Osztott rendszerek

Kidolgozott vizsgatematika

Jurányi Zsolt (JUZRAAI.ELTE)

Utolsó módosítás: 2012.06.11. 12:43

Megjegyzés.

Ezt a doksit elsősorban magamnak készítettem, az előadáson készített jegyzeteim és a letöltött diasorok alapján [1]. **Ez nem a diasorok teljes fordítása és nem is feltétlenül a teljes anyag.**

Ha valaki hibát találna benne, kérem, jelezze a zsolt.juranyi@gmail.com címen, vagy esetleg Facebook-on. A legfrissebb verziót mindig megtaláljátok a blogomon [2].

Jó készülést, sok sikert!

Zsolt

[1] Az előadás diasorai Tóth Melinda tanárnő, előadó tárhelyén:
http://people.inf.elte.hu/toth_m/or/

[2] juzraai blogja:
http://juzraai.blogspot.com/

Tételek

1. Az elosztott rendszer fogalma és főbb tervezési tulajdonságai 2
2. Az elosztott rendszerek típusai 4
3. Architektúra stílusok 5
4. Elosztott rendszerek különböző architektúrái 5
5. A köztesréteg kapcsolata az elosztott rendszer architektúrájával 7
6. Ön-szervezés elosztott rendszerekben 7
7. Processzek és szálak nem elosztott és elosztott rendszerekben 8
8. Virtualizáció 9
9. Kliens folyamatok (+ példák) 9
10. Szerver folyamatok (+ példák) 10
11. Szerver klaszterek 10
12. Kód áthelyezése 11
13. Kommunikációs protokollok és típusok 12
14. Távoli eljárás hívások (RPC) 13
15. Üzenet-orientált kommunikáció 15
16. Adatfolyam(stream)-orientált kommunikáció 15
17. Multicast kommunikáció 16
18. Órák szinkronizálása 17
19. Logikai órák szinkronizálása 18
20. Kölcsönös kizárás 19
21. Globális állapot meghatározása 20
22. Választó/szavazó algoritmusok 20
23. Replikátumok(többszörözés) használata és konzisztenciával való kapcsolata 21
24. Adat-centrikus konzisztencia model 21
25. Kliens-centrikus konzisztencia model 22
26. Replikátumok kezelése 22
27. Konzisztencia protokolok 23
28. Elnevezésések 24
29. Flat(sima) elnevezések 24
30. Strukturált elnevezések 25
31. Attributum alapú elnevezések 25
Tételek vázlatosan. 26

Diasor #1 – Bevezetés

1. Az elosztott rendszer fogalma és főbb tervezési tulajdonságai

1.1. Fogalma

- **független gépek** sokasága, melyek a felhasználó számára egy **összefüggő rendszert** alkotnak.
- Az osztott rendszer egy **köztes réteg** a gépek és alkalmazások között.

1.2. Tervezési szempontok

- erőforrások elérhetővé tétele a felhasználónak (pl. nyomtató megosztás) (az erőforrás adat is lehet)
- átlátszóság: felhasználó csak az adatot kapja, nem kell tudni az üzemeltetőről, a rendszer „elfedi" mi van alatta (~átlátszatlanság – relatív :D)
- nyíltság
- skálázhatóság: különböző terhelésekre tudjon reagálni

1.2.1. Átlátszóság

- **hozzáférési** (access): adatok tárolási módjának, belső reprezentációjának elfedése (pl. '/' vagy '\' – közös interfész)
- **hely** (location): fizikai hely elfedése, pl. globális névvel (pl. URL)
- **mozgatási** (migration): szerver fizikai mozgatása ne érintse a felhasználót
- **relokációs**: a kliens ne tudjon arról, hogy másik kiszolgálóra váltott a rendszer (pl. váltás egyik adatbázisról a másikra, vagy wifi esetén séta -> másik router)
- **replikációs**: felhasználó ne lássa, hogy melyik replikátumból kapja az adatot
- **konkurencia**: az alkalmazásnak nem kell tudnia, hogy más is használja az erőforrást
- **hibatűrés**: felhasználó ne lássa a hibát

De: hardver hibákat elfedni nehéz (pl. ha a szerver nem válaszol, nem tudni, hogy csak késik vagy összeomlott), az elfedés költséges

Cél: minél jobban elfedni

1.2.2. Nyíltság

- milyen nyelv? Java? C? -> kell egy jól definiált interfész. ez különböző erősségű lehet:
  - teljesség: megmondjuk, mit kell csinálni
  - semlegesség: nem mondjuk meg, hogyan csinálja
- hordozhatóság: egyik gépről a másikra (UNIX -> Windows)
- műveletek közötti átjárhatóság (pl. komponens csere után is működik a rendszer)

Megvalósítása: interfésszel, amely függetleníti a rendszert az alatta levő környezet heterogenitásától (különböző hardver, platform, nyelv).

Célkitűzések, amiket támogatni kell:

- milyen konzisztenciát követelünk meg a kliens által cache-elt adatoknál?
- milyen műveleteket engedünk meg a letöltött kódnak?
- milyen QoS követelményeket állítunk a változó sávszélesség alapján?
- milyen titkosítást követelünk meg a kommunikációnál?

Idális esetben az elosztott rendszer csak mechanizmusokat biztosít:

- cache-elés (dinamikus) szabályozásának megengedése
- különböző bizalmi szintek a hordozható kódokhoz
- adatfolyamonként állítható QoS paraméterek
- különböző titkosító algorimusok

1.2.3. Skálázhatóság

- **méretbeli**: felhasználók, processzek száma
- **földrajzi**: gépek közötti távolság
- **adminisztratív**: adat tárolási/kiadási stratégiák összehangolása

Az első megoldható erős gépekkel, az utóbbi kettőnél van nehézség.

Megvalósítás:

- kommunikációs késleltetés elfedése: aszinkron kommunikációval (nem vár a válaszra, közben dolgozik tovább, amíg tud), a bejövő válasz független feldolgozásával. Nem minden alkalmazás illik erre a modellre.
- elosztás: számítási feladat átadása a kliensnek (pl. Java applet), decentralizált elnevezési szolgáltatás (DNS) és információs rendszer (WWW)
- replikálás, cache-elés: replikált fájl és adatbázis szerverek, tükrözött weboldalak, web cache a böngészőben és proxy-kban, fájl cache szerver és kliens oldalon

De: meg kell oldani a konzisztenciát, a szinkronizálást. A teljes szinkronizálás erőforrás igényes. Az inkonzisztencia tolerálhatósága alkalmazásfüggő.

Hibás feltételezések:

- a hálózat megbízható, biztonságos, homogén
- a topológia fix
- késleltetés és a szállítási költség nulla, sávszélesség végtelen
- egy adminisztrátor van

2. Az elosztott rendszerek típusai

- számítási
- információs
- beágyazott

2.1. Számítási rendszerek

- nagy számítási kapacitást adnak algoritmusok alá
- típusai
  - **cluster**: hasonló gépek LAN-on összekapcsolva, 1 vezérgép osztja ki a feladatokat
  - **grid**: különböző gépek bárhonnan, lassabb kommunikáció, de több lehetőség, rétegek: application, collective, connectivity, resource, fabric

2.2. Információs rendszerek

- több adatbázis használatához biztosít egységes interfészt (akár különböző típusúakhoz is)
- tranzakciók (=atomi műveletek), ACID tulajdonságok (atomicity, cons., isolation, durab.)
- szubtranzakciók (pl. airline és hotel)
  - egyik lehetőség: TP Monitor az adatbázis szerverek fölé
  - másik lehetőség: kommunikációs réteg, nagyobb szabadságot ad
    - RPC (Remote Proc. Call) – adatbázisnak aktívnak kell lennie
    - MOM (Msg Oriented Mw) – megvárja, míg aktív lesz

