

## Záróvizsga tételek

### 19. Osztott rendszerek és konkurens programozás

#### Osztott rendszerek és konkurens programozás

**A, C:** Folyamat fogalma, elosztott rendszerek tulajdonságai és felépítése, elnevezési rendszerek, kommunikáció, szinkronizáció, konzisztencia.

**B:** Feladatok specifikációja biztonsági és haladási feltételekkel, absztrakt párhuzamos program tulajdonságai, megoldás fogalma, nevezetes feladatok megoldása párhuzamos és elosztott programokkal.

**2018:** Szálkezelés. Ütemezés, kontextusváltás. Race condition. Szinkronizáció. Blokkoló műveletek. Memória (stack és heap) használata a szálakban. A konkurens programozás nyelvi eszközei. Szinkronizációhoz és kommunikációhoz használható adatszerkezetek.

#### 1 Folyamatok, szálak

**Szál:** A szál (thread) a processzor egyfajta szoftveres megfelelője, minimális kontextussal. Ha a szálat megállítjuk, a kontextus elmenthető és továbbfuttatáshoz visszatölthető.

**Folyamat:** A folyamat (process vagy task) egy vagy több szálat összefogó nagyobb egység. Egy folyamat szálai közös memóriaterületen (címtartományon) dolgoznak, azonban különböző folyamatok nem látják egymás memóriaterületét.

**Kontextusváltás:** A másik folyamatnak/szálnak történő vezérlésátadás, így egy processzor több szálat/ folyamatot is végre tud hajtani.

**Szál vs. folyamat:** A szálak közötti váltáshoz nem kell igénybe venni az operációs rendszer szolgáltatásait, míg a folyamatok közötti váltásnál ahhoz, hogy a régi és új folyamat memóriaterülete elkülönüljön a memóriavezérlő (MMU) tartalmának jó részét át kell írni, amihez csak a kernel szintnek van joga. A folyamatok létrehozása, törlése és a kontextusváltás közöttük sokkal költségesebb a szálakénál.

#### 2 Elosztott rendszerek tulajdonságai és felépítése

**Elosztott rendszer fogalma:** Az elosztott rendszer önálló számítógépek olyan összessége, amely kezelői számára egyetlen koherens rendszernek tűnik.

##### 2.1 Az elosztott rendszer céljai, tulajdonságai

Az elosztott rendszer céljai a következők:

- Távoli erőforrások elérhetővé tétele

- Átlátszóság (transparency)
- Nyitottság (openness)
- Skálázhatóság

### 2.1.1 Átlátszóság

Az átlátszóság nem más, mint az erőforrásokkal kapcsolatos különböző információk elrejtése a felhasználó elől. Az alapján, hogy mit rejtünk el, többféle fajtája létezik:

Fajta	Angolul	Mit rejt el az erőforrással kapcsolatban?
Hozzáférési/elérési	Access	Adatábrázolás; elérés technikai részletei
Elhelyezési	Location	Fizikai elhelyezkedés
Áthelyezési	Migration	Elhelyezési + a hely meg is változhat
Mozgatási	Relocation	Áthelyezési + használat közben is történhet az áthelyezés
Többszörözési	Replication	Az erőforrásnak több másolata is lehet a rendszerben
Egyidejűségi	Concurrency	Több versenyhelyzetű felhasználó is elérheti egyszerre
Meghibásodási	Failure	Meghibásodhat és újra üzembe állhat

ábra 1: Az átlátszóság különböző típusai.

### 2.1.2 Nyitottság

A rendszer képes más nyitott rendszerek számára szolgáltatásokat nyújtani, és azok szolgáltatásait igénybe venni:

- A rendszerek jól definiált interfészekkel rendelkeznek.
- Az alkalmazások hordozhatóságát (portability) minél inkább támogatják.
- Könnyen elérhető a rendszerek együttműködése.

A nyitott elosztott rendszer legyen könnyen alkalmazható heterogén környezetben, azaz különböző hardvereken, platformokon, programozási nyelveken.

#### Implementálása:

- Fontos, hogy a rendszer könnyen cserélhető elemekből álljon.
- Belső interfészek használata, nem egyetlen monolitikus rendszer.
- A rendszernek minél jobban paraméterezhetőnek kell lennie.
- Egyetlen komponens megváltoztatása/cseréje lehetőleg minél kevésbé hasson a rendszer más részeire.

### 2.1.3 Skálázhatóság

Többféle jelentése van, 3 fontos dimenzió:

1. méret szerinti skálázhatóság: a felhasználók és/vagy folyamatok száma
2. földrajzi skálázhatóság: a csúcsok közötti legnagyobb távolság
3. adminisztrációs skálázhatóság: az adminisztrációs tartományok száma

Ezek közül a legtöbb rendszer a méret szerinti skálázhatóságot kezeli, ennek egy lehetséges megvalósítási módja erősebb szerverek használata. A másik kettőt nehezebb kezelni.

Technikák a skálázhatóság megvalósítására:

- A kommunikációs késleltetés elfedése azzal, hogy a válaszra várás közben más tevékenységet végzünk. Ehhez aszinkron kommunikáció szükséges.
- Elosztás: az adatokat és számításokat több számítógép tárolja/végzi (pl. amit lehet, a klienssel számoltatunk ki, elosztott elnevezési rendszerek használata, stb.)
- Replikáció/cache-elés: Több számítógép tárolja egy adat másolatait

A skálázhatóságnak ára van. Több másolat fenntartása inkonzisztenciához vezethet (ha módosítjuk az egyiket, az eltérhet a többitől). Ez globális szinkronizációval kikerülhető (minden egyes változtatás után az összes másolatot frissítjük), viszont a globális szinkronizáció rosszul skálázódik. Emiatt sok esetben fel kell hagynunk a globális szinkronizációval, ez viszont bizonyos mértékű inkonzisztenciát eredményez. Rendszerfüggő, hogy ez milyen mértékben megengedett. A cél az, hogy az inkonzisztencia mértéke a megengedett szint alatt maradjon.

## 2.2 Elosztott rendszerek típusai

Főbb típusok:

- Elosztott számítási rendszerek:
- Elosztott információs rendszerek
- Elosztott átható rendszerek

### 2.2.1 Elosztott számítási rendszerek

Célja számítások végzése nagy teljesítménnyel.

**Cluster (fürt):** Lokális hálózatra kapcsolt számítógépek összessége. Homogén rendszer (ugyanaz az oprendszer, hardveresen hasonlóak), központosított vezérléssel (általában egy gépre).

**Grid (rács)** Nagyméretű hálózatokra is kiterjedhet, akár több szervezeti egységen is átívelhet. Heterogén architektúra jellemzi.

**Cloud(felhő):** Többrétegű architektúra: hardver, infrastruktúra, platform, alkalmazás.

### 2.2.2 Elosztott információs rendszerek

Az elsődleges cél általában adatok kezelése, illetve más információs rendszerek elérése. Például tranzakciókezelő rendszerek.

