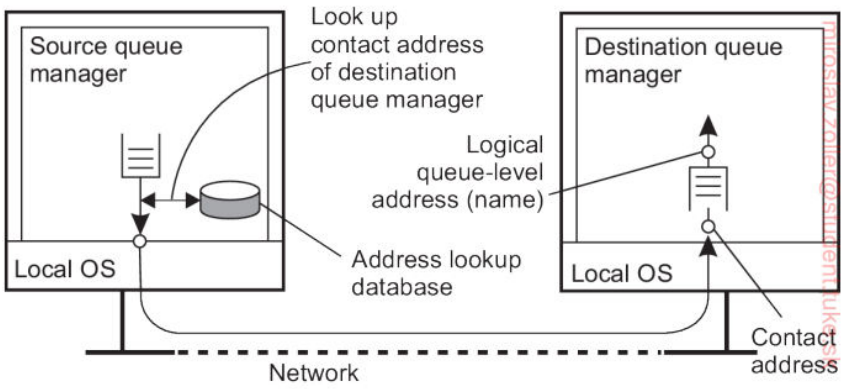
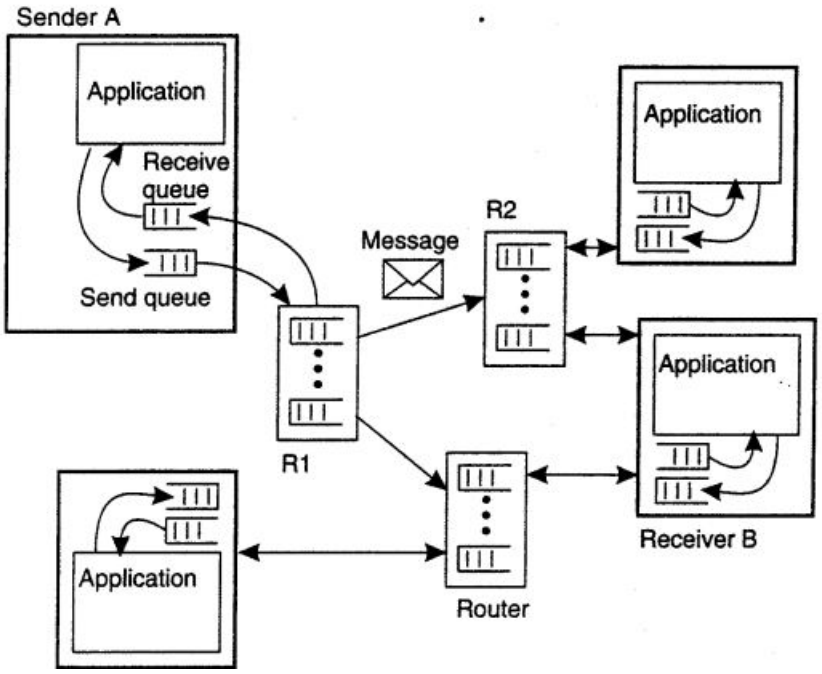


Figure 4.28: The relationship between queue-level naming and network-level addressing.

Queue manager (manažer frontov):

- Interaguje s aplikáciou, ktorá odosiela alebo prijíma správy
- Smerovače alebo relé na preposielanie prichádzajúcich správ iným manažérov frontov



4.7 Message Brokers – sprostredkovatelia sprav

Sprostredkovateľ správ (tiež známy ako integračný spostredkovatel alebo modul rozhrania) je sprostredkujúci modul počítačového programu, ktorý prekladá správu z **formálneho protokolu odosielania správ** do **formálneho protokolu prijímania správ**. Sprostredkovatelia správ sú prvky v telekomunikačných alebo počítačových sieťach, kde softvérové aplikácie komunikujú prostredníctvom výmeny formálne definovaných správ. Sprostredkovatelia správ sú stavebným blokom middlewaru orientovaného na správy (MOM), ale zvyčajne nenahrádzajú tradičný middleware, ako je MOM a vzdialené volanie procedúr (RPC).

- Konvertuje prichádzajúce správy tak, aby boli **zrozumiteľné** pre cieľovú aplikáciu
- Sprostredkovateľ správ sa dá v jednoduchosti opísať ako reformatter pre správy

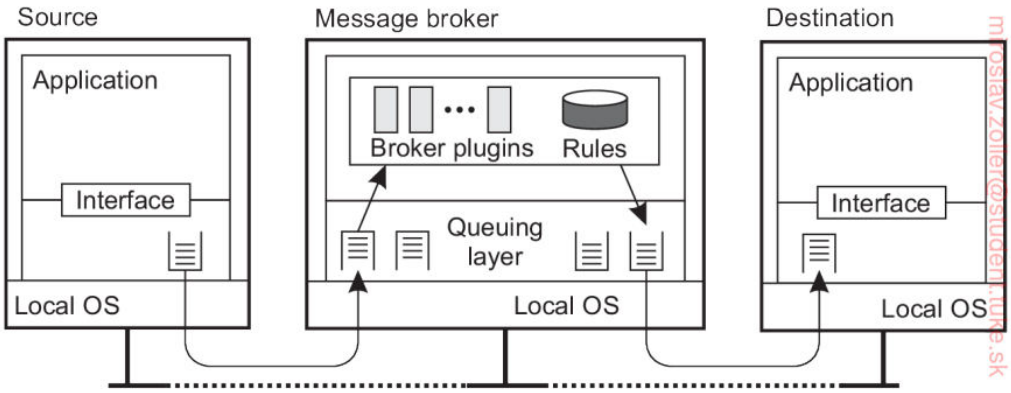


Figure 4.29: The general organization of a message broker in a message-queuing system.

4.8 Multicast komunikácia

- Odosielanie sprav viacero prijimatelom

4.8.1 Aplication level multicasting

Nastavenie explicitných komunikačných ciest v správe prekrytia štruktúr

Konštrukcia prekryvajucej siete:

STROMOVA ORGANIZACIA - Jedinečná cesta medzi každou dvojicou uzlov

MESH ORGANIZACIA - Viaceré cesty medzi každou dvojicou uzlov

Multicast Tree a meranie vykonu:

Stres spojenia – ako často pakety prechádzajú cez odkaz

Stretch alebo Relative Delay Penalty (RDP) – pomer oneskorenia medzi dvoma uzlami v prekryvnej aj základnej sieti

Stromové náklady (Tree cost) – minimalizácia agregovaných nákladov na prepojenie

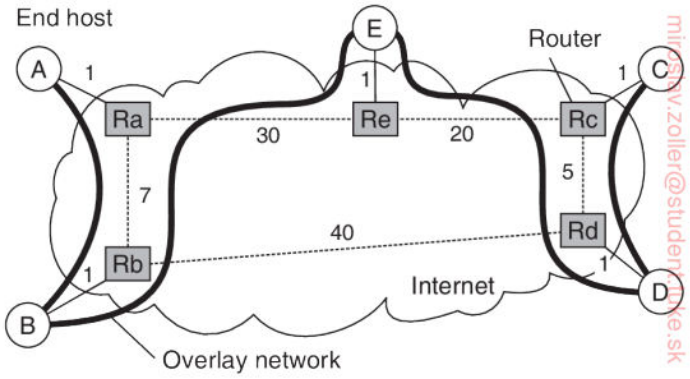


Figure 4.35: The relation between links in an overlay and actual network-level routes.

4.8.2 Šírenie informácií na základe záplav a klebiet

Jednoduché spôsoby (menej efektívne) multicastingu bez explicitných komunikačných ciest

4.8.2.1 Zaplavy:

- Kazda node posuva spravu svojim susedom, okrem tych od ktorych spravu ziskala
- Noda ignoruje duplikaty
- Vykon pre strom je n-1 sprav
- Vykon pre mesh strukturu: (n nad 2)

4.8.2.2 Klebety:

- Edidemicke spravanie nody spreduju informacie v distribuovanom systeme
- Vykon je O(log n)
- Mazanie dat je narocne

5 6 Prednaska Naming

5.1 Mena, identifikatory a adresy

Meno – referuje na entitu distribuovaného systému, **typicke priklady** su disky, subory, tlaciarne, servery, ale aj mailové sluzby, procesy, spravy, sietove spojenia atd.

Access Point – specialny typ entity, ktora poskytuje **pristup** do inej entity, jeho meno je **adresa**. Ak sa pripojime na adresu mozeme vzdialenu entitu spravovat, resp ju pouzivat.

- Entita **moze menit access point** takisto **access point sa moze presunut na inu Entitu**
- Preto aby sme nemuseli riesit konkretnu Adresu entity mozeme ju **pomenovat** nezavisle od adresy (location independent).

Identifikátor – (jednoducho jednozacne meno servera s adresou) - **spravny identifikator** by mal splnať tieto vlastnosti:

1. Referuje na maximalne jednu Entitu
2. Na každú entitu sa odkazuje maximálne jedným identifikátorom
3. Identifikátor sa vždy odkazuje na tu istú entitu

Human-friendly names - Znakový reťazec vo formáte zrozumiteľnom pre ľudí (názov DNS, e-mailová adresa)

5.2 Binding mena k adrese

Ako teda spajame mena a identifikatory k adresam ?

- Najjednoduchsia forma je key-value (name,address) centralizovana databaza ale ta je dost nepouzitelna vo vacsich distribuovanych systemoch
- Dekompozicia mien sa niekolko casti (napr. tak ako funguje DNS), rekurzivne vyhladavanie v domenach takze pre vyhladanie www.tuke.sk budeme mat takuto posutpnost:

  (**root server**) ns(.) -> ns(“sk“) -> ns(“tuke.sk“) -> vysledok

5.3 Simple Flat Naming

mena su retazce znakov bez akejkolvek struktury

5.3.1 Broadcast a multicast

Nove identifikatory su broadcastovane pre vsetkych zariadenia v sieti a kazda masina si skontroluje ci tuto entitu ma ulozenu.

Pouziva sa to v ARP protokoloch. Neefektivne vo velkych sietiach, multicastovanie je lepsia moznost.

5.3.2 Posuvanie pointerov

Ked sa entita presunie, ponecha referenciu na novu lokaciu v starej lokacii, klient sa uz dohlada pomocou referencii

Nevyhody:

- vznika chain pointerov ktory moze byt zbytocne dlhy a drahy na zdroje
- vsetky lokacie z ktorych sa servery presuvali musia udrziavat poiintery
- ak sa strati pointer v akejkolvek entite tak hladana entita sa stane nedostupnou

5.3.2.1 SSP CHAIN

Ked entita odide zo starej lokacie tak na tomto mieste sa nainstaluje „CLIENT STUB“ a v novej lokacii sa nainstaluje „Server Stub“

Migracia je pre klienta transapretna.

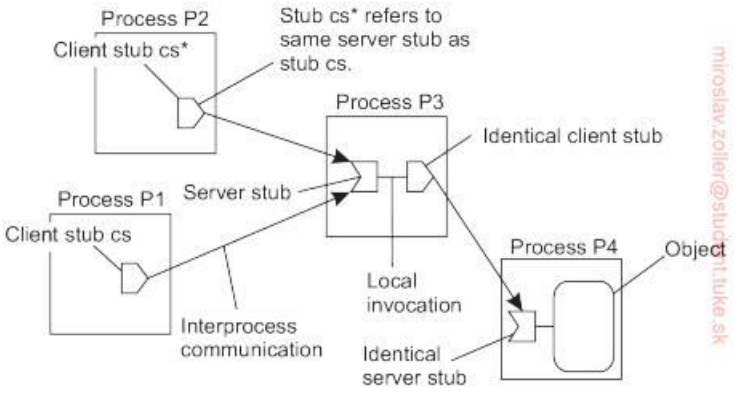


Figure 5.1: The principle of forwarding pointers using *(client stub, server stub)* pairs.

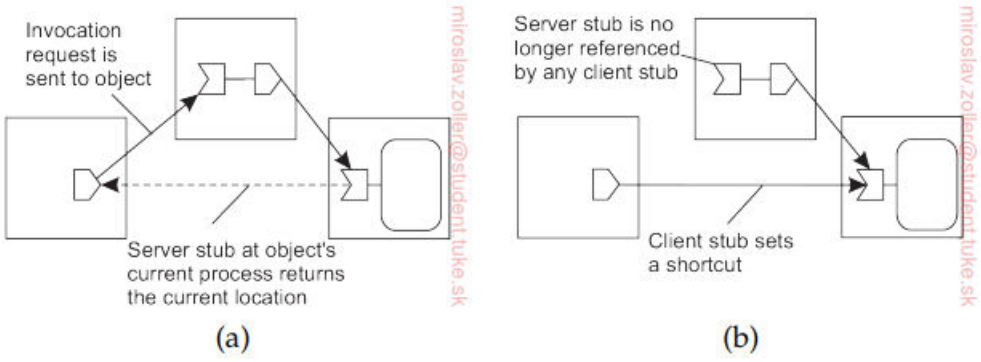
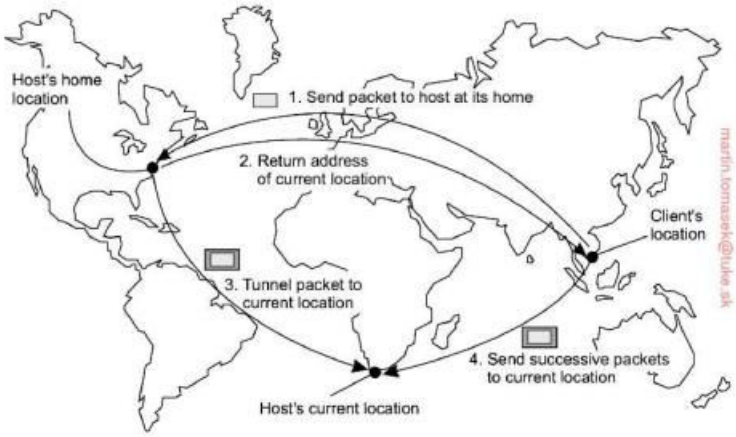


Figure 5.2: Redirecting a forwarding pointer by storing a shortcut in a client stub.

5.3.3 Home based approach

Nastavene na aktualnu lokaciu entity. V praxi, je to lokacia kde entita bola vytvorena

- Mobile IP



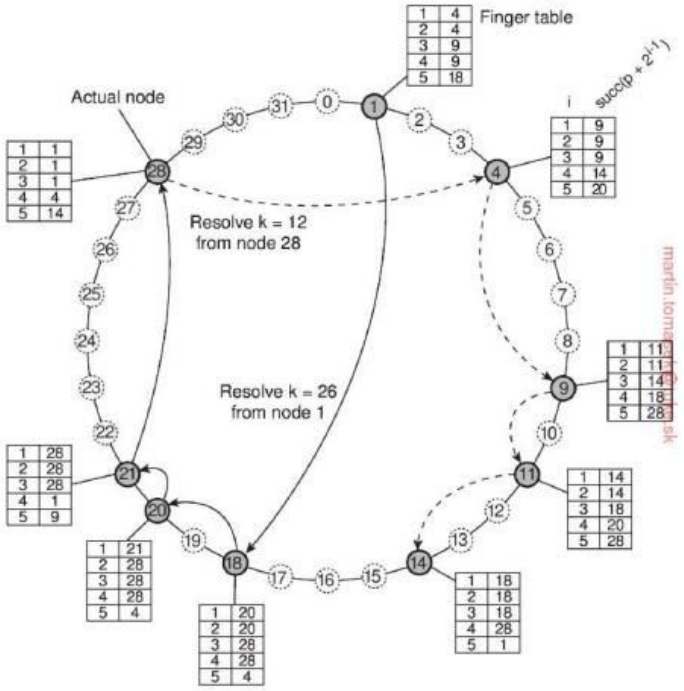
- Nevyhody :
  - Vysoka latencia
  - Domaca lokacia musi vzdy existovat

5.3.4 Distribuovane hashovacie tabulky

Posledne desatrocie bol pouzivany tento princip vyhladavanie hostov v sieti. Ku kazdemu peeru bol dosadeny kluc a

Distributed hash tables

- Chord system
- $FT_p[i] = succ(p + 2^{i-1})$
- $q = \begin{cases} FT_p[j] \leq k < FT_p[j+1] \\ FT_p[1] \text{ when } p < k < FT_p[1] \end{cases}$
- Lookup require $O(\log n)$ steps
- Exploiting network proximity
  - Topology-based assignment
  - Proximity routing
  - Proximity neighbor selection



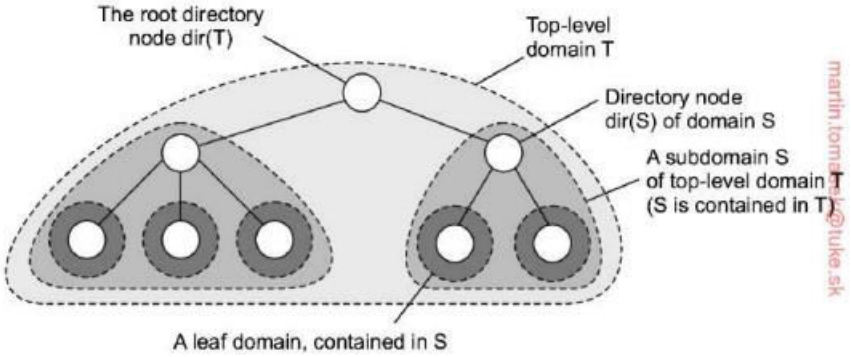
$$q = \mathsf{FT_p}[j] \leq k < \mathsf{FT_p}[j+1]$$

or $q = \mathsf{FT_p}[1]$ when $p < k < \mathsf{FT_p}[1]$. (For clarity, we ignore modulo arithmetic.) Note that when the finger-table size s is equal to 1, a Chord lookup corresponds to naively traversing the ring linearly as we just discussed.

To illustrate this lookup, consider resolving $k = 26$ from node 1 as shown in Figure 5.4. First, node 1 will look up $k = 26$ in its finger table to discover that this value is larger than $\mathsf{FT}_1[5]$, meaning that the request will be forwarded to node $18 = \mathsf{FT}_1[5]$. Node 18, in turn, will select node 20, as $\mathsf{FT}_{18}[2] \leq k < \mathsf{FT}_{18}[3]$. Finally, the request is forwarded from node 20 to node 21 and from there to node 28, which is responsible for $k = 26$. At that point, the address of node 28 is returned to node 1 and the key has been resolved. For similar reasons, when node 28 is requested to resolve the key $k = 12$, a request will be routed as shown by the dashed line in Figure 5.4. It can be shown that a lookup will generally require $\mathcal{O}(\log(N))$ steps, with N being the number of nodes in the system.

5.3.5 Hierarchicke postupy

Kolekcia domen rozdelenych do subdomen. Nachadza sa tu **ROOT DIRECTORY** node so vsetkymi lokaciami pod sebou ako listami stromu.



