# Elosztott rendszerek: Alapelvek és paradigmák
# Distributed Systems: Principles and Paradigms

Maarten van Steen[1] Kitlei Róbert [2]

[1]VU Amsterdam, Dept. Computer Science
[2]ELTE Informatikai Kar

## 7. rész: Konzisztencia & replikáció

2015. május 24.

# Tartalomjegyzék

# Hatékonyság és átméretezhetőség

## Konfliktusos műveletek

A replikátumok konzisztensen tartásához garantálni kell, hogy az egymással konfliktusba kerül(het)ő műveletek minden replikátumon egyforma sorrendben futnak le.
Ahogy a tranzakcióknál, írás–olvasás és írás–írás konfliktusok fordulhatnak elő.

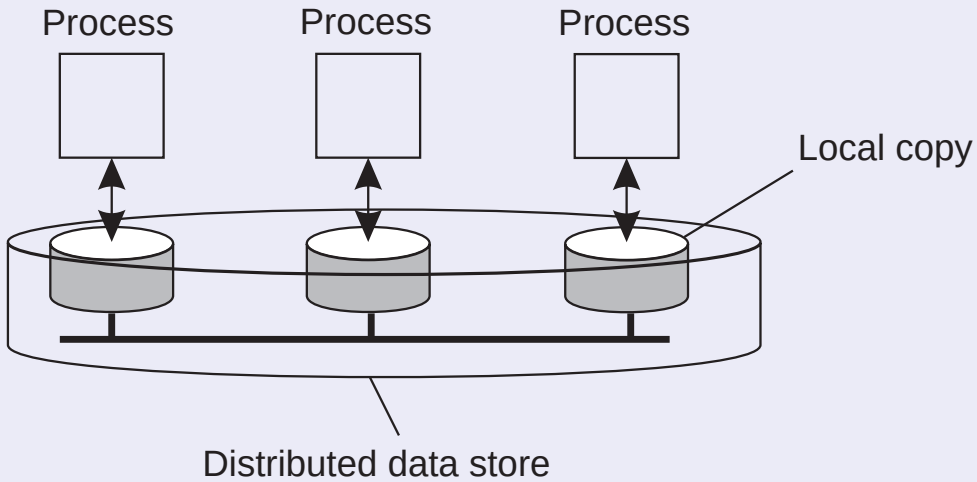
## Terv: kevesebb szinkronizáció

Az összes konfliktusos művelet globális sorbarendezése általában túl költséges.
Megvizsgáljuk, hogyan lehet a konzisztenciakövetelményeket gyengíteni.
Jellemzően minél megengedőbbek a feltételek, annál kevesebb szinkronizáció szükséges a biztosításukhoz.

# Adatközpontú konzisztencia

## Konzisztenciamodell

A konzisztenciamodell megszabja, milyen módokon használhatják a folyamatok az adatbázist. Elosztott adattár (lásd az ábrát) esetén legfőképpen az egyidejű írási és olvasási műveletekre ad előírásokat. Ha a feltételek teljesülnek, az adattárat érvényesnek tekintjük.



# Folyamatos konzisztencia

## Konzisztencia mértéke

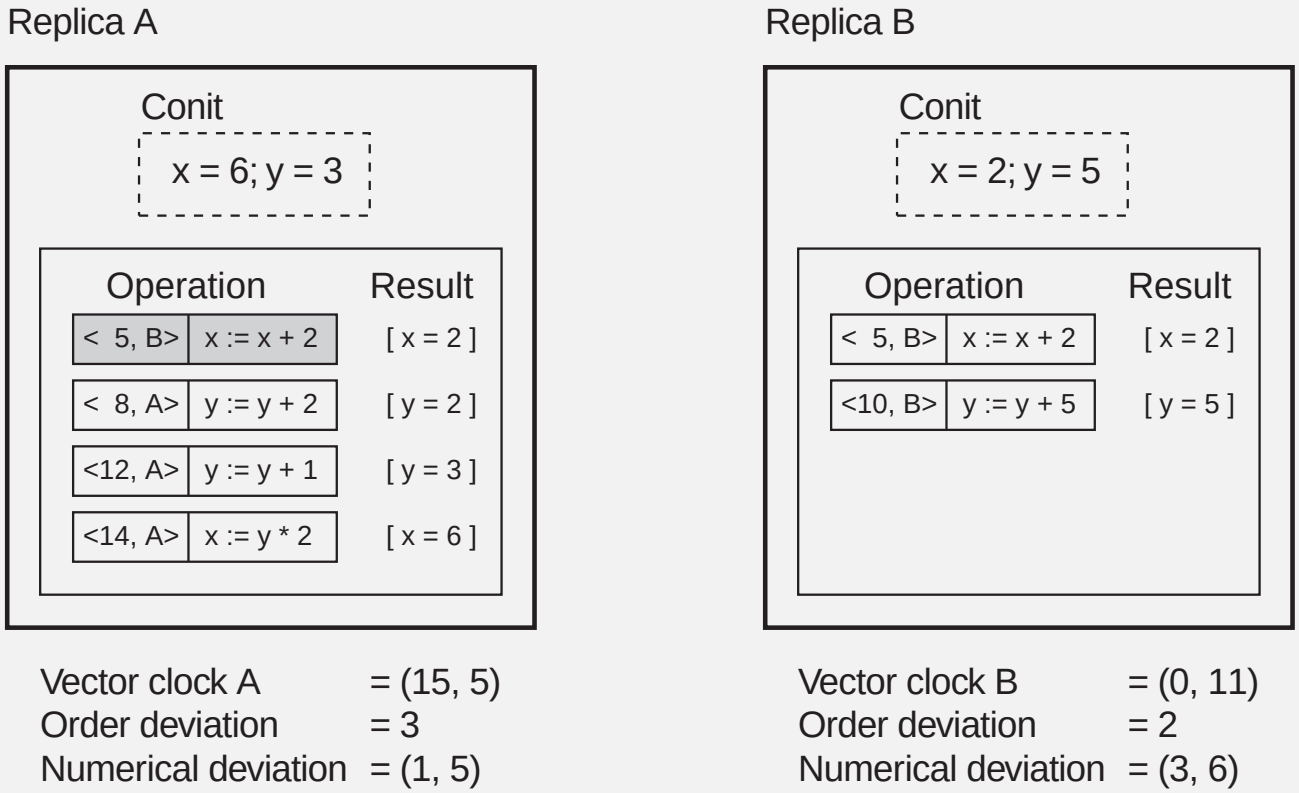
A konzisztencia több módon is sérülhet:

- eltérhet a replikátumok számértéke
- eltérhet, hogy mennyire frissek az egyes replikátumok
- eltérhet, hogy hány frissítési művelet nem történt még meg (illetve: sorrendben melyik műveletek hiányoznak)

## Conit

Ha megtehetjük, a konzisztenciafeltételeket nem a teljes adatbázisra írjuk fel, hanem az adatoknak minél szűkebb körére. Az olyan adategység, amelyre közös feltételrendszer vonatkozik, a conit (**con**sistency un**it**).

# Példa: Conit

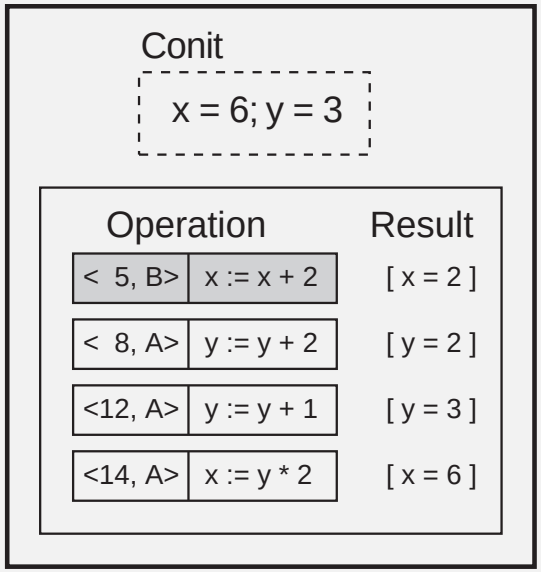



## Conit (most az $x$ és $y$ változó alkotja)

- Mindkét replikátumnak van vektorórája: (A-beli idő, B-beli idő)
- $B$ elküldi $A$-nak a $[\langle 5, B\rangle: x := x + 2]$ műveletet
- $A$ véglegesíti a kijött eredményt (nem vonható vissza)

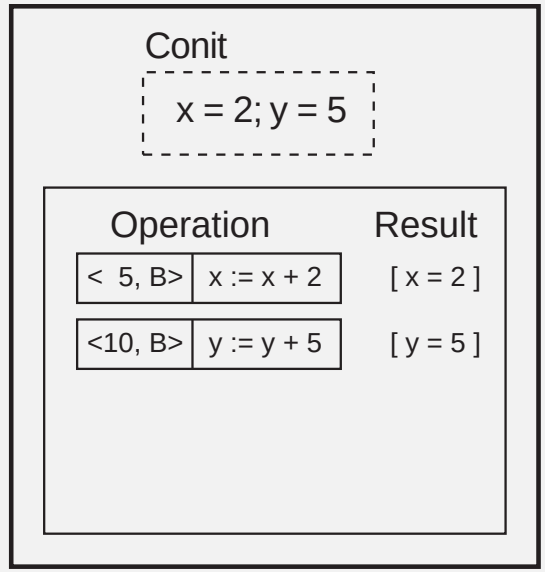
# Példa: Conit

Replica A




Vector clock A = (15, 5)
Order deviation = 3
Numerical deviation = (1, 5)

Replica B




Vector clock B = (0, 11)
Order deviation = 2
Numerical deviation = (3, 6)

## Conit (most az *x* és *y* változó alkotja)

- *A* három műveletet nem végzett még el
- *A*-ban egy *B*-beli művelet még nem hajtódott végre, a számérték legfeljebb 5-tel térhet el $\Rightarrow$ (1,5)