A tranzakció adatok összességén (pl. egy adatbázison, adatbázis objektumon, stb.) végzett művelet (lehetnek részműveletei). A tranzakciókkal szemben az alábbi követelményeket szokás támasztani (ACID):

- Oszthatatlan, elemi (atomicity): Vagy a teljes tranzakció végbemegy minden részműveletével, vagy az adattárház egyáltalán nem változik.
- Konzisztens (consistency): Az adattárra akkor mondjuk, hogy érvényes, ha bizonyos, az adott adattárra megfogalmazott feltételek teljesülnek. Egy tranzakció konzisztens, ha érvényes állapotot állít elő a tranzakció végén.

- Elkülöníthető, sorosítható (isolation): Egyszerre zajló tranzakciók olyan eredményt adnak, mintha egymás után hajtottak volna végre.
- Tartósság (durability): Végrehajtás után az eredményt tartós adattárolóra mentjük, így az összeomlás esetén visszaállítható.

## 2.3 Elosztott rendszerek felépítése

Alapötlet: A rendszer elemeit szervezzük logikai szerepük szerint különböző komponensekbe, és ezeket osszuk el a rendszer gépein.

### 2.3.1 Központosított architektúrák

**Kliens-szerver modell:** Egyes folyamatok (szerverek) szolgáltatásokat ajánlanak, míg más folyamatok (kliensek) ezeket a szolgáltatásokat szeretnék használni. A kliens kérést küld a szervernek, amire a szerver válaszol, így veszi igénybe a szolgáltatást. A kliens és szerver folyamatok különböző gépeken lehetnek.

### 2.3.2 Többrétegű architektúrák

Az elosztott információs rendszerek gyakran három logikai rétegre (layer vagy tier) vannak tagolva:

- Megjelenítés: az alkalmazás felhasználói felületét alkotó komponensekből áll.
- Üzleti logika: az alkalmazás működését írja le konkrét adatok nélkül
- Perzisztencia: az adatok tartós tárolása

### 2.3.3 Decentralizált architektúrák

**Peer-to-peer (P2P):** A csúcsok (peer-ek) között többnyire nincsenek kitüntetett szerepűek.

**Overlay hálózat:** A gráfban szomszédos csúcsok fizikailag lehetnek távol egymástól, a rendszer elfedi, hogy a közöttük lévő kommunikáció több gépen keresztül zajlik. A legtöbb P2P rendszer overlay hálózatra épül.

**P2P rendszerek fajtái:**

- Strukturált P2P: A csúcsok által kiadott gráfszerkezet rögzített. A csúcsokat valamilyen struktúra szerint overlay hálózatba szervezzük és a csúcsoktól az azonosítójuk alapján lehet szolgáltatásokat igénybe venni. Pl.: elosztott hasítótábla (DHT).
- Strukturálatlan P2P: Az ilyen rendszerek igyekeznek véletlen gráfstruktúrát fenntartani. Mindegyik csúcsnak csak részleges nézete van a gráfról. Minden  $P$  csúcs időnként véletlenszerűen kiválaszt egy  $Q$  szomszédot.  $P$  és  $Q$  információt cserélnek és elküldik egymásnak az általuk ismert csúcsokat.
- Hibrid P2P: néhány csúcsnak speciális szerepe van

**Superpeer:** Olyan csúcs, aminek külön feladata van, pl. kereséshez index fenntartása, a hálózat állapotának felügyelete, csúcsok közötti kapcsolatok létrehozása.

## 3 Elnevezési rendszerek

Az elosztott rendszerek entitásai a kapcsolódási pontjaikon (access point) keresztül érhetőek el. Ezeket távolról a címük azonosítja, amely megnevezi az adott pontot.

Célszerű lehet az entitást a kapcsolódási pontjaitól függetlenül is elnevezni. Az ilyen nevek helyfüggetlenek (location independent).

**Egyszerű név:** Nincs szerkezete, tartalmaz véletlen szöveg. Csak összehasonlításra használható.

**Azonosító:** Egy név azonosító, ha egy-egy kapcsolatban áll a megnevezett entitással, és ez a hozzárendelés maradandó, azaz a név később nem hivatkozhat más egyedre.

### 3.1 Strukturálatlan nevek

#### 3.1.1 Egyszerű megoldások

**Broadcasting:** Kihirdetjük az azonosítót a hálózaton. Az egyed visszaküldi jelenlegi címét. Hátrányai:

- Lokális hálózatokon túl nem skálázódik.
- A hálózaton minden gépnek figyelnie kell a beérkező kérésre.

**Továbbítómутató:** Amikor az egyed elköltözik, egy mutató marad utána az új helyére.

- A kliens elől el van fedve, hogy a szoftver továbbítómутató-láncot old fel.
- A megtalált címet vissza lehet küldeni a klienshez, így a további feloldások gyorsabban mennek.
- Földrajzi skálázási problémák:
  - A hosszú láncok nem hibátűrőek.
  - A feloldás hosszú időbe telik.
  - Külön mechanizmus szükséges a láncok rövidítésére.

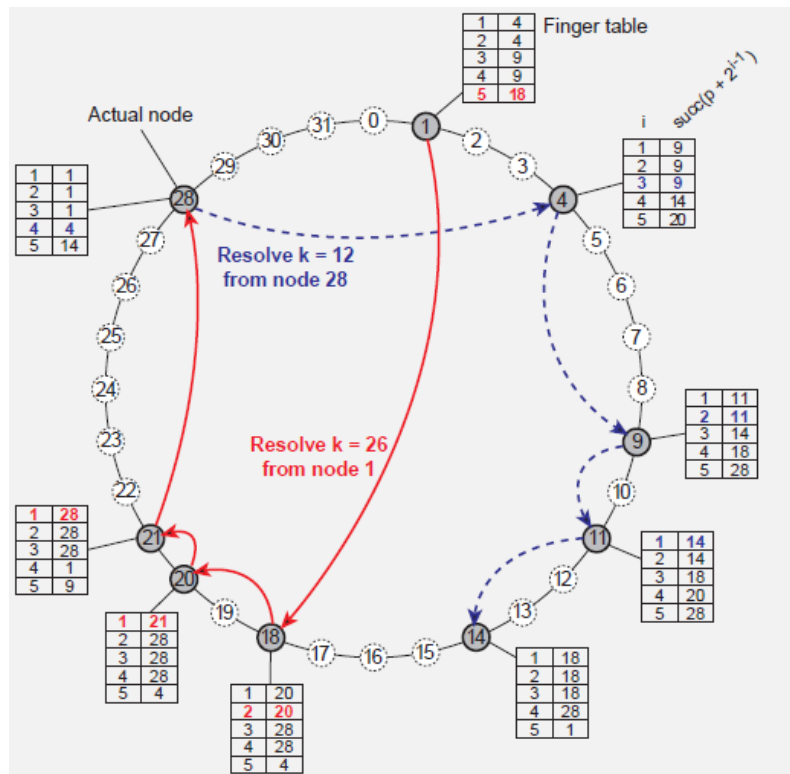
#### 3.1.2 Otthon alapú megoldások

**Egyrétegű rendszer:** Az egyedhez tartozik egy otthon, ez tartja számon az egyed jelenlegi címét. Az egyed otthoni címe (home address - HA) be van jegyezve egy névszolgáltatásba. Az otthon számon tartja a jelenlegi címet (foreign address - FA). A kliens az otthonhoz kapcsolódik, onnan kapja meg a címet.