Co ak entita ma viacero adries ? Napr ze je replikovana ? Ak Entita ma adresu v listovych domenach D1 a D2 tak directory node najmensej domeny obsahuje pointer na obidve domeny ako na obrazku 5.7

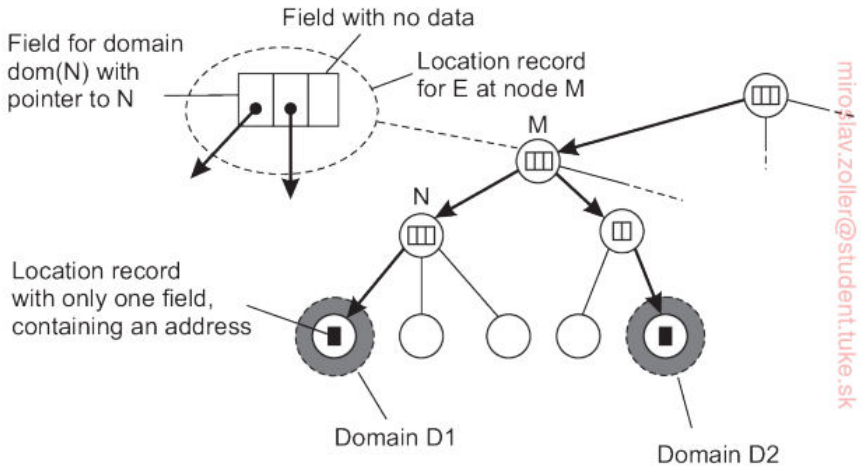
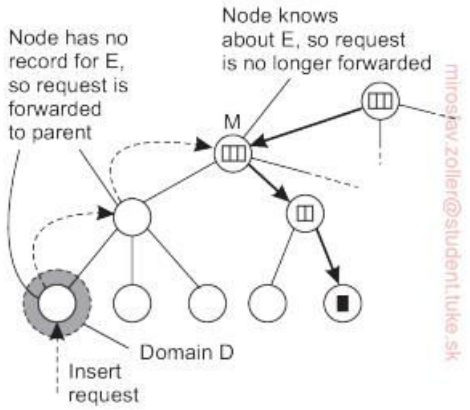
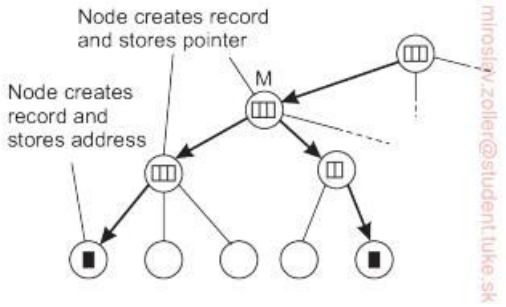


Figure 5.7: An example of storing information of an entity having two addresses in different leaf domains.

Teraz si predtavme ze **noda D** chce urobit lookup a **najst entitu E**, mozno sa bude nachadzat v replikovanej instancii? Na obrazku 5.8 je vidno ako **klient z D chce najst entitu E** a bude posielat lookup do directory node, ta noda to dalej posunie parentovi kym sa nenajde pointer kde by entita E bola. Ak sa taka domena najde v node ktora pozna E teda **M noda z obrazku** tak sa cez subdomeny dostane ku entite. Potom **M noda posunie pointer** najblizsej domene z ktorej bola poziadavka na zaciatku odoslana.



(a)



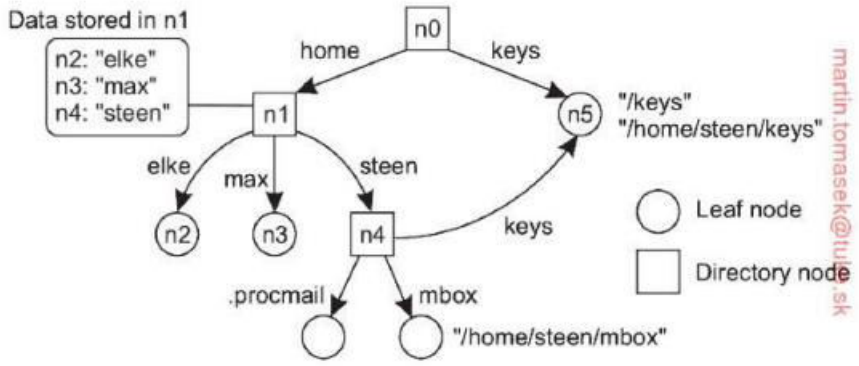
(b)

5.4 Štruktúrovane pomenovania

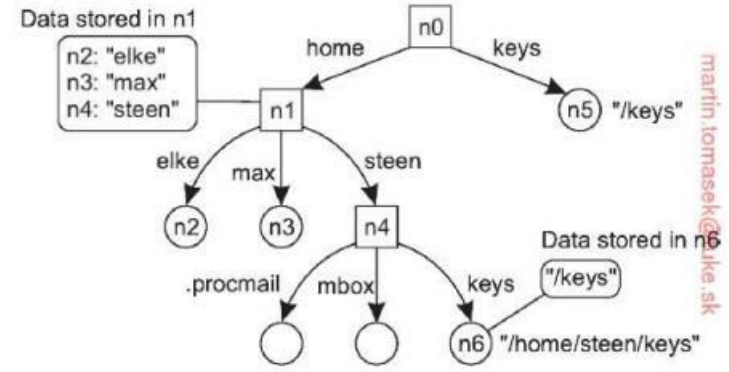
Mena sú organizované v **menových priestoroch (name space)**

Name space je organizovaný v **orientovanom grafe:**

- Listy su pomenovane entity
- Vnutorne nody su adresare
- Kazdy pozna **root node**
- **Mena su realne cesty (PATH) skrz menny system**



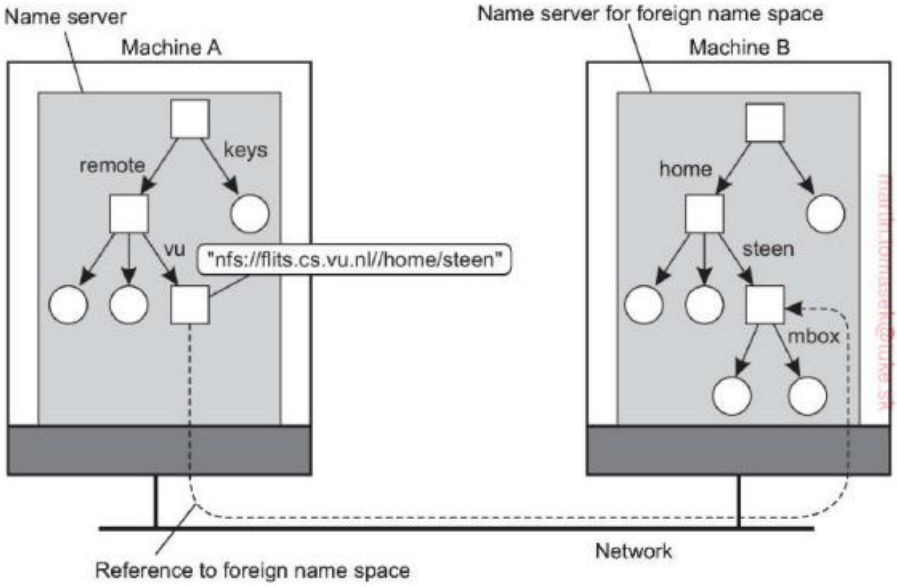
- Pozname inicialnu nodu a ako zacat s vyhladavanim v grafe
- Na **Linkovanie v mennom priestore pouzivame:**
  - ALIASY
  - HARD LINKY
  - SOFT LINKY
- Na **Linkovanie mimo menneho priestoru pouzivae:**
  - **Mount pointy –** noda ktora pouziva remote node identifier/ da sa cez siet pripojit inde
  - **Mounting pointy –** koren menneho priestoru na ktory sa chceme pripojit ()
  - **URL**



5.5 Remote name space (vzdialeny menny priestor)

Potrebujeme Mountnut vzdialeny file system

- V **nasom adresari potrebujeme nastavit nodu** ktora bude obsahovat **identifikator adresara vo vzdialenom mennom priestore.**
- Nasa noda sa nazyva **MOUNT POINT**
- Vzdialeny adresar na ktory sa pripajame sa nazyva **MOUTING POINT**

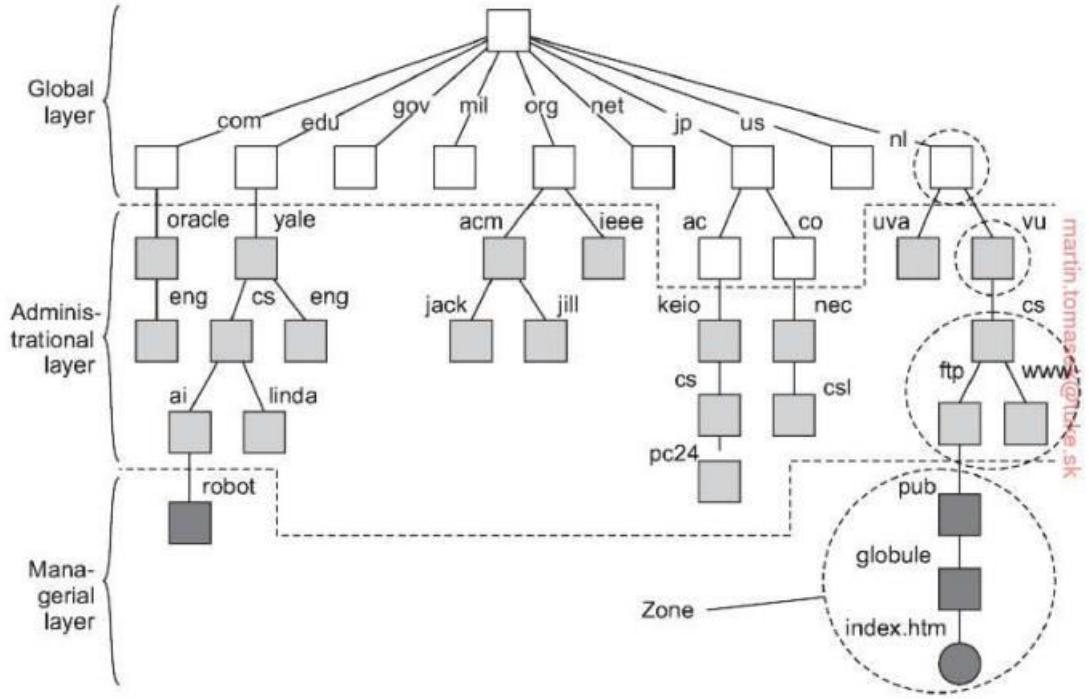


5.6 Distribuovanie menneho priestoru

- Menne priestory su rozlozene hierarchicky
- Menny priestor je distribuovany a ma **3 logicke vrstvy:**
    1. **Globalna vrstva** – najvyssi level, najmenej zmien
    2. **Administrativna vrstva –** kazda noda pre organizaciu napr. , obcasne zmeny
    3. **Manazerska vrstva -** najnizzsi level, reprezentuje hostov, adresare, subory, caste zmeny

5.7 DNS

Ako by take rozdelenie menneho priestoru podla 3 logickych vrsteiv vyzeralo v pripade DNS?



Comparison of name servers at different levels

	Global	Administrational	Managerial
• Geographical scale of network	Worldwide	Organization	Department
• Total number of nodes	Few	Many	Vast number
• Responsiveness to lookup	Seconds	Milliseconds	Immediate
• Update propagation	Lazy	Immediate	Immediate
• Number of replicas	Many	None or few	None
• Client-side caching	Yes	Yes	Sometimes

5.7.1 Implementacia ziskavania mien (name resolution)

- **Iterativne** – opakovanie kontaktuj hierarchiu nod aby ziskali casti mena
- **Rekurzivne** – kontaktuj lokalny menny priestor a nechaj poziadavku rekurzivne posielat dalej, **best option**

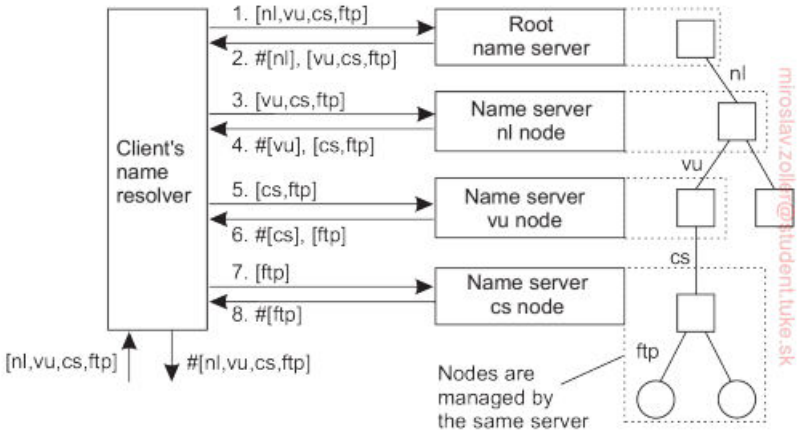


Figure 5.17: The principle of iterative name resolution.

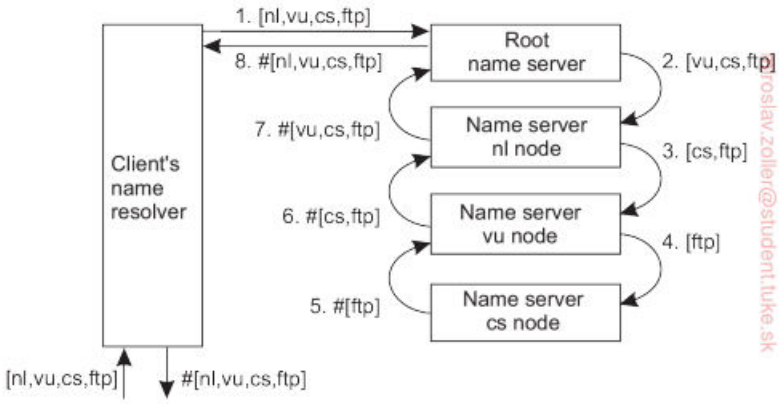


Figure 5.18: The principle of recursive name resolution.

5.8 Pomenovania zalozene na atributoch

- Adresare ukladaju (attribute,value) pary
- Pre rovnaky atribut moze byt viacero hodnot
- Moze byt kombinovana s hierarchickou strukturou
- Taktiez volane ako **sluzby adresarov (directory services)**

5.8.1 LDAP

https://cs.wikipedia.org/wiki/LDAP

https://www.youtube.com/watch?v=SK8Yw-CiRHk&ab_channel=EyeonTech

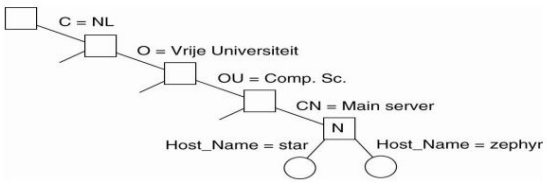
- Adresar na serveri ma u seba ulozenych kopu (attribute,value) pairs pre nejaku entitu
- Lookup sa moze vykonat cez meno/atribut/hodnotu

5.8.2 Priklad

LDAP entry in a directory information base (DIB)

Attribute	Abbr.	Value
Country	C	NL
Locality	L	Amsterdam
Organization	O	Vrije Universiteit
OrganizationalUnit	OU	Comp. Sc.
CommonName	CN	Main server
Mail_Servers	—	137.37.20.3, 130.37.24.6, 137.37.20.10
FTP_Server	—	130.37.20.20
WWW_Server	—	130.37.20.20

LDAP directory information tree (DIT)



Attribute	Value
Country	NL
Locality	Amsterdam
Organization	Vrije Universiteit
OrganizationalUnit	Comp. Sc.
CommonName	Main server
Host_Name	star
Host_Address	192.31.231.42

Attribute	Value
Country	NL
Locality	Amsterdam
Organization	Vrije Universiteit
OrganizationalUnit	Comp. Sc.
CommonName	Main server
Host_Name	zephyr
Host_Address	137.37.20.10

• `Host_Name` represents a relative distinguished name (RDN)

5.8.3 Implementacia LDAP

- Je podobna ako pre DNS ale robi sa **viac LOOKUP operacii**

- Strom je **distribuovany po viaceorch serveroch** ktore sa volaju ANO UHADLI STE **AGENTI**.. (directory service agent - DSA)
- **Klient pouziva agenta** cez ktoreho robi lookup (directory user agent DUA)

Pr. answer = search("&(C=NL)(O=Vrije Universiteit)(OU=*)(CN=Main server)")

Ldap sa casto pouziva v kombinacii s Active Directory.

6 7 Prednaska koordinacia

6.1 Problemy koordinacie

Uz len syncrhonizacia v jednom systeme je dost tazka pomocou Semaforov,Sprav a Monitorov. Synchronizacia medzi procesmi v distribuovanych systemoch je **omnoho tazsia.**

Prikladom moze byt NFS-ko kde je potrebne mat **lock manazera** pri pristupe do jedneho adresaru v tom istom case.

(NFS – **Network file system** je sieťový protokol zdieľania súborov, ktorý definuje spôsob, akým sa súbory ukladajú a získavajú z úložných zariadení v sieťach.)

6.2 Globalny cas

- Je **nemozne** aby fyzikalne hodiny bezali na rovnakej frekvencii.
- Hoci mame Universal Coordianted Time (UTC) – stale to treba synchronizovat na astronomicky cas

6.2.1 Network time protokol (NTP)

Bezny sposob ako ziskat aktualny cas je kontaktovat **TIME SERVER,** lenze hoc tento server poskytuje presny cas problem vznika uz len pri latencii a odoslani spravy spat klientovi.

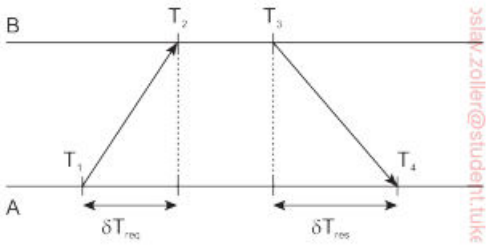


Figure 6.5: Getting the current time from a time server.

Cize ked **A** (klient) odosle poziadavku aby jej **B** (time server) poslalo aktualny cas, tak treba vypocitat . Tento model odhaduje ze **T2-T1 = T4-T3,** pretoze **response by mohol byt rovnaky** jak pri prijati tak pri odoslani

tzn. **A** by malo ziskat od **B** cas T3 + (T2-T1).

to iste co bolo v prezentacii len hnusne rozpisane:

$$\delta = \frac{(T_4 - T_1) - (T_3 - T_2)}{2}$$

- B's clock at T_4 reads approximately

$$T_B = T_3 + \delta = T_3 + \frac{(T_4 - T_1) - (T_3 - T_2)}{2} = \frac{(T_4 - T_1) + (T_2 + T_3)}{2}$$

6.2.1.1 Stratumy

Stratum 0 je server ktory sa nastavuje priamo z astronomickych hodin

Statum 1 je server ktory sa nastavuje podla server Stratum 0

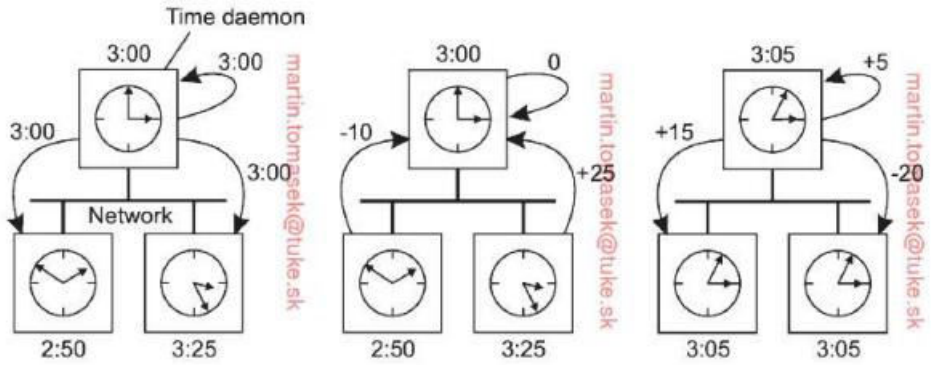
… etc …

Ak T_A je pomale prida sa nejaka konstanta

Ak je T_A rychle odoberie sa nejaka konstanta

6.2.2 Berkeleyho algoritums

Časový **Daemon sa opýta** všetkých zariadení **na ich casy**. Všetky zariadenia pošlú odpoveď. Daemon vypočíta **priemer** všetkých časov a **povie všetkým o koľko si majú zmeniť čas**. Vid obrazok:



6.2.3 Synchronizacia casu v bezdrotovych sietach

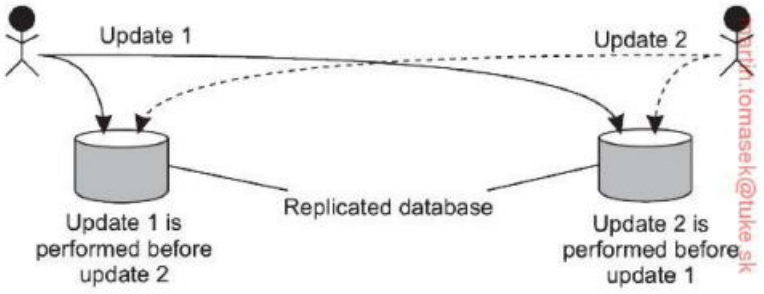
Ak sa pocita narocnost multi-hop operacii je narocne nastavit presny cas. Pouziva sa podobny algoritmus ako Berkeleyho, len to ze prijimatelia su synchronizovaní.