2.3. Beágyazott rendszerek

- kicsi node-ok, gyakran nagyobb rendszerbe ágyazva
- követelmények:
  - szolgáltatások, információk megosztása
  - ad hoc: nincs fix séma/topológia
  - környezet változásaihoz könnyen tudjon alkalmazkodni
- pl.
  - intelligens ház
  - egészségmonitorozó rendszer, szenzorokkal (sok kérdés: tárolás hol-hogyan, mikor figyelmeztessen, hogyan kezelje az adatvesztést, hogyan kommunikáljon)
- szenzorok tulajdonságai:
  - lehet saját tárolójuk, vagy egy központi tárolóra küldhetik az infót
  - lehet elemmel vagy anélkül működő

Diasor #2 – Architektúrák

3. Architektúra stílusok

- rétegelt, információ a rétegek közt áramlik le-föl (pl. OSI hálózatmodell)
- objektum alapú felépítés, metódushívásokkal kommunikálnak
- event bus és publish/subscribe mechanizmus a komponensek között (anonimitás)
- megosztott adattér (anonimitás és aszinkronitás)

4. Elosztott rendszerek különböző architektúrái

4.1. Centralizált

- kliens-szerver modell
- **szerver**: szolgáltatást nyújtó processz
- **kliens**: szolgáltatást használó processz
- különböző gépen lehetnek
- a kliensek a kérés-válasz modellt követik
- alkalmazás rétegek:
  - UI
  - feldolgozó
  - adat
- pl. keresőoldal: UI -> query generator -> weblap adatbázis -> weblap metaadatok -> rangsorolás -> HTML felépítés -> UI
- architektúra lehet
  - egyrétegű: minden egy gépen
  - kétrétegű: kliens-szerver
  - többrétegű: minden réteg külön gépen
- rétegek határai változhatnak kliens-szerver között

4.2. Decentralizált (P2P)

- Strukturált P2P
- Strukturálatlan P2P
- Hibrid P2P
  - centralizált/decentralizált
  - strukturált/nem strukturált

Mindhárom eset overlay hálózatokkal dolgozik, mely a node-ok fölé épül.

4.2.1. Strukturált

- ring
  - csomópontok, melyek a nála kisebb értékű node-okat kezelik
  - lookup(key) -> csomópont
  - fájlmegosztás, tartalomlekérés, szolgáltatás
  - ha egy node kiesik, újraszervezés kell (pl. ha 1-es kiesik, a 4-es veszi át a 0-s node-ot), szomszédos csomópontokat kell csak figyelni

- koordinátás/téglalapos rendszer
    - egy csomópont egy területért felel
    - minél nagyobb terület -> annál több kérést kell kiszolgálnia
    - új node esetén kettévágjuk a téglalapot
    - kiesés esetén egyesítünk

4.2.2. Nem strukturált

- random felépítés
- egy node csak egy részét ismeri a hálózatnak
- kérést el kell küldenie minden általa ismertnek
- ha egy node kiesik, más node-ok elérhetetlenné válhatnak
- bizonyos időközönként a node-ok infót cserélhetnek egymással, bővítik ismereteiket a hálózatról
- robosztus

4.2.3. Overlay hálózatok

- 2 réteg
    - alsó: random felépítés
    - felső: strukturált felépítés
- az alsó réteg random node-okkal látja el a felsőt, a felső pedig eldönti, hogy melyik hivatkozást tartja meg
- kialakul egy tórusz (fánk) alakú hálózat, melyben mindenki össze van kötve a legközelebbi szomszédjával
- (tórusz elképzelése: papírap hengerré formálása, majd a henger kör alakúra görbítése)

4.2.4. Superpeer

- funkcionalitás/erőforrás szempontjából néhány peer lehet speciális
- pl. nyomtató (ne kérjünk tőle fájlt), vagy kapcsolatépítő, indexelő, állapotfigyelő node

4.2.5. Hibrid

- kliens-szerver + P2P
    - kliensek, tartalom megosztó, edge szerverek
    - edge szerver: tudja, [hogy ki tudja,] hogy mi hol van, optimális utat is adhat
- kliens-szerver + P2P: torrent
    - tracker: elérhető tartalmak, seederek nyilvántartása
    - centralizált: weboldal, tracker
    - decentralizált: seedek
    - kliens kapcsolódik a weboldalra, letölti a torrent fájlt, azzal kapcsolódik a tracker-re, aki megmondja, mely node-októl éri el a darabokat, onnantól közvetlenül azokkal kommunikál

5. A köztesréteg kapcsolata az elosztott rendszer architektúrájával

Gond: az alkalmazást konkrét architektúrára írni nem mindig jó, mert változhat.

Kell: alkalmazkodó köztes réteg (middleware).

Interceptor: megszakító. Megszakítja az általános végrehajtást távoli objektum hívásakor, és a konkrét rendszerhez igazodva hajtja végre a hívást. Van kérés szintű (app-> obj mw) és üzenet szintű (obj mw -> os) interceptor.

5.1. Adaptív middleware

- **válasszuk szét** az alap és speciális funkciókat, hogy külön lehessen módosítani (pl. extra: hibakezelés, biztonság)
- engedjük, hogy a programok **vizsgálják önmagukat** és alkalmazkodjanak a változásokhoz (Java Reflection API)
- az elosztott alkalmazást **komponensekből** építsük fel, melyek cserélhetőek, ha szükséges – komplex lehet a függőségek miatt

Kérdés: a szoftver alkalmazkodjon, vagy a rendszer?

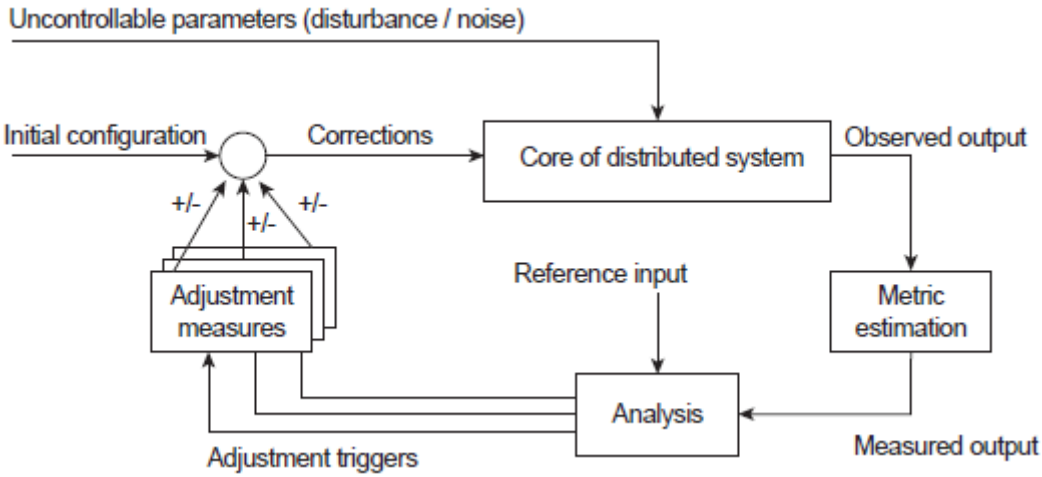
6. Ön-szervezés elosztott rendszerekben

Az automatikus alkalmazkodást önszervezéssel lehet megoldani.

Önkonfiguráló, önkezelő, öngyógyító, önoptimalizáló, ön* rendszerek.

6.1. Feedback Control Model

Az ön* rendszerek sok esetben a Feedback Control Model-t követik:



Elemzi a paramétereket, tulajdonságokat, reference inputhoz viszonyítja a mért adatokat, és optimalizálja a rendszert. Vannak nem kontrollálható paraméterek is.

6.2. Globule

Tartalommegosztó rendszer, ahol az önoptimalizálási döntések egy költségfüggvény alapján történnek: súlyozva összegzi a paraméterek értékeit.