**Kétrétegű rendszer:** Az egyes környékeken feljegyezzük, hogy mely egyedek tartózkodnak a közelben. A névfeloldás először ezt a jegyzéket vizsgálja meg és ha az egyed nincs a környéken, akkor kell az otthonhoz fordulni.

#### 3.1.3 Elosztott hasítótábla

Elosztott hasítótáblát (DHT) készítünk, ebben csúcsok tárolnak egyedeket. Az  $N$  csúcs gyűrű overlay szerkezetbe van szervezve. Minden csúcshoz hozzárendelünk egy  $m$  bites azonosítót, és mindegyik entitáshoz egy  $m$  bites kulcsot ( $N \leq 2^m$ ). A  $k$  kulcsú egyed felelőse az az  $id$  azonosítójú csúcs, amelyre  $k \leq id$ , és nincs köztük másik csúcs. Ezt a csúcsot a kulcs rákövetkezőjének is szokás nevezni:  $succ(k)$ . Mindegyik  $p$  csúcs egy  $FT_p$  finger table-t tárol  $m$  bejegyzéssel:  $FT_p[i] = succ(p + 2^{i-1})$ . Bináris (jellegű) keresést szeretnénk elérni, ezért minden lépés felezi a keresési tartományt. A  $k$  kulcsú egyed kikereséséhez (ha nem a jelenlegi csúcs tartalmazza) a kérést továbbítjuk ahhoz a  $j$  indexű csúcshoz, melyre  $FT_p[j] \leq k < FT_p[j + 1]$ , illetve, ha  $p < k < FT_p[1]$ , akkor is  $FT_p[1]$ -hez irányítjuk a kérést.



ábra 2: Példa DHT-re finger table-el.

### 3.1.4 Hierarchikus módszerek

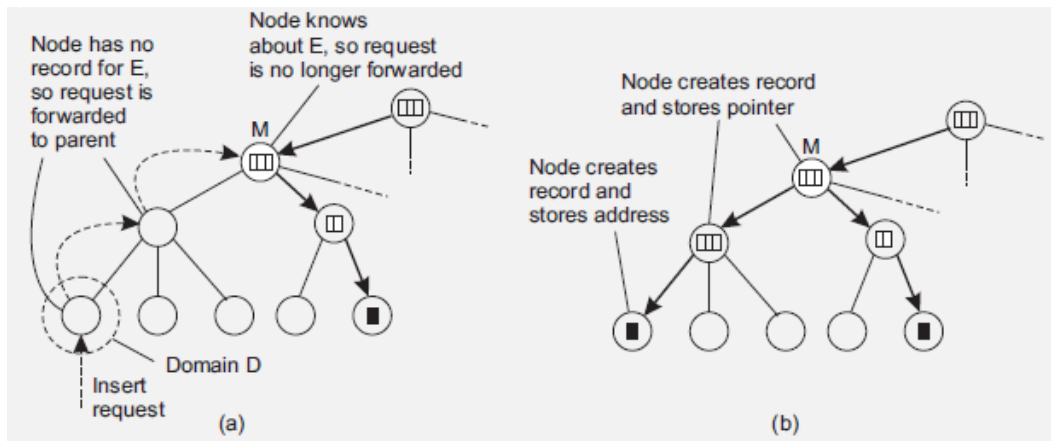
**Hierarchical Location Services(HLS):** A hálózatot osszuk fel tartományokra, és mindegyik tartományhoz tartozzon katalógus. Építsünk hierarchiát a katalógusokból.

A csúcsokban tárolt adatok:

- Az  $E$  egyed címe egy levélben található.
- A gyökértől az  $E$  leveléig vezető úton minden belső csúcsban van egy mutató a lefelé következő csúcsra az úton.
- Mivel a gyökér minden út kiindulópontja, minden egyedről van információja.

**Keresés a fában:** A kliens tartományából indul a keresés. Felmegyünk addig a fában, amíg olyan csúcsba nem érünk, amelyik tud  $E$ -ről, majd követjük a mutatókat a levélig, amely tudja  $E$  címét. Mivel a gyökér minden egyedet ismer, a terminálás garantált.

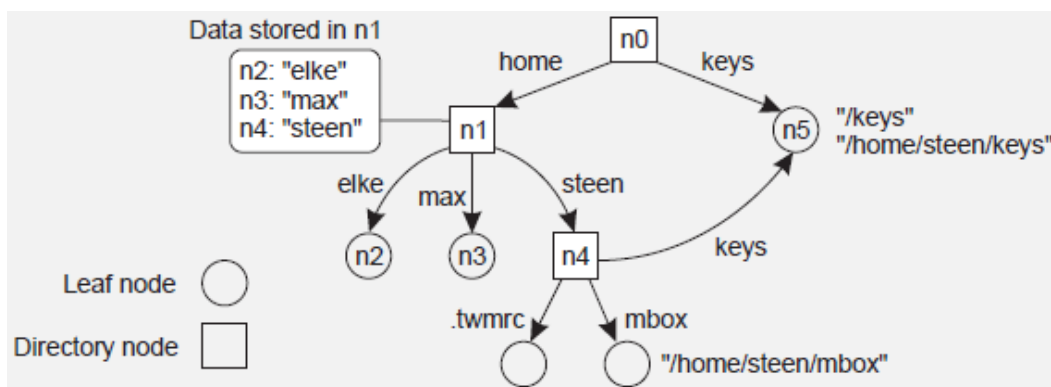
**Beszúrás a fában:** Ugyanaddig megyünk felfelé a fában, mint keresésnél, majd a belső csúcsokban mutatókat helyezünk el.



ábra 3: Beszúrás a fában HLS-nél.

### 3.2 Strukturált nevek

**Névtér:** Gyökeres, irányított, élcímkézett gráf, a levelek tartalmazzák a megnevezett egyedeket, a belső csúcsokat katalógusoknak vagy könyvtáraknak nevezzük. Az egyedhez vezető út címkeit összeolvasva kapjuk az egyed egy nevét. A bejárt út, ha a gyökértől indul, abszolút útvonalnév, ha belső csúcsból indul, relatív útvonalnév. Mivel egy egyedhez több út is vezethet, több neve is lehet.



ábra 4: Példa névtérre.

A névtér csúcsaiban (akár levélben, akár belső csúcsban) különféle attribútumokat is eltárolhatunk, pl. az egyed típusát, azonosítóját, helyét/címét, más neveit, stb.