6.3 Problem casu

Čas **nie je spolahlivy sposob na urcenie synchronizacie**. Pouzivatelia mozu mat **zle nastaveny čas** dokonca aj **časové pásmo** spolu s neodhadnuteľným **oneskorením** internetu.

Example

- At midnight EDT (UTC –4), bank posts interest to your account based on current balance
- At 5:00 CET (UTC +1), you withdraw some cash
- Does interest get paid on the cash you just withdrew?
  - Depends upon which event came first!
- What if transactions made on different replicas?



6.4 Lamportove logicke hodinky

- Nie su to „hodinky“, len monotonne pocitadla
- A -> B : znamena ze A sa vykona pred B a potom sa vykona B
  - Napr. send(message) -> receive(message)
- **Tranzitivnost**
  - Ak A->B a B->C tak A->C

? Ako synchronizujem lokalne hodinky 3 roznych procesov ?

6.4.1 Lamportov algoritumus

Kazdy **proces P_i** udržiava vlastné **logické „hodiny“ - počítadlo C_i**

ODOSIELANIE

- **Time = time + 1**
- **TimeStamp = time;**
- **Send (message,timestamp)**

PRIJIMANIE

- **(message,timestamp) = receive()**
- **Time = max(timestamp – time) + 1**

(a) obrazok

Vsimnite si ze P3 ked posiela spravu P2 (m3) , tak cas je realne mensi u prijimatela ako odosielatela, preto:

(b) obrazok

P2 si nastavi casu na cas P3 + 1 a zmeni vsetky dalsie hodnoty casu vopred. Tzn [61+8,69+8,77+8]

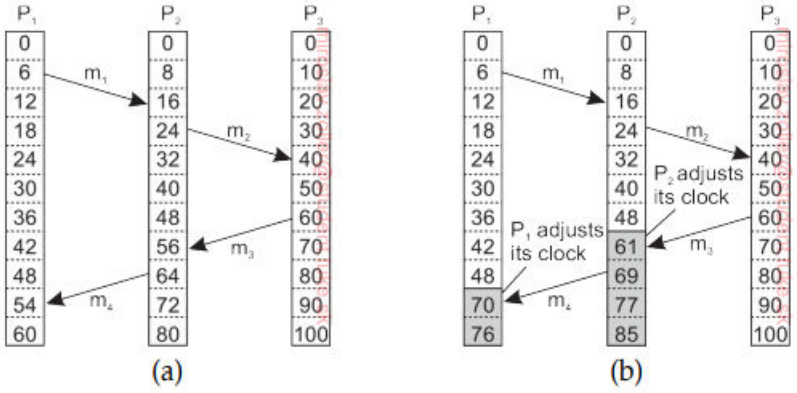
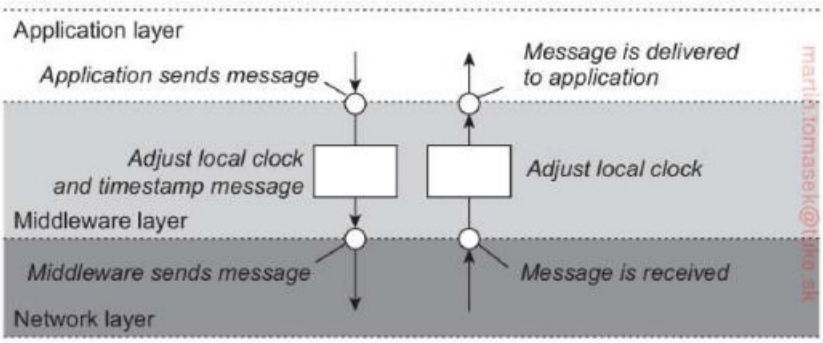


Figure 6.8: (a) Three processes, each with its own (logical) clock. The clocks run at different rates. (b) Lamport's algorithm corrects their values.

Lamport's logical clock as a part of middleware



6.4.2 Uplne usporiadany multicasting

Ako skupinka procesov komunikujem medzi sebou zapomoci lamportovych hodin?

1. Skupina procesov si multicastuje spravy medzi sebou. Sprava je oznackovana **lokalnym logickym casom.**
2. Prijimatel prebera spravu a ulozi si ju do lokalneho frontu zoradeneho podla casu
3. Prijimatel posiela ACK ostatnym procesom (Lamportovy algoritmus riesi synchronizaciu)
4. Vsetky procesy budu eventualne mat rovnaku kopiu frontu
5. Sprava sa popne z frontu a prebera
6. Vsetky procesy su syncrhonizovane a fronty rovnake

6.5 Kauzálny problém (Kauzálny znamená súvislý)

- Lamportove hodiny zoradia eventy:

Ak e -> e' tak C(e) < C(e')

- Na druhej strane pomocou lamportovych timestampov nevieme zaistiť, ktoré eventy sú kauzálne v relácii.

Ak C(e) < C(e') **nemusi znamenat implikaciu** e -> e'

Riešením sú **vektorové hodiny**.

6.6 Vektorové hodiny

https://en.wikipedia.org/wiki/Vector_clock

V podstate je to to iste ako lamprotove hodiny len spravy su v nejakom arrayliste a su syncrhonizovane pomocou pocitadla.

7 8 Prednáška – Coordination #2

7.1 Mutual Exclusion (Vzajomne vylucenie)

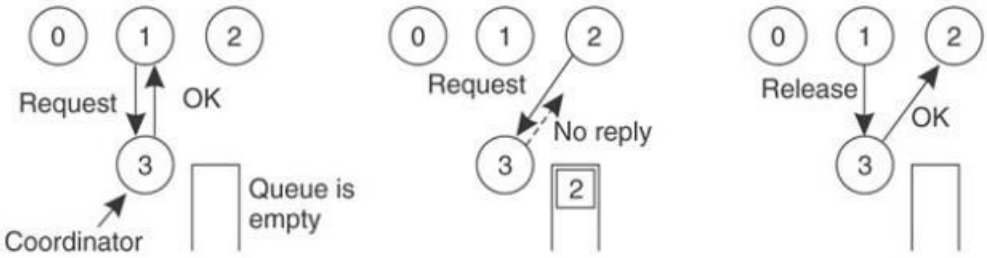
Mutual Exclusion je vlastnost synchronizacie procesov kde “ziadne 2 procesy nemozu vykonavat cast instrukcii kritickej sekcii v tom istom case”. Tzn treba zachovat syncrhonizaciu medzi menenim/mazanim zdielanych dat.

Pozname 2 pristupy **vzajomneho vylucenia**

7.2 Permission-based pristup

Proces potrebuje schvalenie od ineho procesu. Budu rozobrane sposoby nizzsie

7.2.1 Centralizovany sposob na zaklade povolenia (Centralized permission approach)



- Jeden process je **coordinator** pre zdroje
- Vsetky nody si pytaju od koordinatora **povolenie (permission)**
- Odpovede su: OK, Denied (retry) ,None (wait)
- V podstate tak jak sme robili centralizovany Lockserver

Vyhody :

- Vzajomne vylucenie je garantovane koordinatorom
- “Ferove” zdielanie pristupu **bez starvacie**
- Jednoducha implementacia

Nevyhody:

- Single point of failure (jeden bod zlyhania)
- Bottleneck pre performance

POINT OF FAILURE - Bod zlyhania je časť systému, ktorá, ak zlyhá, zastaví fungovanie celého systému.

7.2.2 Decentralizovany sposob povolenia

Mame v sieti **n** koordinatorov a opytame sa vsetkych, **ak vacsina suhlasi tak peer ziska pristup.**

Vyhody :

- Viacero "lockserverov" (nie je to jediny bod zlyhania)

Nevyhody:

- Velmi vela sprav medzi peermi
- Velmi nepriehladne to je

7.2.3 Distribuovane povolenia

- Pouzijeme **Lamportove logicke hodiny**.
- Ziadatel posle ziadost o povolenie **vsetkym** zariadeniam (vratane seba) v sieti a caka na OK response od kazdeho
- Odpoved na ziadost >

- If the receiver is not accessing the resource and does not want to access it, it sends back an OK message to the sender.
- If the receiver already has access to the resource, it simply does not reply. Instead, it queues the request.
- If the receiver wants to access the resource as well but has not yet done so, it compares the timestamp of the incoming message with the one contained in the message that it has sent everyone. The lowest one wins. If the incoming message has a lower timestamp, the receiver sends back an OK message. If its own message has a lower timestamp, the receiver queues the incoming request and sends nothing.

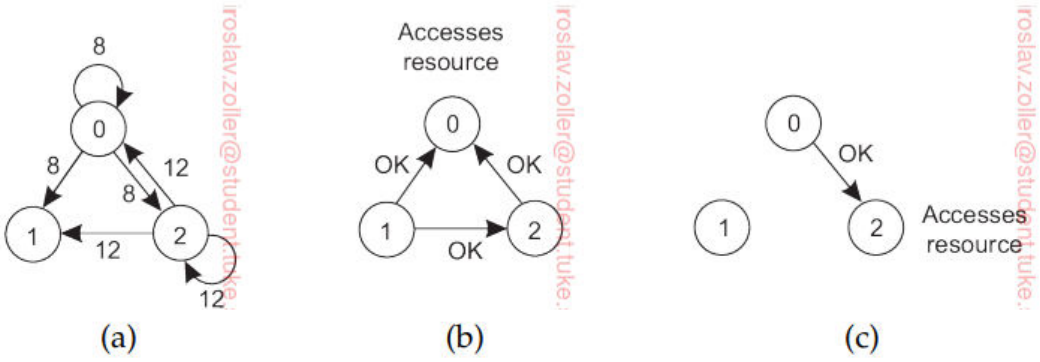


Figure 6.16: (a) Two processes want to access a shared resource at the same moment. (b) P_0 has the lowest timestamp, so it wins. (c) When process P_0 is done, it sends an OK also, so P_2 can now go ahead.

Vyhody:

- Ziadny centralny bottleneck
- Menej sprav ako v decentralizovanej sieti

Nevyhody

- N points of failure (n bodov zlyhania)
- Ak jeden bod zlyha tak sa lockne cely system

7.3 Pristup zalozeny na tokenoch

V sieti je jeden jediny TOKEN, a hocikto kto vlastní token smie pristupovat do zdielaneho zdroja.

Vyhody su ze vyhneme sa **DEADLOCKU** (kde procesy cakaju vzajomne na seba kym skoncia) **a STARVATION** (kazdy klient eventualne ziska pristup k zdroju).

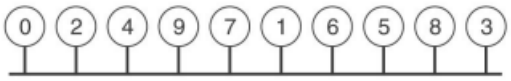
Najvacsou nevyhodou je ze ak sa token strati (vypadkom nejakeho servera) tak sa potrebuje vytvorit novy token tak aby ostal vzdy ako jediny.

7.3.1 Logical Ring

- Kazdy process **pozna svojho nastupcu (successor)**
- **Token** je posuvany v ringu (v tom kruhu peerov podla obrazku nizsie)
- Ked process chce robit zmeny v zdielanom subore **potrebuje token** a **ponecha si ho kym svoju pracu nedokonci**
- Ak je Node mrtva, tak sa prejde na successora mrtveho procesu.

A

A.



B.

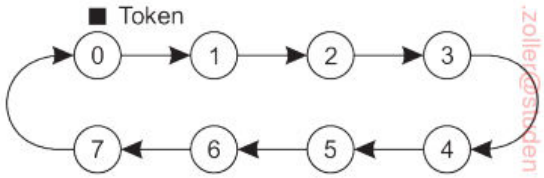


Figure 6.17: An overlay network constructed as a logical ring with a token circulating between its members.

Vyhody:

- **Spravodlivost,** ziadna starvacia (vyhladovanie)
- Jednoducha recovery ak sa node **Bez tokenu** odpoji

Nevyhody:

- **Pad procesu ktory vlastni token**
- Tazke najst strateny token popripade vytvorit novy aby bol iba 1 token

7.4 Election algoritmy (Volebné algoritmy)

- Vsetky procesy sa zhodnu na jednom **koordinatorovi**
- Ak aktualny koordinator sa **odpoji** tak sa zvoli **novy koordinator**
- Ak sa stary koordinator vrati do siete, pomocou novych volieb sa **moze stat koordinatorom znova**

7.4.1 Bully Algoritmus

Pre celu siet predpokladame ze:

- Vsetky procesy sa medzi sebou poznaju
- Kazdy proces je ocislovany **UNIKATNOU hodnotou**
- Nepoznaju navzajom svoje stavy

Predpokladajme ze **Proces P** zisti ze neexistuje koordinator

- Posle **election message** vsetkym vyssie ocislovanym procesom
- Ak ziadny vyssi proces sa neozve tak P sa stane koordinatorom
- Ak sa ozve vyssi proces, tak P sa vzdá

Predpokladajme ze **Proces Q** dostane **election message (volebnu spravu)**

- Odpoveda OK odosielatelovi, stym ze prebera proces elekcie
- Zacne rozposielat **nove election message** procesom s vyssimi hodnotami

Tento proces sa opakuje **kym neostane posledny proces s najvyssim cislom**. A ten **informuje vsetky procesy** o **vyhre**.

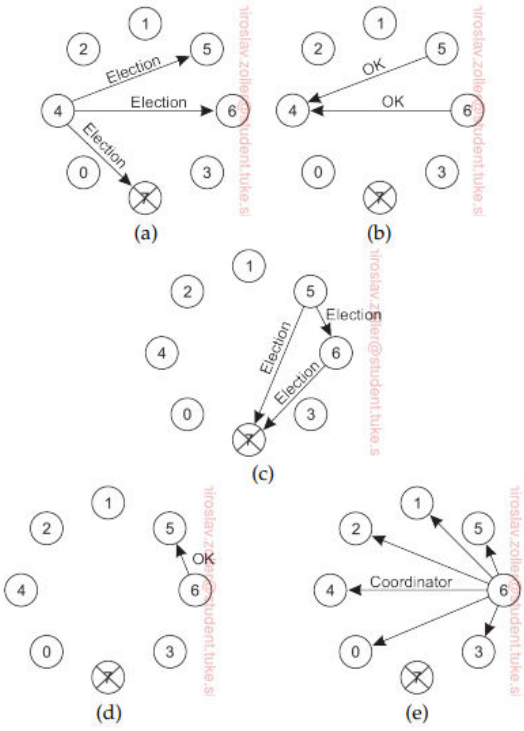


Figure 6.20: The bully election algorithm. (a) Process 4 holds an election. (b) Processes 5 and 6 respond, telling 4 to stop. (c) Now 5 and 6 each hold an election. (d) Process 6 tells 5 to stop. (e) Process 6 wins and tells everyone.

7.4.2 Ring algoritmus (kruhovy algoritmus)

Vsetky procesy su zoskupene v kruhovej topologii (ringu) ale tentokrat bez tokenu. Procesy taktiez vlastnia unikatne cislo.

Predpokladajme ze **Proces P** zisti ze neexistuje koordinator

- P odosle **election message** nasledujucemu procesu spolu s **vlastnym cislom**
- (Samozrejme ak nasledujuci proces nefunguje tak ho preskoci)

Predpokladajme ze **Proces Q** dostane election message

- Prida vlastne procesne cislo (tu unikatnu hodnotu)

Cely proces sa opakuje dookola kym proces ktory zacal inicializaciu neziska konecne data. Teda:

Predpokladajme ze **Proces P** dostane **election message** s jeho vlastnym procesnym cislom v tele

- Tak odosle znova spravu do kruhu s typom spravy **kto jen novy koordinator (najvyssie cislo)**
- Vsetky procesy si ulozia noveho koordinatora

□

Ak by sa odoslalo viacero sprav naraz, eventualne by bol vysledok rovanky.

Ak sa **pripoji novy proces** zacne sa **nova elekcia.**

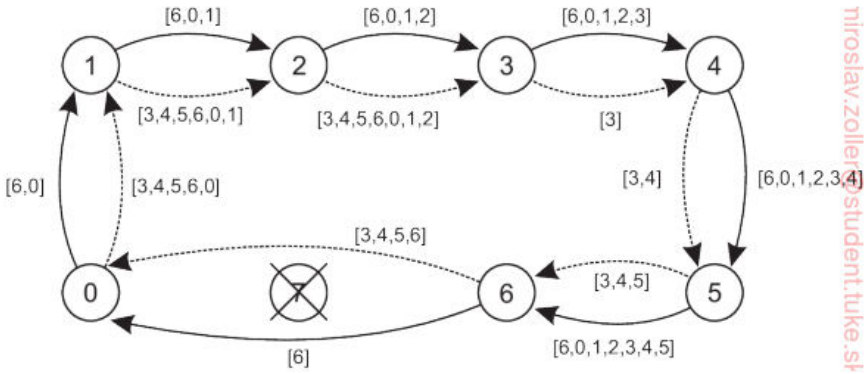


Figure 6.21: Election algorithm using a ring. The solid line shows the election messages initiated by P_6; the dashed one those by P_3.

7.5 Elekcie v bezdrotovych prostrediach

- Sú nespolahlivé a procesy sa možu prenášať (napr. mobilne zariadenia v sieti)
- Topologia sa konštante mení

7.5.1 Algoritmus elekcie v bezdrotovej sieti:

1. Hocijaka noda odosle **election message** svojim **susedom.**
2. Ked noda ziska **election message** prvykrat tak ju **posunie susedom** a **nastavi odosielatela ako parenta**
3. Pocka na odpovede susedov kym nevznikne **kostra grafu**
4. Ak noda dostane druhykrat **election message** tak len posle OK odpoved.
5. Nakoniec sa rozdristribuuje kto je **zvolený koordinator,** tym ze **kazda noda posiela svojemu parentovi len to najvyssie cislo** vid obrazok f.

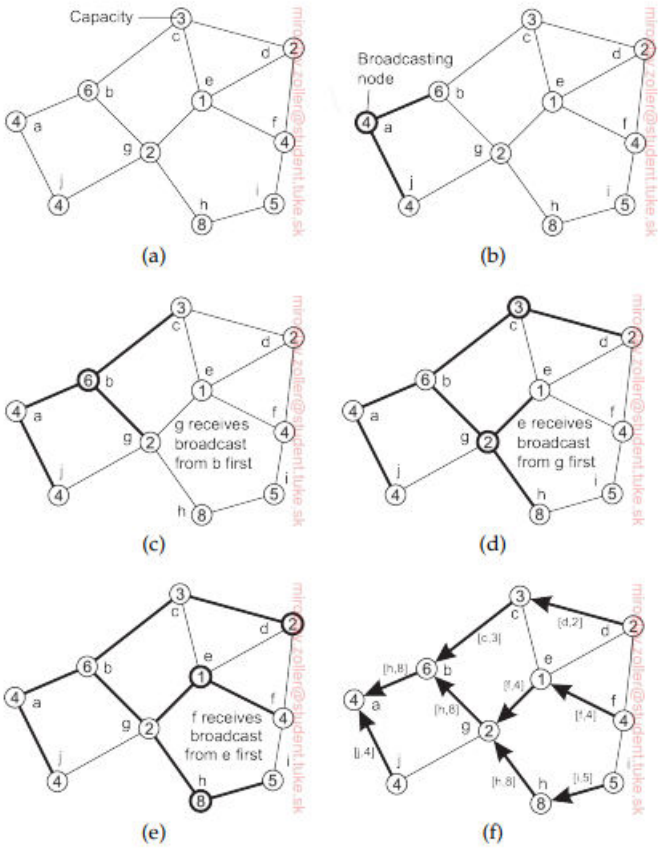


Figure 6.22: Election algorithm in a wireless network, with node a as the source. (a) Initial network. (b)–(e) The build-tree phase (last broadcast step by nodes f and i not shown). (f) Reporting of best node to source.