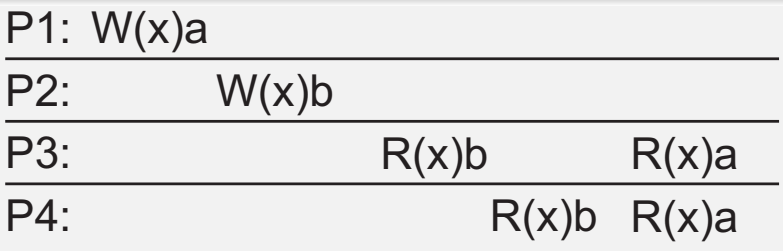
# Soros konzisztencia

## Jelölések

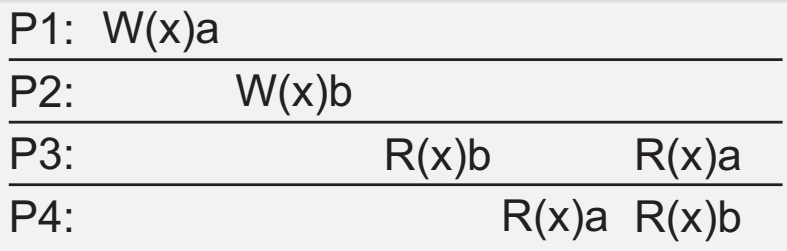
Sokszor a feltételeket nem a számértékekre alapozzuk, hanem csupán az írások/olvasások tényére. Jelölje W(x) azt, hogy az *x* változót írta egy megadott folyamat, R(x) azt, hogy olvasta, a mellettük levő betűk pedig azt jelölik, hogy az olvasás melyik írással írt értéket látja.

## Soros konzisztencia

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 műveleteinek sorrendjét. (a) teljesíti, (b) nem




(a)




(b)

# Okozati konzisztencia

## Okozati konzisztencia

A potenciálisan okozatilag összefüggő műveleteket kell mindegyik folyamatnak azonos sorrendben látnia. A konkurens írásokat különböző folyamatok különböző sorrendben láthatják.

(b) teljesíti; (a) nem, mert ott $P_1$ és $P_2$ írásait „összeköti” az olvasás

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

(a)

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

(b)

# Műveletek csoportosítása

## Szinkronizáció

Most szinkronizációs változókat használunk, ezek elérései sorosan konzisztensek. Háromfajta megközelítés:

- Egy rendszerszintű $S$ változó használata. $S$ egy elérése után garantált, hogy a korábbi elérései előtti írások megtörténtek.
- Igényeljük (acquire) és aztán feloldjuk (release) a változókat, ezzel kritikus területeket (más néven: védett területeket) alakítunk ki.
  - Több rendszerszintű szinkronizációs változó használata.
    A védelem a területen írt/olvasott adatokra terjed ki.
  - Minden adatelemhez külön változó használata.
    A védelem a változó adatelemére terjed ki.

Ha a fenti szinkronizációs határokon belül több írás is történik, a határon belül ezek nem definiált sorrendben látszanak, és a szinkronizáció után csak a végeredmény látszik, az nem, hogy milyen sorrendben történtek az írások.

# Kliensközpontú konzisztencia

## Cél

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.

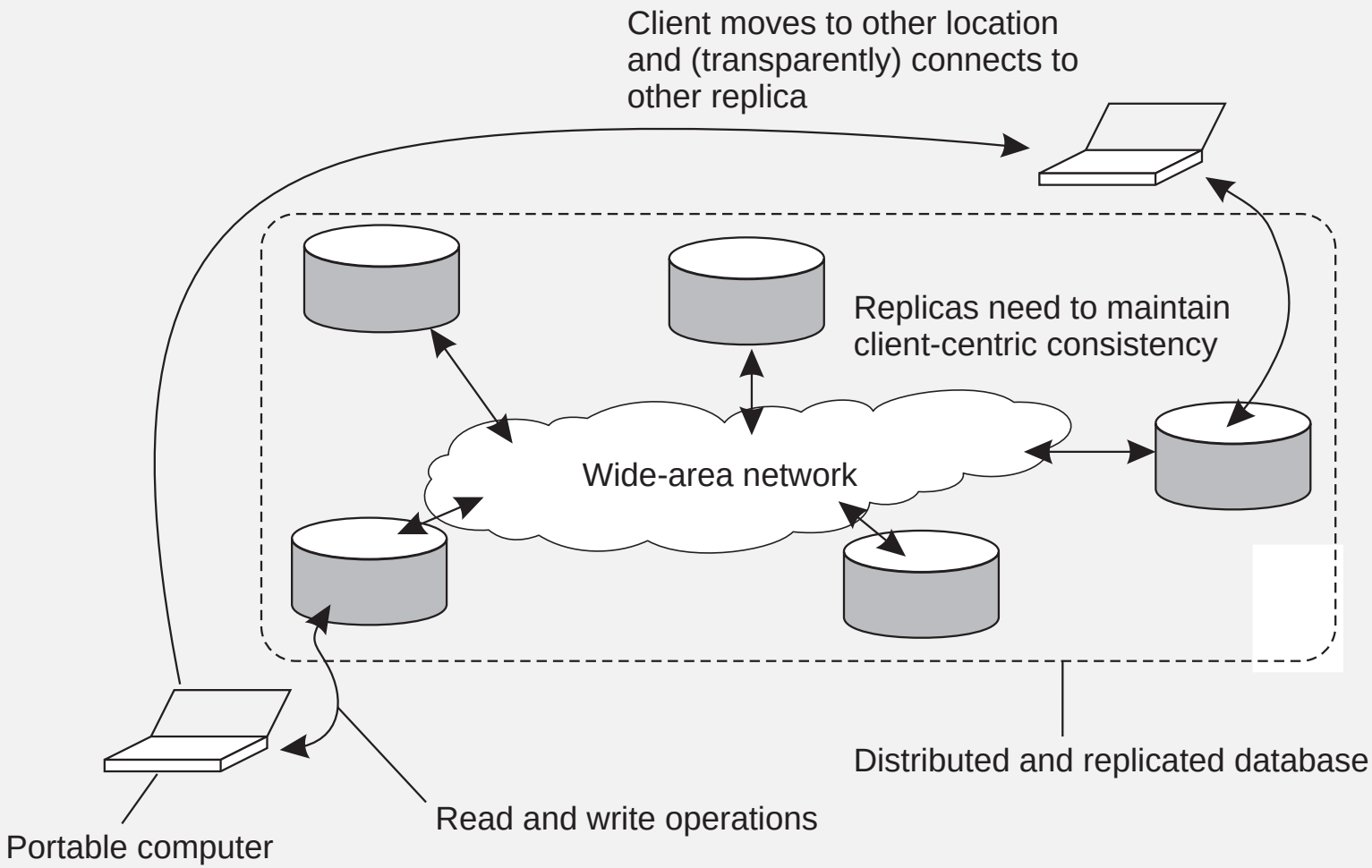
Az *A* szerver után a *B* szerverhez csatlakozva különböző problémák léphetnek fel:

- az *A*-ra feltöltött frissítések lehet, hogy még nem jutottak el *B*-hez
- *B*-n lehet, hogy újabb adatok találhatóak meg, mint *A*-n
- a *B*-re feltöltött frissítések ütközhetnek az *A*-ra feltöltöttekkel

Cél: a kliens azokat az adatokat, amelyeket az *A* szerveren kezelt, ugyanolyan állapotban lássa *B*-n. Ekkor az adatbázis konzisztensnek látszik a kliens számára.

Négyfajta kombináció: a kliens *A*-n (olvasott/írt) adatokat (olvas/ír) *B*-n.

# Kliensközpontú konzisztencia



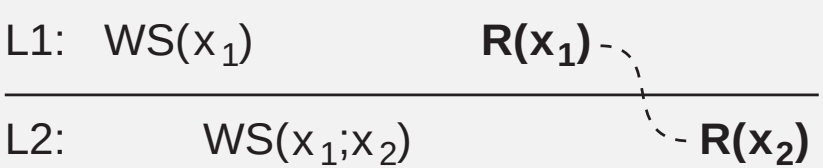
# Kliensközpontú konzisztencia: olvasás után olvasás

## Jelölések

- $WS(x_i)$: írási műveletek összessége (write set) az $x$ változóra
- $WS(x_i; x_j)$: olyan $WS(x_j)$, amely tartalmazza $WS(x_i)$ frissítéseit

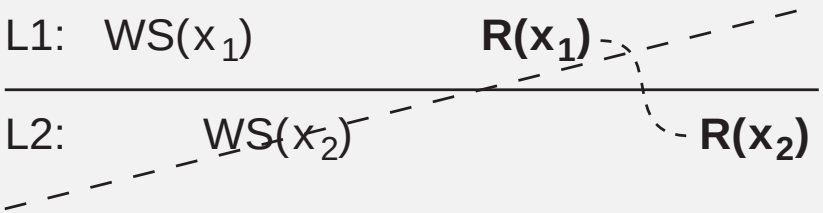
## Monoton olvasás (olvasás után olvasás)

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




### Példa: határidőnapló

Minden korábbi bejegyzésünknek meg kell lennie az új szerveren is.



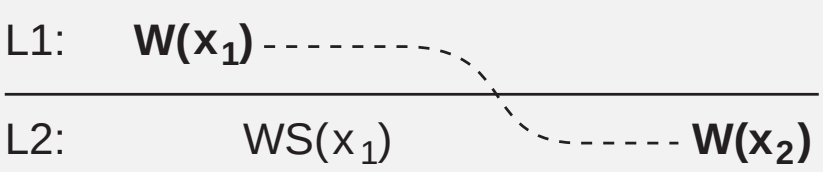

### Példa: levelezőkliens

Minden korábban letöltött levelünknek meg kell lennie az új szerveren is.

# Kliensközpontú konzisztencia: írás után írás

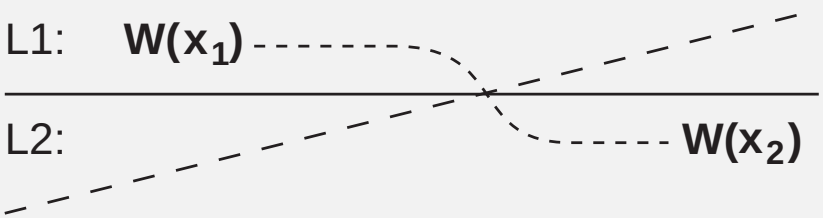
## Monoton írás (írás után írás)

A kliens csak akkor írhatja *x*-et, ha a kliens korábbi írásai *x*-re már befejeződtek.




## Példa: forráskód

A kliens egy programot szerkeszt több szerveren. Minden korábbi szerkesztésnek jelen kell lennie, mert az újonnan írt kód függ tőlük.