**Névfeloldás:** Kiinduló csúcsra van szükség a névfeloldás megkezdéséhez. A gyökér elérhetőségét a név jellegétől függő környezet biztosítja, pl.:

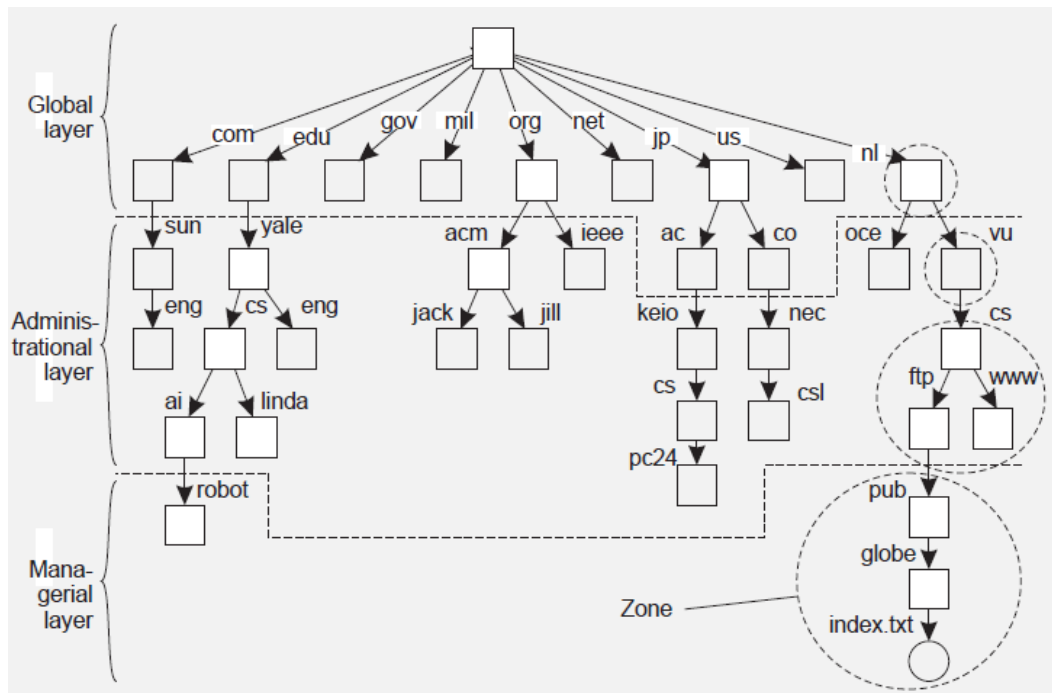
- www.inf.elte.hu : egy DNS névszerver
- /home/steen/mbox : a lokális NFS fájlserver
- 0031204447784 : a telefonos hálózat
- 157.181.161.79 : a www.inf.elte.hu webszerverhez vezető út

**Névtér implementációja - DNS:** Ha nagy névtérünk van, el kell osztani a gráfot a gépek között, hogy hatékonyra tegyük a névfeloldást és a névtér kezelését. Ilyen nagy névtér a DNS (Domain Name System).

A DNS névtérnek alapvetően 3 szintjét különböztetjük meg:

- Globális szint: Ide tartozik a gyökér és a felsőbb csúcsok (TLD-k, pl. országokhoz tartozó csúcsok - .hu, .uk, stb.). A szervezetek ezt közösen kezelik.

- Szervezeti szint: Egy-egy szervezet által kezelt csúcsok szintje (pl. elte.hu, stb.).
- Kezelői szint: Egy adott szervezeten belül kezelt csúcsok (pl. elte.hu-n belüli csúcsok)



ábra 5: A DNS névtér egy része.

**A névfeloldás különböző megközelítései:** DNS névtér esetén alapvetően két különböző névfeloldási megközelítést alkalmazunk:

- **Rekurzív névfeloldás:** A rekurzív névfeloldás során a névszerverek egymás között kommunikálva oldják fel a neveket, a kliensoldali névfeloldóhoz rögtön a válasz érkezik.
- **Iteratív névfeloldás:** A névfeloldást a gyökeri névszerverek egyikétől indítjuk. Az iteratív névfeloldás során a névnek mindig csak egy komponensét oldjuk fel, a megszólított névszerver az ehhez tartozó névszerver címét adja vissza (ha a kliensoldali névfeloldó megkapja ezt a címet, a következő komponens feloldását ettől a névszervertől kéri - ez addig megy, míg teljesen fel nem oldjuk a nevet).

**Skálázhatóság:** Mivel sok kérést kell kezelni rövid idő alatt, ezért a globális szint névszerverei nagy terhelést kapnának. Mivel a felső szinteken a gráf ritkán változik, ezért az ezeken a szinteken található csúcsok adatairól több szerveren is tarthatunk másolatot, így a keresést közelebből indíthatjuk (pl. van több gyökeri névszerver, a hozzánk legközelebbihez fordulunk).

### 3.2.1 Attribútumalapú nevek

Az egyedet sokszor kényelmes lehet tulajdonságai (attribútumai) alapján keresni, viszont ha bármilyen kombinációban megadhatunk attribútumértékeket, akkor a kereséshez az összes egyedet érintenünk kell, ami nem hatékony.

**X.500, LDAP:** A katalógusszolgáltatásokban az attribútumokra megkötések érvényesek (X.500 szabvány), amelyet az LDAP protokollon keresztül szokás elérni. Az elnevezési rendszer fastruktúrájú, élei attribútum-érték párokkal címeztek. Az egyedekre az útjuk jellemzői vonatkoznak, és további párokat is tartalmazhatnak.



## 4 Kommunikáció

### 4.1 Köztesréteg

A köztesrétegbe (middleware) olyan szolgáltatásokat és protokollokat szokás sorolni, amelyek sokfajta alkalmazáshoz lehetnek hasznosak és alapvetően a rendszer egyedei közötti összekötő kapocsként szolgálnak.

- Kommunikációs protokollok
- Sorosítás (szerializáció, marshalling), adatok reprezentációjának átalakítása
- Elnevezési protokollok az erőforrások megosztásának könnyítésére
- Biztonsági protokollok a kommunikáció biztonságosabbá tételére
- Skálázási mechanizmusok adatok replikációjára és gyorsítótárazására

### 4.2 A kommunikáció fajtái

A kommunikáció lehet:

- időleges (transient) vagy megtartó (persistent):
  - időleges: a kommunikációs rendszer elveti az üzenetet, ha az nem kézbesíthető
  - megtartó: a kommunikációs rendszer hajlandó huzamosabb ideig tárolni az üzenetet
- szinkron vagy aszinkron
  - szinkron: a küldő vár a válasza, addig blokkolódik
  - aszinkron: a küldő nem vár a válasza, hanem más tevékenységet folytat

