Consistenţa şi replicarea datelor în Internet

Bazat pe “Distributed Systems” de AS Tanenbaum

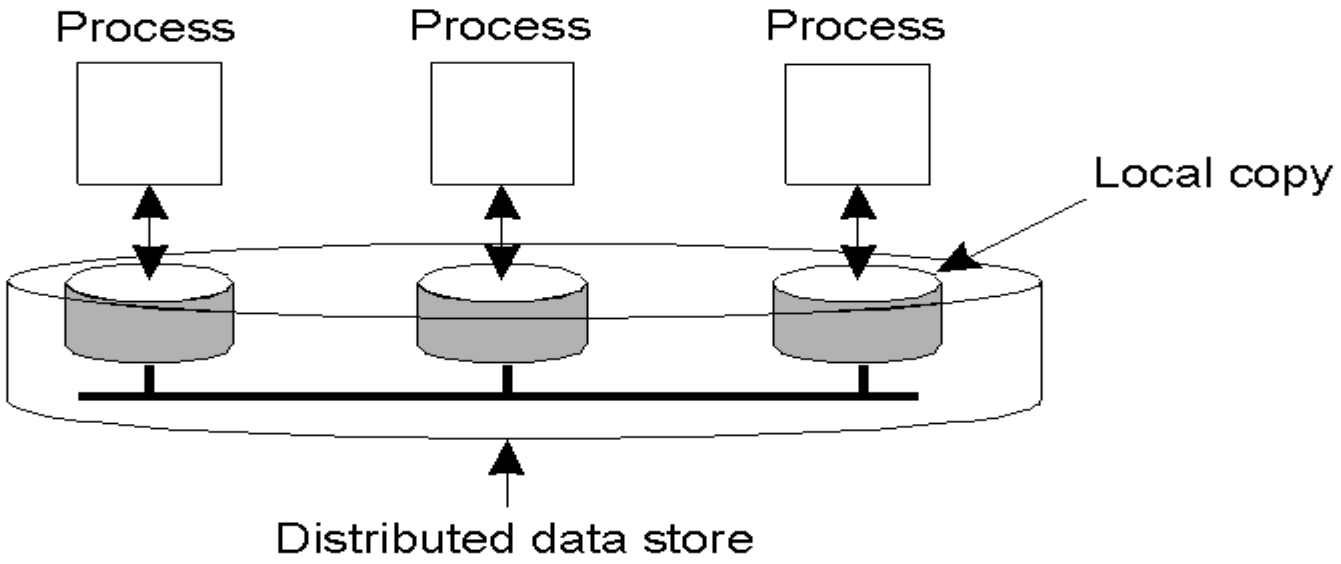
Motive pentru replicare

- Siguranta
  - Toleranta la defectari - sistemul continua sa ofere un serviciu corect chiar in cazul defectarii unei replici
  - Protectie mai buna impotriva datelor corupte (generali bizantini)
- Performanta
  - Timp de acces mai mic la replici apropiate de client
  - Incarcare mai mica a fiecarei replici
  - Disponibilitate mai mare
- Forme de replicare
  - Date – pastrarea unor copii ale datelor in mai multe calculatoare
  - Actiuni – executia mai multor replici identice ale unui proces
- Probleme: consistenta replicilor

Modele de consistenţă a datelor

- Consistenta stricta
  - modificarea unei replici se regaseste imediat in celelalte
  - greu de pastrat
- Solutie = constrangeri mai slabe de consistenta
- Categorii de modele de consistenta
  - centrate pe date
    - se aplica tuturor replicilor indiferent de utilizatorul care le foloseste
  - centrate pe client
    - consistenta replicilor apelate de un acelasi client

Modele de consistenta centrate pe date



- Organizare generala depozit de date distribuit si replicat
  - corespunde sistemelor distribuite actuale, inclusiv NoW, clustere …
- Operatii: **read** (copie locala), **write** (propagata la celelalte copii)
- Model de consistenta = contract intre proces si data store
- Ideal, read ar trebui sa intoarca rezultatul ultimei operatii write

Consistenţa Strictă

- Orice citire a unei date x intoarce valoarea rezultata din cea mai recenta scriere a lui x.

- Ex:

Doua masini A si B

Doua procese: Pa pe A si Pb pe B

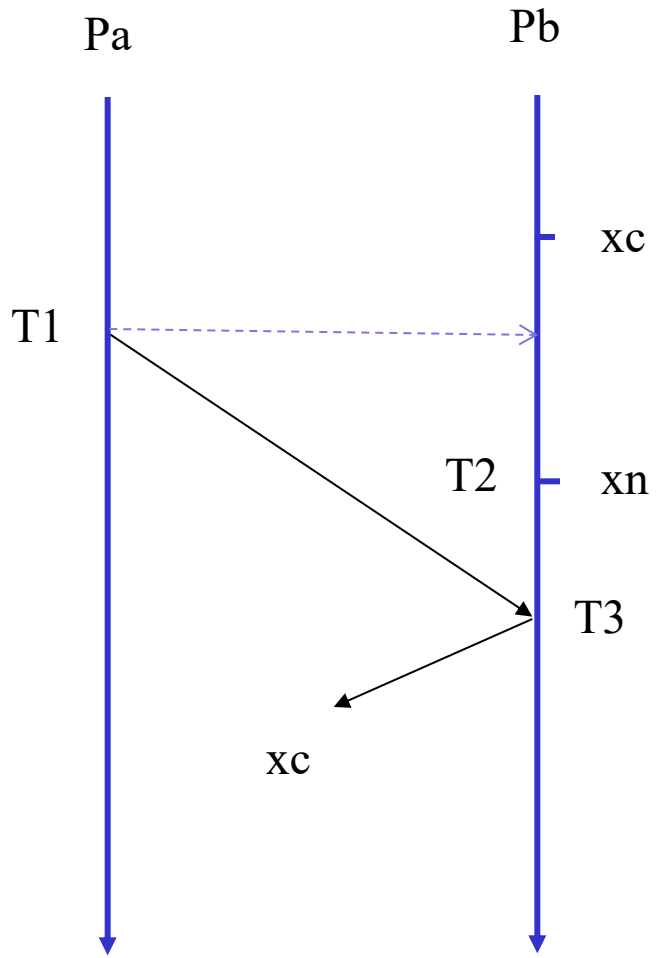
x memorat pe B (doar); valoarea curenta este **xc**

La T1, Pa citeste **x** - trimite un mesaj lui B sa citeasca x (la T1 valoare x = xc)

La T2>T1, Pb scrie **x**; noua valoare este **xn**

La T3>T2, mesajul de citire de la Pa soseste la B

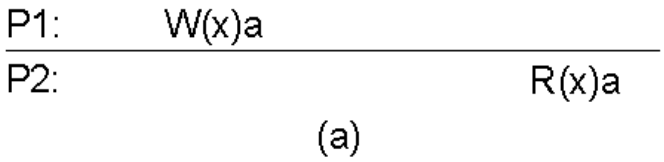
Consistenta stricta → B ar trebui sa intoarca **xc** (greu de implementat)



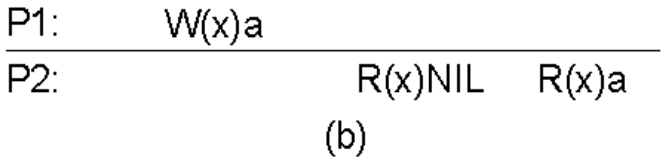
Consistenţa Strictă (2)

Se pastreaza ordinea absoluta globala in timp

Toate scrierile sunt instantaneu vizibile tuturor proceselor



(a)



(b)

W(x)a = scrie in x valoarea a R(x)a = citeste din x valoarea a

Axa orizontala este timpul fizic

Comportarea a doua procese operand pe acelasi item x

a) Memorie strict consistenta.

b) Memorie care nu este strict consistenta. DE CE ???