7.6 Elekcie v rozsiahlych systémoch

- Niekedy sa zvoli **viac nez jeden coordinator**
- Nody su organizovane ako **peers** a **superpeers**
- **V kazdej** skupinke peerov ktoru riadi superpeer sa vykonavaju elekcie
- **Medzi superpeermi** sa tiez vykonvaju elekcie

8 Prednaska 9 Konzistenstnost a Replikácia (Consistency and Replication)

Konzistentnost - vlastnosť distribuovaného systému, ktorá zabezpečuje, že každý uzol alebo replika má rovnaký pohľad na dáta v danom čase, bez ohľadu na to, ktorý klient dáta aktualizoval. Poskytuje **kvalitu jednostnosti.**

Kauzalnost – príčinnosť, vzťah medzi príčinou a dôsledkom (z minulej prednasky pri lamportovych hodinach)

Koherentnosť – súvisly, neprerušeny a ako všetko v distribuovanom systeme do seba zapada.

8.1 Dôvody replikácie

Co replikujeme v distribuovanom systeme?

- Data
- Servery

Preco replikujeme veci?

- Zvysujeme **vykon**
- Zvysujeme **spolahlivost**

Co je **hlavnym problemom** pri udrziavani repliacie?

- Udrziavat **konzistenciu** replik

8.2 Výkon a škálovateľnosť

Vo všeobecnosti musíme zabezpečiť, aby sa všetky konfliktné operácie vykonávali všade v rovnakom poradí.

Konfliktne operácie sú:

- **Read-write konflikt** – citanie a zapisovanie sa vykonavaju **súbežne**
- **Write-write konflikt** – 2 **súbežné** zápisy

Globálne zoradenie (global ordering) na konfliktne operacie napr. Pomocou lamportovych hodin moze byt draha operacia, ktora znizuje škalovatelnosť. Takze mame dilemu na 2 rozhodnutia

1. **Na jednej strane je možné problémy so škálovateľnosťou zmierniť použitím replikácie a ukladania do vyrovnávacej pamäte, čo vedie k zlepšeniu výkonu**
2. **Na druhej strane je vo všeobecnosti potrebné zachovať konzistentnosť všetkých kópií globálnou synchronizáciou, ktorá je vo svojej podstate nákladná z hľadiska na výkon.**

Spravnym riesenim je **oslabit konzistencne poziadavky,** aby sme sa globalnej syncrhonizacii vyhli.

8.3 Oslabenie konzistencnych poziadaviek

Co znamena „oslabit konzistenčne poziadavky“ ?:

- Zjemnit poziadavku, ze vsetky zmeny potrebuju byt vykonavane ako atomicke operacie
- Nepouzivat **ziadnu globalnu synchronizaciu**
- Kopie nemusia byt vzdy vsade rovnake

Do akej miery sa dá oslabiť konzistentnosť?

- Závisi od mnozstva updatov na replikovane data, nejake zname patterny napr.
- Závisi na pouzitelnosti replikovanych dat (resp. ako ich apklikacia pouziva)

8.4 Data-centricke konzistenčne modely

Dohoda medzi (distribuovaným) úložiskom údajov a procesmi, v ktorom úložisko údajov **presne** špecifikuje, aké sú výsledky operácií čítania a zápisu v prítomnosti súbežnosti. **Čím je slabši model konzistencie, tym je jednoduchsie vytvoriť škálovateľné riešenie.**

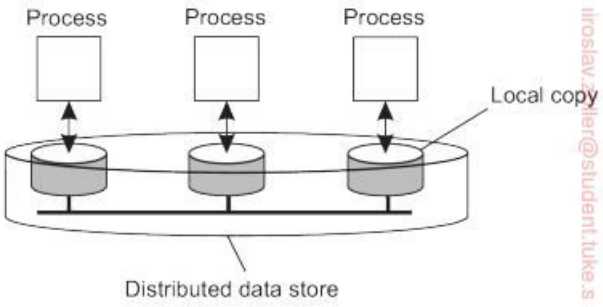


Figure 7.1: The general organization of a logical data store, physically distributed and replicated across multiple processes.

Modely silnej konzistencie – operacie na zdielanych datach **su synchronizovane** (modely nepoužívajúce synchronizačné operácie, ale konzistentné poradie operácií). Sem patria:

- Strict consistency (silna konzistencia)
- Sequential consistency (sekvencna konzistencia)
- Causal consistency (kauzalna konzistencia)

Modely slabej konzistencie - synchronizácia prebieha iba vtedy, keď sú zdieľané údaje uzamykané a odomykané (LOCK & ACQUIRE) (modely so **synchronizačnými operáciami**). Sem patri:

- Entry consistency (Vstupna konzistencia)

8.4.1 Strict consitency

Kazda READ operacia dat vrati vysledok poslednej WRITE operacie.

- Vďaka prísnej konzistencii sú všetky zápisy okamžite viditeľné pre všetky procesy a v distribuovanom systéme sa zachováva **globalna synchronizacia**.

Priklad: P1 zapisuje (W) data **a** do premennej **x.** P2 to neskor cita (R) a ziskava data **a**. Predpokladajme ze tato horizontalna ciara urcuje cas.

- Strictly consistent data store (a)
- Data store that is not strictly consistent (b)

V striktnej konzistencii v priklade **b** sa takato situacia nesmie stat to uz je **sekvencna konzistencia, nie striktna.**

8.4.2 Sequential consistency

Výsledok akéhokoľvek vykonávania **je rovnaký**, ako keby operácie (čítanie a zápis) všetkými procesmi v dátovom úložisku boli vykonávané v určitom sekvenčnom poradí a operácie každého jednotlivého procesu sa objavujú v tomto poradí v poradí určenom jeho programom. Tzn. všetky procesy vidia **rovnaké prelynanie** operácií.

Spravenie sa dvoch procesov ktore pracuju na rovnakej datovej položke:

Figure 7.4: Behavior of two processes operating on the same data item. The horizontal axis is time.

POZNÁMKA: Rozšírenie aktualizácie x na P2 nejaký čas trvalo, čo je úplne prijateľné.

Všimnite si, že sa **nič nehovorí o čase**; to znamená, že **neexistuje žiadny odkaz na „najnovšiu“ Write operáciu** na dátovú položku. Proces tiež „vidí“ zápisy zo všetkých procesov, ale iba prostredníctvom svojich **Read** operaci. Radšej tu pacnem vysvetlenie z knihy:

That time does not play a role can be seen from Figure 7.5. Consider four processes operating on the same data item x. In Figure 7.5(a) process P_1 first performs $W_1(x)a$ on x. Later (in absolute time), process P_2 also performs a write operation $W_2(x)b$, by setting the value of x to b. However, both processes P_3 and P_4 *first* read value b, and *later* value a. In other words, the write operation $W_2(x)b$ of process P_2 appears to have taken place before $W_1(x)a$ of P_1.

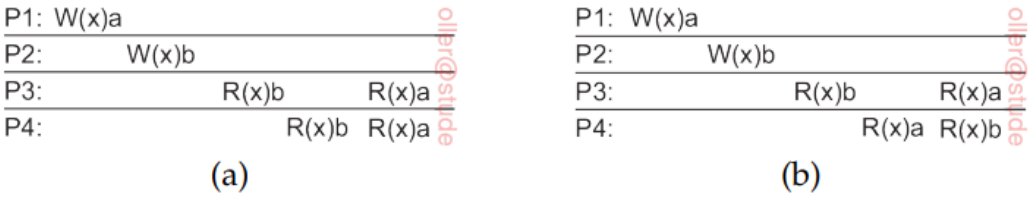


Figure 7.5: (a) A sequentially consistent data store. (b) A data store that is not sequentially consistent.

In contrast, Figure 7.5(b) violates sequential consistency because not all processes see the same interleaving of write operations. In particular, to process P_3, it appears as if the data item has first been changed to b, and later to a. On the other hand, P_4 will conclude that the final value is b.

Process P_1	Process P_2	Process P_3
`x ← 1;`	`y ← 1;`	`z ← 1;`
`print(y,z);`	`print(x,z);`	`print(x,y);`

Figure 7.6: Three concurrently executing processes.

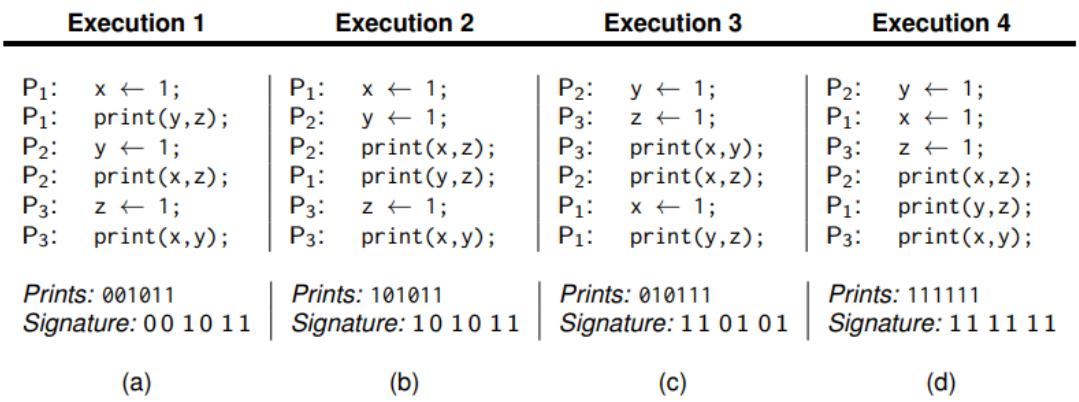


Figure 7.7: Four valid execution sequences for the processes of Figure 7.6. The vertical axis is time.

Signatura je vystup Procesov v poradi P1,P2,P3
Prints: su vypisy v ramci sekvencnej postupnosti vykonania

Signatura ako napriklad 00 00 00 , 00 10 01 nie su validne.

8.4.3 Kauzalna konzistencia

Kauzalny model konzistencie reprezentuje **oslabenie (weakening) sekvencnej konzistencie.**

Zápisy, ktoré sú potenciálne kauzálne súvisiace, musia vidieť všetky procesy v rovnakom poradí. Súbežné zápisy môžu byť na rôznych počítačoch viditeľné v rôznom poradí.

As an example of causal consistency, consider Figure 7.10. Here we have an event sequence that is allowed with a causally consistent store, but which is forbidden with a sequentially consistent store or a strictly consistent store. The thing to note is that the writes $W_2(x)b$ and $W_1(x)c$ are concurrent, so it is not required that all processes see them in the same order.

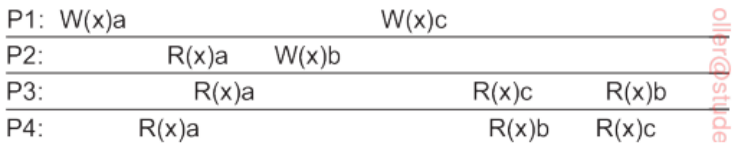


Figure 7.10: This sequence is allowed with a causally-consistent store, but not with a sequentially consistent store.

Now consider a second example. In Figure 7.11(a) we have $W_2(x)b$ potentially depending on $W_1(x)a$ because writing the value b into x may be a result of a computation involving the previously read value by $R_2(x)a$. The two writes are causally related, so all processes must see them in the same order. Therefore, Figure 7.11(a) is incorrect. On the other hand, in Figure 7.11(b) the read has been removed, so $W_1(x)a$ and $W_2(x)b$ are now concurrent writes. A causally consistent store does not require concurrent writes to be globally ordered, so Figure 7.11(b) is correct. Note that Figure 7.11(b) reflects a situation that would not be acceptable for a sequentially consistent store.

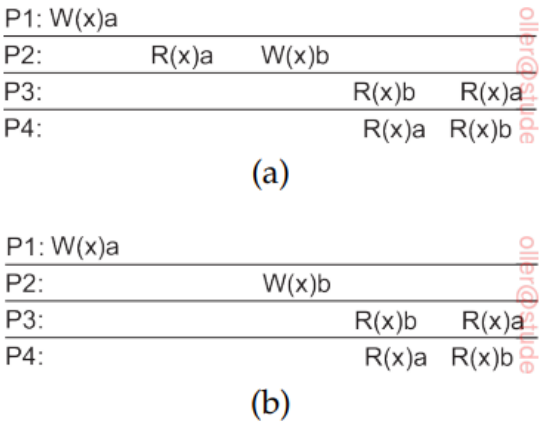


Figure 7.11: (a) A violation of a causally-consistent store. (b) A correct sequence of events in a causally-consistent store.

Ako som to pochopil ja:

Pre pochopenie v (a) je W1(x)a a W2(x)b kauzalne zavisle pretoze pred b-ckom je R2(x)a. Tzn. Tieto 2 operacie by sa mali vykonat sekvencne a preto P3 a P4 **by mali mat vo svojich READOCH len vysledok b.**

Na druhu stranu v (b) W1(x)a a W2(x)b nie su kauzalne zavisle na sebe tzn. Vykonavaju sa **konkurentne.** Preto sa ready v case striedaju.

8.4.4 Zoskupenie operacii

Mnoho modelov konzistencie je definovaných na úrovni základných operácií čítania a zápisu. Táto úroveň granularity je z historických dôvodov: tieto modely boli pôvodne vyvinuté pre multiprocesorové systémy so zdieľanou pamäťou av skutočnosti boli implementované na hardvérovej úrovni.

Na spravne zoskupenie operacii potreubujeme **Zdieľané synchronizačné premenné** alebo v jednoduchosti **zámky (locks).** Takze tak ako sme mali zadanie acquire/release pre urcity blok dat (kritickej sekcii).

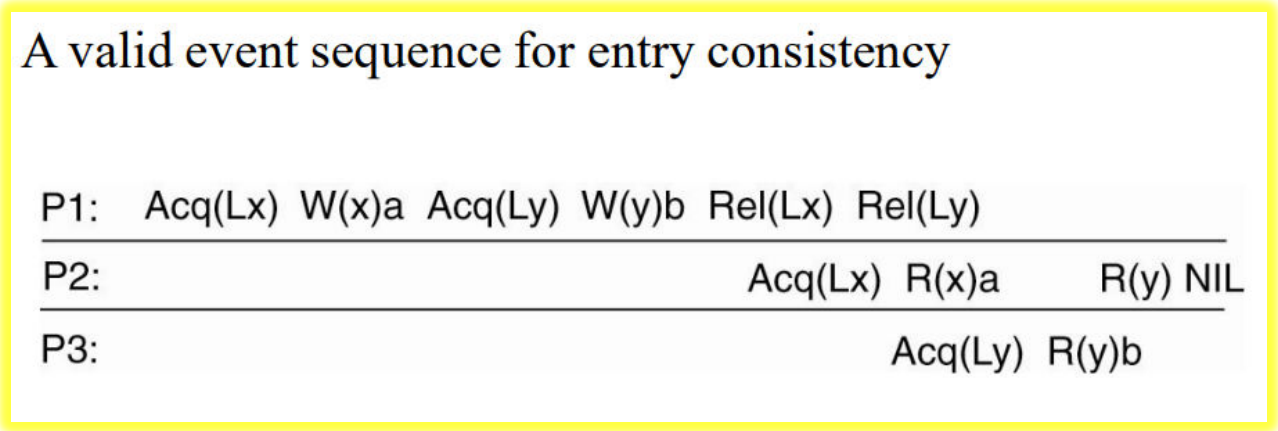
Exclusive access (exkluzivny pristup) – mozeme robit read/write operacie

Nonexclusive access (neexkluzivny pristup) – mozeme robit **len read operacie**

8.4.5 Nevyhnutné kritériá pre správnu synchronizáciu pomocou lockov:

- Ziskat acquire na lock moze byt upsesny len ak vsetky updaty, pre ktore ten lock je rezerovany sa dokoncia
- **Exkluzivny pristup** k locku ziskame len ak tento **lock nik nepouziva**
- **Neexkluzivny pristup** k locku ziskame iba ak lock nedrzi iny proces v **exkluzivnom rezime**

8.4.6 Entry consistency



8.5 Consistency vs. coherence

Model konzistencie rieši:

- Čo možno očakávať v súvislosti so **súborom údajov**, keď na údajoch súčasne funguje viacero procesov
- Súbor údajov je konzistentný, ak je v súlade s pravidlami modelu

Model koherencie rieši:

- Čo možno očakávať, že bude platiť len pre **jeden údaj**
- Sekvencna konzistencia
- V prípade súbežných zápisov všetky procesy nakoniec uvidia rovnaké poradie aktualizácií

Eventualna konzistencia

- Súbežnosť v obmedzenej forme (veľa operácií čítania a takmer vôbec zápisu)
- Ako rýchlo by sa mali vykonávať aktualizácie replík? Najmä keď väčšina procesov používa rovnakú repliku
- Ak sa dlhší čas neuskutočnia žiadne aktualizácie, všetky repliky sa postupne stanú konzistentnými (majú rovnaké údaje)

8.6 Client-centric consistency model

Problem konzistentneho datoveho modelu**:** Eventualne konzistentné dátové modely vo všeobecnosti fungujú dobre, pokiaľ klienti vždy pristupujú k rovnakej replike. Problémy však nastávajú, keď v priebehu krátkeho času pristupujú k rôznym replikám. Najlepšie to ilustruje pohľad mobilného užívateľa na prístup k distribuovanej databáze, ako je znázornené nizsie:

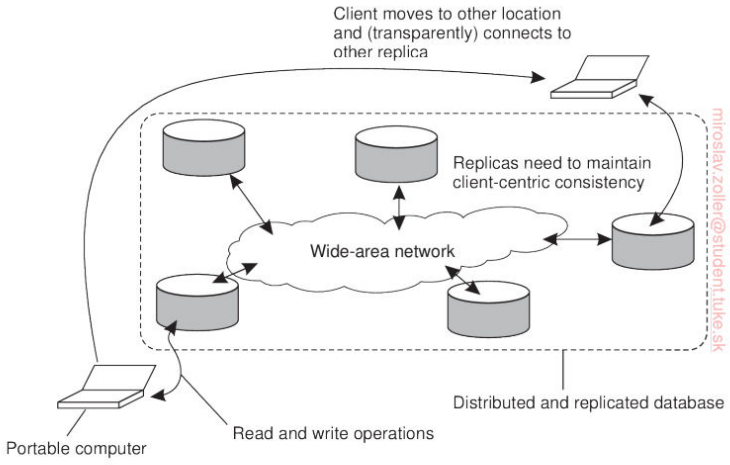


Figure 7.15: The principle of a mobile user accessing different replicas of a distributed database.

Problem, kde sa klient pripojil na inu repliku, ktora mozno este nema replikovane data z predchadzajucich zmien sa da riešiť **klient-centrickou konzistenciou.**

Model konzistencie orientovaný na klienta (client-centric consistency model) definuje, ako úložisko údajov prezentuje hodnotu údajov **jednotlivému klientovi**, keď klientsky proces pristupuje k hodnote údajov cez rôzne repliky.

- Dátové úložiská s nedostatkom simultánnych aktualizácií (jednoduché riešenia konfliktov)
- Modely slabej konzistencie, ako je **eventualna konzistencia**, zvyčajne postačujú
- Konzistentnosť poskytuje istotu pre **jedného používateľa**

8.7 Client consistency guarantees

Konzistencia zameraná na klienta poskytuje záruky pre jedného klienta za **jeho prístupy k úložisku údajov**. Zamerame sa na 4 typy client-centrickych konzistencnych modelov.