Diasor #3 – Processzek

7. Processzek és szálak nem elosztott és elosztott rendszerekben

7.1. Objektumok

- **Processzor**: utasításhalmazt biztosít és lehetőséget, hogy egy utasítássorozatot végrehajtson.
- **Szál**: minimális szoftver processzor, mely utasításhalmazt biztosít. Az állapota elmenthető, a szál megállítható és később az állapota betölthető, a végrehajtása folytatható.
- **Processz**: minimális szoftver processzor, mely szálakat tud futtatni. A szálak utasítássorozatok végrehajtását jelentik, az adott szál saját környezetben.

7.2. Környezetek (contexts)

- **Processzor környezete**: minimális számú érték a processzor regisztereiben, melyek az utasítássorozat végrehajtásához kellenek.
- **Szál környezete**: minimális számú érték a regiszterekben és a memóriában, az utasítássorozat végrehajtásához (processzor környezete + állapot).
- **Processz környezete**: minimális számú érték a regiszterekben és a memóriában, szálak futtatásához (szál környezete + legalább az MMU regiszter tartalma).

7.3. Megfigyelések

- a szálak egy címterületen vannak, a szálak környezetének váltása végrehajtható OS-től függetlenül
- processz váltás sokkal költségesebb, mert OS hívásokkal jár
- szálak létrehozása és felszabadítása olcsóbb, mint processzek esetén

7.4. Megoldások

- user szinten: ha egy szál blokkolva van -> blokkolja a processzt is
- kernel szinten: ha egy processz blokkolva van, az OS egy másik processzt ütemez (drága)
- hibrid megoldás (Solaris): lightweight (könnyűsúlyú) processzek, melyek futtatják a szálakat. Ha egy szál blokkolódik, blokkolja az LWP-t, az OS egy másik LWP-t ütemez, amelyiknek van futó szála. Egy LWP-hez bármilyen user szintű szál hozzáköthető.

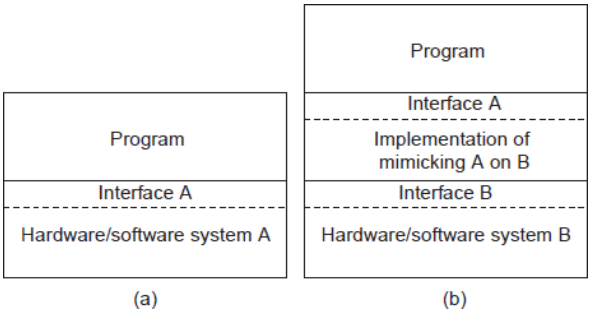
7.5. Szálak elosztott rendszerekben

- többszálú webkliens
  - a HTML oldal által igényelt fájlok lekérését külön szálakon indítja el (blokkoló HTTP kérések)
  - amikor mindegyik fájl megérkezett, megjeleníti a böngészőben
- többszálú RPC hívások
  - a kliens sok RPC hívást végez, mindegyiket külön szálon
  - megvárja, míg mindegyik válasz megérkezik
  - ha különböző szervereket hívott, akkor akár lineáris gyorsítás is elérhető
- teljesítmény javítása
  - egy szál indítása gyorsabb, mint egy processz indítása
  - egyszálú alkalmazás nem használja ki a többprocesszoros rendszert
  - a hálózati késés elfedhető a kliensnél a párhuzamosítással

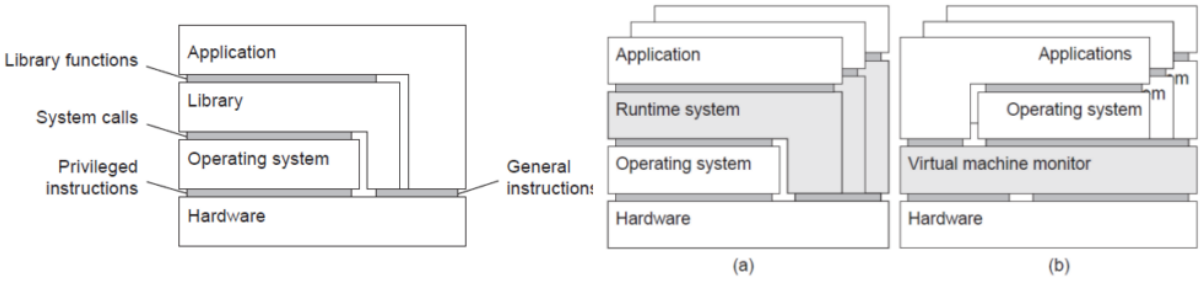
- jobb felépítés
    - egyszerű, jól érthető blokkoló utasítások egyszerűsítik a struktúrát
    - többszálú alkalmazások kisebbek és jól érthetőbbek

8. Virtualizáció

Fontos, mert a hardver gyorsan változik, a kódot hordozni kell, és a hibás komponenseket is el kell tudni különíteni.



Különböző szinteken is lehet virtualizálni, a rendszerkomponensek interfészeitől függően.



- **Process VM**: a programok egy hordozható kódra lesznek fordítva, melyet a VM futtat le. Csak a VM-et kell a különböző rendszerekre implementálni. (pl. JVM)
- **VM Monitor**: hardver utasításkészletet mutat a felette levő rétegnek, ami egy komplett oprendszer lehet a saját alkalmazásaival. Az utasításokat lefordítja az alatta levő hardvernek. (pl. VirtualBox, VMware)

9. Kliens folyamatok (+ példák)

A kliens oldali program jelentős része a (G)UI. A GUI biztosíthat alkalmazások közötti kommunikációt is:

- drag-n-drop: objektumok képernyőn történő mozgatásával interakciót váltunk ki más alkalmazással
- helyben szerkesztés: több alkalmazás integrálása a szerkesztőbe, pl. helyesírás ellenőrző, rajzoló eszközök

9.1. X window system

- Alapesetben: billentyűzet -> jel -> gép -> képernyő
- X esetében: billentyűzet -> jel -> X -> másik jel -> [[másik] gép -> másik jel] -> képernyő
- felhasználó terminálján a driverek felett: X kernel
- szerver oldalon: ablakkezelő

9.2. Átlátszóság

- access: RPC csonkok a kliens oldalon
- hely/migrációs: kliens ne tudja, hol fut le a parancs
- replikációs: akár több replikációt is megszólíthat a kliens, a kliens csonk kezeli le
- hibák: elrejtési mechanizmus a kliensben: elrejti a szerver- és kommunikációs hibákat

10. Szerver folyamatok (+ példák)

- A **szerver** egy olyan processz, amely a bejövő szolgáltatás kéréseket fogadja egy adott címen. A gyakorlatban egy szolgáltatás egy porthoz tartozik (1-1 kapcsolat).
- **Szuperszerver**: több porton figyel, több szolgáltatást nyújt. Bejövő kérésekre új processzt indít.
- **Iteratív szerver**: egyszerre csak egyetlen klienssel kommunikál
- **Konkurens szerver**: több klienssel is tud (szálak)

10.1. Kérés abortálása

- egyik lehetőség: **külön szál** a sürgős üzenetek figyelésére, mely magasabb prioritáson fut
- másik lehetőség: **külön portot** tart fenn a sürgős üzeneteknek

10.2. Mit figyeljen/tároljon a szerver?

- szerverleállásnál ugyanonnan lenne jó folytatni
- kliens és szerver állapotának konzisztenciáját fenn kéne tartani
- replikációk konzisztenciáját is
- **stateless** (állapot nélküli) **szerver**: nem tárol semmilyen infót
- **statefull** (teljes állapottal rendelkező) **szerver**: mindent tárol
    - fájl megnyitása
    - tudja, mit cache-elt a kliens

11. Szerver klaszterek

- 3 réteg
    - logikai switch: szétosztja a kéréseket
    - feldolgozó szerverek: feldolgozzák a kérést
    - adatbázisok
- az első rétegnek megbízhatónak kell lennie
- az első réteg leterhelődhet, ha minden kérés és válasz rajta megy keresztül
    - TCP-handoff (lepasszolás): miután a kérés eljutott egy szerverhez, a szerver a választ már direktben a kliensnek küldi vissza; a switch átírja a TCP header-ben a küldő címét a kliens címére