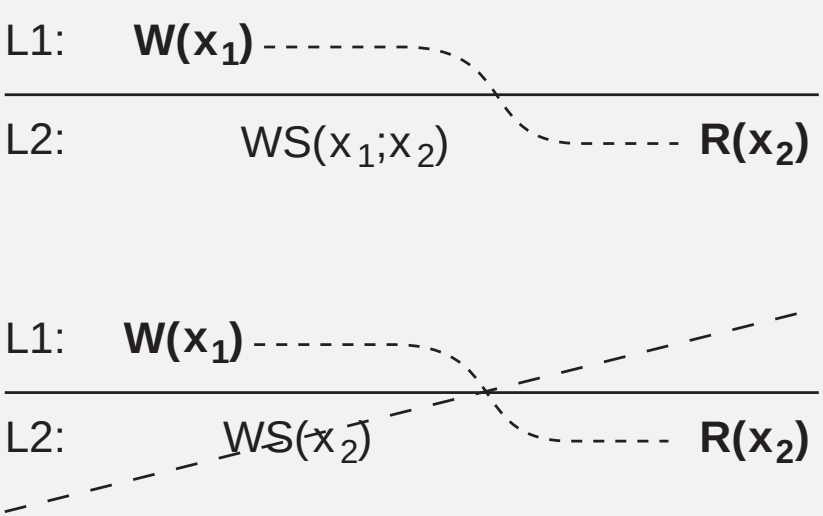

## Példa: verziókezelés

A kliens verziószámokkal ellátott fájlokat készít. Minden korábbi verziónak meg kell lennie ahhoz, hogy újat készíthessen.

# Kliensközpontú konzisztencia: írás után olvasás

## Olvasd az írásodat (írás után olvasás)

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



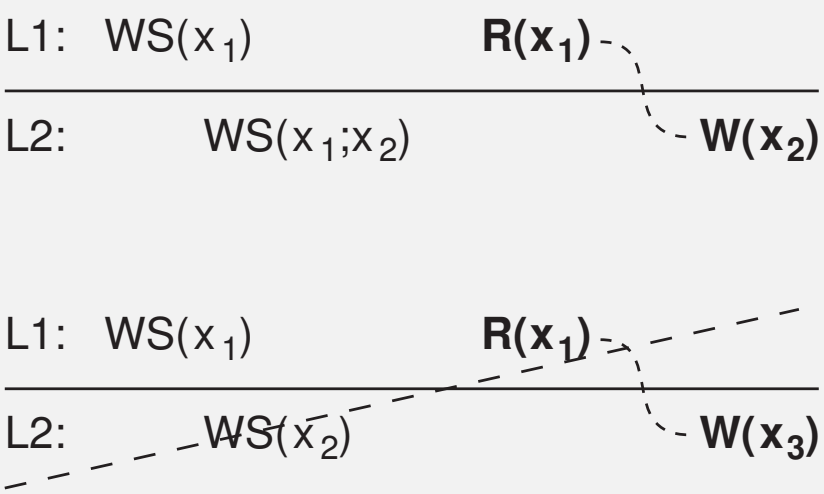

## Példa: honlap szerkesztése

A kliens a honlapját szerkeszti, majd megnézi a szerkesztés eredményét. Ahelyett, hogy a böngésző gyorsítótárából egy régebbi verzió kerülne elő, a legfrissebb változatot szeretné látni.

# Kliensközpontú konzisztencia: olvasás után írás

**Írás olvasás után (olvasás után írás)**

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




**Példa: fórum**

A kliens csak olyan hírre tud választ beküldeni, amelyet már látott.

# Replikátumszerverek elhelyezése

Tegyük fel, hogy $N$ lehetséges helyre szeretnénk összesen $K$ darab szervert telepíteni. Melyik helyeket válasszuk?

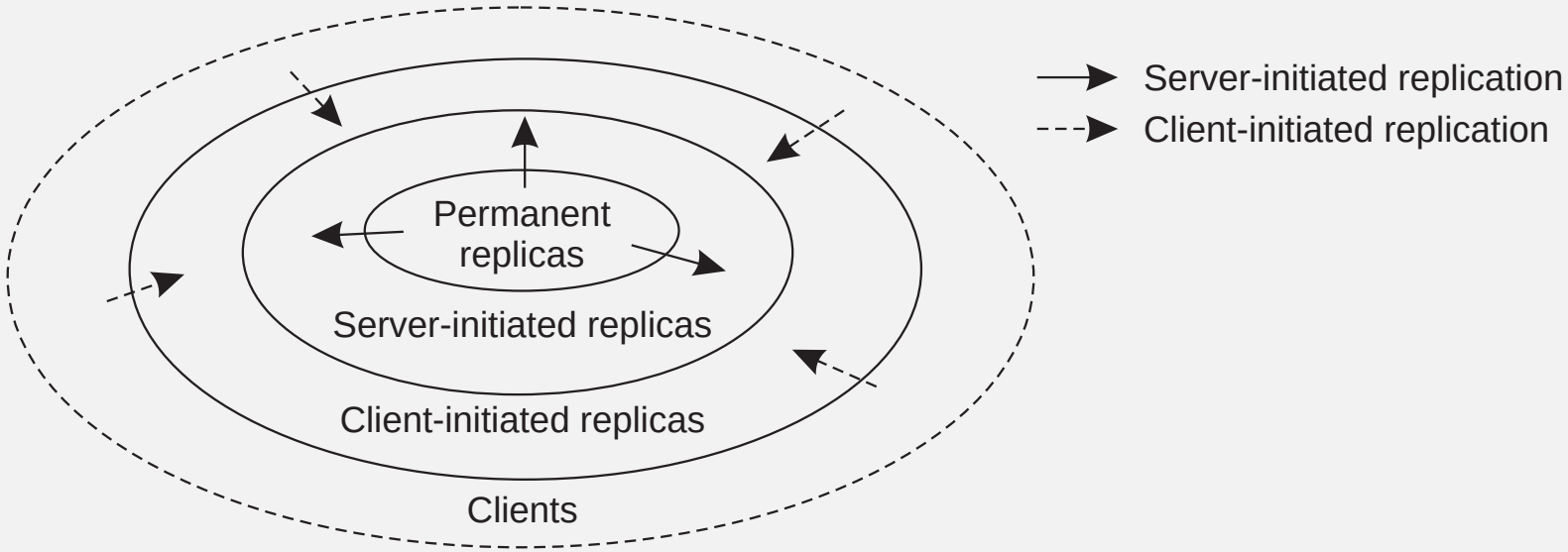
- Tegyük fel, hogy ismert a kliensek helye. Válasszuk meg úgy a szervereket, hogy azok átlagtávolsága a kliensektől minimális legyen. Egzakt kiszámítása költséges, heurisztika szükséges.
- Jellemzően a kliensek több autonóm rendszerben[a] találhatóak meg. A $K$ legnagyobb rendszerben helyezzünk el egy-egy szervert, mindig a rendszeren belül leginkább központi helyre. Szintén magas a számítási költsége.
- Mint korábban a csúcsok globális pozícionálásánál, ábrázoljuk a csúcsokat egy $d$-dimenziós térben, ahol a távolság mutatja a késleltetést. Keressük meg a $K$ „legsűrűbb” részt, és oda helyezzünk szervereket. Számítási költsége alacsonyabb.