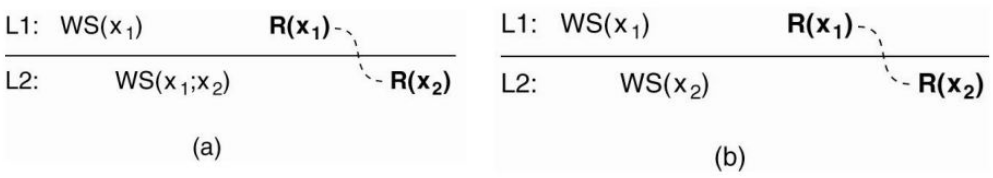
1. Monotonic Reads
2. Monotinic Writes
3. Read your Writes
4. Write Follow Reads

8.7.1 Monotonic reads

Ak proces načíta hodnotu údajovej položky x, každá následná operácia čítania na x týmto procesom vždy vráti rovnakú hodnotu alebo novšiu hodnotu.

Inými slovami, konzistencia monotónneho čítania zaručuje, že keď proces raz uvidí hodnotu x, už nikdy neuvidí staršiu verziu x.

- A monotonic-read consistent data store (a)
- A data store that does not provide monotonic reads (b)

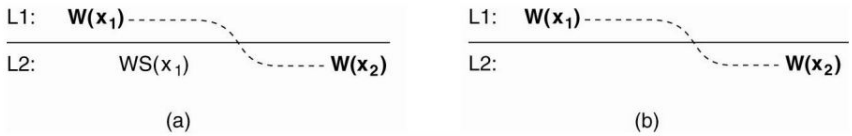


Preco (b) nie je monotonna read operacia ? **Pretoze potrebuje mat celu historiu zapisov ako v predoslom priklade a.. WS(x1;x2) najpr sa ulozi hodnota x na verziu 1 potom na verziu 2 a az potom pokracujeme na read.** To zabezpeci tu garanciu citania.

8.7.2 Monotonic writes

Operácia zápisu procesom na dátovú položku x je dokončená pred akoukoľvek následnou operáciou zápisu na x tým istým procesom

- A monotonic-write consistent data store (a)
- A data store that does not provide monotonic-write consistency (b)

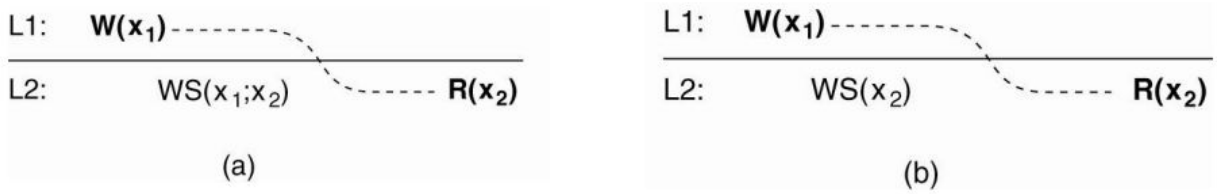


Same shit, potrebujeme mat garanciu ako v (a) ze v L2 bol WS(x1) aby mohla vzniknut nova prepisana verzia. Takze mame ukladane verzie za sebou. Dobry prikladom su replikovane subory, ktore su oznacene svojou verziou.

8.7.3 Read your writes

Účinok operácie zápisu procesom na údajovú položku x sa vždy prejaví následnou operáciou čítania na x tým istým procesom

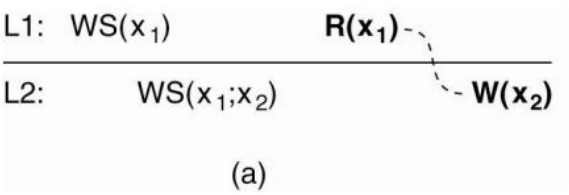
A data store that provides read-your-writes consistency (a)
A data store that does not (b)



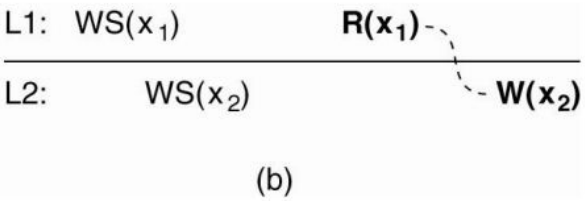
8.7.4 Writes follow reads

Operácia zápisu procesom na dátovú položku x, ktorá nasleduje po predchádzajúcej operácii čítania na x tým istým procesom, je zaručená, že sa uskutoční na rovnakej alebo novšej hodnote x, ktorá bola načítaná.

- A writes-follow-reads consistent data store (a)
- A data store that does not provide writes-follow-reads consistency (b)



(a)



(b)

8.8 Manazment replik

Kde by sme umiestnili repliky a aky mechanizmus sa pouzije na konzistenciu?

- Repliky ukladame podla roznych heuristickych strategii

Content replication and placement

When it comes to content replication and placement, three different types of replicas can be distinguished logically organized as shown in Figure 7.21.

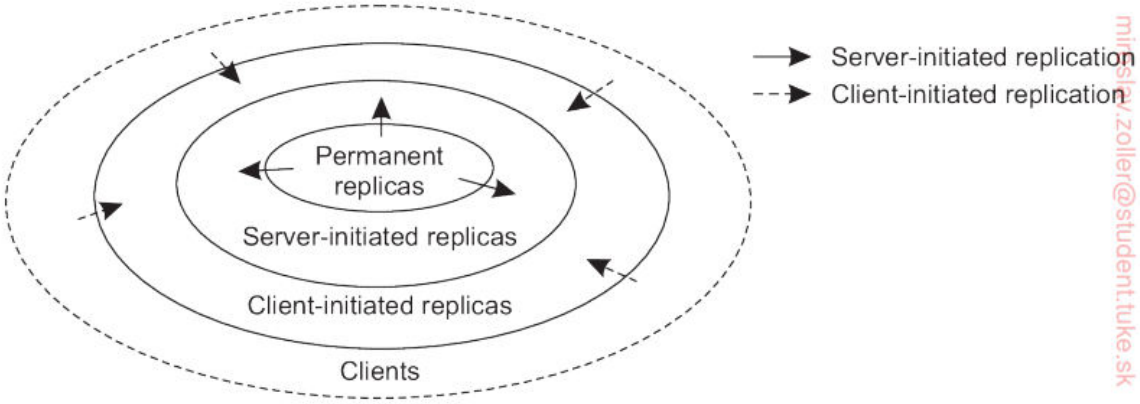


Figure 7.21: The logical organization of different kinds of copies of a data store into three concentric rings.

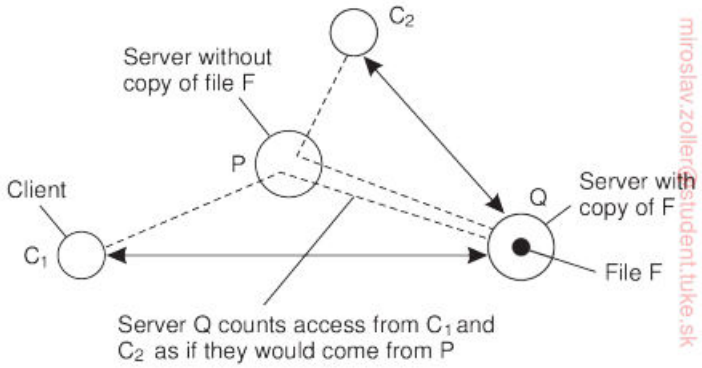
8.8.1 Permanentne repliky

- Statická organizácia distribuovaných serverov
- Servery su replikovane na jednej lokacii
- Mirrored repliky

8.8.2 Serverove repliky

Počítanie žiadostí o prístup od rôznych klientov.

Serverom iniciované repliky sú kópie dátového uloziska, ktoré existujú na zvýšenie výkonu a sú vytvorené z iniciatívy (vlastníka) dátového uloziska. Predstavte si napríklad webový server umiestnený v New Yorku. Za normálnych okolností tento server dokáže spracovať prichádzajúce požiadavky pomerne ľahko, ale môže sa stať, že v priebehu niekoľkých dní príde náhly naval požiadaviek z neočakávaného miesta ďaleko od servera. V takom prípade môže byť užitočné nainštalovať niekoľko dočasných replík v oblastiach, z ktorých prichádzajú požiadavky.



8.8.3 Klientske repliky

- klient cache
- Vyuzivaju sa len na performance pre pristup k datam

8.9 Content distribution

Pushing updates: na servery sa odosiela update, bez hociakeho opytania sa od servera

Pulling updates: klientsky pristup – klient requestuje server aby ho updatol

8.10 Ako implementovat klient-centricku konzistentu siet

1. Invocation service
    a. Build middlewaru na spracovanie poziadaviek
2. Replication manager
    a. Pozna vsetky repliky
    b. Pozna rolu kazdej repliky (primary,backup)
    c. Moze zmenit rolu repliky
3. Protokol replikacie
    a. Logika repliakcie
    b. Fault tolerance, koherencia replik, rekonfiguracia
4. Monitorovacia sluzba
    a. Detekcia vypadkov peerov
    b. Spusta rekonfiguraciu
5. Multicastovacia sluzba
    a. Posiela spravy a updaty ostatnym peerom v sieti
6. Sluzba pre transakcie
    a. Komunikacne kanaly

9 Prednaska 10 – Fault Tolerance

Konsezmus - Cieľom distribuovaného konsenzuálneho algoritmu je umožniť skupine počítačov, aby sa všetci zhodli na jedinej hodnote, ktorú navrhol jeden z uzlov v systéme (na rozdiel od vytvárania náhodnej hodnoty) VIAC V PROBLEME BYZANTSKEJ DOHODY

9.1 Zakladne koncepty

Distribuovany system by mal byt **odolny voci chybam (fault-tolerant)** ak nastane nejaka menej zavazna chyba tak by mal pokracovat v operative.

Odolnost voci chybam je v relacii so **kvalitou spoľahlivosti** - **(dependability):**

- Dostupnost (Availability)
- Spolahlivost (Reliabilty)
- Bezpecnost (Safety)
- Udržiavateľnosť (Maintainability)

9.1.1 Dostupnost

Znamena ze **server je v prevadzke** a okamzite pouzitelny.

9.1.2 Spolahlivost

Je meranie systemu, kde system dokaze fungovat **nepretržite bez poruchy.**

Priklady:

- Ak system spadne kazdu 1ms/hod tak dostpunost je viac ako 99.99%, ale nie je to spolahlive
- Ak system nikdy nespadol ale tyzden v roku je vypnuty tak spolahlivy je na 100% ale dostupny na 98%

9.1.3 Bezpecnost

Meria ako velmi **bezpecne tie poruchy** sú.

- Ak bezny system spadne (napr webovy server pre obchod) nic hrozne sa nedeje.
- Ale system ktory ma na starosti napr nukleane zalezitosti by jednoducho spadnut nemal

9.1.4 Udrziavatelnost

Meria ako **jednoduche je tento system prevadzkovat alebo opravit**

- Vysoko udrziavany system moze tiez vykazovat vysoku dostupnost
- Niektore chyby sa dokazu opravit same systemom (Self-healing systems)

9.2 Poruchy

- System **spadne** ak nedokaze splnit nejake specifikacie.
- **Chyba** je cast systemoveho stavu, ktora moze viest k spadnutiu (failure).
- **Porucha** je príčinou chyby (hladanie chyb je dolezite)
- **Fault tolerance** – znamena ze system vie poskytovat sluzby aj ked vznikaju chyby
- **Chyby delime**:
    - **Transient (kratkodobe,prechodne)** – raz sa vyskytnu a zmiznu
    - **Intermittent (prerušovane) –** pride odide a znova pride
    - **Permanent –** otravuju az kym ich niekto neopravi

9.3 Modely poruch

Crash failure – bezne spadnutie

- Server sa zastavi, ale potom pokracuje v pohode po restarte

Omission failure – zlyhanie vynechania

- Server nedokaze odpovedat na spravy
- Server nedokaze prijimat spravy
- Server nedokaze odosielat spravy

Timing failure – zlyhanie nacasovania

- Server neodpoveda v casovom intervale

Response failure – zlyhanie pri odpovedi

- Odpoved serveru je nespravna
- Hodnoty odpovedi su zle
- Stav je zly

Arbitrary failure – Svojvolne zlyhanie (Byzantine failure)

- Server produkuje lubovolne odpovede v lubovolnom case (random veci)

Proces P nedostava spravy od servera Q, ako zisti P ze server Q sa zastavil?

- Synchronny a Asyncrhonny pristup

Fail-stop failures

- Lahko sa najde chyba

Fail-noisy failures

- Chyba sa eventualne najde

Fail-silent failures

- Nedokazeme zistit server sa stopol, alebo je chyba medzi odosielanim sprav

Fail-safe failures

- Chyba vobec neuskodi

Fail-arbitrary failures

- Chyby su nedohladatelne aby boli skodlive, nevieme zistit…

9.4 Maskovanie zlyhania

Redundancia je kľúčovou technikou na skrytie zlyhaní.

Typy redundancie:

- Informacie – pridame viac chybovych hlaseni
- Cas - vytrvalo vykonávať akciu, kým nebude úspešná
- Fyzicka – pridavanie extra komponentov (replikacia atd.)

9.5 Odolnosť procesov

- Maskovanie zlyhania procesov replikáciou
- **Usporiadajte procesy do skupín**, správa odoslaná skupine je doručená všetkým členom
    - Ak niektorý člen spadne, mal by ho doplniť iný

9.5.1 Organizacia skupin a ich manazovanie

Pozname **2 druhy skupin : Flat Group a Hierarchical group**

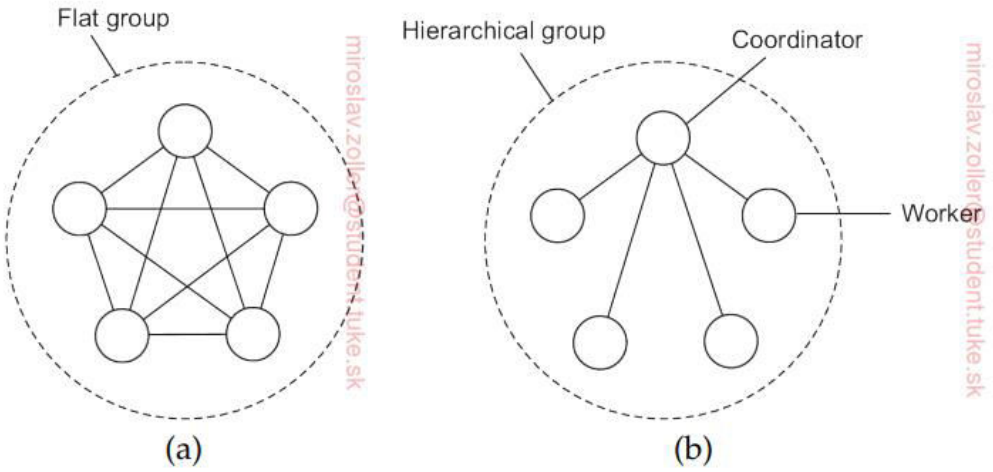


Figure 8.4: Communication in a (a) flat group and in a (b) hierarchical group.

Manazovanie:

- Centralizovane (skupinovy sever) vs Distribuovane (Multicasting)
- Synchronizacia pripojenia a odpojenia
- Znovuvytvorenie skupin

9.6 Replikacia procesov

Replikujeme procesy a skupiny replik v jednej skupine. **Kolko replik vlastne potrebujeme ?**

Predpokladajme replikovany write only system:

> **Systém je považovaný za tolerantný ku k chybám, ak dokáže prežiť chyby v k komponentoch a stále spĺňa svoje špecifikácie aj po znovusputeni.** Ak mame **k+1** replik tak vieme zotavit system. Pretoze jedna replika zachrani vsetko.

Pre byzantske poruchy (cize zly proces produkuje zle odpovede v nahodnych casoch):

- Potrebujeme 3k+1 replik z toho (2k +1 musia byt spravne)

Priklad: Predpokladajme k = 1;
Mame **3k+1=4** repliky takze nato aby sa system zotavil spolu stym aby boli data spravne odosielane potrebujeme **2k+1 = 3** spravne servery. Preco? Pretoze **vacsina prehovori mensinu** ze rozposielava ostatnym serverom bludy.

Ak by boli spravne 2 servery a 2 nespravne repliky tak by sa nikdy nedohodli kde je pravda alebo by preklopili nahodne na pravdu/nepravdu sprav

Ak by bol len 1 spravny server a 3 nepsravne tak potom by sa dohodli vsetci a tym padom by bolo 4 nespravne servery.

Tomuto problemu sa venuje **Problem byzantskej dohody.**
https://www.youtube.com/watch?v=dfsRQyYXOsQ&ab_channel=MarkReddick

https://www.youtube.com/watch?v=LoGx_ldRBU0&ab_channel=MartinKleppmann

9.7 Limitacia dohody v chybovom systeme

Vsetky mozne nasledky

- Syncrhonne vs asyncrhonne systemy
- Sprava je ohranicena v casovom intervale ?
- Spravy su usporiadane alebo nie?
- Distribucia sprav je cez unicast / mulitcast

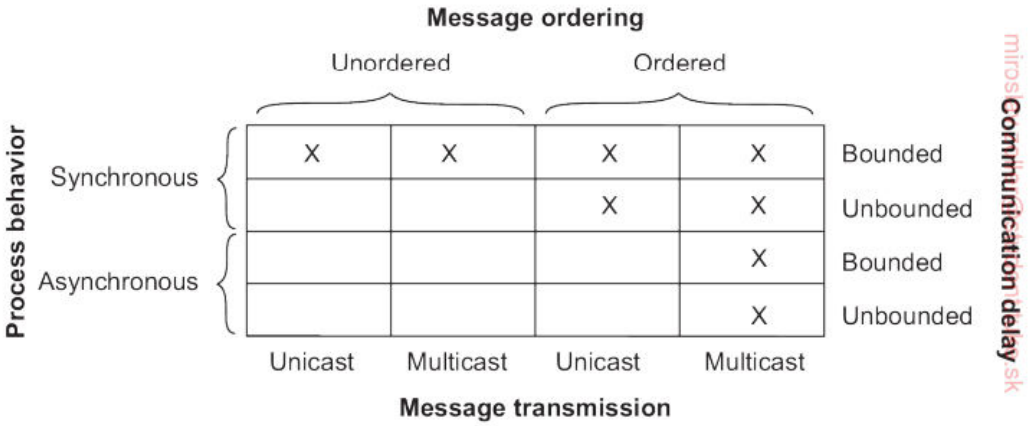


Figure 8.17: Circumstances under which distributed consensus can be reached.

9.8 CAP

Akýkoľvek sieťový systém poskytujúci zdieľané údaje môže poskytnúť **iba dve** z nasledujúcich vlastností:

- Konzistencia
- Dostupnost
- Partitioning (rozdelenie procesov na skupiny kvoli vypadkom siete)

V sieti s poruchami komunikácie nie je možné realizovať atomicke čítanie/zapisovanie ktoru bude garantovat odpoved na kazdu poziadavku.

9.9 Detekcia poruch

- Aktivne Pingovanie „si nazive ?“
- Periodicky heatbeat : multicast message ze „I am alive!“
- Pasivne, kym pride sprava

9.10 Poruchy v RPC

- Klient nemoze najst server
- Poziadavka od klienta k serveru je stratena „Timeout“
- Server spadne po ziskani spravy
- Odpoved od servera ku klientovy je stratena, klient opakuje request, ale su potrebne **idempotentne poziadavky** tzn viacero poziadaviek na server neovplyvni negativne proces
- Klient spadne po odoslani poziadavky

10 Prednaska 11 Fault tolerance #2

10.1 Spolahliva skupinova komunikacia (Reliable group multicast)

Spolahliva skupinova komunikacia -> Intuitívne to znamená, že správa odoslaná skupine procesov by mala byť doručená každému členovi tejto skupiny. Ak rozdelime funkcionalitu **spracovania správ od jadra člena (**Message handling component**)**, mozeme pohodlne rozlisovat medzi **prijimanim** sprav a **odosielanim** sprav. Cize message handling component je middleware (dufam).