11.1. PlanetLab

- különböző szervezetek alkalmazásai, kutatáshoz
- meg kell oldani, hogy az alkalmazások ne ütközzenek -> virtualizáció
- rétegek (alulról):
  - hardver
  - Linux alapú OS
  - VServerek
  - VServerenként több processz
- vannak kiemelt VServerek, melyek kiváltságos processzeket futtatnak

12. Kód áthelyezése

- processz áthelyezése: gép -> gép
- pl. szerver klaszter esetén, ha egy szerver leterhelt, átadja a várakozó processzt egy másiknak
- kliens is futtathat kódot, pl. Java applet

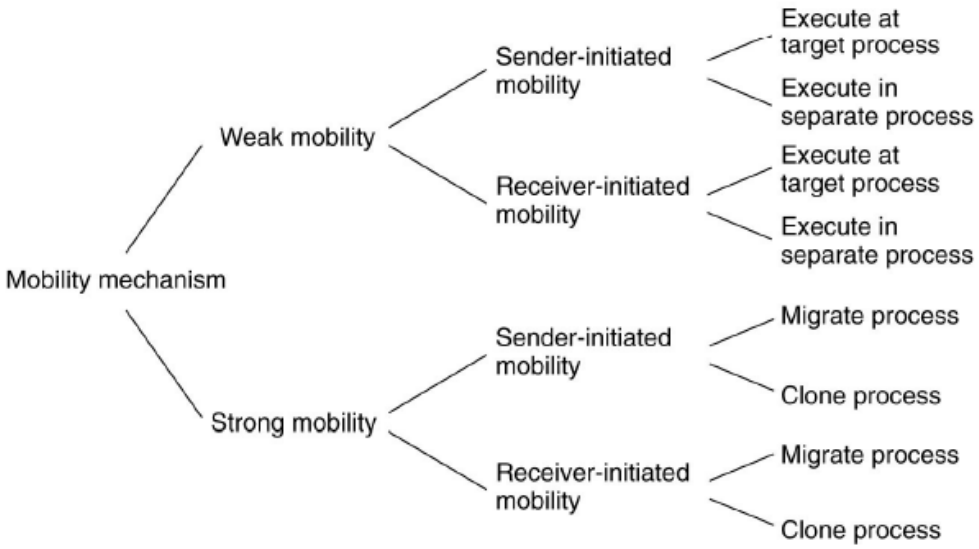
12.1. Komponensek

- kód szegmens
- adat szegmens (paraméter, részeredmény)
- végrehajtási állapot (hol tart a végrehajtás)

12.2. Modellek

- mobilitás lehet
  - **gyenge** (weak): kód és adat szegmens; újraindítja a kódot
    - végrehajthatja az adott processzben vagy újban
  - **erős** (strong): mindhárom rész; folytatja a végrehajtást
    - költöztetheti vagy klónozhatja a processzt
- mobilitást
  - kezdeményezheti a küldő
  - vagy kérheti a fogadó

12.3. Modellek gráfja



12.4. Erőforrások

- típusai
  - szigorúan csatolt (fix): nem migrálható (pl. hardver)
  - drágán migrálható (pl. nyomtató)
  - leválasztahtó és migrálható (pl. cache)
- erőforrás meghatározása
  - azonosító alapján konkrét erőforrás (pl. adatbázis)
  - érték alapján (pl. cache)
  - típus alapján (pl. nyomtató)

	Unattached	Fastened	Fixed
ID	MV (or GR)	GR (or MV)	GR
Value	CP (or MV, GR)	GR (or CP)	GR
Type	RB (or MV, GR)	RB (or GR, CP)	RB (or GR)

GR = Establish global systemwide reference
MV = Move the resource
CP = Copy the value of the resource
RB = Re-bind to a locally available resource

12.5. Heterogén rendszereknél

- nem biztos, hogy a célgép végre tudja hajtani a kódot
- a szál/processz környezet szerkezete függ a rendszertől
- megoldás: virtuális gép (az OS fölé)

Diasor #4 – Kommunikáció

13. Kommunikációs protokollok és típusok

13.1. OSI modell

- alacsony szintű rétegek
  - **fizikai réteg**: a biteket és azok továbbítását specifikálja
  - **adatkapcsolati réteg**: előírja, hogyan legyenek keretekbe szervezve a bitek
  - **hálózati réteg**: leírja, hogyan történjen a csomagok célbajuttatása a hálózatban
- **szállító réteg**: ez biztosítja igazából a kommunikációt az elosztott rendszernek
  - TCP: kapcsolat orientált, megbízható, folyam orientált
  - UDP: nem megbízható datagram kapcsolat
- **middleware layer**: gyakori szolgáltatásokat és protokollokat implementál
  - (OSI: session réteg és prezentációs réteg)
  - kommunikációs protokollok
  - adatok be-/kicsomagolása (marshalling)
  - elnevezési protokollok
  - biztonsági protokollok
  - skálázási mechanizmusok, pl. replikáció, cache-elés
- a fennmaradó dolgok már alkalmazás specifikusak

13.2. Kommunikáció

- szinkronitás
  - **szinkron**: vár, blokkol
  - **késleltetett szinkron**: küldés után még dolgozik, amennyit tud, utána blokkol csak
- tartósság
  - **ideiglenes**: az üzenetet eldobja, ha nem sikerül kézbesíteni
  - **tartós**: az üzenet el lesz tárolva, és amikor a célgép aktív lesz, megkapja (pl. mailbox)
- hol lehet szinkronizálni (pl. ACK-t, értesítést küldeni)
  - kérés fogadva
  - kérés célba juttatva
  - kérés feldolgozva

13.3. Kliens-szerver modell

- ideiglenes, szinkron kommunikáció
- mindkét oldalnak aktívnak kell lennie
- kliens blokkolódik, míg választ nem kap a szervertől
  - nem tud közben mást csinálni
- szerver csak bejövő kérésekre vár, azután feldolgozza azokat
  - a hibákat le kell kezelni azonnal, mert a kliens vár a válaszra

13.4. MOM

- tartós, aszinkron kommunikáció
- a processzek egymásnak üzennek, az üzenetek várakoztatva vannak
- a küldő nem vár a válaszra, közben dolgozik
- a köztes réteg hibatűrést biztosíthat

14. Távoli eljárás hívások (RPC)

- az egyszerű procedurális modellt könnyebb használni a programozóknak
- az eljárások izolált környezetben (black box) futnak
- nincs igazi ok, hogy egy eljárást ne másik gépen futtassunk
- a hívó és hívott közötti kommunikáció ezzel az eljárás hívásra épülő rendszerrel elfedhető

14.1. Kommunikáció menete

- Kliens meghív egy eljárást
- A **kliens csonk** felépíti az üzenetet (eljárás neve és paraméterei) és továbbítja az OS-nek
- OS átküldi az üzenetet a hálózaton
- Szerver OS fogadja az üzenetet, továbbítja a **szerver csonk**nak
- A szerver csonk kicsomagolja az adatokat és meghívja az eljárást a paraméterekkel
- Az eljárás visszatér a csonkba
- A szerver csonk felépíti az üzenetet (visszatérési érték) és továbbítja az OS-nek
- OS átküldi az üzenetet a hálózaton
- Kliens OS fogadja az üzenetet, továbbítja a kliens csonknak
- Kliens csonk kicsomagolja az értéket és továbbítja a kliensnek

14.2. Paraméterek üzenetté alakítása

- az üzenetté alakítás **bájtsorozattá alakítást** jelent
- a kliensnek és a szervernek egyező reprezentációt kell használnia
  - bájtsorrend
  - egyszerű típusok (int, float, char) reprezentációja
  - összetett típusok (tömb, unió) reprezentációja
- a kliensnek és a szervernek jól kell interpretálniuk az üzeneteket és le kell tudni fordítaniuk az adott géptől függő reprezentációra