#### 4.2.1 Kliens-szerver modell

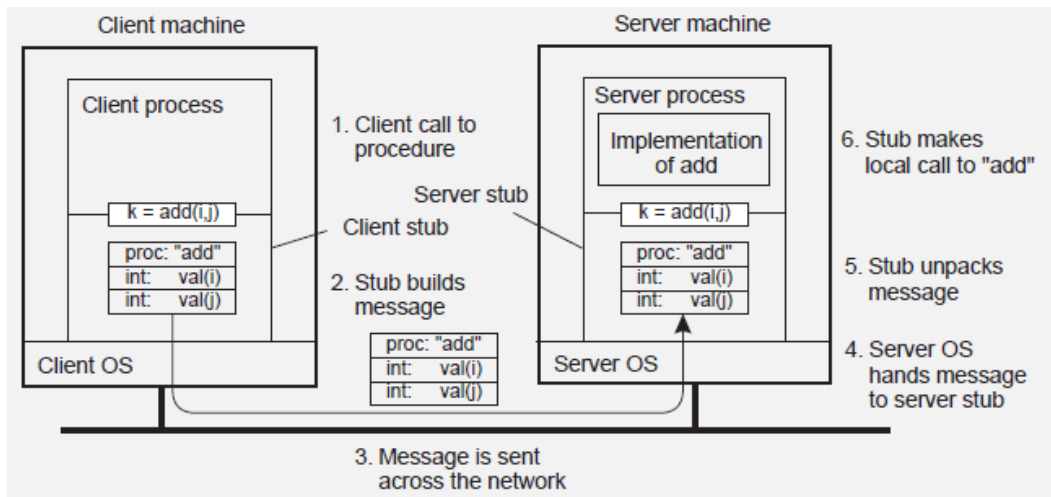
A kliens-szerver modell jellemzően időleges, szinkron kommunikációt végez, ahol a kliensnek és a szervernek egyidejűleg kell aktívnak lenni. A kliens a kérés küldése után blokkolódik, vár a szerver válaszára. A szerver csak a kliensek fogadásával és a kérések feldolgozásával foglalkozik.

#### 4.2.2 Távoli eljáráshívás (RPC)

A távoli eljáráshívásnál egy távoli gépen szeretnénk futtatni egy alprogramot. Ehhez hálózati kommunikáció szükséges, amit elfedünk egy eljáráshívással.

**A hívás lépései:**

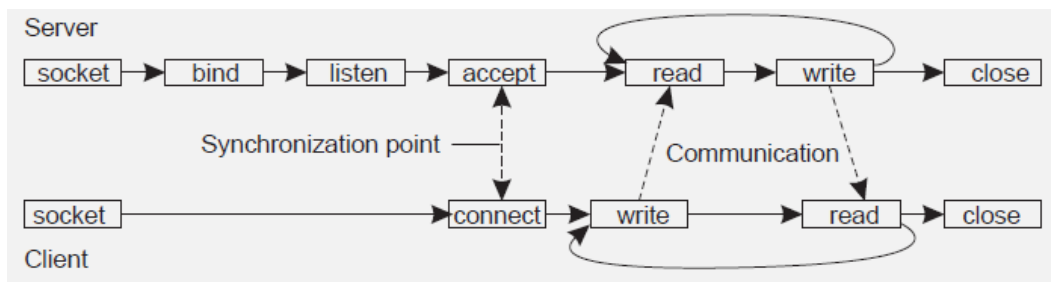
1. A kliensfolyamat lokálisan meghívja a klienscsonkot (client stub).
2. A klienscsonk becsomagolja az eljárás azonosítóját és paramétereit. Meghívja az oprendszert.
3. A lokális gép oprendszere elküldi a csomagot a távoli gép oprendszerének.
4. Az átadja az üzenetet a szervercsonknak (server stub).
5. A szervercsonk kicsomagolja az azonosítót és a paramétereket, amiket átad a szerverfolyamatnak.
6. A szerverfolyamat lokálisan meghívja az eljárást, megkapja a visszatérési értéket.
7. A visszatérési érték visszaküldése a kliensfolyamatnak hasonlóan történik, fordított irányban.



ábra 6: A távoli eljáráshívás lépései.

#### 4.2.3 Socket

Az időleges kommunikáció egy módja.



ábra 7: Kommunikáció socket-el.

#### 4.2.4 Üzenetorientált köztesréteg (MOM)

Az üzenetorientált köztesréteg (MOM - message-oriented middleware) egy megtartó, aszinkron kommunikációs architektúra. Segítségével a folyamatok üzeneteket küldhetnek egymásnak. A küldő félnek nem kell a válaszra várnia, addig foglalkozhat mással.

A MOM várakozási sorokat tart fenn a rendszer gépein. A kliensek az alábbi műveleteket használhatják a várakozási sorokra:

- PUT: Üzenetet tesz a sor végére.
- GET: Blokkol, amíg a sor üres, majd kiveszi az első üzenetet
- POLL: Lekérdezi, hogy van-e üzenet. Ha van, leveszi az első. Ha nincs, nem blokkol, folytatja a tevékenységét.
- NOTIFY: Kezelőrutint telepít a várakozási sorhoz, amely minden beérkező üzenetre meghívódik.

Az üzenetsorkezelő rendszerek feltételezik, hogy a rendszer minden eleme közös protokollt használ, azaz az üzenetek szerkezete és adatábrázolása megegyezik. A kérdés: mi van akkor, ha heterogén a rendszerünk? Erre szolgál az üzenetközvetítő (message broker), amely heterogén rendszerben gondoskodik a megfelelő konverziókról, azaz átalakítja az üzenetet a fogadó által használt formátumra. Általában proxy-ként is működik, azaz a közvetítés mellett más funkciókat is nyújt, pl. biztonsági funkciókat.

#### 4.2.5 Folyam (stream)

Az eddig tárgyalt kommunikációfajtákban közös, hogy az adategységek közötti időbeli kapcsolat nem befolyásolja azok jelentését, folyamatos médiánál (pl. audio, videó, szenzoradatok) viszont az adatok időfüggőek, ezért a kommunikáció időbeliségével kapcsolatban izokrón megkötést teszünk, ami felső és alsó korlátot is ad a csomagok átvitelének idejére.

**Folyam:** Ilyen izokrón adatátvitelt lehetővé tevő kommunikációs forma a folyam. Főbb jellemzői:

- Egyirányú
- Legtöbbször egy forrástól irányul egy vagy több nyelő felé
- A forrás és/vagy nyelő gyakran közvetlenül kapcsolódik olyan hardverelemekhez, mint pl. egy kamera, képernyő, mikrofon, stb.

**Főbb típusai:**

- Egyszerű folyam: egyfajta adatot továbbít, pl. egyetlen audiocsatornát, vagy csak videót.
- Összetett folyam: Többfajta adatot továbbít egyszerre, pl. videót többcsatornájú audióval (sztereó, 5.1, stb.). Az összetett folyam esetében biztosítani kell, hogy az alfolyamok a nyelőnél időben ne csússzanak el egymáshoz képest. Ennek egyik módja a szinkronizáció. Egy másik lehetséges módszer a multiplexálás és demultiplexálás. Ekkor a forrás egyetlen folyamot készít (multiplexálás). Itt az alfolyamok garantáltan szinkronban vannak egymással. A nyelőnél kell szétbontani a folyamot alfolyamokra (demultiplexálás).