[a] A teljes hálózat útválasztás szempontjából egységként adminisztrált része, gyakran egy Internet-szolgáltató (ISP) felügyeli.

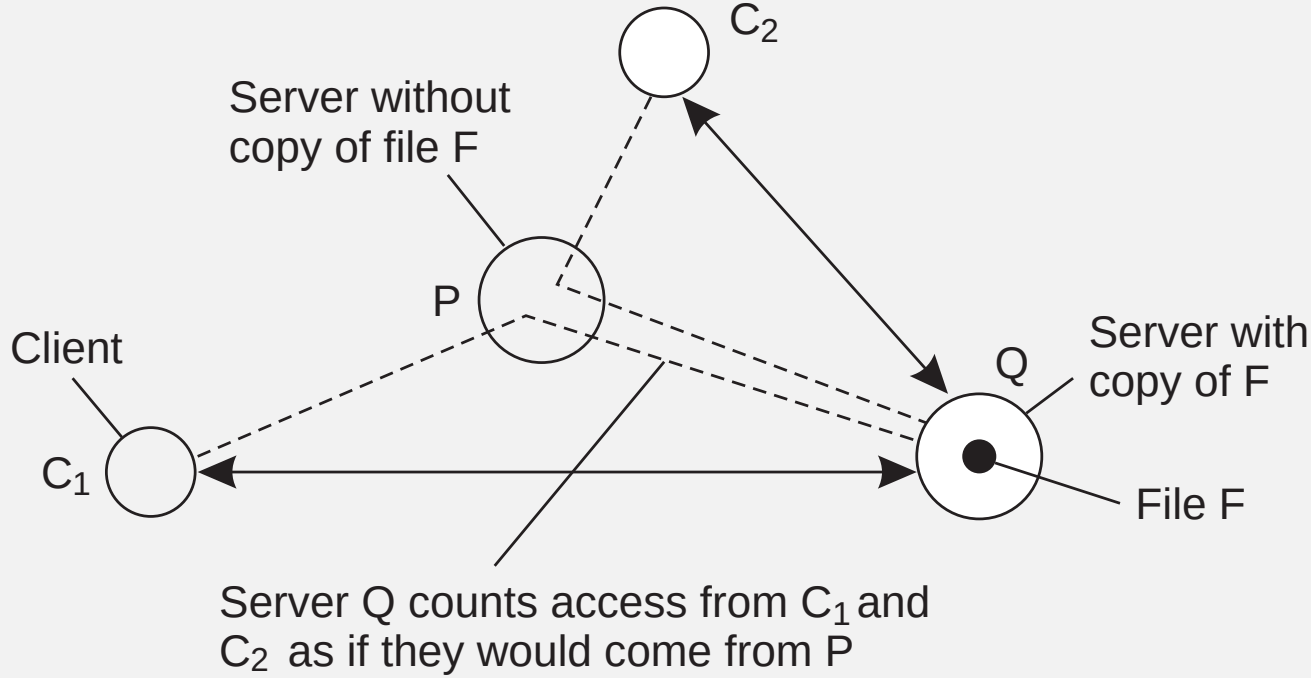
# Tartalom replikálása

Különböző jellegű folyamatok tárolhatják az adatokat:

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



# Szerver által kezdeményezett másolatok




A rendszer figyeli, hányszor fértek hozzá a fájlhoz ($A$), úgy számolva, mintha a kérés a klienshez legközelebbi szerverhez ($P$) érkezett volna be. Adott két szám, $D$ és $R$, melyekre $D < R$. Mit tegyünk a fájllal?