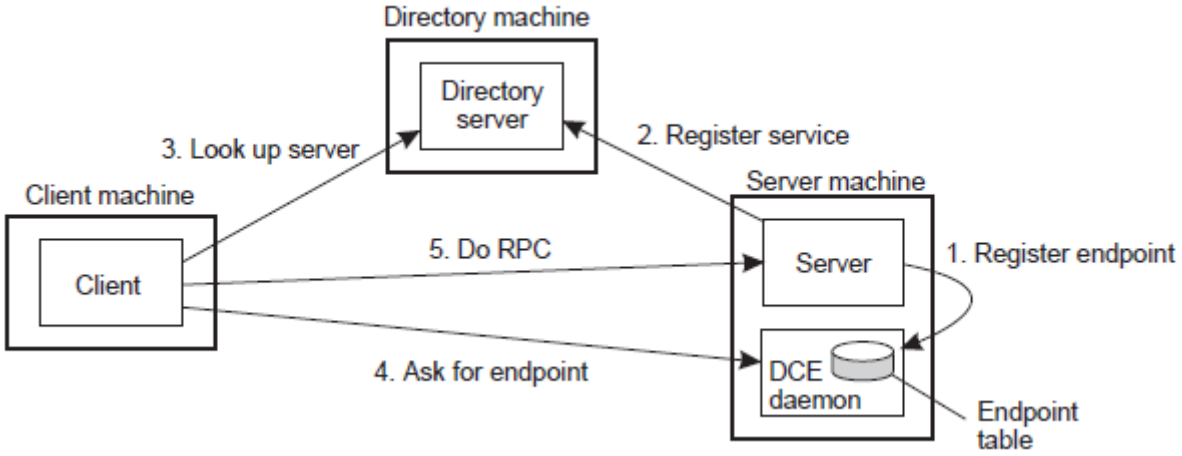
14.3. Paraméter átadás

- amíg az eljárás fut, semmit sem tudni a paraméterek pillanatnyi értékéről
- minden szükséges adatot át kell adni paraméterként
- távoli hivatkozás (remote reference) mechanizmus növeli a hozzáférési átlátszóságot
  - egységesíti a hozzáférést távoli adatokhoz
  - paraméterként átadható RPC hívásban

14.4. További RPC modellek

- aszinkron RPC
  - a szerver a hívás fogadásáról értesíti a klienst
  - a kliens dolgozhat tovább
- késleltetett szinkron RPC
  - aszinkron RPC és
  - amikor a szerver kész, visszaküldi a választ, mely megszakítja a klienst
  - a kliens ACK-t küld
  - megoldható az is, hogy a kliens nem blokkoló poll utasítással kérje le az eredményt

14.5. Kliens-szerver összeköttetés



15. Üzenet-orientált kommunikáció

15.1. Socket kommunikáció

- szokásos parancsok:
  - szerver: SOCKET, BIND, LISTEN, ACCEPT
  - kliens: CONNECT
  - kliens és szerver: SEND, RECEIVE, CLOSE
- szinkronizációs pont: CONNECT <-> ACCEPT
- read-write loop

Nem nagyon használatos megoldás üzenetküldésre, a socketek mellett: MPI (Message Passing Interface). Low-level parancsok.

15.2. MOM

- tartós, aszinkron kommunikáció, üzenet várakozási sorral a köztes rétegben (Message Queueing System)
- parancsok
  - PUT: hozzáad egy üzenetet a sorhoz
  - GET: blokkol, amíg az üzenetsor nem-üres lesz, és leveszi az első üzenetet
  - PULL: ha van üzenet, akkor leveszi az elsőt, nem blokkol
  - NOTIFY: beállít egy eseménykezelőt, arra az esetre, ha üzenet érkezne a sorra

15.3. Message Broker

- eddig az üzenetek formájában meg kellett egyezniük a feleknek
- a MB lehetővé teszi a heterogenitást: köztes rétegként felel az üzenetek átformálásáért, egyénre szabottan
- címeket és aliasokat is tárol

15.4. IBM WebSphere MQ

- üzenetsorok
- üzenetsor managerek
- az üzenetek üzenetsorok között mennek
- a csatornák végén Message Channel Agent-ek vannak, ezek felelnek a kapcsolatok kezeléséért és az üzenetek csomagolásáért
- irányítatlan csatornák
- az utak manuálisan vannak beállítva
- rounting table és alias table van

16. Adatfolyam(stream)-orientált kommunikáció

- időfüggő adatok: audio, video, animáció, szenzor adatok
- átviteli módok
  - **aszinkron**: nincs megkötés arra, mikor érkezzenek meg az adatok
  - **szinkron**: maximális célba érkezési idő a csomagokra
  - **izokron**: minimális és maximális idő a célba érkezésre (jitter)

16.1. Stream (adatfolyam) fogalma

- egy folyamatos **adatfolyam** olyan kapcsolat orientált lehetőség, mely támogatja az izokron átvitelt
- egyirányú
- többnyire egy küldő, több fogadó
- egyszerű stream: egy adatfolyam, audio vagy videó
- komplex stream: több adatfolyam, audio és kép, vagy sztereo audio

16.2. QoS

- folyamatosság
  - bufferelés kliens oldalon, X ideig
  - be kell lőni a méretet – ez hálózat és alkalmazás specifikus
  - akár dinamikusan, pl. Feedback Control Model-lel
- csomagelvesztés
  - 1,2,3,4|5,6,7,8|9,10,11,12|… helyett 1,5,9,13|2,6,10,14|3,7,11,15|4,8,12,16
  - így kisebb részek esnek ki, ha egy csomag elveszik
  - de így 16-os buffer kell lejátszás előtt, hogy sorba lehessen rakni
- több stream szinkronizálása
  - szerver egyetlen stream-be pakolja (multiplexálás) (pl. MPEG)
  - a kliens pedig demultiplexálja

17. Multicast kommunikáció

- egy üzenetet több állomásnak el kell küldeni (pl. UPDATE a replikátumoknak)
- lehet szekvenciálisan implementálni (A->B; A->C; A->D)
- de hatékonyabb fát építeni (gyökér, továbbító és levél csúcsok)
- algoritmus:
  - a kezdeményező generál egy azonosítót
  - meghatározza a következő node-ot
  - elküldi annak a kérést, az a node lesz a gyökér
  - ha P csatlakozni akar, küld egy join kérést a gyökérnek
  - amikor Q fogad egy join kérést
    - ha még nem fogadott korábban, akkor Q továbbító lesz, P pedig Q gyereke, a join kérés továbbítódik
    - ha tud a fáról, akkor P Q gyereke lesz, nincs továbbítás

17.1. Járvány alapú algoritmusok

- tegyük fel, hogy nincs olvasás-írás konfliktus
- a frissítési műveletek egy szerveren hajtódnak végre
- egy replikátum csak néhány szomszédnak küldi el a frissített állapotot
- frissítés lusta, nem azonnali
- minden frissítésnek el kell érnie minden replikátumot

17.1.1. Anti-entrópia algoritmus

- minden replikátum random választ egy szomszédot és állapotot cserél, a cél az azonos állapot elérése
- push, pull, push-pull
- push-pull esetén O(log(n)) műveletigénnyel frissül minden node

17.1.2. Pletyka alapú algoritmus

- a frissült replikátum szétkürtöli a frissítést a szomszédainak
- ha találkozik olyannal, aki már tudott a frissítésről, 1/k valószínűséggel nem kürtöli tovább

17.1.3. Törlés

- találkozik két állomás, egyiknél van adat, másiknál nincs – ki a frissebb?
- megoldás: a törlést frissítésként, **halotti bizonyítvány**ként küldjük
- de mikor töröljük a bizonyítványt?
  - időnként az összeset, mint egy GC
  - adjunk a bizonyítványoknak lejárati időt
- a bizonyítványoknak el kell érniük az összes szervert
- a globális algoritmus meg skálázhatósági gondokat szül