- Model greu de implementat
- ne-realist - Programele concurente nu sunt bazate pe timpul global sau pe viteza proceselor (ex. Producer / Consumer)
- Sunt necesare modele mai slabe

Consistenta Secvenţială

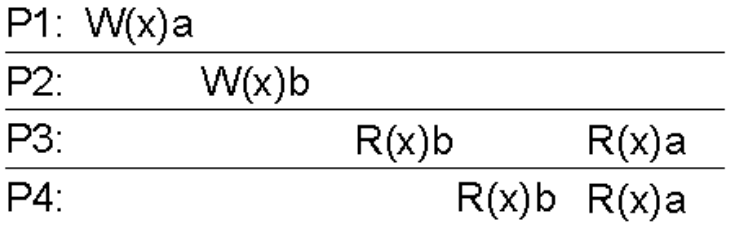
Rezultatul oricarei executii este acelasi cu cel obtinut daca
(1)operatiile (read, write) ale tuturor proceselor asupra depozitului de date sunt executate intr-o **secventa** oarecare si
(2)operatiile fiecarui proces individual apar in aceasta secventa in **ordinea** specificata de **programul** sau.

Orice intretesere valida a operatiilor read si write este acceptata

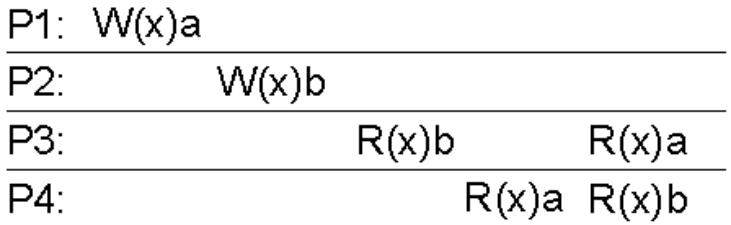
Toate procesele vad aceeasi intretesere de operatii

Operatiile nu au amprente de timp

Notatie: W(x)a – procesul scrie valoarea a in variabila x



(a)



(b)

Depozit **(a)** secvential consistent si **(b)** nu este secvential consistent. **DE CE ?**

Consistenţa Secvenţială – ordonari valide

x, y, si z sunt initializate la 0

Proces P1	Proces P2	Proces P3
x = 1;	y = 1;	z = 1;
print (y, z);	print (x, z);	print (x, y);

- Trei procese concurente.
  - assignment = write
  - print = doua read simultane

- Sunt posibile 90 ordonari valide diferite ale instructiunilor

  → 90 = 720 (=6!) / 8 **Interpretati formula !**

- Aplicatia care foloseste acest model ar trebui sa le accepte pe toate!

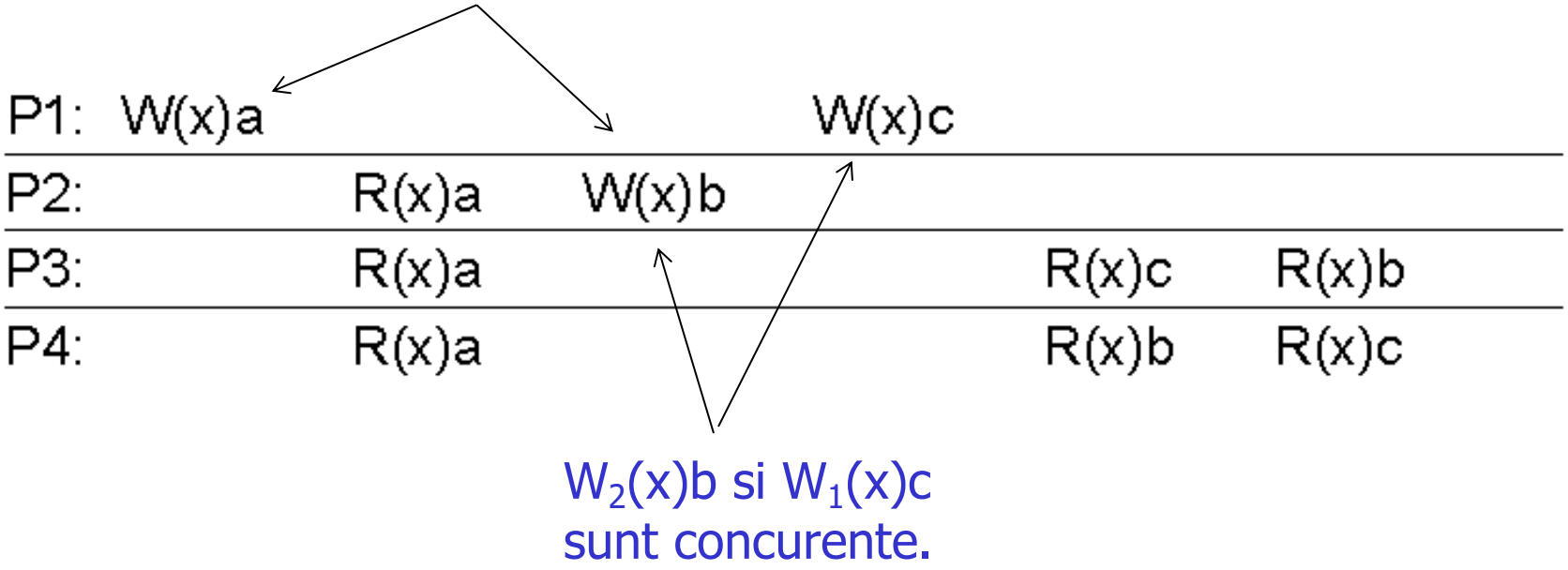
Consistenţa Cauzală (1)

- Doua categorii de operatii:
  1. Legate Cauzal (dependente cauzal)
     - ex:
       - procesul p executa (1) write x;
       - ulterior, procesul q executa (2) read x; (3) write y
       - (1) si (3) sunt legate cauzal
  2. Concurente

- Conditie necesara:
  Operatiile "write" care sunt potential legate cauzal trebuie sa fie vazute de toate procesele in aceeasi ordine. Scrierile concurente pot fi vazute in diferite ordini pe masini diferite.