- Ha $A < D \Rightarrow$ töröljük $Q$-ról (ha megvan máshol a rendszerben)
- Ha $D < A < R \Rightarrow$ migráljuk $Q$-ról $P$-re
- Ha $R < A \Rightarrow$ replikáljuk $P$-re

# Frissítés terjesztése

## Modellek

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

- Kizárólag a frissítésről szóló értesítés/érvénytelenítés elterjesztése (pl. gyorsítótáraknál ez egy egyszerű lehetőség)
- Passzív replikáció: adatok átvitele egyik másolatról a másikra
- Aktív replikáció: frissítési művelet átvitele

## Melyiket érdemes választani?

A sávszélesség és az írási/olvasási műveletek aránya a replikátumon nagyban befolyásolja, melyik módszer a leghatékonyabb adott esetben.

# Frissítés terjesztése

- Küldésalapú: a szerver a kliens kérése nélkül küldi a frissítést
- Rendelésalapú: a kliens kérvényezi a frissítést

| Témakör | Küldésalapú | Rendelésalapú |
|---|---|---|
| Kezdeményező | Szerver | Kliens |
| Szerverállapot | Klienscache-ek listája | Nincsen |
| Küldött üzenetek | Frissítés* | Lekérdezés és frissítés |
| Válaszidő a kliensnél | Azonnali* | Letöltő frissítés ideje |

*: A kliens később még alkalmazhat letöltő frissítést (fetch update).

# Frissítés terjesztése

## Haszonbérlet

Haszonbérlet (lease): a szerver ígéretet tesz a kliensnek, hogy át elküldi neki a frissítéseket, amíg a haszonbérlet aktív

## Rugalmas haszonbérlet

Fix lejárat helyett rugalmasabb, ha a rendszer állapotától függhet a haszonbérlet időtartama:

- Kor szerinti: minél régebben változott egy objektum, annál valószínűbb, hogy sokáig változatlan is marad, ezért hosszabb lejárat adható
- Igénylés gyakorisága szerinti: minél gyakrabban igényli a kliens az objektumot, annál hosszabb időtartamokra kap haszonbérletet rá
- Terhelés szerinti: minél nagyobb a szerver terhelése, annál rövidebb haszonbérleteket ad ki

# Folyamatos konzisztencia: számszerű eltérések

## Alapelv

- Az egyszerűség kedvéért most egyetlen adatelemet vizsgálunk.
- Az $S_i$ szerver a $log(S_i)$ naplóba írja az általa végrehajtott műveleteket.
- A $W$ írási műveletet elsőként végrehajtó szervert jelölje $origin(W)$, a művelet hatására bekövetkező értékváltozást pedig $weight(W)$. Tegyük fel, hogy ez mindig pozitív szám.
- $S_i$ olyan írásait, amelyek $S_j$-ről származnak, jelölje $TW[i,j]$:

$$TW[i,j] = \sum \{ weight(W) \,|\, origin(W) = S_j \;\&\; W \in log(S_i) \}$$

- Ekkor a változó összértéke (v) és értéke az $i$-edik másolaton ($v_i$):

$$v = v_{kezdeti} + \sum_k TW[k,k] \qquad v_i = v_{kezdeti} + \sum_k TW[i,k]$$

# Folyamatos konzisztencia: számszerű eltérések

Cél: minden $S_i$ szerveren teljesüljön: $v - v_i < \delta_i$ (rögzített $\delta_i$ értékre).

## Algoritmus

$TW[i,j]$ értékét minden $S_k$ szerver becsülje egy $TW_k[i,j]$ értékkel.
(Azaz: $S_k$ „mit tud”, $S_i$ milyen frissítéseket kapott már meg $S_j$-től.)

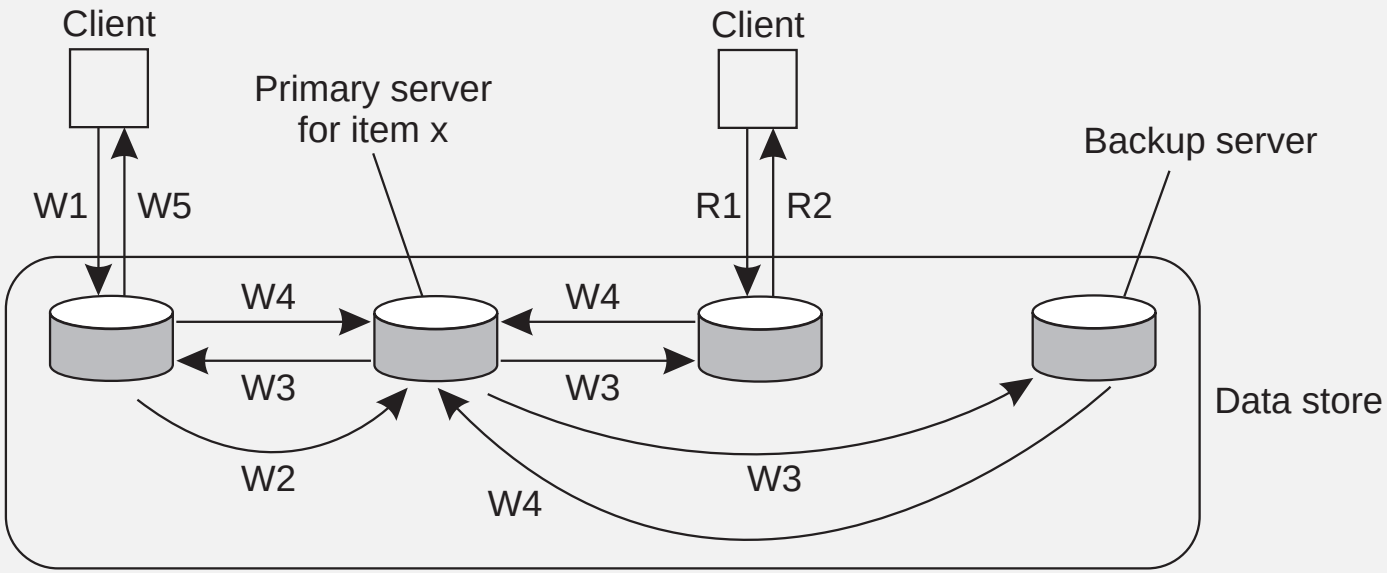
Mivel a számértékeink nemnegatívak, fennáll:

$$0 \leq TW_k[i,j] \leq TW[i,j] \leq TW[j,j] = TW_j[j,j]$$

- Amikor új frissítési művelet érkezik be egy szerverre, pletykálással értesíti erről a többi szervert.
- Amikor $S_j$ azt látja, hogy $TW_j[i,j]$ túl messzire kerül $TW[j,j]$-től[a], akkor küldje el $S_i$-nek $log(S_j)$ műveleteit.

[a] $TW[j,j] - TW_j[i,j] > \delta_i/(N-1)$, ahol $N$ a replikátumok száma

# Elsődleges másolaton alapuló protokoll távoli írással

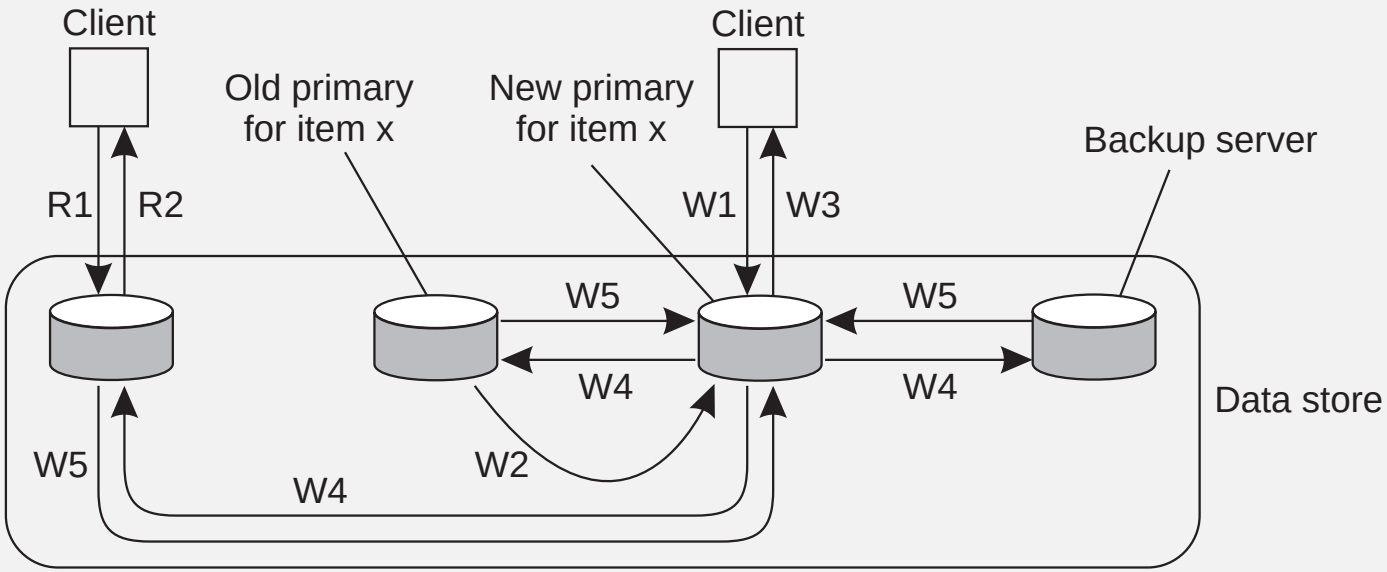



W1. Write request
W2. Forward request to primary
W3. Tell backups to update
W4. Acknowledge update
W5. Acknowledge write completed

R1. Read request
R2. Response to read

Példa: nagy hibatűrést igénylő elosztott adatbázisokban és fájlrendszerekben használatos. A másolatok gyakran egy lokális hálózatra kerülnek.

# Elsődleges másolaton alapuló protokoll helyi írással




W1. Write request
W2. Move item x to new primary
W3. Acknowledge write completed
W4. Tell backups to update
W5. Acknowledge update

R1. Read request
R2. Response to read

Példa: kapcsolat nélküli munka, időnként szinkronizálás a rendszerrel annak különböző pontjaihoz csatlakozva.

# Többszörözöttírás-protokoll

## Testületalapú protokoll

Többszörözött írás: az írási műveletet több szerveren hajtjuk végre.

Testület (quorum): egy művelet végrehajtása előtt meghatározott számú szervertől kell engedélyt kérni. Jelölés: írási $N_W$, olvasási $N_R$.

Egy írási művelet ütközhetne egy olvasási művelettel, vagy egy másik írásival; az első elkerüléséhez $N_R + N_W > N$, a másodikhoz $N_W + N_W > N$, azaz $N_W > N/2$ szükséges a skatulya-elv alapján.