Diasor #6 – Szinkronizáció

18. Órák szinkronizálása

- bizonyos erőforrásokat nem lehet aszinkron módon elérni (pl. bankszámla)
- műveletek sorrendje is fontos
- időbélyeg kell
- de biztosítani kell a valós sorrendet -> szinkronizáció kell

18.1. Fizikai órák szinkronizálása

- megoldás: **UTC** (Universal Coordinated Time)
  - a cézium atom felezési ideje
  - kb. 50 ilyen óra van a világon
  - sugározzák rádióhullámon (kb. +/- fél ms késés lehet)
- bizonyos időközönként szinkronizálás (újraállítás)
  - dC: clock time, dt: UTC time
  - dC/dt az eltérés
  - 1-p <= dC/dt <= 1+p
  - cél: 2 óra ne különbözzön deltánál több időegységben
  - tehát: **minden delta/2p másodpercben szinkronizálni kell**
- ha az óra késett és előre kell állítani, az nem gond
- de ha sietett és vissza kell állítani az problémás
- **GPS** (Global Positioning System): 2 adótól vett távolság (két kör) pozitív tartományba eső metszéspontja

- szinkronizálás:
  - lekérjük az időt
  - lekérjük a távolságot -> késleltetési idő
  - a kettő különbsége a pontos idő
  - jelátvitel ideje (delta i): pontos idő – én időm + késés (delta r)
  - távolság: fénysebesség * jelátvitel ideje

18.2. Szinkronizálás hálózatban

- egyik megoldás:
  - time daemon: ez az etalon, ez szolgáltatja az időt
  - minden gép tőle kéri az időt, delta/2p másodpercenként
  - de jól kell kiszámolni a távolságot, beleszámítva a megszakítás kezelést és az üzenet fogadását!
- másik megoldás:
  - a szerver lekéri mindenkitől az óráját
  - átlagot számol
  - és mindenkinek megmondja, hogyan állítsa az óráját
  - de a visszafelé állítás még mindig probléma

19. Logikai órák szinkronizálása

- cél: csak előre állítsuk az órákat
- pontos idő nem fontos, csak a sorrend

19.1. A happened-before (előbb történt) reláció

- ha a és b ugyanannak a processznek két eseménye, és a előbb jön, mint b, akkor: a->b
- ha a egy üzenet elküldése és b ennek az üzenetnek a fogadása, akkor: a->b
- ha a->b és b->c, akkor a->c
- ez bevezeti események részleges sorbarendezését egy konkurens processzekkel dolgozó rendszerben

19.2. Logikai órák

- minden eseményhez rendeljünk egy **időbélyeget**: C(e)
- P1: ha a és b ugyannak a processznek két eseménye, és a->b, akkor C(a)<C(b)
- P2: ha a egy üzenet elküldése és b ennek az üzenetnek a fogadása, akkor is C(a)<C(b)
- de hogyan adjuk időbélyeget az eseményekhez, ha nincs globális óra?
- -> legyenek konzisztens logikai óráink, minden processznek egy

19.3. Megoldás (Lamport)

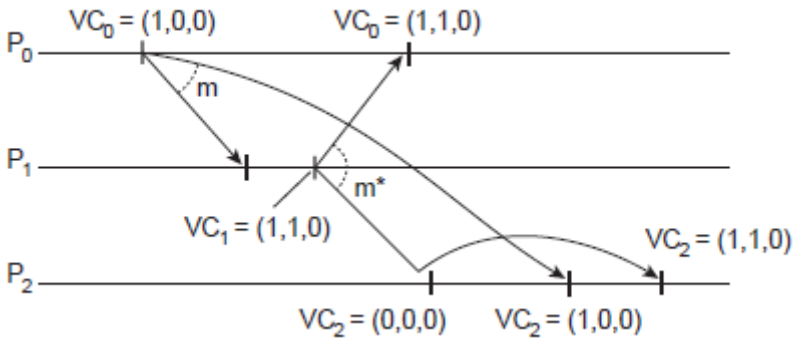
- minden Pi processz fenntart egy Ci **számlálót**
- 1: minden két egymást követő, Pi-ben történő esemény esetén: Ci++
- 2: minden m üzenethez, amit Pi küld, hozzárendelődik egy időbélyeg: ts(m)=Ci
- 3: amikor az m üzenet megérkezik Pj-be, Pj beállítja a saját óráját: Cj:=max(Cj, ts(m)), majd végrehatja az 1. pontot, mielőtt az üzenetet továbbítja az alkalmazásnak
- implementálása célszerű a köztes rétegben (hálózat <-> köztes réteg <-> alkalmazás)

19.4. Vektor órák

- az is kéne, hogy minden replikátumban azonos sorrendben fussanak le a tranzakciók (bankos példa)
- cél: okozati kapcsolatok nyomon követése az órák között
- minden processz nyilván tart egy **vektort**, a többi processz időbélyegeiről: VCi[]
- amikor Pi elküld egy üzenetet, megnöveli a VCi[i]-t és csatolja a vektort az üzenethez
- így a fogadó tudni fogja Pi időbélyegét
- amikor Pj fogadja az m üzenetet ts(m) időbélyeggel Pi-től:
  - frissíti a saját vektorát: VCj[k]:= max(VCj[k], ts(m)[k]) (minden k-ra)
  - és növeli a saját időbélyegét: VCj[j]++

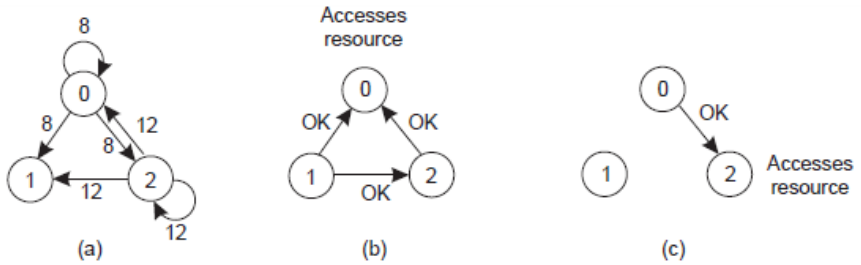
19.5. Javítás

- Pi csak akkor növeli VCi[i]-t, ha üzenetet küld
- Pj csak akkor állítja VCj-t, ha üzenetet fogad
- Pj **késlelteti** az üzenet továbbítását az alkalmazásnak, amíg
  - ts(m)[i] = VCj[i]+1 (amíg meg nem kapja azt az üzenetet, ahol csak egy eltérés van)
  - ts(m)[k]<=VCj[k] (k!=i)



20. Kölcsönös kizárás

- erőforrások védelme: egyszerre csak egy processz férjen hozzá
- centralizált megoldás: egy szerver egy sorral
- decentralizált megoldás
  - n db szerver, n/2+1 jogot kell megszerezni a hozzáféréshez
  - nagy kommunikációs költség
  - ha valaki kiesik, az nem gond, az n/2+1 jog megszerezhető még
- elosztott megoldás (Ricart & Agrawala)
  - OK választ küld egy processz, ha
    - nincs szüksége az erőforrásra
    - ha várakozik az erőforrásra, de alacsonyabb prioritással (magasabb számláló érték === később kérte)
  - minden más esetben a válasz halogatva van
  - egy processz akkor használhatja az erőforrást, ha mindenkitől OK-t kapott
  - gond, ha kisesik akár egy node is



- token ring
  - processzek rögzített sorrendben
  - egy token jár körbe
  - akinél tart a token, az használhatja az erőforrást
  - gond: ha elveszik a token (pl. kiesik egy node, akinek tovább kéne küldenie)

21. Globális állapot meghatározása

- egy node hogyan tudja megmérni a késleltetést egy másik node-hoz?
- megoldás: konstruáljunk egy geometrikus overlay hálózatot, ahol P és Q távolsága adja meg a P és Q között fellépő késleltetést
- a pozícionáláshoz k+1 pont kell egy k-dimenziós térben
- minden ponthoz távolságot kell számolni: di=sqrt((xi-xP)^2+(yi-yP)^2)
- gond:
  - a távolságok (késleltetések) ingadoznak
  - nem konzisztensek, pl. P-Q: 1, Q-R: 2, P-R: 3.2
- megoldás:
  - a pontok párosával kiszámolják a késleltetéseket
  - a P minimalizál
  - … (nem vágom)