Consistenţa Cauzală (2)

$W_1(x)a$ si $W_2(x)b$ sunt dependente cauzal;



$W_2(x)b$ si $W_1(x)c$ sunt concurente.

Aceasta secventa este permisa cu o memorie cauzal-consistenta, dar nu cu una secvential sau strict consistenta **DE CE ???**

Consistenta Cauzala (3)

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)

a) Violare a consistentei cauzale. (Cele doua scrieri sunt legate cauzal.)

b) O secventa corecta de evenimente intr-o memorie cauzal-consistenta. (Cele doua scrieri nu mai sunt legate cauzal.)

Consistenta FIFO (1)

- Conditie Necesara:
  Scrierile facute de un singur proces sunt vazute de toate celelalte procese in ordinea in care au fost executate, dar scrierile din procese diferite pot fi vazute in ordini diferite de procese diferite.

- Usor de implementat
  - etichetand fiecare operatie write cu perechea (proces, numar secventa)

Consistenta FIFO (2)

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

- O secventa valida de evenimente pentru consistenta FIFO
- Consistenta cauzala?

Consistenta FIFO (3)

Process P1	Process P2
x = 1;	y = 1;
if (y == 0) kill (P2);	if (x == 0) kill (P1);

Doua procese concurente.

- Consistenta FIFO: ambele procese pot fi omorate DE CE ???
  - deoarece scrierile sunt vazute in ordini diferite
- Prin contrast
  - Consistenta Secventiala: sase posibile intreteseri, nici una cu ambele procese omorate

DE CE ???

Gruparea operatiilor

- Datele partajate pot fi asociate cu o variabila de sincronizare.
- Un proces foloseste acquire si release pe variabilele de sincronizare
  - acquire – cand intra in sectiunea critica - datele protejate (replicile) de variabila de sincronizare sunt facute consistente
  - release - cand iese din sectiunea critica
- Model specific de folosire variabile de sincronizare
  - Fiecare variabila de sincronizare are un proprietar curent
  - Proprietarul poate modifica datele protejate de variabila de sincronizare, in mai multe sectiuni critice (ramane proprietar)
  - Un proces care vrea sa acapareze (acquire) trebuie sa faca o cerere proprietarului, devenind noul proprietar
  - Mod ne-exclusiv – Mai multe procese detin variabila doar pentru citirea datelor protejate

Consistenta la Intrare

P1:	Acq(Lx)	W(x)a	Acq(Ly)	W(y)b	Rel(Lx)	Rel(Ly)						
P2:							Acq(Lx)	R(x)a			R(y)	NIL
P3:									Acq(Ly)	R(y)b		

- Fiecare variabila partajata **x** este asociata cu o variabila de sincronizare (lock) **Lx**.
- Pentru ca un proces sa modifice date partajate, el trebuie sa capate accesul exclusiv la variabila de sincronizare care protejeaza acele date
  - P1 executa aquire pe x si y inainte de modificare
- Un acquire pe o variabila de sincronizare actualizeaza toate datele partajate protejate de acea variabila
  - P2 citeste corect pe **x** (s-a facut acquire)
  - dar nu si pe **y** (nu s-a facut acquire)

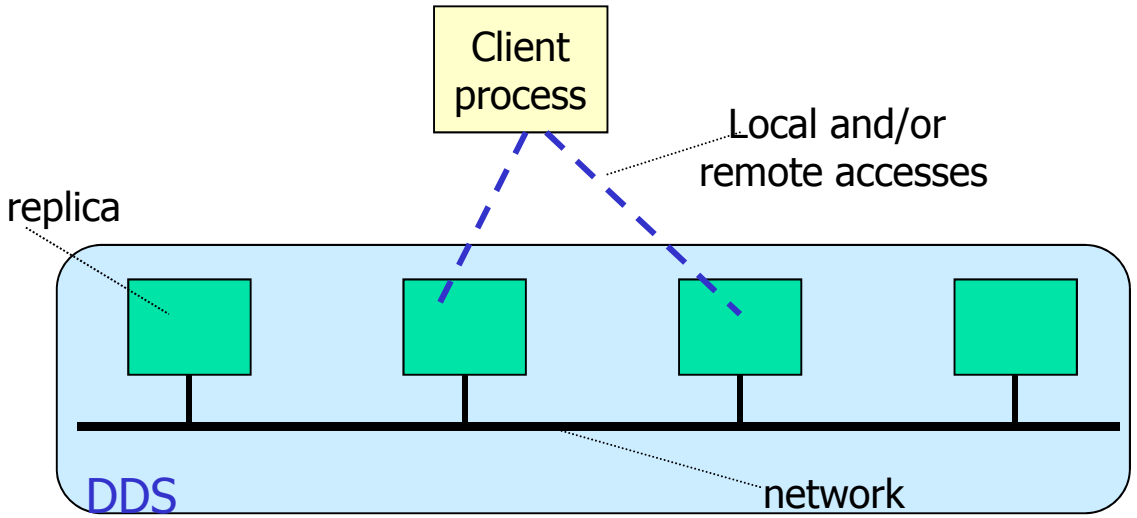
Modele de Consistenta Centrata pe Client

- Ofera garantii de consistenta pentru **un singur client**
  - efectul unei operatii depinde de istoria operatiilor clientului
  - nu se ofera garantii de consistenta pentru accesele intretesute ale diferitilor clienti
- Este un caz special: **lipsesc actualizarile simultane** ale datelor
- Exemple
  - Sisteme de baze de date – cele mai multe procese citesc din BD
  - DNS – pentru fiecare domeniu este asignata o autoritate care face actualizarile -> nu se fac actualizari simultane
  - WWW – paginile actualizate de webmaster sau de proprietar