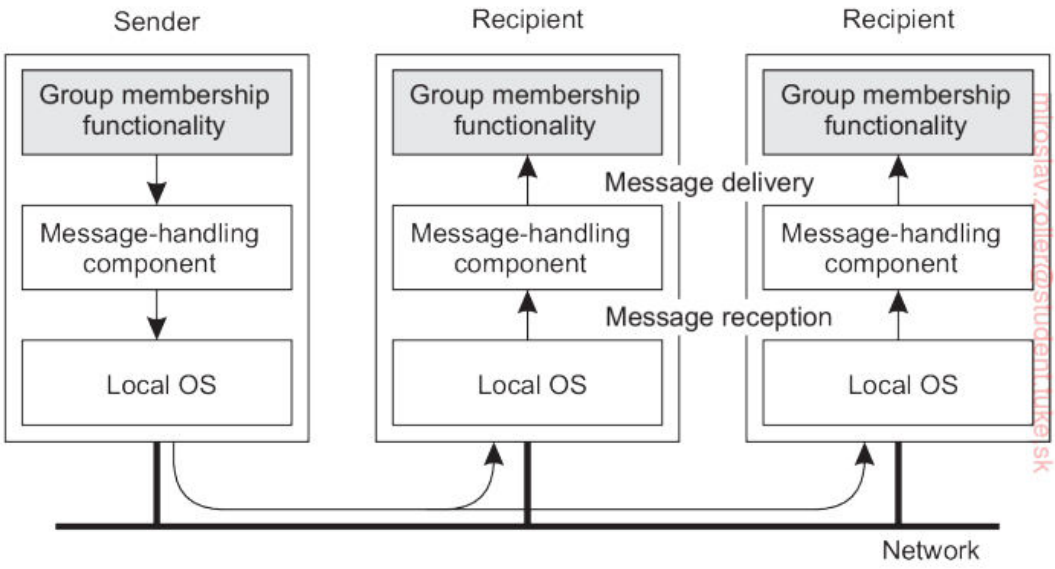


Figure 8.20: The distinction between receiving and delivering messages.

Zacneme jednoduchym prikladom odmyslime si ze mozu vznikat vypadky, ze odosielatel nema uplny zoznam peerov, ze mozu sa tu nachadzat chybne procesy a ze odignorujeme konsenzus (zabranenie odosielania nahodnych sprav v sieti).

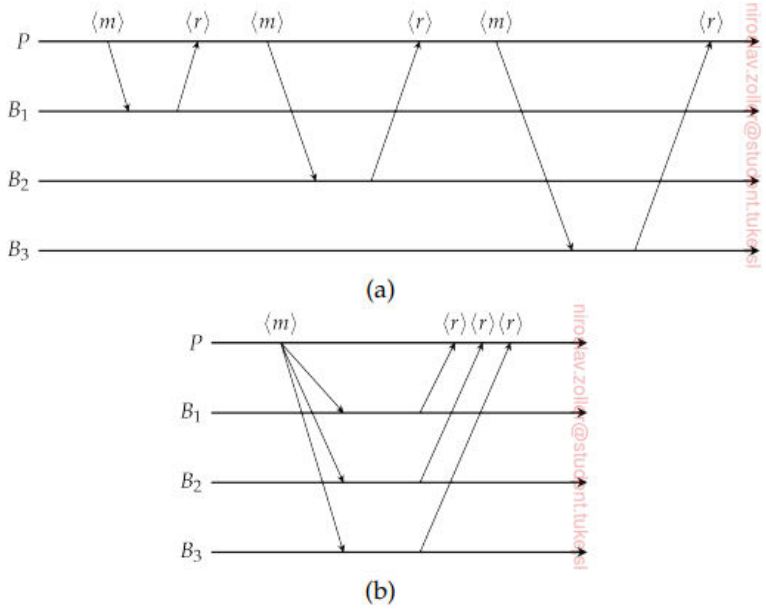


Figure 8.21: (a) A sender sends out requests, but waits for a response before sending out the next one. (b) Requests are sent out in parallel, after which the sender waits for incoming responses.

Jednoduché riešenie spoľahlivého multicastingu, keď sú známe všetky prijímače a predpokladá sa, že nezlyhajú je ze odosielatel posiela **sekvencne cislo spravy** vsetkym a **prijmatelia si cachuju v buffri poslednu hodnotu sekvencie.** Ak sa sekvencia narusi. Prijimatel informuje Odosielatela o chybajucej sprave, ostatne servery odoslu ACK.

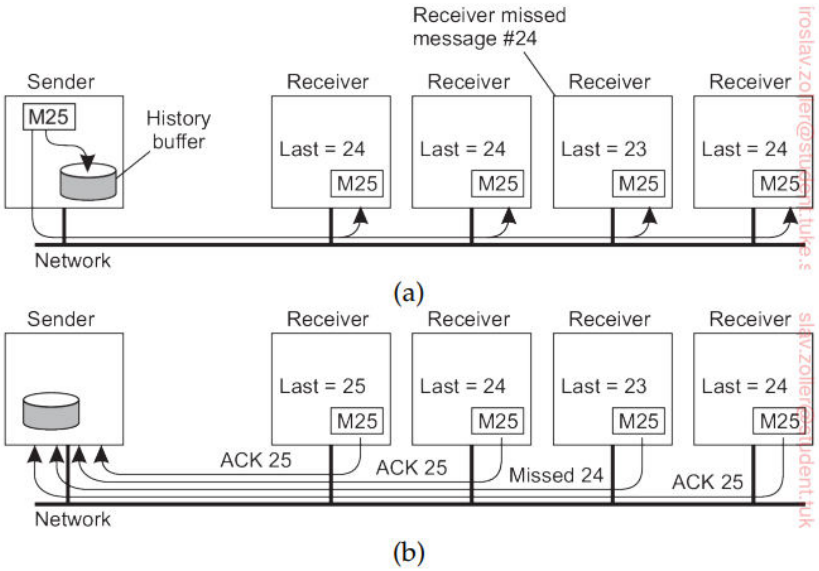


Figure 8.22: A solution for reliable multicasting. (a) Message transmission. (b) Reporting feedback.

10.2 Atomicky multicast

Vratme sa teda trocha do realneho sveta a pocitajme s tym ze procesy mozu **zlyhat.** Co to teda znamena atomicky multicast ?

- **Vsetky spravy su odoslane a v jednotnom poradi vsetkym procesom alebo ziadnym**

V skupine procesov sa moze stat ze proces zlyha v case odosielania spravy a teda skupina je komfrontovana neprijemnej situacii kde nejaka skupina **ziskala spravu** no niektora nie. Ak by cast procesov ulozila spravy do databazy vznikol by **inkonzistentny stav databazy**.

10.2.1 Group view

- Pohlad na **skupinu procesov** ktore su sucastou skupiny, ktorej odosielatel poslal **mulitcastovu** spravu
- Prijata sprava je odoslana **vsetkym** procesom skupiny **alebo nikomu**.

10.2.2 Virtualny synchronny multicast

Správa m multicast do skupinového zobrazenia G je doručená všetkým **neporuchovým procesom (non-faulty processes)** v G alebo je všetkými ignorovaná.

10.2.2.1 Princip virtualnej syncrhonnej multicast spravy

Pomocou virtuálnej synchronizácie sa proces v rámci skupiny (jeden z replikovaných serverov) môže **pripojiť** k skupine alebo ju **opustiť** – alebo môže byť zo skupiny **vylúčený**. Každý proces môže poslať správu skupine a softvér virtuálnej synchronizácie implementuje atómove multicast spravy do skupiny.

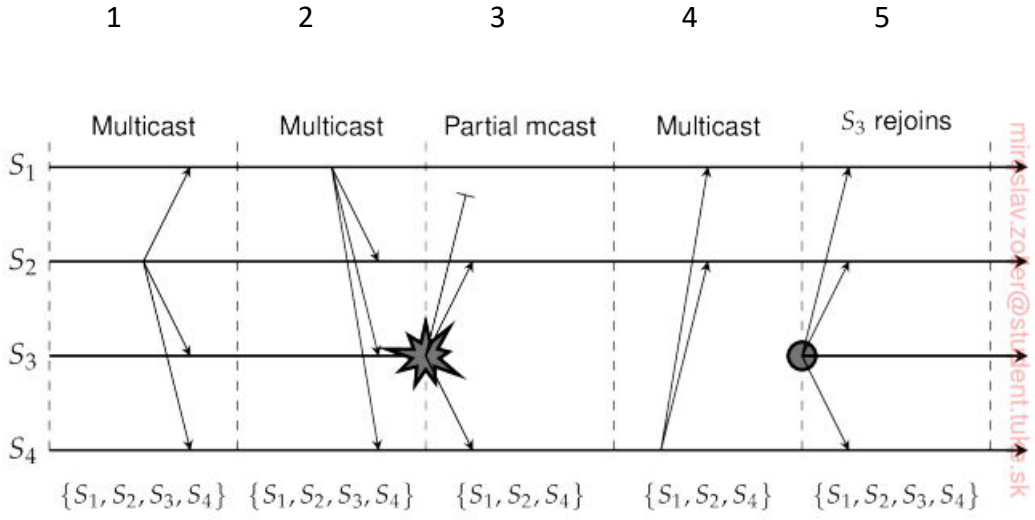


Figure 8.25: The principle of virtual synchronous multicast.

Tu je pekne vidno ze 1 a 2 su valid Multicasty. V 3jke S3 crashol takze sa vykonal len **ciastkovy multicast.** Tzn vsetky procesy **nedostali spravu** a preto virtualna syncrhonizacia zabezpecila aby sa tento multicast **discardol.** Potom v 4 S4 odoslala multicast zvysnej skupine co je uplne v poriadku lebo vsetci spravu prijali. No a nakoniec ked sa S3 naspat napojil tak odoslal multicast znova a tentokrat uspesne.

10.2.2.2 Zoradenie sprav

Pozname 4 druhy zoradenia:

- **Nezoradene multicast spravy (UNORDERED)**

Event order	Process P_1	Process P_2	Process P_3
1	sends m_1	receives m_1	receives m_2
2	sends m_2	receives m_2	receives m_1

Figure 8.26: Three communicating processes in the same group. The ordering of events per process is shown along the vertical axis.

- **FIFO- zoradene multicasty (first in first out - P1 a P4 su concurrent procesy) (FIFO ORDERED)**

Event order	Process P_1	Process P_2	Process P_3	Process P_4
1	sends m_1	receives m_1	receives m_3	sends m_3
2	sends m_2	receives m_3	receives m_1	sends m_4
3		receives m_2	receives m_2	
4		receives m_4	receives m_4	

Figure 8.27: Four processes in the same group with two different senders, and a possible delivery order of messages under FIFO-ordered multicasting.

- **Kauzálne-zoradene multicasty (CAUSALY ORDERED)**
  - Kauzalita medzi rôznymi správami je zachovaná (vektorové timestampy)
- **“Specialny” Uplne-zoradeny mutlicast (atomicky) (TOTALLY ORDERED)**
  - Spravy su doruceny v jednotnom poradi celej skupine takze P2 a P3 by mali byt zhodne

- Six different versions of virtually synchronous reliable multicasting

Multicast	Basic Message Ordering	Total-Ordered Delivery?
Reliable multicast	None	No
FIFO multicast	FIFO-ordered delivery	No
Causal multicast	Causal-ordered delivery	No
Atomic multicast	None	Yes
FIFO atomic multicast	FIFO-ordered delivery	Yes
Causal atomic multicast	Causal-ordered delivery	Yes

Cize aby sme **“Specialny” Uplne-zoradeny mutlicast (atomicky) (TOTALLY ORDERED)** pomocou tychto kombinacii MULTICAST + ORDERING zabezpecime aby napr. Podla prikladu 8.27 procesy P2 a P3 mali **rovnaky** vysledok po prijati sprav.

10.2.2.3 Implementacia virtualnej syncrhonizacie

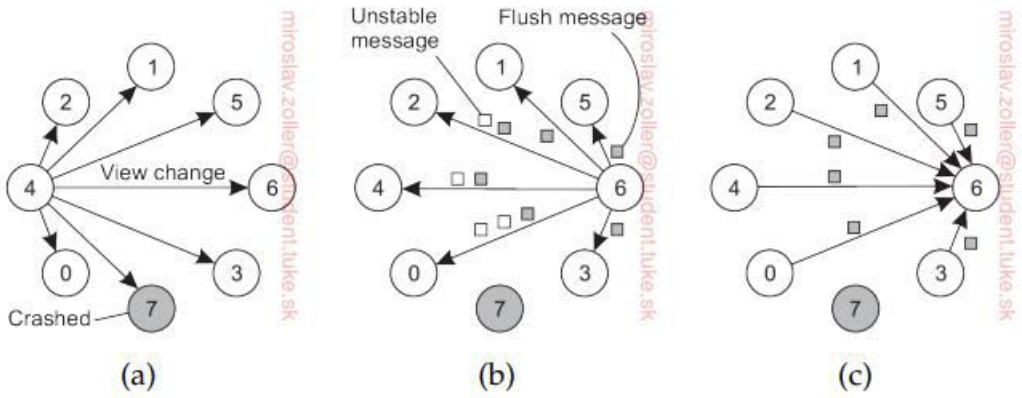


Figure 8.29: (a) Process 4 notices that process 7 has crashed and sends a view change. (b) Process 6 sends out all its unstable messages, followed by a flush message. (c) Process 6 installs the new view when it has received a flush message from everyone else.

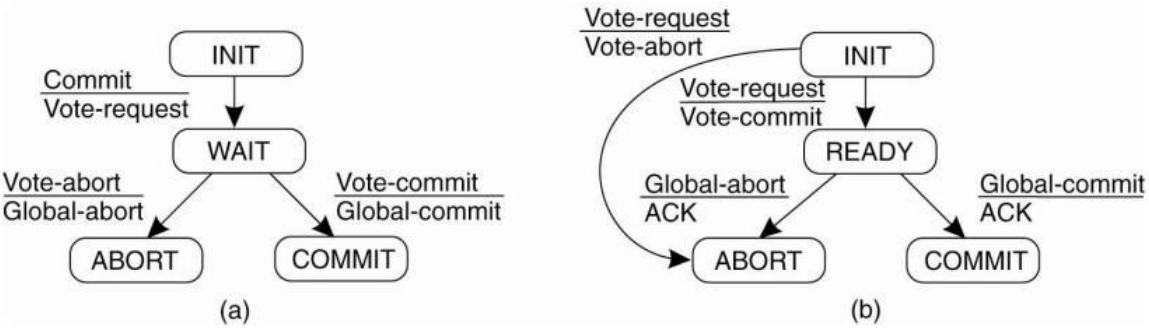
Ako proces 4 zistil ze 7 vypadla ? – Asi ked 7micka vypadla ta neodpovedala na predoslu message tak 4 sa rozhodla poslat spravu.

10.32-fazovy commit

- Koordinator odosle VOTE_REQUEST vsetkym procesom
- Proces vrati VOTE_COMMIT ked chce urobit lokalny commit alebo VOTE_ABORT na zrusenie
- Koordinator vyzbiera vsetky odpovede a posle GLOBAL_COMMIT. Ak vsetky procesy suhlasili s commitom tak OK inak GLOBAL_ABORT
- Kazdy proces pocka na poslednu odpoved, a potom lokalne urobia commit alebo zrusia transakciu

Two-phase commit (2)

- The finite state machine for the coordinator in 2PC (a)
- The finite state machine for a participant (b)



10.43-fazovy commit

Problém s protokolom dvojfázového commitu je, že **keď koordinátor zlyhá**, účastníci nemusia byť schopní dospieť ku konečnému rozhodnutiu. V dôsledku toho môže byť nastat, ze účastníci zostali zablokovaní, kým sa koordinátor nezotavil.

Podstatou 3 phase commit protokolu je, že stavy koordinátora a každého účastníka spĺňajú tieto dve podmienky:

1. Neexistuje jediný stav, z ktorého je možné prejsť priamo do stavu COMMIT alebo ABORT.

2. Neexistuje stav, v ktorom by nebolo možné urobiť konečné rozhodnutie a z ktorého by sa dalo prejsť do stavu COMMIT.

The coordinator in 3PC starts with sending a VOTE-REQUEST message to all participants, after which it waits for incoming responses. If any participant votes to abort the transaction, the final decision will be to abort as well, so the coordinator sends GLOBAL-ABORT. However, when the transaction can be committed, a PREPARE-COMMIT message is sent. Only after each participant has acknowledged it is now prepared to commit, will the coordinator send the final GLOBAL-COMMIT message by which the transaction is actually committed.

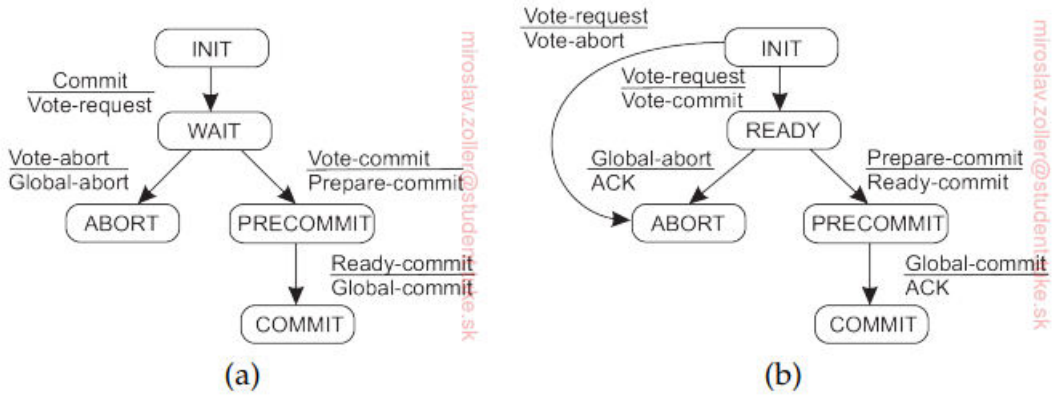


Figure 8.34: (a) The finite state machine for the coordinator in 3PC. (b) The finite state machine for a participant.

10.5 Zotavenie

Ak nejaky proces vykazuje zlyhanie tak potrebuje byt **zotaveny**.

Na taketo operacie potrebuje **stabilne ulozisko**.

Spatne zotavenie

- system sleduje a uklada stavy.. robi si **checkpointy (zachytneb body)** predoslych stavov z ktorych sa dokaze neskor zotavit. Nieco ako snapshot/backup
- prikladom je napr. Packet retransmission (opätovné odoslanie paketov, ktoré boli poškodené alebo stratené počas ich počiatočného prenosu)

Dopredne zotavenie

- Zotavime system tym ze ho vyskladame odznova do spravneho stavu, nie spätne.

10.6 Checkpointing

Obnova po zlyhaní procesu alebo systému vyžaduje, aby sme vytvorili **konzistentný globálny stav z lokálnych stavov**, ktoré si každý proces uloží. Predovšetkým je najlepšie obnoviť **najaktuálnejšii distribuovany snapshot**, ktory sa označuje aj ako **ciara obnovy (recovery line)**. Inými slovami, obnovovacia línia zodpovedá **poslednému konzistentnému súboru kontrolných bodov**, ako je znázornené na obrázku 8.35

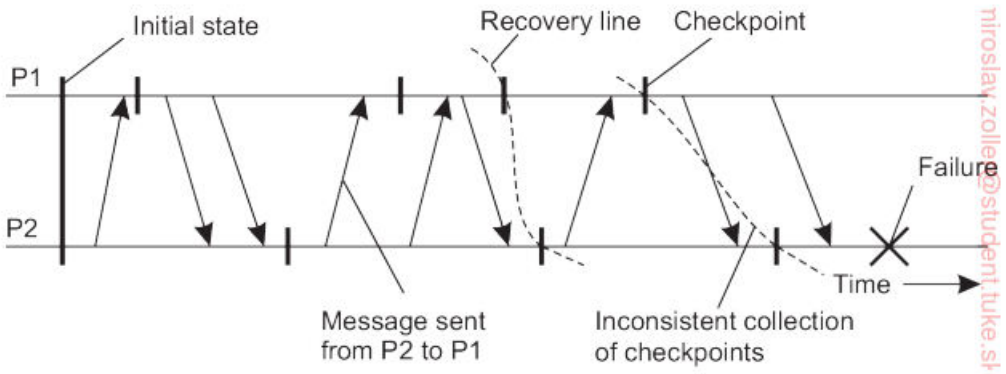


Figure 8.35: A recovery line.