22. Választó/szavazó algoritmusok

- cél: vezérgép kiválasztása
- zsarnok/erőszakos algoritmus
  - minden node-nak van egy súlya/prioritása
  - bárki kezdeményezheti, pl. ha észleli, hogy kiesett a vezér
  - nála nehezebb node-okat szólítja meg
  - a nehezebb node értesíti a könnyebbet, hogy átvette a választás feladatát, ő is indít egy választást
  - ha a node nem kap üzenetet a választás átvételéről, akkor ő lesz a vezér, és erről értesít mindenkit
  - ehhez mindenkinek ismernie kell a súlyokat
  - ha kommunikációs hiba volt csak (amikor kiesettnek tűnt a vezér) -> nem történik semmi

- választás ringben
    - mindenki számon tartja, ki a vezető
    - bárki indíthat választást: egy token fog körbemenni
    - mindenki hozzáfűzi az id-jét a tokenhez
    - amikor a token visszaér a kezdeményezőhöz, kiválasztja a legmagasabb prioritásút vezetőnek
    - és körbeküldi a tokent, hogy mindenki értesüljön róla
- választás overlay hálózatban
    - feszítőfa építés
    - kezdeményező megszólítja a szomszédait
    - azok kapcsolódnak a fához és megszólítják a szomszédaikat
    - egy node ahhoz a node-hoz kapcsolódik, akitől előbb jött kérés
- superpeer választás
    - normal node és superpeer között alacsony késleltetés kell
    - superpeereknek egyenletesen kell eloszlaniuk a rendszerben
    - a peerek előre meghatározott hányadának kell superpeernek lennie
    - egy superpeernek nem kell kiszolgálnia egy fix számnál több peert
- DHT-ben
    - ID egy tartománya lehet a superpeereké (hasonló, mint az IP maszkolás)

Diasor #7 – Konzisztencia és replikáció

23. Replikátumok(többszörözés) használata és konzisztenciával való kapcsolata

- Google szerverparkokat, replikációkat telepít a különböző országokba, az internetszolgáltatókhoz
- a böngésző cache is egyfajta replikáció
- konzisztensen kell tartani, biztosítani kell, hogy minden replikációban ugyanabban a **sorrendben** futnak le az utasítások
- read-write és write-write **konfliktus**, mint tranzakcióknál
- a teljes konzisztencia fenntartása költséges
- gyengébb konzisztencia nem igényel teljes szinkronizációt

24. Adat-centrikus konzisztencia model

- cél: elsődleges adattárak konzisztensen tartása
- fontos a sorrend, pl. vektor órákkal megoldható
- folyamatos konzisztencia
    - numerikus értékben eltérhet
    - conit: adat egység
    - minden replikátumban van egy vektor óra
    - a műveleteket összefésülik az időbélyegek alapján

- szekvenciális konzisztencia
  - 2 processz olvas 1 replikátumban
  - ugyanolyan sorrendben kell olvasniuk (is)
- okozati konzisztencia
  - megköveteljük a sorrendet, ha volt előtte read(a);write(b)
  - ha nem volt read, akkor mindegy
- műveletek csoportosítása
  - szinkronizációs változók
    - amíg az összes előző írás be nem fejeződött, nincs hozzáférési lehetőség
    - a lényeg, hogy az utasítássorozat eredménye látszódjon (az egyes utasításoké külön nem kell közben)
  - zárak: gyenge konzisztencia maga után vonja
    - elengedéskor a köztes réteg frissíti minden replikációban
    - megengedőbb olvasás: már frissült replikátumból lehet olvasni

25. Kliens-centrikus konzisztencia model

- cél: kliens lássa a módosításait, más replikátumból is
- monoton olvasás: minden írásnak végre kell hajtódnia olvasás előtt (pl. naptár, email)
- monoton írás: csak akkor lehet írni, ha az előző írás végbe ment (pl. egy program frissítése szerveren)
- olvasd az írásaid: amit írt, legközelebb mindenképpen tudnia kell olvasni (pl. weboldal készítés)
- olvasás utáni írás: előbb olvasás, aztán írás (pl. twitter posztra kommentelés)

26. Replikátumok kezelése

26.1. Replikátum elhelyezése

- replikátum és kliens távolságának minimalizálása
- autonóm (független) rendszerekhez keresünk közeli szervert
- hol a legolcsóbb

26.2. Tartalom elhelyezése

- nem kell minden adatot replikálni (pl. szótár)
- tartós replikátum: mindig meg van neki az adat
- szerver által kezdeményezett replikátum: ha egy adathoz sok kérés jön, akkor replikálni fogja
  - számolja a kéréseket az egyes fájlokhoz
  - ha a kérések száma kisebb, mint az alsó küszöb, akkor törli a fájlt
  - ha a kérések száma nagyonbb, mint a felső küszöb, akkor replikálja
  - ha a két küszöb között van, akkor költözteti
- kliens által kezdeményezett replikátum: cache
- egymásba ágyazott halmazok (belülről): tartós, szerver á. k., kliens á. k., kliensek

26.3. Tartalom elosztás

- modellek
  - csak jelzést küld a módosulásról (pl. cache)
  - átküldi a teljes fájlt
  - átküldi a módosítási műveletet
- frissítés iránya
  - push: szerver indítja, ha kérte a kliens, ha nem (tudnia kell a kliensekről)
  - pull: kliens kéri
- frissítés gyakoriság: lejáratok
  - régen frissült adat: nem is fog frissülni, hosszú lejárat
  - minél gyakoribban kéri a kliens, annál hosszabb lesz a lejárat
  - leterhelt szervernél rövid ideig tároljuk

27. Konzisztencia protokolok

27.1. Folyamatos

- súlyozzuk az írásokat
- origin: ahol először végrehajtódott
- mátrix: TW[i,j]: az Si által végrehajtott írás, melynek originja Sj
- aktuális érték: kezdeti érték + a mátrix átlójának összege
- érték az i. replikátumban: kezdeti érték + a mátrix i. sorának összege (szumma(k): TW[i,k])
- biztosítani kell, hogy az aktuális érték és a replikátumokban levő érték eltérése kicsi legyen
- pletyka alapú algoritmussal szét lehet kürtölni a változást

27.2. Elsődlegesség-alapú

- minden adatnak van egy elsődleges replikátuma
- az írási kérelem ide továbbítódik
- az fog update-elni mindenkit
- mindenki ACK-t küld az elsődleges replikátumnak a frissítés után
- módosítás lokális írásokkal
  - írási kérelemnél a replikátum átveszi az elsődlegességet
  - írás után azonnal vissza tud szólni a kliensnek
  - és ezután frissíti a többi replikátumot

27.3. Replikált írás

- az íráshoz megfelelő számú szavazat kell
- NR: olvasási lehetőség
- NW: írási lehetőség
- kell: NR+NW>N és NW>N/2

Diasor #5 – Elnevezések

28. Elnevezések

- node-okat azonosítani kell
- nevek
  - egyszerű nevek, jelentés nélkül, pl. random string
  - azonosítók
    - 1-1 kapcsolat az entitásokkal
    - azonosítót nem lehet újrafelhasználni