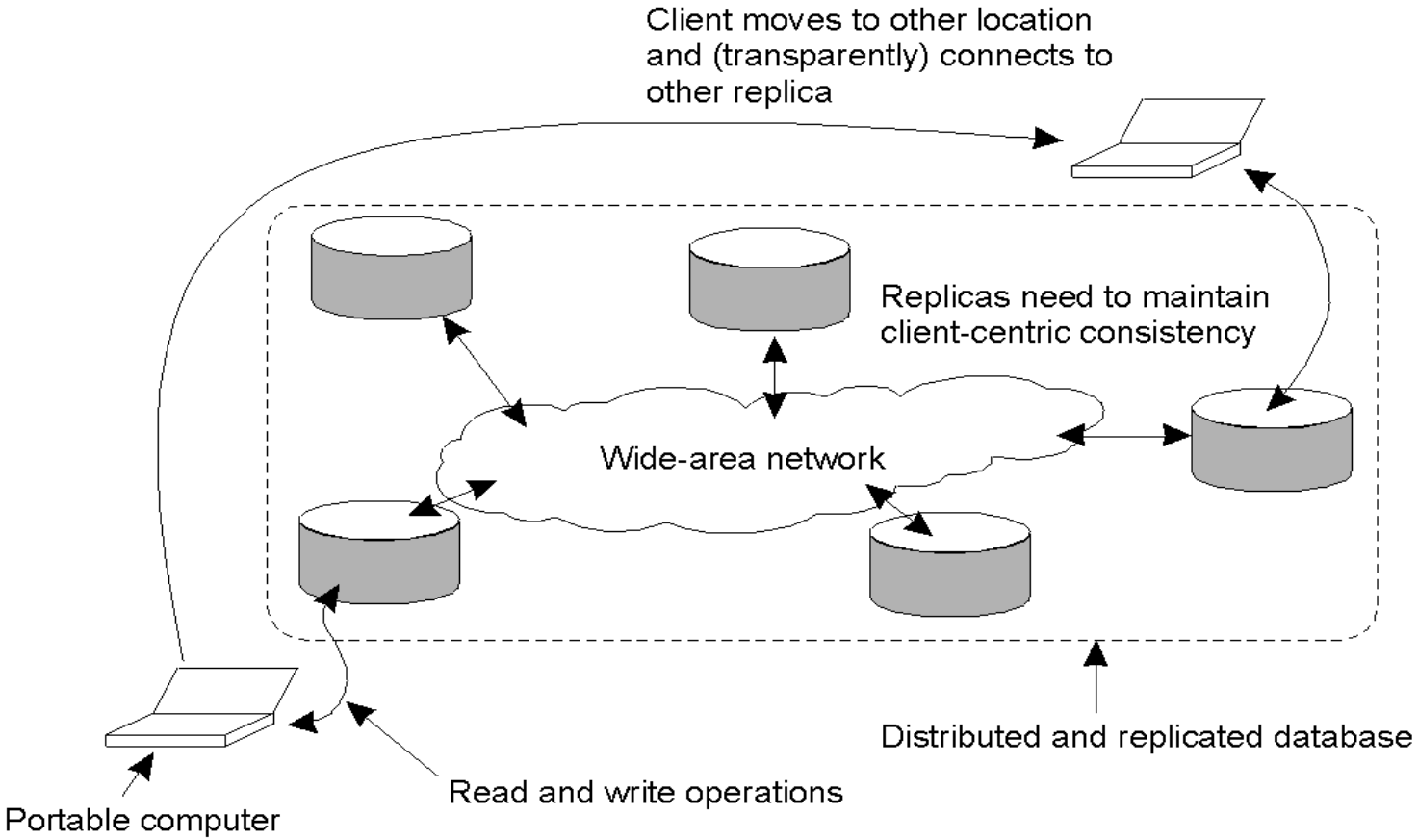
Consistenta "in cele din urma" (eventual consistency):

- in orice moment **pot exista replici neactualizate**
- in absenta indelungata a unor actualizari, toate **replicile vor deveni "in cele din urma" consistente**
- **conditie**: sa existe un mecanism de propagare a oricarei schimbari la toate replicile
- consistenta acceptabila cand clientul nu acceseaza replici diferite intr-un interval scurt de timp

Datele DDS (Distributed Data Store) vor deveni consistente "dupa un timp"



Consistenta Client-centric: un exemplu



- **Un utilizator mobil accesand diferite replici ale unei baze de date distribuite**
  - valorile citite si modificarile facute pe o replica sa fie regasite la cealalata replica
  - da garantii unui singur client privind consistenta acceselor la replici diferite ale acelui client

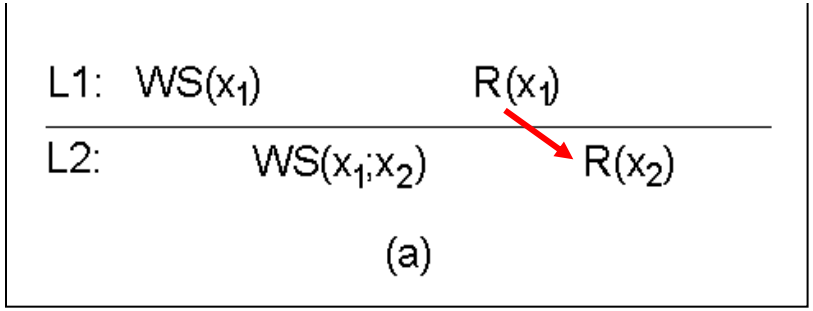
Citiri monotone (Monotonic Reads)

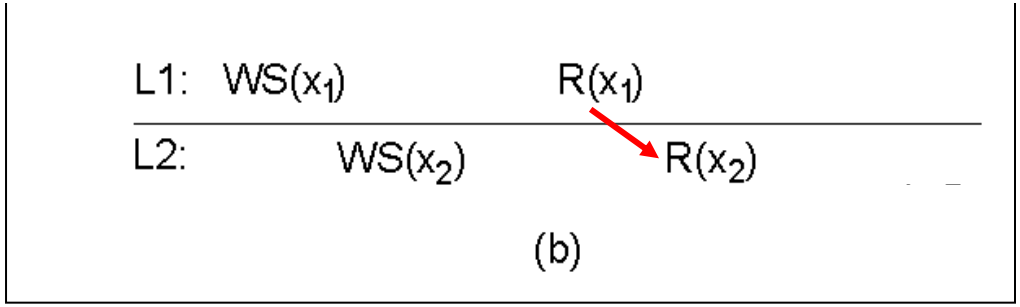
Exemplu: baza de date distribuita, pentru e-mail.

- Fiecare utilizator are o cutie postala distribuita si replicata pe mai multe masini
- Mail-urile pot fi inserate in diferite locatii
- Actualizarile se fac mai lent, cand este necesar (la cerere)
  - de ex. cand utilizatorul deschide cutia postala in alta locatie (acceseaza o alta copie)
- Presupunem ca utilizatorul citeste mesajele din cutia postala in locatia A si ulterior in B
- Citirea nu modifica cutia postala (de ex. nu sterge mesajele citite)
- Consistenta "Monotonic Reads" asigura ca mesajele citite de un proces in A vor fi regasite, de acelasi proces, la accesul in B

Citiri monotone (2)

Daca un proces citeste valoarea unei date x, orice citire succesiva a lui x de catre acel proces va returna acea valoare sau o valoare mai noua.





Operatiile *read* excutate de un singur proces *P* pe doua copii locale diferite ale aceleiasi baze de date.

Notatii: WS(x1) – Write Set - este seria de operatii de scriere pe x executate in locatia 1 de la initializare;

WS(x1;x2) => WS(x1) este parte a lui **WS(x2)**

a) Memorie **consistenta** "monotonic-reads" DE CE ???