**QoS:** A folyamokkal kapcsolatban sokfajta követelmény írható elő, ezeket összefoglaló néven a szolgáltatás minőségének (QoS - Quality of Service) nevezzük. Ilyen jellemzők például a következők:

- Az átviteli sebesség, azaz a bitráta.
- A folyam elindításának legnagyobb megengedett késleltetése.
- A folyam adategységeinek megadott idő alatt el kell jutniuk a forrástól a nyelőig.
- Remegés (jitter): az adategységek beérkezési idejének egyenetlensége. Ennek csökkentésének egy módja a puffereelés.

## 5 Szinkronizáció

### 5.1 Órák szinkronizálása

Néha a pontos időt szeretnénk megtudni, néha elég, hogy ha két időpont közül megállapítható, hogy melyik volt korábban. A világidő: UTC.

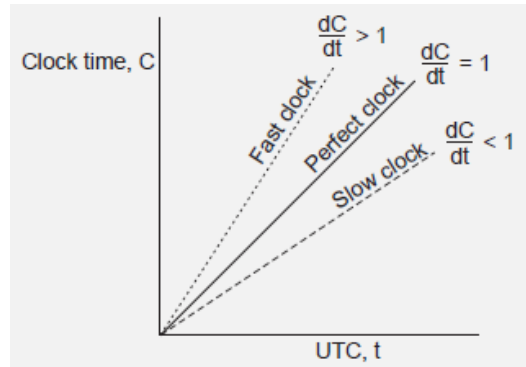
#### 5.1.1 Fizikai órák

**A fizikai idő elterjesztése:** Ha a rendszerünkben van UTC-vevő, az megkapja a pontos időt. Ezt a következők figyelembevételével terjeszthetjük el a rendszeren belül.

- A  $p$  gép saját órája szerint az idő a  $t$  UTC-időpillanatban  $C_p(t)$
- Ideális esetben az óra mindig pontos, azaz  $C_p(t) = t$  minden  $t$  UTC-időpillanatra. Másképpen fogalmazva az óra sebessége mindig 1, azaz  $dC/dt = 1$ .

- A valóságban  $p$  órája vagy túl gyors, vagy túl lassú, de viszonylag pontos:

$$1 - \rho \leq \frac{dC}{dt} \leq 1 + \rho$$



ábra 8: Az óra sebessége.

**Cristian algoritmus:** Csak megadott  $\delta$  eltérést akarunk megengedni az óra sebességében. Mindegyik gép egy központi időszerverről kéri le a pontos időt legfeljebb  $\frac{\delta}{2\rho}$  másodpercenként (ekkor tudunk  $\delta$  eltérésen belül maradni). Az órát nem a megkapott időpontra kell állítani: bele kell számolni, hogy a szerver kezelte a kérést és a válasznak vissza kellett érkeznie a hálózaton.

**Berkeley algoritmus:** Nem a pontos idő beállítása a cél, csak az, hogy a rendszeren belül minden gép ideje azonos legyen. Az időszerver időnként minden gép idejét bekéri, amiből átlagot von, majd mindenkit értesít, hogy a saját óráját mennyivel kell átállítania. Az idő egyik gépnél sem folyhat visszafelé, ezért ha valamelyik órát vissza kellene állítani, akkor ehelyett lelassítja az óráját addig, amíg a kívánt idő be nem áll.

### 5.1.2 Logikai órák

**Az előbb-történt reláció:** Az előbb-történt (happened-before) reláció az alábbi tulajdonságokkal bíró reláció. Annak jelölése, hogy  $a$  előbb történt, mint  $b$ :  $a \rightarrow b$ .

- Ha ugyanabban a folyamatban  $a$  előbb következett be, mint  $b$ , akkor  $a \rightarrow b$ .
- Ha  $a$  esemény egy üzenet küldése,  $b$  pedig ennek az üzenetnek a fogadása, akkor  $a \rightarrow b$ .
- Tranzitív: Ha  $a \rightarrow b$  és  $b \rightarrow c$ , akkor  $a \rightarrow c$ .

**Az idő és az előbb-történt reláció:** Minden  $e$  eseményhez időbélyeget rendelünk, ami egy egész szám. Jelölése:  $C(e)$ , és megköveteljük az alábbi tulajdonságokat:

- Ha  $a \rightarrow b$  egy folyamat eseményeire, akkor  $C(a) < C(b)$
- Ha  $a$  esemény egy üzenet küldése,  $b$  pedig ennek az üzenetnek a fogadása, akkor  $C(a) < C(b)$ .

Ha van globális óra, akkor az időbélyeg elkészíthető. A továbbiakban azzal foglalkozunk, hogy mi van akkor, ha nincs globális óra.

**Lampert-féle időbélyeg:** Minden  $P_i$  folyamat egy  $C_i$  számlálót tart nyilván az alábbiak szerint:

- $P_i$  minden eseménye eggyel növeli  $C_i$ -t.
- Az elküldött  $m$  üzenetre ráírjuk az időbélyeget:  $ts(m) = C_i$ .
- Ha az  $m$  üzenet beérkezik  $P_j$  folyamathoz, ott a számláló új értéke  $C_j = \max\{C_j, ts(m)\} + 1$  lesz

- $P_i$  és  $P_j$  egybeeső időbélyegjei közül tekintsük a  $P_i$ -belit elsőnek, ha  $i < j$ .

**Pontosan sorbarendezett csoportcímnzés:** A  $P_i$  folyamat minden műveletet időbélyeggel ellátott üzenetben küld el.  $P_i$  egyúttal beteszi a küldött üzenetet a saját  $queue_i$  prioritásos sorába. A  $P_j$  folyamat a beérkező üzeneteket az ő  $queue_j$  prioritásos sorába teszi be az időbélyegnek megfelelő prioritással. Az üzenet érkezéséről mindegyik folyamatot értesíti.  $P_j$  akkor adja át a  $msg_i$  üzenet feldolgozásra, ha:

- $msg_i$  a  $queue_j$  elején található, azaz az ő időbélyege a legkisebb
- a  $queue_j$  sorban minden  $P_k, k \neq i$  folyamatnak megtalálható legalább egy üzenete, amelynek  $msg_i$ -nél későbbi az időbélyege

**Időbélyeg-vektor:**

- $P_i$  most már az összes folyamat idejét is számon tartja egy  $VC_i[1..n]$  tömbben, ahol  $VC_i[j]$  azon  $P_j$ -ben bekövetkezett események száma, amiről  $P_i$  tud.
- Az  $m$  üzenet elküldése során  $P_i$  megnöveli eggyel  $VC_i[i]$  értékét és a teljes  $VC_i$  időbélyeg-vektort ráírja az üzenetre.
- Amikor az  $m$  üzenet megérkezik  $P_j$ -hez, amelyen a  $ts(m)$  időbélyeg van, akkor

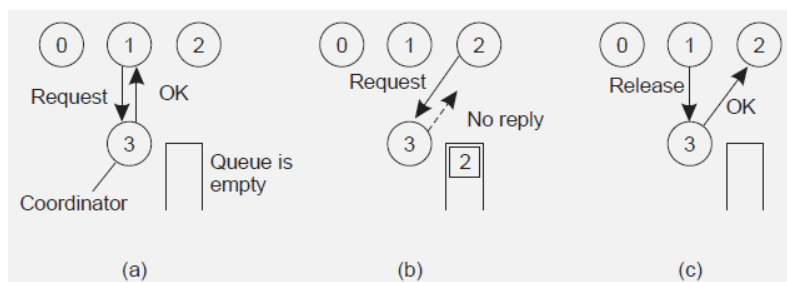
1.  $VC_j[k] := \max\{VC_j[k], ts_m[k]\}$
2.  $VC_j[j]$  megnő eggyel

## 5.2 Kölcsönös kizárás

Több folyamat egyszerre szeretne hozzáférni egy adott erőforráshoz. Ezt egyszerre csak egynek engedhetjük meg közülük, különben az erőforrás helytelen állapotba kerülhet.