29. Flat(sima) elnevezések

- gond: nem tudjuk, kinek szól a kérés
- megoldások:
- broadcast: körbekérdezősködik
  - nagy hálózatnál nem jó, költséges
  - ha egy node elköltözött, forwarding pointert hagyhat hátra
  - ezt el lehet fedni, csak követni kell a pointereket
  - minél hosszabb a lánc, annál költségesebb és annál több esély van hibára
  - földrajzi hely miatt késleltetés is nőhet
- otthon-alapú elérés
  - egy home regisztrál egy elnevezési szolgáltatást
  - ő mondja meg, hol a node
  - a kliens a home-ot szólítja meg elöbb, a kliens cache-elhet is
  - de a home-nak fix-nek kell lennie és követnie az entitás helyét egész életében
- DHT (Distributed Hash Table)
  - minden node-hoz tartozik egy azonosító
  - minden entitáshoz egy kulcs
  - **logical ring** (az előtte levőket szolgálja ki)
  - finger táblák
    - minden p node-nak van egy FTp[] táblája
    - FTp[i] = succ(p+2^(i-1))
    - FTp[i] az első p-t követő node-ra mutat, de legalább 2^(i-1)-re
    - k megtalálásához p elküldi a kérést a q node-nak:
    - q = FTp[j] <= k < FTp[j+1]
    - vagy ha p < k < FTp[1], akkor FTp[1]-nek
- HLS (Hierarchical Location Service)
  - B-fa szerű felépítés, a belső csúcsokban a gyerekek infói
  - gond: a gyökér sok infót tárol
  - keresés egy levélből: ha tud a keresett node-ról, akkor lefelé megy, különben felfelé

30. Strukturált elnevezések

- névterek, mint a domain neveknél, elérési utaknál
- levél node: erőforrás/adat/fájl
- belső csúcs: **directory node**
- a node-okban sok infót tárolhatunk
- keresést a gyökérből kell indítani
    - de a feloldáshoz a gyökeret meg kell találni
- linkek
    - **hard link**: abszolút vagy relatív, feloldáshoz a gráfot kell követni
    - **soft link**: forwarding pointer szerű, fává alakítja a gráfot
- megvalósítás
    - **globális** szint: magas szintű node-ok, nem változnak, hozzájuk tartozik a DNS (pl. hu)
    - **adminisztrációs** szint: ritkán változó, szervezeti egységek DNS-ei (pl. elte, inf.elte)
    - **igazgatási** szint: kisebb egységek, gyakran változhatnak
- feloldás
    - **iteratív**: a NS-eket végigkérdezi sorban
    - **rekurzív**: a NS-ek kérdezik végig egymást, a válasz visszacsorog az elsőként megkérdezettbe, kliens csak a végeredményt kapja
        - NS-ek csak egy részt látnak, érdemes cache-lniük
- skálázhatóság
    - a magas szinteken sok kérés érkezik, de az adat ritkán változik: replikálás
    - földrajzi skálázhatóság megoldható a rekurzív feloldással
- DNS-ek használhatnak DHT-t
- el lehet osztani, pl. egy node csak olyan ID-jű node-ok adatait tartja nyilván, melyek ID-i csak olyan számjegyekből állnak, amiből a saját ID-je
- replikálás: a node-okat növekvő elemszámú szintekre bontjuk népszerűség szerint, a legnépszerűbbeket (legmagasabb szint) kell a sokszor replikálni

31. Attributum alapú elnevezések

- néha arra van szükség, hogy bizonyos tulajdonságok alapján keressünk egy node-ot
- gond: több attribútum értéket kell összehasonlítani node-onként és minden node-ot vizsgálni kell
- megoldás: adatbázis, mely a node-ok attribútumait tárolja, kombinálva egy strukturált névtér rendszerrel

Tételek vázlatosan

1. Az elosztott rendszer fogalma és főbb tervezési tulajdonságai
    - fogalma
    - 4 tulajdonság (erőforrás, átlátszóság, nyíltság, skálázhatóság)
    - átlátszóság 7 fajtája, megjegyzések
    - nyíltság fogalma, megvalósítása, célok
    - skálázhatóság 3 fajtája, megvalósítása
2. Az elosztott rendszerek típusai
    - számítási (cluster, grid)
    - információs (tranzakciók, szubtranzakciók 2 megvalósítása)
    - beágyazott (2 példa, szenzorok)
3. Architektúra stílusok
    - 4 stílus (rétegelt, objektum elvű, event bus, osztott adattér)
4. Elosztott rendszerek különböző architektúrái
    - centralizált: kliens-szerver (fogalmak, rétegek)
    - decentralizált:
        - strukturált: ring, téglalapos
        - nem strukturált: random
        - fogalmak: overlay hálózat, superpeer
        - hibrid: torrent
5. A köztesréteg kapcsolata az elosztott rendszer architektúrájával
    - miért van rá szükség
    - interceptor fogalma
    - adaptív mw 3 elve
6. Ön-szervezés elosztott rendszerekben
    - ön*
    - FCM
    - Globule
7. Processzek és szálak nem elosztott és elosztott rendszerekben
    - 3 objektum, 3 környezet
    - megfigyelések
    - 3 féle megoldás
    - szálak elosztott rendszerekben: webkliens, RPC, javítás
8. Virtualizáció
    - miért fontos
    - 2 féle elhelyezkedés
9. Kliens folyamatok (+ példák)
    - GUI
    - X window system
    - átlátszóság

10. Szerver folyamatok (+ példák)
    - szerver és szuperszerver
    - iteratív és konkurens szerver
    - kérés abortálásának 2 megoldása
    - mit figyeljen a szerver
11. Szerver klaszterek
    - felépítés, meggondolások
    - PlanetLab
12. Kód áthelyezése
    - 3 komponens
    - modellek
    - erőforrások fajtái, elérése
    - heterogén rendszerek
13. Kommunikációs protokolok és típusok
    - OSI modell
    - kommunikáció típusai 3 szempontból (szinkronitás, tartósság, szinkronizálás)
    - kliens-szerver
    - MOM
14. Távoli eljárás hívások (RPC)
    - fogalma
    - paraméterek átalakítása, átadása
    - további RPC modellek
    - kliens-szerver összeköttetés
15. Üzenet-orientált kommunikáció
    - socketek
    - MOM
    - Message Broker
    - IBM WebSphere MQ
16. Adatfolyam(stream)-orientált kommunikáció
    - 3 átviteli mód
    - stream fogalma
    - QoS biztosítása
17. Multicast kommunikáció
    - faépítés
    - járvány alapú: feltételek
        - anti-entrópia
        - gossiping
        - törlési probléma megoldása
18. Órák szinkronizálása
    - bevezetés
    - fizikai órák szinkronizálása (UTC, GPS)
    - szinkronizálás hálózatban: 2 mód

19. Logikai órák szinkronizálása
    - Happened-before reláció
    - Erre épülő logikai órák, és Lamport algoritmus
    - Vektor órák, algoritmus és javítása
20. Kölcsönös kizárás
    - 4 megoldás (cent, decent, elosztott, token ring)
21. Globális állapot meghatározása
    - távolság~késleltetés, gond, megoldás
22. Választó/szavazó algoritmusok
    - zsarnok
    - ring
    - overlay hálózatban (faépítés)
    - szuperpeer választási szempontok
23. Replikátumok(többszörözés) használata és konzisztenciával való kapcsolata
    - sorrend és konfliktusok
    - meggondolások
24. Adat-centrikus konzisztencia model
    - folyamatos, szekvenciális, okozati
25. Kliens-centrikus konzisztencia model
    - 4 modell: monoton read, monoton write, write->read, read->write
26. Replikátumok kezelése
    - replikátum elhelyezési szempontok (3 szempont)
    - tartalom elhelyezése (3 fajta replikátum)
    - tartalom elosztása (3 szempont: mód, irány, lejárat)
27. Konzisztencia protokolok
    - folyamatos (mátrixos)
    - elsődlegesség-alapú
    - replikált írás (quorum-os)
28. Elnevezések
    - 2 típusú név
29. Flat(sima) elnevezések
    - broadcast, fwd pointer
    - home-based
    - DHT
    - HLS
30. Strukturált elnevezések
    - bevezetés, linkek
    - megvalósítás
    - feloldás
    - skálázhatóság, egyéb meggondolások
31. Attribútum alapú elnevezések
    - probléma - megoldás