- operatiile pe **x** in **L1** au fost propagate la **L2**

b) Memorie **ne-consistenta** "monotonic-reads"

Scrieri monotone (Monotonic Writes)

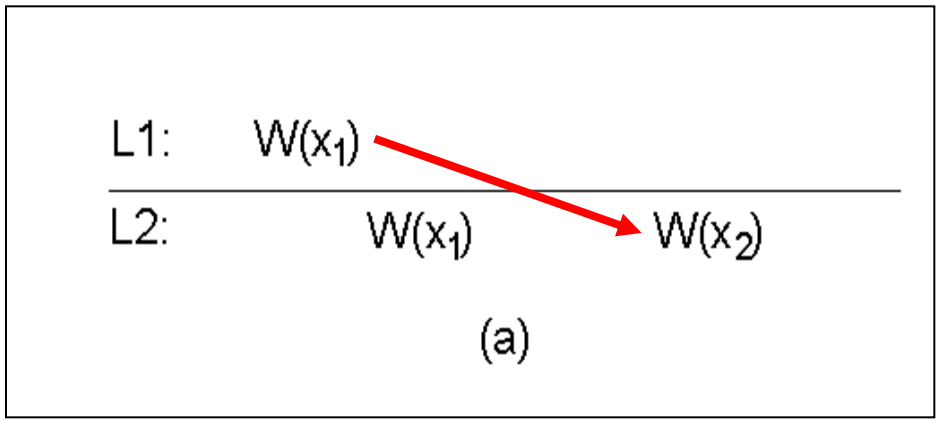
Exemplu: actualizarea unei biblioteci software

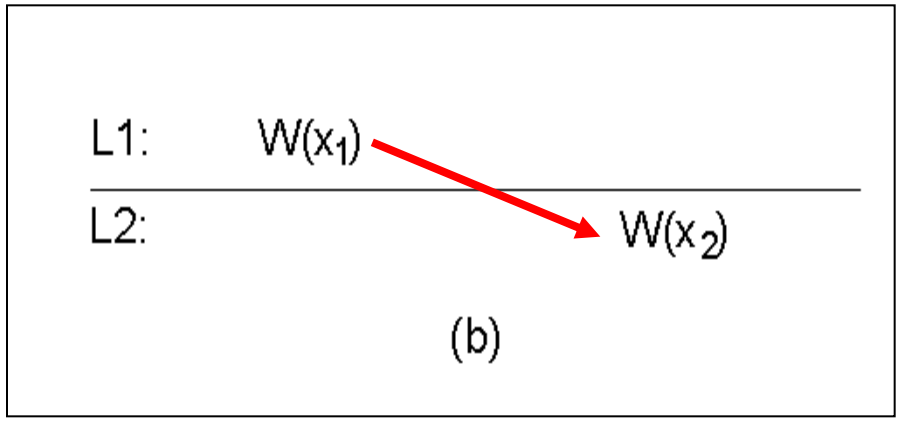
- actualizarea inseamna inlocuirea unei functii (sau a catorva functii), ducand la o noua versiune
  - schimba doar "o parte a valorii" bibliotecii
  - se face pe o replica
- o noua actualizare facuta pe o alta replica cere ca asupra ei sa se fi operat deja actualizarea anterioara
  - asigura obtinerea unei noi versiuni a bibliotecii
  - noua versiune incorporeaza toate actualizarile precedente
  - operatiile de scriere se aplica in aceeasi ordine tuturor replicilor

Scrieri monotone (2)

O operatie *write x* a unui proces este terminata inaintea oricarei operatii *write x* ulterioare a aceluiasi proces.

Obs. Seamana cu consistenta FIFO, dar se refera la un singur proces





Operatiile de scriere executate de un proces *P* pe doua copii locale diferite ale aceluiasi depozit de date

a) Un depozit consistent "monotonic-writes" - operatia W(x1) asupra lui x in L1, a fost propagata la L2 inainte de W(x2)

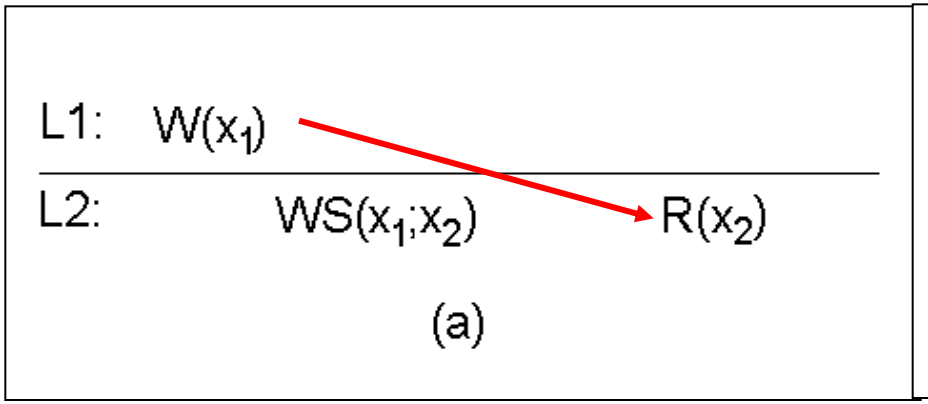
b) Un depozit ne-consistent "monotonic-writes".

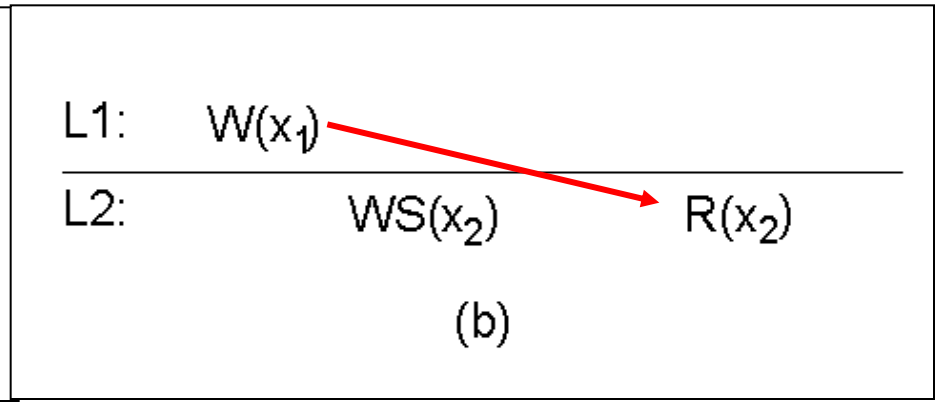
Citirea scrierilor proprii (Read Your Writes)

- **Exemplu**: actualizarea paginilor Web si observarea efectelor.
  - Actualizarea unei pagini Web are ca efect scrierea in fisierului pastrat de server
  - Citirea ulterioara a paginii se poate face dintr-o copie a fisierului pastrata de o replica a serverului
  - Read Your Writes asigura ca fisierul returnat de replica include modificarile facute anterior de utilizator
  - Cazul unei copii cache:
    - Daca browserul returneaza o copie din cache, Read Your Writes asigura invalidarea cache-ului la modificarea paginii Web si obliga la aducerea paginii actualizate de la server

Citirea scrierilor proprii (2)

Efectul unei operatii *write x* a unui proces P va fi totdeauna vazut de operatiile *read x* executate ulterior de acelasi proces P.





a) Un depozit care **ofera** consistenta "read-your-writes" (efectele scrierii lui x la L1 au fost propagate la L2 inainte de citire)

b) Un depozit care **nu ofera** consistenta "read-your-writes".