### 5.2.1 Kölcsönös kizárás központi szerver használatával

Egy központi szerver a koordinátor, ő szabályozza az erőforráshoz való hozzáférést. Van egy várakozási sora. Ha az erőforrás szabad, akkor ha kérés érkezik rá, a szerver megadja a hozzáférést és foglalttá teszi. Ezután ha valaki más hozzá akar férni az erőforráshoz, akkor bekerül a várakozási sorba. Miután az első kliens elengedte az erőforrást, az ahhoz kerül, aki a sor elején van. Ha kiürült a sor és az utolsó kliens is elengedte az erőforrást, az újra szabaddá válik.



ábra 9: Példa központosított kölcsönös kizárásra.

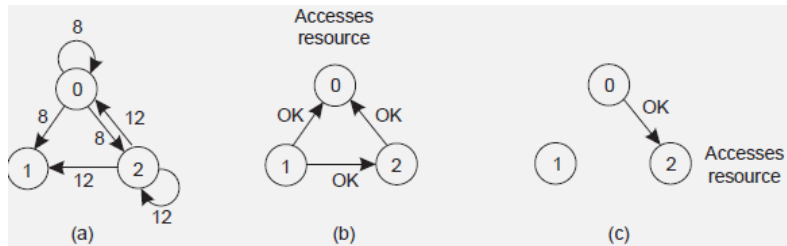
### 5.2.2 Decentralizált kölcsönös kizárás

Tegyük fel, hogy az erőforrás  $n$ -szeresen többszörözött, és minden replikátumhoz tartozik egy azt kezelő koordinátor. A hozzáférésről többségi szavazás dönt: legalább  $m$  koordinátor szükséges, ahol  $m > \frac{n}{2}$ . Feltesszük, hogy egy esetleges összeomlás után a koordinátor felépül, de a kiadott engedélyeket elfelejti.

### 5.2.3 Elosztott kölcsönös kizárás

Többszörözött az erőforrás. Amikor a kliens hozzá szeretne férni az erőforráshoz, kérést küld a koordinátornak időbélyeggel ellátva. Választ (hozzáférési engedélyt) akkor kap, ha:

- A koordinátor nem igényli az erőforrást, vagy
- a koordinátor is igényli az erőforrást, de kisebb az időbélyege.
- Különben a koordinátor átmenetileg nem válaszol.



ábra 10: Példa elosztott kölcsönös kizárásra.

### 5.2.4 Kölcsönös kizárás token ring-gel

A folyamatokat egy logikai gyűrűbe szervezzük. Egy tokenet küldünk körbe. Amelyik folyamat birtokolja a tokenet, az férhet hozzá az erőforráshoz.

## 5.3 Vezetőválasztás

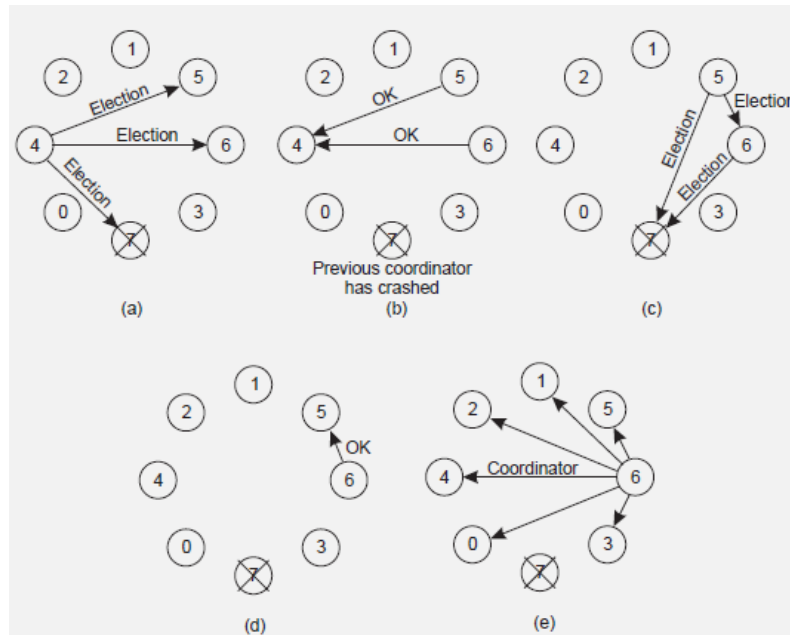
Sok algoritmusnak szüksége van arra, hogy kijelöljön egy folyamatot, amely a további lépéseket koordinálja.

### 5.3.1 Zsarnok-algoritmus

A folyamatoknak sorszámot adunk, melyek közül a legnagyobb sorszámút szeretnénk vezetőnek választani.

A zsarnok-algoritmus lépései:

1. A vezetőválasztás kezdeményezése. Bármelyik folyamat kezdeményezheti. Mindegyik olyan folyamatnak, amelyről nem tudja, hogy kisebb lenne az övéénél a sorszáma, elküld egy üzenetet.
2. Ha a nagyobb sorszámú folyamat üzenetet kap egy kisebb sorszámútól, akkor visszaküld neki egy olyan üzenetet, amivel kiveszi a kisebb sorszámút a választásból.
3. Amelyik folyamat nem kap letiltó üzenet egy bizonyos időn belül, akkor ő lesz a vezető. Erről értesíti a többi folyamatot egy-egy üzenettel.



ábra 11: Példa a zsarnok-algoritmus működésére.

### 5.3.2 Vezetőválasztás gyűrűben

Logikai gyűrűnk van, a folyamatoknak vannak sorszámai. A legnagyobb sorszámú folyamatot szeretnénk vezetőnek választani. Bármelyik folyamat kezdeményezhet vezetőválasztást: elindít egy üzenetet a gyűrűn körbe, amelyre mindenki ráírja a sorszámát. Ha egy folyamat összeomlott, az kimarad az üzenetküldésből. Amikor az üzenet visszajut a kezdeményezőhöz, minden aktív folyamat sorszáma szerepel rajta. Ezek közül a legnagyobb sorszámú lesz a vezető. Ezt egy másik üzenet körbeküldése tudatja mindenkivel.