10.6.1 Koordinovany checkpointing

- Procesy musia byt syncrhonizovane pri checkpointe
- 2 fazovy blokovaci protokol
    - Koordinator multicastuje CHECKPOINT_REQUEST
    - Procesy urobia lokalny checkpoint a ACK koordinatorovi
    - Koordinator multicastuje CHECKPOINT_DONE

10.6.2 Nezavisly checkpointing

Teraz zvážte prípad, v ktorom každý proces **z času na čas jednoducho urobi checkpoint** nekoordinovaným spôsobom. Na zistenie linky obnovy je potrebné, aby bol každý proces vrátený späť do posledného uloženého stavu. Ak tieto lokalne stavy spoločne nevytvoria distribuovanú snímku, je potrebné ďalšie vrátenie. Tento proces kaskádového rollbacku môže viesť k tomu, čo sa nazýva **dominový efekt** a je znázornené na obrázku 8.36.

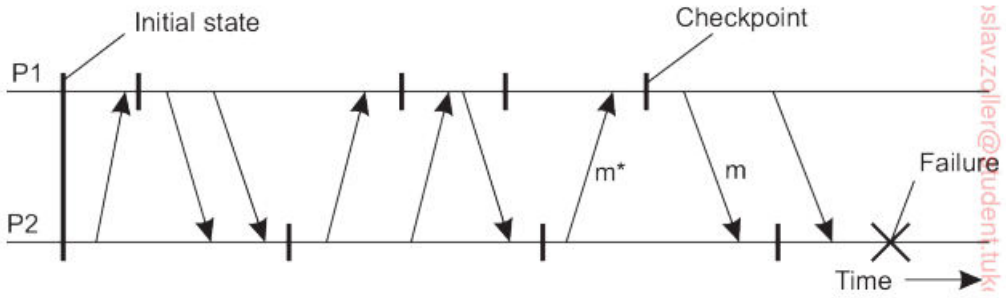


Figure 8.36: The domino effect.

When process P_2 crashes, we need to restore its state to the most recently saved checkpoint. As a consequence, process P_1 will also need to be rolled back. Unfortunately, the two most recently saved local states do not form a consistent global state: the state saved by P_2 indicates the receipt of a message m, but no other process can be identified as its sender. Consequently, P_2 needs to be rolled back to an earlier state.

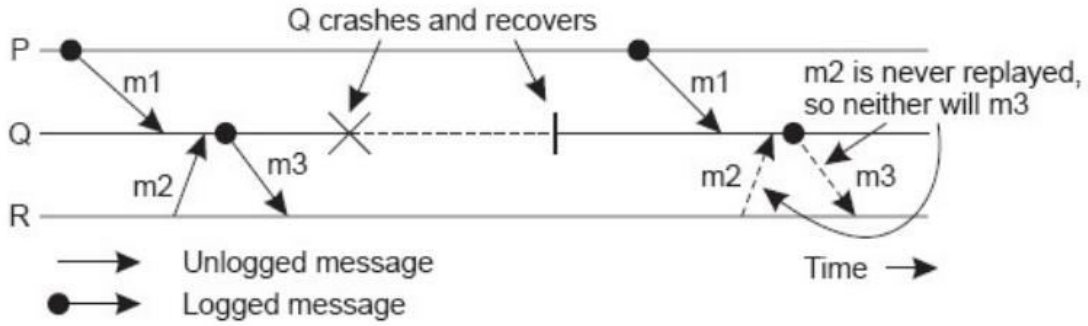
However, the next state to which P_2 is rolled back also cannot be used as part of a distributed snapshot. In this case, P_1 will have recorded the receipt of message m^*, but there is no recorded event of this message being sent. It is therefore necessary to also roll P_1 back to a previous state. In this example, it turns out that the recovery line is actually the initial state of the system.

10.7 Logovanie sprav

- Checkpoint je draha operacia
- Logovanie sprav pomoze **znovu vykonat operacie vykonania** (replay) v urcitom poradi namiesto snapshotov
- Logovanie moze byt **Pesimisticke a Optimisticke**

10.7.1 Proces – sirota

Prikladom nespravneho **replayu**, moze zapricinit ze vznikne sirota. Tak ako dole v priklade proces R.



10.7.2 Schemy pre logovanie sprav

Kazda sprava by mala splnat minimalne tieto poziadavky ak by sa mohla v buducnosti pouzit na zotavenie:

- Sprava obsahuje hlavicku s dolezitimy info (sequence,sender,receiver)
- Sprava sa nestratila pocas prenosu
- Kazda sprava obsahuje postupy zavislych procesov a vykonani
- Each message m leads to a set of dependent processes DEP(m), to which either m or a message causally dependent on m has been delivered

• The set COPY(m) consists of the processes that have a copy of m, but not in their local stable storage

• Any process in COPY(m) could deliver a copy of m on request

• Process Q is an orphan process if there is a nonstable message m, such that Qis contained in DEP(m), and every process in COPY(m) has crashed

• To avoid orphan processes, we need to ensure that if all processes in COPY(m) crash, no processes remain in DEP(m). It means, all processes in DEP(m) should crash. Whenever a process becomes dependent on m, it will keep a copy of m

10.7.3 Pesimisticke logovanie

- Pre každú nestabilnú správu m sa uistite, že najviac jeden proces P je závislý od m
- Najhoršie, čo sa môže stať, je, že P havaruje bez m po prihlásení
- Žiadny iný proces sa nemohol stať závislým od m, pretože m bol nestabilný, takže nezostanú žiadne siroty

10.7.4 Optimisticke logovanie

- Loguje sa az po spadnuti procesu (The work is done after a crash occurs, not before)
- Ak pre niektoré m každý proces v COPY(m) zlyhal, potom každý osirotený proces v DEP(m) sa vráti späť do stavu v ktorý už nepatrí do DEP(m)
- Závislosti musia byť explicitne sledované, čo znamená je to ťažko implementovatelny sposob logovania
    - V dôsledku toho sa **v reálnom svete uprednostňujú pesimistické prístupy**

11 Prednaska 12 Security

11.1 CIA Triad model

- Confidentiality – Dôvernosť – informaciu vydia len autorizovani pouzivatelia
- Integrity – Integrita – data su mene len autorizovanymi pouzivatelmi
- Availability – Dostupnosť – system je chraneny pred utokmi aby mohol nadalej poskytovat sluzby

11.2 Bezpecnostne hrozby

- **Interception** – odpočuvanie – neautorizovana strana ma pristup k datam alebo sluzbe
- **Interruption** – prerušenie – sluzba alebo data su nedostupne
- **Modification** – modifikacia – neatorizovana zmena dat
- **Fabrication** – zhotovenie – doplnujuce data alebo aktivita ktora by v systeme existovat nemala

11.3 Bezpečnostná politika a bezpečnostný mechanizmus

- **Security policy** – bezpecnostne politky
  - Presny opis povolenych a zakazanych akcii v systeme
- **Security mechanism** – bezpecnostny mechanizmus
  - Mechanizmy pomocou ktorych politiky mozno presadit

Sem patria:

- Encryption
- Authentification – overime ci osoba je ta za ktoru sa vydava (passwd,2FA,cipove karty)
- Authorization – overime ci osoba ma povolenie vykonat akcie (ci ma prava/rolu)
- Auditing

11.4 Problemy návrhu

Distribuovaný systém alebo akýkoľvek počítačový systém musí poskytovať bezpečnostné služby, pomocou ktorých možno implementovať širokú škálu bezpečnostných politík. Pri implementácii všeobecných bezpečnostných služieb je potrebné **vziať do úvahy množstvo dôležitých problémov s dizajnom**. Rozlisujeme tri z týchto problémov: **zameranie kontroly, vrstvenie bezpečnostných mechanizmov a jednoduchosť** (focus of control, layering of security mechanisms, and simplicity)

11.4.1 Problemy navrhu - zameranie kontroly

Prvy pristup je zamerany na ochranu dat ktore su v zavislosti s aplikaciou. Prvoradým záujmom je **zabezpečiť integritu dát**.

Druhým prístupom je sústrediť sa na ochranu tým, že sa presne špecifikuje, **ktoré operácie a kto môže vyvolať.** V tomto prípade sa dôraz kladie na kontrolu súvisiacu s mechanizmami kontroly prístupu.

Tretím prístupom je zamerať sa **priamo na používateľov** prijatím opatrení, podľa ktorých majú k aplikácii prístup iba konkrétni ľudia, bez ohľadu na operácie, ktoré chcú vykonávať.

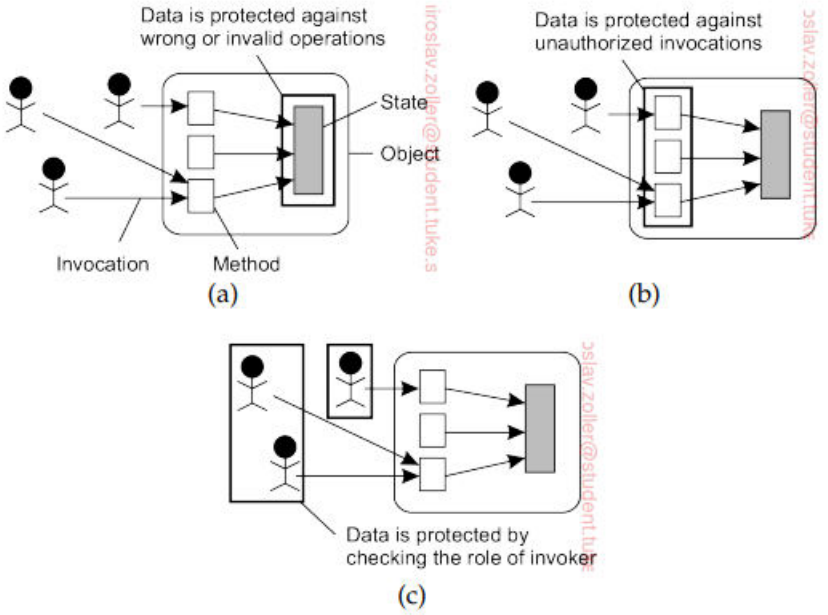


Figure 9.1: Three approaches for protection against security threats. (a) Protection against invalid operations (b) Protection against unauthorized invocations. (c) Protection against unauthorized users.

11.4.2 Problemy navrhu - vrstvenie bezpečnostných mechanizmov

Na akú úroveň by sa mal umiestniť bezpečnostný mechanizmus? Závisí to od dôvery klienta v to, ako bezpečná je služba na konkrétnej úrovni. Bezpečnostné mechanizmy sú zvyčajne umiestnené na **úrovni middlewaru**

Napríklad: Použitie služby TLS sa používa na bezpečné odosielanie správ cez pripojenie TCP. Takze :

- Všetky správy na úrovni transportu sú šifrované (t. j. všetky vrstvy nižšej úrovne sú bezpecne)
- **Dôvera** je na TLS (t. j. veríme, že TLS je **bezpečné**)

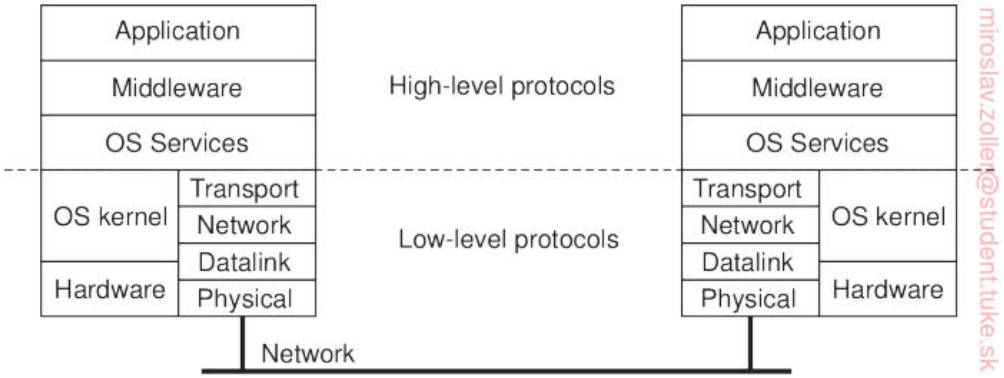


Figure 9.2: The logical organization of a distributed system into several layers.

Rozdiel medzi dôverou a bezpečnosťou je dôležitý. Systém je buď bezpečný, alebo nie je (pri zohľadnení rôznych pravdepodobnostných opatrení), ale to, či klient považuje systém za bezpečný, je vecou dôvery.

11.4.2.1 Distribucia bezpečnostných mechanizmov

- **TCB** (Trusted computer base) - Súbor všetkých mechanizmov v systéme, ktoré sú potrebné na presadzovanie bezpečnostnej politiky a ktorým **je potrebné dôverovať**

Predstavte si súborový server v distribuovanom súborovom systéme. Takýto server sa **musi spoliehať** na rôzne ochranné mechanizmy, **ktoré ponúka jeho lokálny operačný systém**. Medzi takéto mechanizmy patria nielen tie, ktoré chránia súbory pred prístupmi iných procesov ako súborový server, ale aj mechanizmy na ochranu súborového servera pred hrozbou zvonka.

11.5 Kryptografia

Odosielateľ zašifruje správu m na nezrozumiteľnú správu m' a príjemca zase dešifruje prijatú správu m' na jej pôvodné m. Easy as it is.

V sieti sa mozu nachadzat ludia so zlym umyslom, bud su **aktivni (narusitelia) alebo pasivni (nasluchaci).**

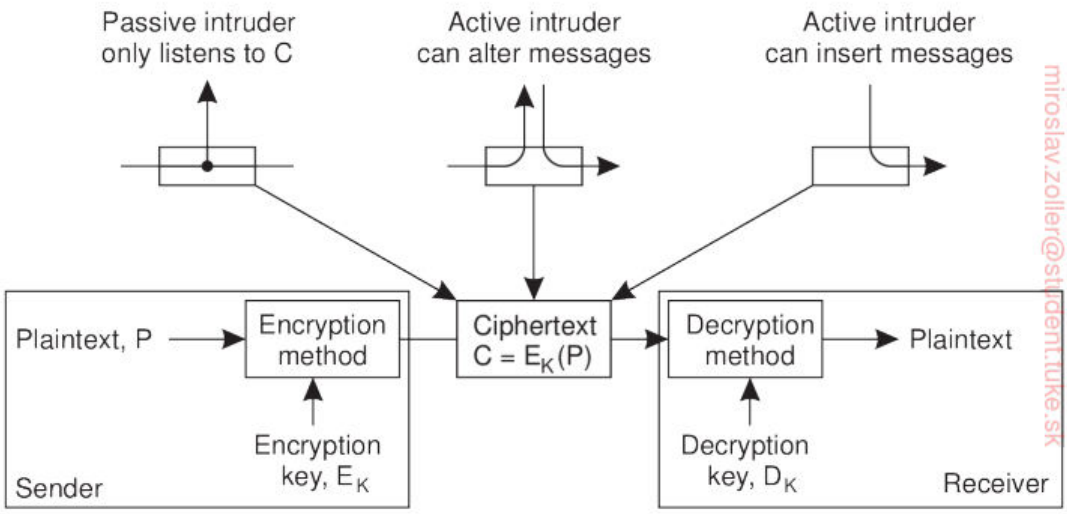


Figure 9.4: Intruders and eavesdroppers in communication.

11.5.1 Symetricke a asymetricke sifrovanie

- Symetricke je viacmenej jasne. Vezmem plaintext a zasifrujem ho s klucom. S tym istym klucom aj desifrujem.

- Pri asymetrickej pouzivame **kluc na sifrovanie a kluc na desifrovanie**

11.5.2 Hashovacia funkcia

- Jednosmerna funkcia (neda sa ziskat povodna sprava z hashu)
- **Slaba kolizna odolnost**: Je velmi tazke **najst dalsii** rovnaky retazec aby vygeneroval rovnaky hash
- **Silna kolizna odolnost:** Je velmi tazke najst **akukolvek dvojicu** retazcov aby vegenerovali rovnaky hash

11.6 Zabezpecene kanaly (Secure channels)

Ako zabezpečiť bezpecnu komunikáciu medzi klientmi a servermi? **Zabezpečený kanál chráni odosielateľov a príjemcov pred zachytením, modifikáciou a fabrikáciou správ**. Noale nemusí nevyhnutne chrániť pred prerušením.

Ochrana správ pred zachytením sa vykonáva **zabezpečením dôvernosti**: zabezpečený kanál zaisťuje, že jeho **správy nemôžu byť odpočúvané**. Ochrana proti modifikácii a fabrikacii pred narušiteľmi sa vykonáva prostredníctvom protokolov pre vzájomnú autentifikáciu a integritu správ.

Ako môže server zistiť, že klient je oprávnený vykonať túto požiadavku? - Práva klienta.

Ako zabezpečiť bezpecnu komunikáciu medzi klientmi a servermi? – Autentifikacia, Zabezpecit dovernost spravy a ochrana komunikacie medzi servermi.

11.6.1 Autentifikacia

Proces alebo činnosť dokazovania alebo preukazovania, že niečo je pravdivé, pravé alebo platné.

Ake autentifikacne mechanizmy pozname ?

- Autentifikacia zalozena na **Tajnom kluci**
- Autentifikacia pomocou **distribucneho centra**
- Autentifikacia pomocou **kryptografie verejneho kluca**

$K_{A,B}$ – Secret key shared by A and B
K_A^+ – Public key of A
K_A^- – Private key of A
$K(d)$ – Some data d encrypted by key K

11.6.1.1 Secret key autentifikacia

Kľúč relácie je zdieľaný (tajný) kľúč, ktorý sa používa na šifrovanie správ pre integritu a prípadne aj dôvernosť. Takýto kľúč sa vo všeobecnosti používa len dovtedy, kým kanál existuje. Keď je kanál zatvorený, jeho priradený kľúč relácie sa zničí.

In the case of authentication based on a shared secret key, the protocol proceeds as shown in Figure 9.6. First, Alice sends her identity to Bob (message 1), indicating that she wants to set up a communication channel between the two. Bob subsequently sends a **challenge** R_B to Alice, shown as message 2. Such a challenge could take the form of a random number. Alice is required to encrypt the challenge with the secret key $K_{A,B}$ that she shares with Bob, and return the encrypted challenge to Bob. This response is shown as message 3 in Figure 9.6 containing $K_{A,B}(R_B)$.

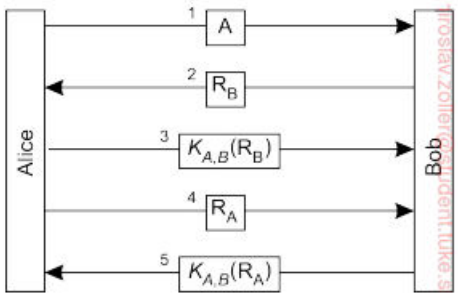


Figure 9.6: Authentication based on a shared secret key.

When Bob receives the response $K_{A,B}(R_B)$ to his challenge R_B, he can decrypt the message using the shared key again to see if it contains R_B. If so, he then knows that Alice is on the other side, for who else could have encrypted R_B with $K_{A,B}$ in the first place? In other words, Bob has now verified that he is indeed talking to Alice. However, note that Alice has not yet verified that it is indeed Bob on the other side of the channel. Therefore, she sends a challenge R_A (message 4), which Bob responds to by returning $K_{A,B}(R_A)$, shown as message 5. When Alice decrypts it with $K_{A,B}$ and sees her R_A, she knows she is talking to Bob.

OSLABENIE SECRET MESSAGE

The attack is illustrated in Figure 9.8 Chuck starts out by sending a message containing Alice's identity A, along with a challenge R_C. Bob returns his challenge R_B and the response $K_{A,B}(R_C)$ in a single message. At that point, Chuck would need to prove he knows the secret key by returning $K_{A,B}(R_B)$ to Bob. Unfortunately, he does not have $K_{A,B}$. Instead, what he does is attempt to set up a second channel to let Bob do the encryption for him.