Scrierile urmeaza citirile (Writes Follow Reads)

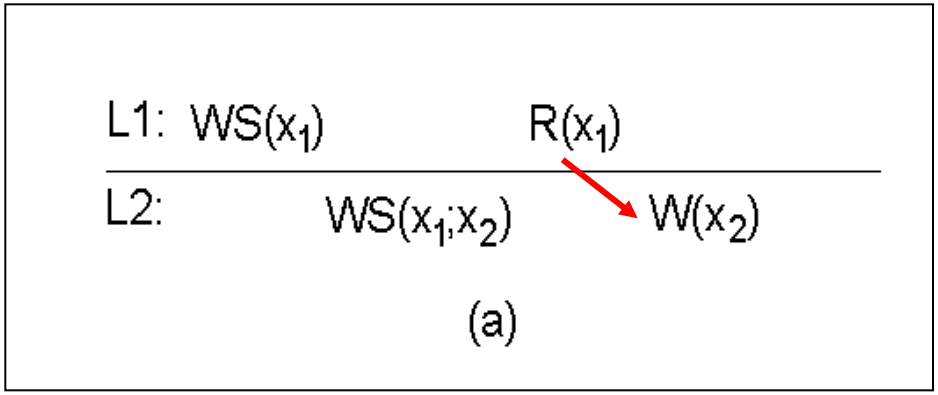
Exemplu: intr-un serviciu de stiri (newsgroup)

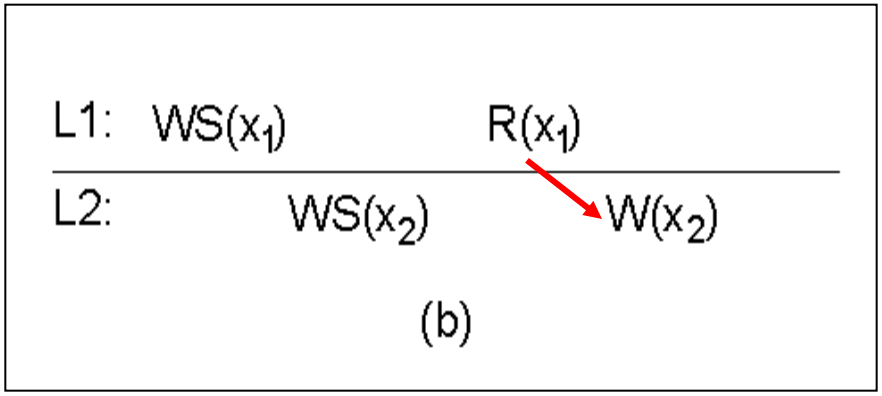
- un utilizator citeste un articol A
- apoi el reactioneaza si scrie (posteaza) un raspuns B
- consistenta "writes follow reads" garanteaza ca, in orice copie a newsgroup-ului, B va fi scris doar dupa ce A a fost scris de asemenea.

Efectul: un utilizator al unui serviciu de stiri intr-o retea nu vede o reactie la un articol fara sa vada si articolul la care se refera reactia

Scrierile urmeaza citirile (Writes Follow Reads)

O operatie *write x* executata de un proces P dupa o operatie prealabila *read x* a aceluiasi proces se executa garantat pe valoarea citita a lui *x* sau pe una mai recenta.





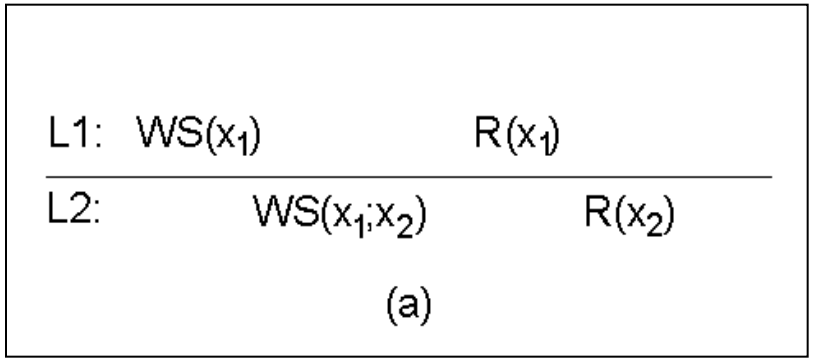
a) Un depozit care **ofera** consistenta "writes-follow-reads"

operatiile de scriere din L1, care au dus la valoarea citita in L1, apar in setul de scrieri de la L2 unde acelasi proces face apoi o scriere

b) Un depozit care **nu ofera** consistenta "writes-follow-reads".

Implementare "naiva" a consistentei centrate pe client

- Fiecarei operatii write i se asociaza un identificator global unic, de catre serverul care a initiat operatia (originea scrierii)
- Pentru fiecare client se pastreaza doua seturi de identificatori:
  - read set – identificatori ai operatiilor write relevante pentru operatiile read executate de client
    - in figura, operatiile de scriere relevante pentru citirea R(x1) sunt incluse in WS(x1)
    - la L2, scrierile relevante pentru R(x2) sunt operatiile din WS(x1;x2)
  - write set – identificatori ai operatiilor write executate de client



(a)

Implementare Monotonic Reads

- Clientul invoca operatia read la un server (replica) si ii paseaza un read set
- Serverul primeste read setul clientului si verifica daca toate operatiile write relevante au fost facute pe replica locala
- Daca nu
  - contacteaza celelalte servere (replici) si face actualizarea
  - eventual paseaza operatia read la serverul care a executat toate operatiile write din setul read al clientului
- Serverul adauga la read set propriile operatii write

- Actiuni similare sunt executate pentru celelalte modele

Probleme de implementare

Problema: seturile **read** si **write** pot deveni prea voluminoase

Solutiie:

- gruparea operatiilor in sesiuni de mai scurta durata (de ex. asociate unei aplicatii) si initializarea lor la inceputul fiecarei sesiuni
- utilizarea unor vectori de timp asociati operatiilor din fiecare sesiune
  - fiecare server Si tine un vector de timp STi asociat unei date,
  - fiecare element STi[j] este amprenta de timp a celei mai recente operatii write cu originea in serverul Sj, pe care Si a procesat-o
  - cand serverul i executa o operatie write, el ii asociaza o noua amprenta de timp pe care o inregistreaza in STi[i]