Ha több folyamat kezdeményez egyszerre választást, az nem probléma, ugyanaz az eredmény adódik. Ha az üzenetek elvesznének, akkor újra lehet kezdeni a választást.

### 5.3.3 Superpeer-választás

A superpeer-eket úgy szeretnénk megválasztani, hogy teljesüljön rájuk:

- A többi csúcs alacsony késleltetéssel éri el őket.
- Egyenletesen vannak elosztva a hálózaton.
- A csúcsok megadott hányadát választjuk superpeer-nek.
- Egy superpeer korlátozott számú peer-t szolgál ki.

**Megvalósítás DHT esetén:** Ha  $m$ -bit-es azonosítókat használunk, és  $S$  superpeer-re van szükség, akkor a  $k = \lceil \log_2 S \rceil$  felső bitet foglaljuk le a superpeer-ek számára. Így  $N$  csúcs esetén kb.  $2^{k-m} N$  superpeer lesz.

A  $p$  kulcshoz tartozó superpeer a  $p$  AND  $\underbrace{11\dots11}_k \underbrace{00\dots00}_{m-k}$  kulcs felelőse lesz.

## 6 Konzisztencia

**Konfliktusos műveletek:** A replikátumok konzisztensen tartásához biztosítani kell, hogy az egymással konfliktusba kerülhető műveletek minden replikátumon egyforma sorrendben futnak le. Írás-olvasás és írás-írás konfliktusok fordulhatnak elő.

**Konzisztenciamodell:** A konzisztenciamodell megszabja, milyen módokon használhatják a folyamatok az adatbázist. Ha a feltételek teljesülnek, az adattárat érvényesnek tekintjük.

**Konzisztencia mértéke:** A konzisztencia többféle módon is sérülhet: eltérhet a replikátumok számértéke, frissessége, meg nem történt frissítési műveletek száma.

**Conit:** Az olyan adategység, amelyre közös feltételrendszer vonatkozik, a conit (consistency unit).

## 6.1 Soros konzisztencia

A feltételeket nem számértékekre, hanem írárok/olvasások tényére alapozzuk. Jelölések:

- $W(x)$  :  $x$  változót írta a folyamat
- $R(x)$  :  $x$  változót olvasta a folyamat

Soros konzisztencia esetén azt várjuk el, hogy a végrehajtás eredménye olyan legyen, mintha az összes folyamat összes művelete egy meghatározott sorrendben történt volna meg, megőrizve bármely adott folyamat saját műveletinek sorrendjét.

P1: $W(x)a$	P1: $W(x)a$
P2: $W(x)b$	P2: $W(x)b$
P3: $R(x)b$ $R(x)a$	P3: $R(x)b$ $R(x)a$
P4: $R(x)b$ $R(x)a$	P4: $R(x)a$ $R(x)b$
(a)	(b)

ábra 12: Példa: az (a) teljesíti, (b) nem a soros konzisztencia követelményeit.

## 6.2 Okozati konzisztencia

A potenciálisan okozati összefüggésben álló műveleteket kell mindegyik folyamatnak azonos sorrendben látnia. A konkurens írárokat a különböző folyamatok különböző sorrendben láthatják.

P1: $W(x)a$	P1: $W(x)a$
P2: $R(x)a$ $W(x)b$	P2: $W(x)b$
P3: $R(x)b$ $R(x)a$	P3: $R(x)b$ $R(x)a$
P4: $R(x)a$ $R(x)b$	P4: $R(x)a$ $R(x)b$
(a)	(b)

ábra 13: Példa: a (b) teljesíti, (a) nem az okozati konzisztencia követelményeit.

## 6.3 Kliensközpontú konzisztencia

Azt helyezzük most előtérbe, hogy a szervereken tárolt adatok hogyan látszanak egy adott kliens számára. A kliens mozog: különböző szerverekhez csatlakozik, és írási/olvasási műveleteket hajt végre.

Az  $A$  szerver után a  $B$  szerverhez csatlakozva különböző problémák léphetnek fel:

- Az  $A$ -ra feltöltött frissítések lehet, hogy nem jutottak még el  $B$ -hez.



- $B$ -n lehet, hogy újabb adatok találhatók, mint  $A$ -n.
- A  $B$ -re feltöltött frissítések ütközhetnek az  $A$ -ra feltöltöttekkel.

A cél az, hogy a kliens azokat az adatokat, amiket az  $A$  szerveren kezelt, ugyanolyan állapotban lássa  $B$ -n is. Ekkor az adatbázis konzisztensnek látszik a kliens számára.

### 6.3.1 Monoton olvasás

Ha egyszer a kliens kiolvasott egy értéket  $x$ -ből, minden ezután következő olvasás ezt adja, vagy ennél frissebb értéket.

Például levelezőkliens esetén minden korábban letöltött levelünknek meg kell lennie az új szerveren is.

### 6.3.2 Monoton írás

A kliens akkor írhatja  $x$ -et, ha kliens korábbi írásai  $x$ -re már befejeződtek.

Például verziókezelésnél minden korábbi verzióknak meg kell lennie a szerveren, ha új verziót akarunk feltölteni.

### 6.3.3 Olvasd az írásodat

Ha kliens olvassa  $x$ -et, a saját legutolsó írásának eredményét kapja, vagy frissebbet.

Például a kliens a honlapját szerkeszti, majd megnézi az eredményt. Ahelyett, hogy a böngésző gyorsítótárából egy régebbi változat kerülne elő, a legfrissebbet szeretné látni.

### 6.3.4 Írás olvasás után

Ha a kliens kiolvasott egy értéket  $x$ -ből, minden ezután kiadott frissítési művelete  $x$ -nek legalább ennyire friss értékét módosítja.

Például egy fórumon a kliens csak olyan hozzászólásra tud válaszolni, amit már látott.

## 6.4 Tartalom replikálása

Különböző jellegű folyamatok tárolhatják a másolatokat:

- Tartós másolat: eredetszerver (origin server)
- Szerver által kezdeményezett másolat: replikátum kihelyezése egy szerverre, amikor az igényli az adatot
- Kliens által kezdeményezett másolat: kliensoldali gyorsítótár

### 6.4.1 Frissítés terjesztése

Megváltozott tartalmat több különféle módon lehet kliens-szerver architektúrában átadni:

- Kizárólag a frissítésről szóló értesítés/érvénytelenítés elterjesztése.
- Passzív replikáció: adatok átvitele egyik másolatról a másikra
- Aktív replikáció: frissítési művelet átvitele

A frissítést kezdeményezheti a szerver (küldésalapú frissítés), ekkor a szerver a kliens kérése nélkül elküldi a frissítést a kliensnek, vagy kezdeményezheti a kliens, aki kérvényezi a frissítést a szervertől (rendelésalapú frissítés).

**Haszonbérlet (lease):** A szerver ígéretet tesz a kliensnek, hogy átküldi a frissítést, amíg a haszonbérlet aktív.