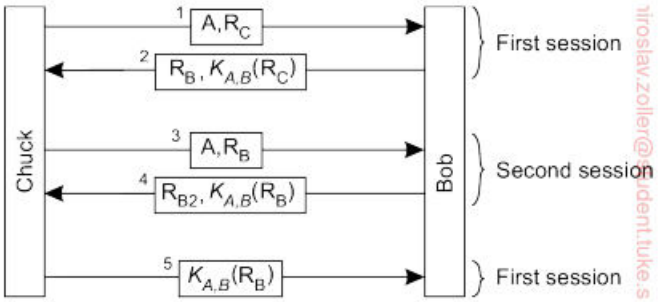


Figure 9.8: The reflection attack.

Therefore, Chuck sends A and R_B in a single message as before, but now pretends that he wants a second channel. This is shown as message 3 in Figure 9.8 Bob, not recognizing that he, himself, had used R_B before as a challenge, responds with $K_{A,B}(R_B)$ and another challenge R_{B2}, shown as message 4. At that point, Chuck has $K_{A,B}(R_B)$ and finishes setting up the first session by returning message 5 containing the response $K_{A,B}(R_B)$, which was originally requested from the challenge sent in message 2.

11.6.1.2 Autentifikacia pomocou KDC (Key distribution center)

Centralizovany sposob distribucie klucov medzi hostov. Takze KDC spravuje secret message. No lenze kde tu v tomto modeli nastava komunikacia ?

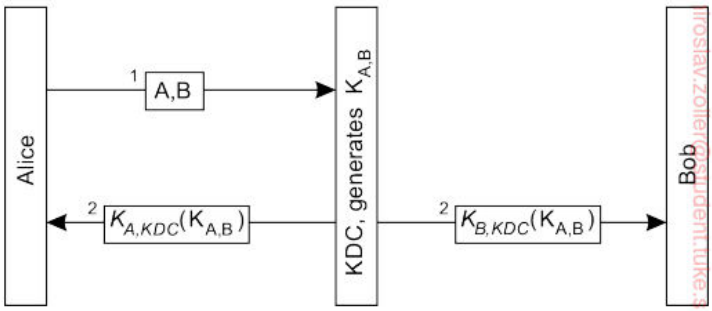


Figure 9.9: The principle of using a Key Distribution Center (KDC).

The main drawback of this approach is that Alice may want to start setting up a secure channel with Bob even before Bob had received the shared key from the KDC. In addition, the KDC is required to get Bob into the loop by passing him the key. These problems can be circumvented if the KDC just passes $K_{B,KDC}(K_{A,B})$ back to Alice, and lets her take care of connecting to Bob. This leads to the protocol shown in Figure 9.10. The message $K_{B,KDC}(K_{A,B})$ is also known as a **ticket**. It is Alice's job to pass this ticket to Bob. Note that Bob is still the only one who can make sensible use of the ticket, as he is the only one besides the KDC who knows how to decrypt the information it contains.

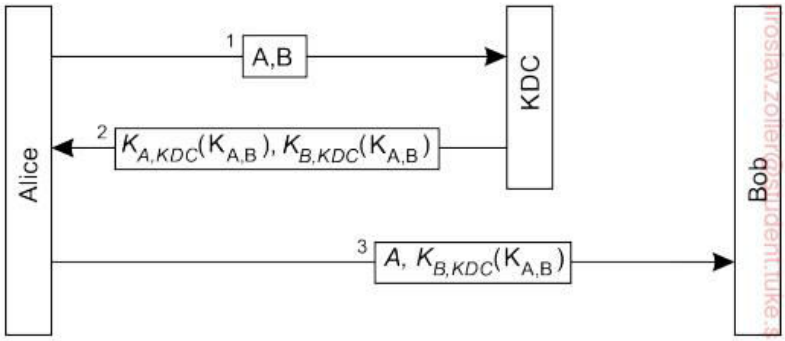
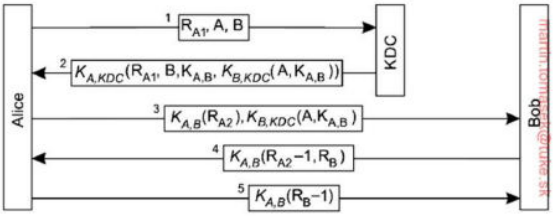


Figure 9.10: Using a ticket and letting Alice set up a connection to Bob.

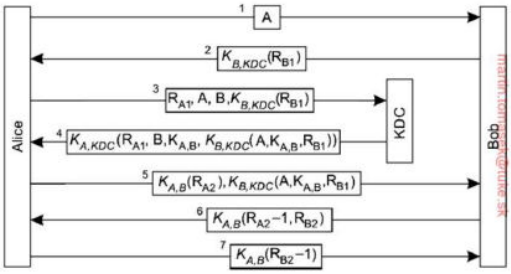
11.6.1.3 Needham-Schroeder protocol (multiway challenge-respone protokol)

Toto uz mi pride jak overkill

Needham-Schroeder protocol (multiway challenge-response protocol)



- Protection against malicious reuse of session key



11.6.1.4 Autentifikacia pomocou public-key kryptografie

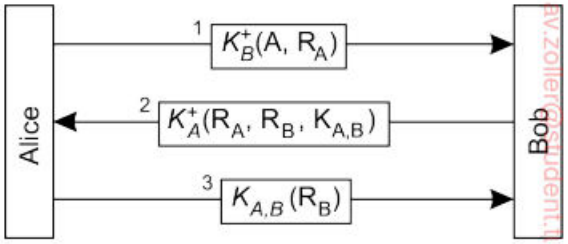


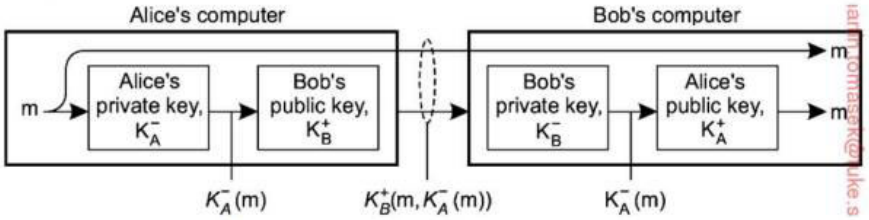
Figure 9.13: Mutual authentication in a public-key cryptosystem.

Toto je klasika ktoru vsetci pozname.. Alice zasifruje Bobovym verejnym klucom spravu, Bob sifruje zas spravu Alicinym verejnym klucom. Obaja maju svoje privatne kluce aby si spravy mohli desifrovat.

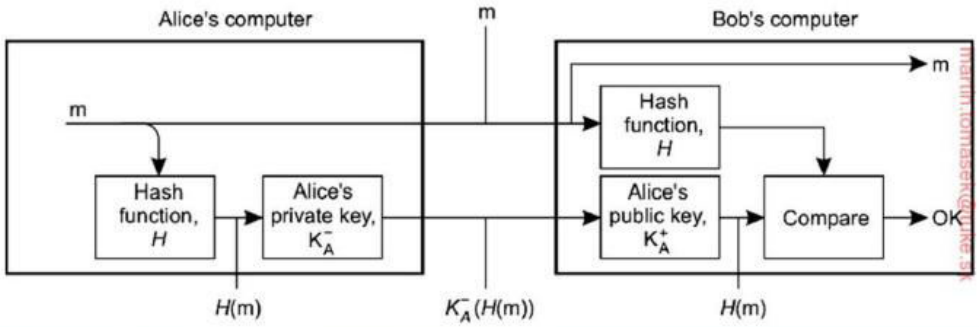
11.7 Integrita a Dôvernosť sprav

- Pouzijeme Digitalny podpis, cize svoj privatnym klucom zasifrujeme napr hash a nasim verejnym klucom si druha strana overi ci skutocne sprava prisla od nas.

• Digital signatures



• Signing a message digest



Session key (kluc relacie):

- Komunikačné strany vo všeobecnosti používajú jedinečný kľúč zdieľanej relácie na zabezpečenie dôvernosti komunikacie
- Kľúč sa zahodí, keď sa kanál už nepoužíva

Vyhody klucov relacie:

- Autentifikačné kľúče sa používajú čo najmenej (ťažšie je ich odhaliť)
- Ochrana pred útokom odpovede (reply attack)
- Keď je kľúč relácie kompromitovaný, je ovplyvnená iba jedna relácia a nie všetky staré konverzácie

11.7.1 Bezpecna skupinova komunikacia

Ako zabezpecit dovernu skupinovu komunikaciu

- Členovia skupiny zdieľajú rovnaký tajný kľúč na šifrovanie a dešifrovanie (ale všetci členovia musia byť dôveryhodní)

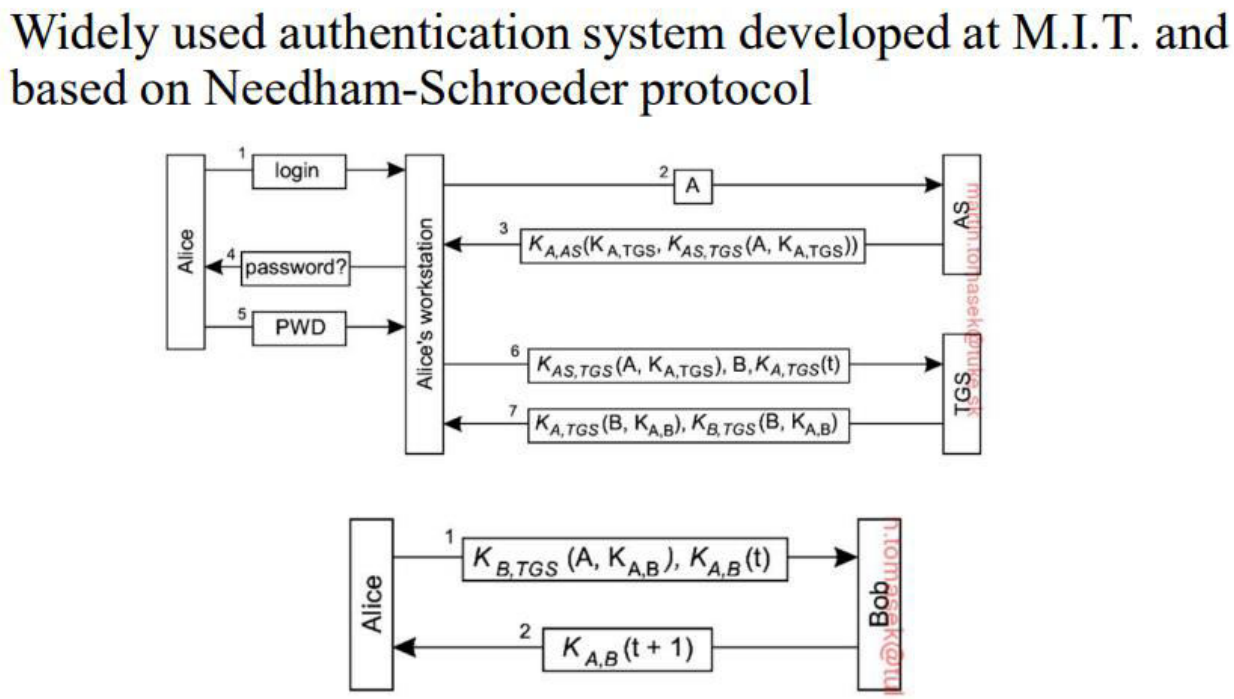
- Alternatívou je držať kluce pre každý pár členov skupiny (ale potrebujeme n (n – 1) klucov
- Kryptosystem s verejnym klucom moze byt pouzity

Bezpecne replikovane servery

- Klient ocakava ze moze verit sprave ktora mu prisla
- Zhromažďujte odpovede zo všetkých serverov a autentifikujte každý z nich, a ak existuje väčšina, klient môže odpovedi dôverovať (ale transparentnosť replikácie je narušená)

Prikladom moze byt **Kerberos,** ten trojhlavy pekelny pes ktory strazi vstup do podsvetia.

Taktiez overkill na tento predmet:



11.8 Access control

Potrebujeme zabezpecit aby uz overeny pouzivatel systemu mohol vykonavat **len povolene operacie**. Takze v tejto casti riesime autorizaciu klienta voci serveru. Cize klient bez urctiych prav/roly nemoze vykonavat operacie na serveri.

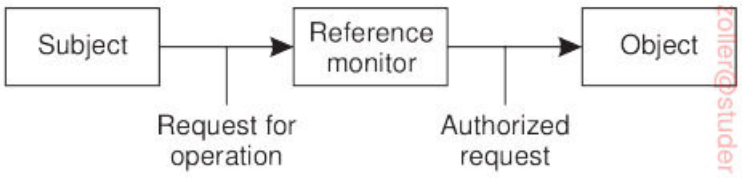


Figure 9.19: General model of controlling access to objects.

11.8.1 Access control matrix

Je len obycajne pole v ktorom sa nachadzaju entries uzivatel/file/prava

11.9 Ohniva stena – firewall

je sieťové zariadenie alebo softvér, ktorého úlohou je oddeliť siete s rôznymi prístupovými právami (typicky napr. Extranet a Intranet) a kontrolovať tok dát medzi týmito sieťami.

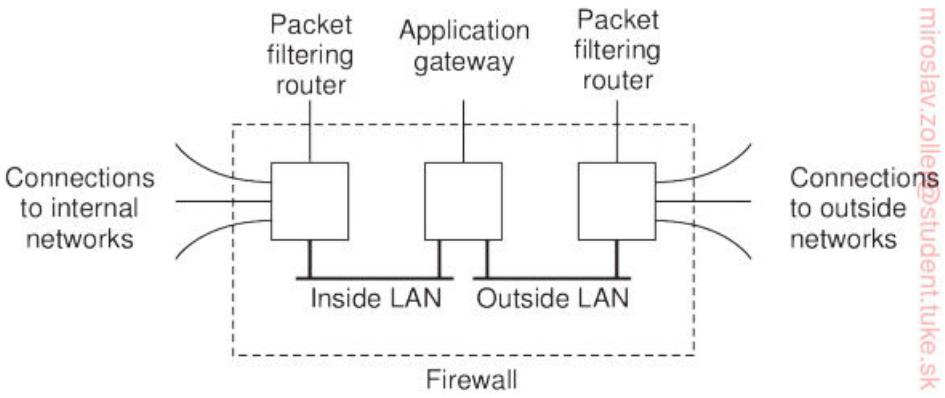


Figure 9.22: A common implementation of a firewall.

11.10 Sandboxing

Z wiki:

Sandbox je v informatice označení pro bezpečnostní mechanismus v rámci počítačové bezpečnosti, který slouží pro oddělování běžících procesů.[1]

Sandbox poskytuje procesům, které v něm běží, omezený přístup ke zdrojům hostitelského počítače. Přístup k disku je typicky omezen na vybrané adresáře, přístup k síti na vybrané servery a porty apod.

Sandbox je často využíván pro dočasné spouštění neotestovaného kódu nebo nedůvěryhodných programů z neověřených třetích stran, od neověřených dodavatelů, či od nedůvěryhodných uživatelů.[2] Sandbox, doslova přeložený jako pískoviště, je vlastně místo, kde se písek nedostane (nemá dostat) mimo vyhrazenou plochu.

11.11 Denial of Service

Škodlivé bránenie autorizovaným procesom v prístupe k zdrojom. Tzn napr zahltenie systemu poziadavkami.

Distributed Denial of Service DDOS – pouzivame viac distribuovanych zdroj na utok na siet napr. botnet.

11.12 Secure naming

A topic that has received increasingly more attention is that of secure naming. Simply put: when a client retrieves an object based on some name, how does it know that it got back the correct object? The whole issue is rather fundamental: when resolving a name in DNS, how does the client know it is returned the correct address? When looking up an object using some combination of a URL and database query, how does the receiver know it is returned what was requested? To be more precise, we have three issues to worry about [Smetters and Jacobson, 2009]:

1. **Validity**: is the object returned a complete, unaltered copy of what was stored at the server?
2. **Provenance**: can the server that returned the object be trusted as a genuine supplier? For example, it may be the case that a client is returned a cached version of the original object.
3. **Relevance**: is what was returned relevant considering what was asked?

11.13 Security management

- Key management
- Secure group management
- Authorization management

11.13.1 Diffie helman key exchange

Je to bezpecny sposob ako vymenit kluce skrz siet.

Princip pomocou farieb z wiki

Description [edit]

General overview [edit]

Diffie–Hellman key exchange establishes a shared secret between two parties that can be used for secret communication for exchanging data over a public network. An analogy illustrates the concept of public key exchange by using colors instead of very large numbers:

The process begins by having the two parties, Alice and Bob, publicly agree on an arbitrary starting color that does not need to be kept secret. In this example, the color is yellow. Each person also selects a secret color that they keep to themselves – in this case, red and cyan. The crucial part of the process is that Alice and Bob each mix their own secret color together with their mutually shared color, resulting in orange-tan and light-blue mixtures respectively, and then publicly exchange the two mixed colors. Finally, each of them mixes the color they received from the partner with their own private color. The result is a final color mixture (yellow-brown in this case) that is identical to their partner's final color mixture.

If a third party listened to the exchange, they would only know the common color (yellow) and the first mixed colors (orange-tan and light-blue), but it would be very hard for them to find out the final secret color (yellow-brown). Bringing the analogy back to a real-life exchange using large numbers rather than colors, this determination is computationally expensive. It is impossible to compute in a practical amount of time even for modern supercomputers.

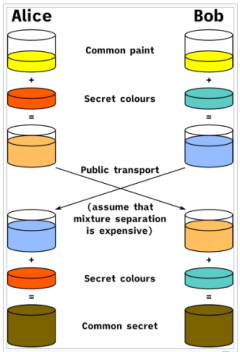


Illustration of the concept behind Diffie–Hellman key exchange

11.13.2 Distribucia klucov

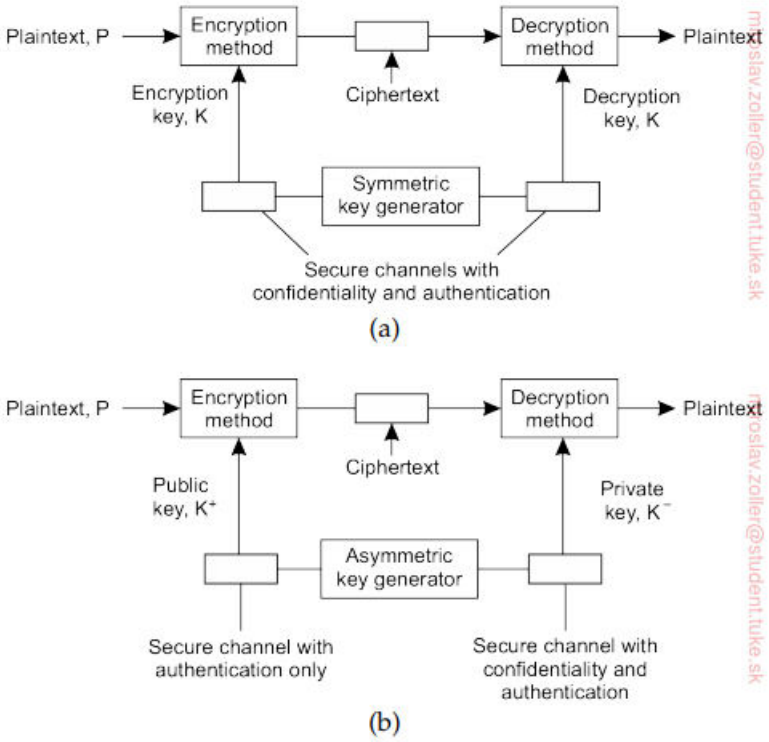


Figure 9.27: (a) Secret-key distribution. (b) Public-key distribution (see also [Menezes et al., 1996]).

...