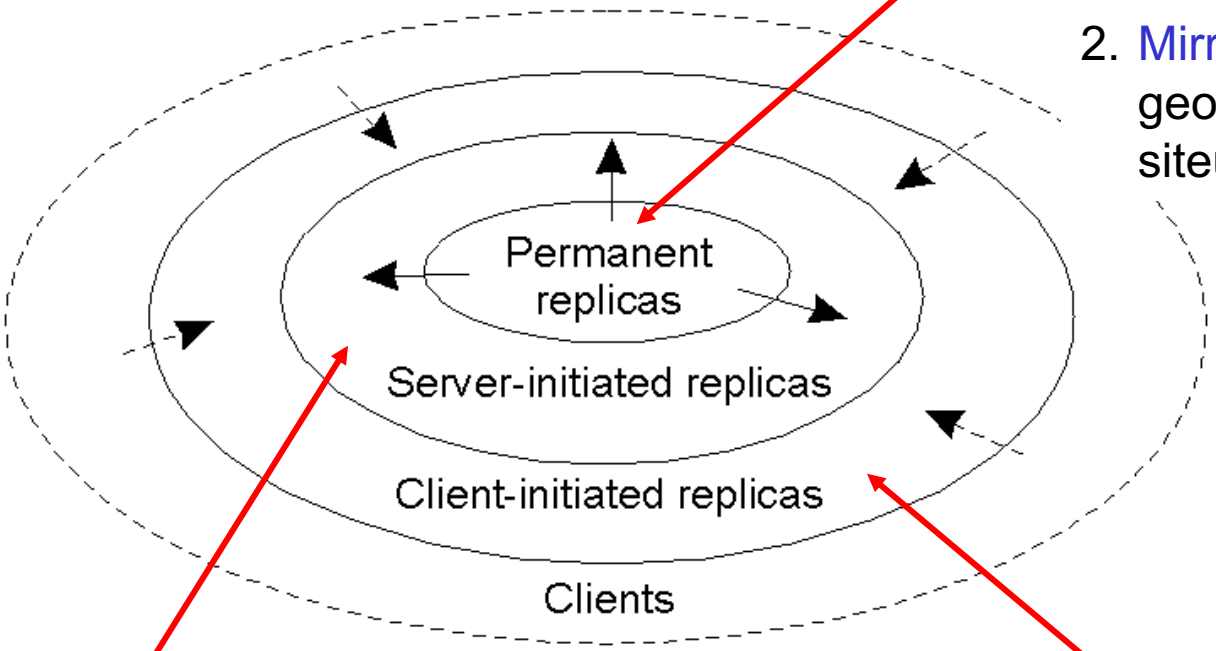
Probleme de implementare (2)

- un vector de timp este mai putin voluminos decat read set
- read set este inlocuit cu un vector de timp notat RS
- fiecare element RS[j] este amprenta maxima a operatiilor write cu originea in j relevante pentru read
- cand clientul invoca un read la serverul i, el trimite RS
- serverul verifica daca a procesat toate operatiile write inregistrate in RS
  - pentru **toti j** la care RS[j] > STi[j], serverul i preia de la Sj si executa operatiile write ne-executate anterior
- serverul actualizeaza eventual RS = max (RS, STi) si trimite raspunsul catre client impreuna cu noul RS

Gestiunea replicilor

- Plasarea replicilor
- Protocoale de propagare a actualizarilor
- Protocoale epidemice

Plasarea Replicilor



Replici permanente

1. Servere "replica" situate in acelasi sistem (cluster)
2. Mirroring – replici larg distribuite geografic - cererile sunt servite de siteul cel mai apropiat

Replici initiate de server pentru a imbunatati performanta

– replici apropiate de clienti

– replici pentru reducerea incarcarii unui server

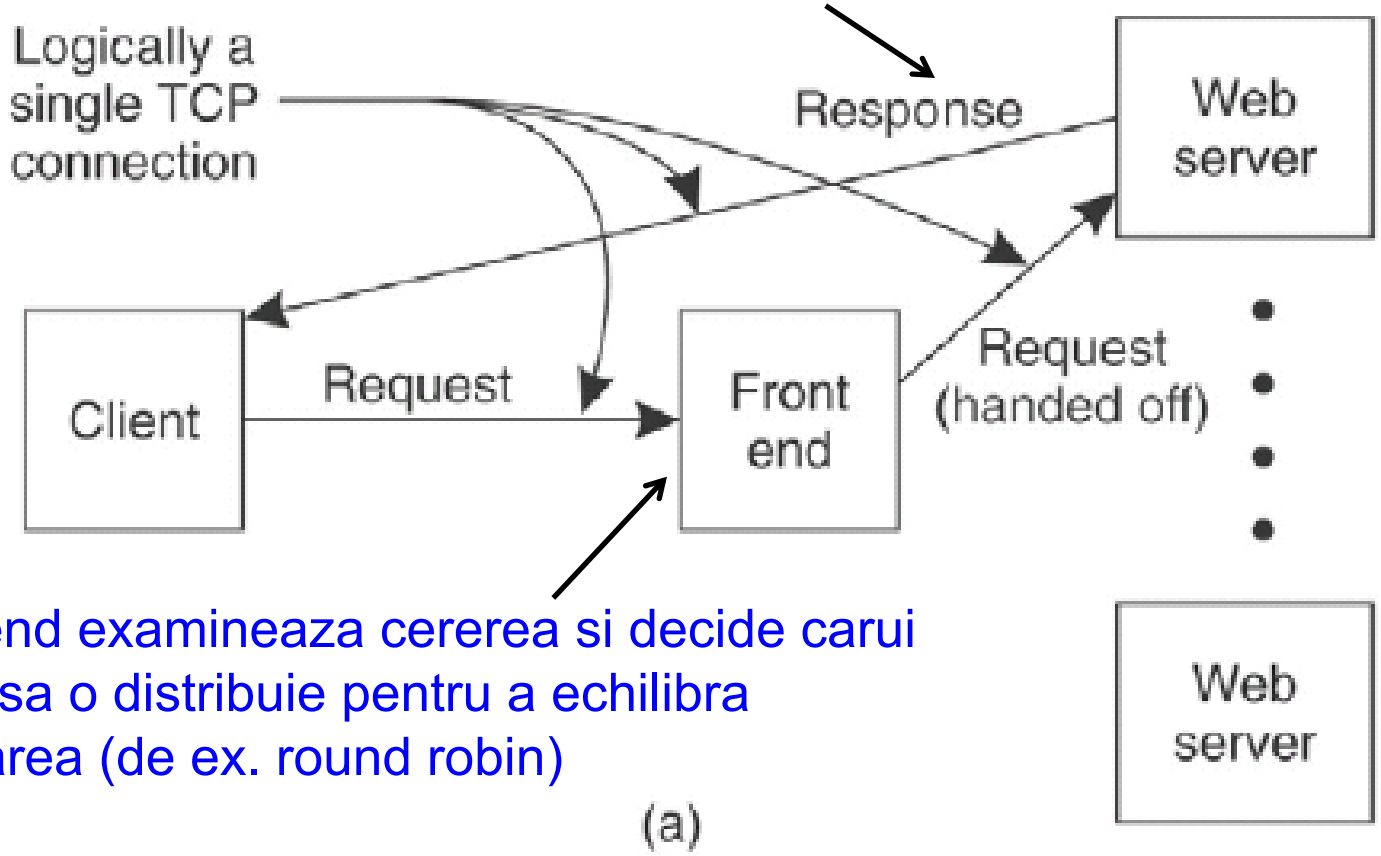
Replici initiate de clienti (caches)

- Utilizate pentru a reduce timp de acces la date
- Date pastrate pentru un timp limitat
- Cache-uri plasate la clienti
- Cache-uri partajate, in apropiere de clienti

Cluster de servere

Paginile sunt replicate pe un numar redus de servere din acelasi cluster

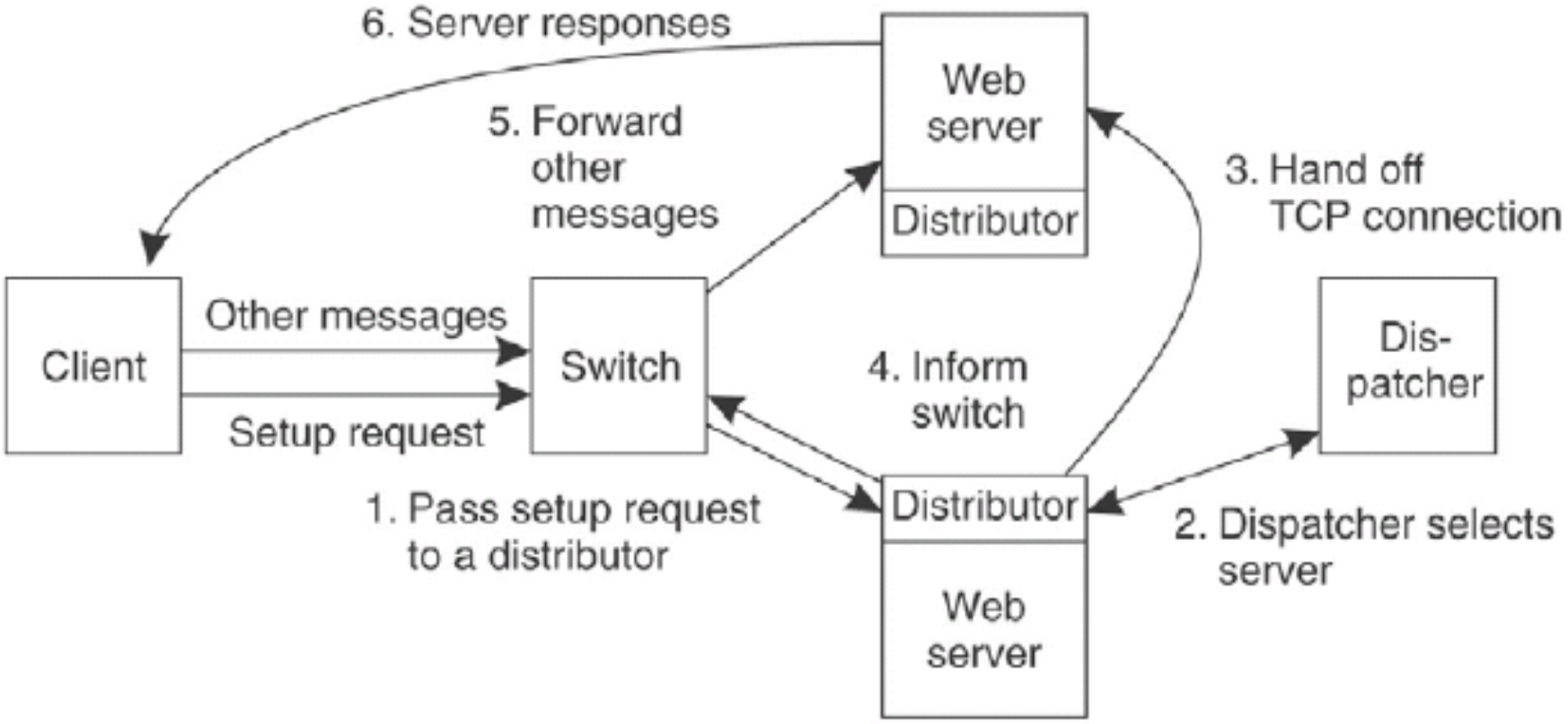
Raspunsul este trimis de serverul Web direct clientului (se foloseste TCP handoff protocol)



Front end examineaza cererea si decide carui server sa o distribuie pentru a echilibra incarcarea (de ex. round robin)

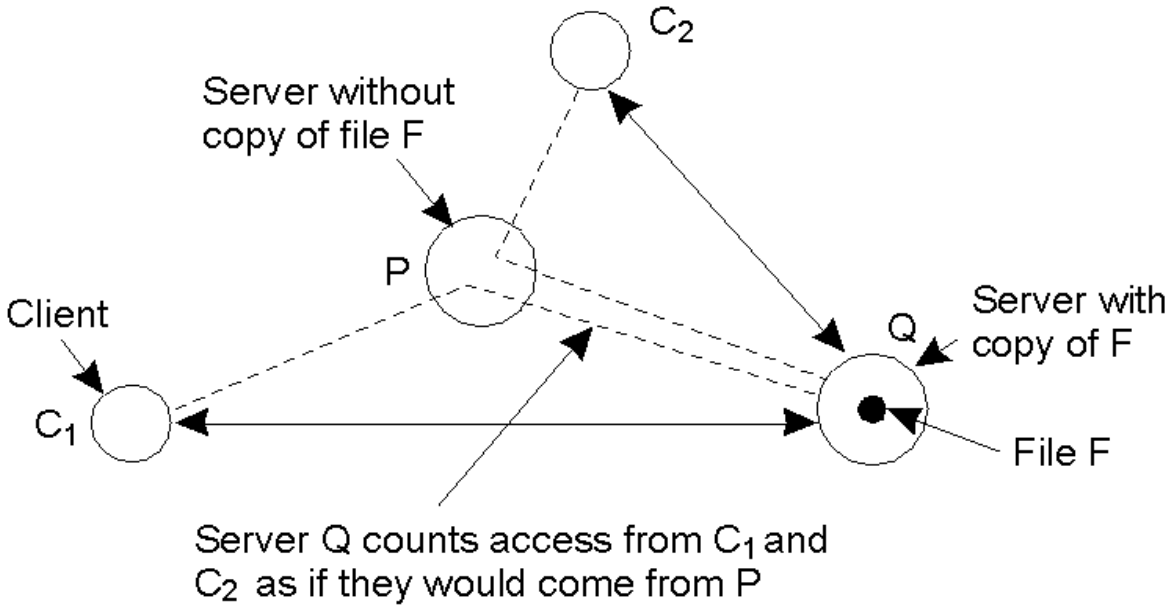
Cluster de servere (2)

Functia Front end este impartita intre un Switch de nivel transport, Distributori si un dispatcher



Replici initiate de server: Cand si unde?

- Fiecare server tine evidenta numarului de accese per fisier si a originii cererilor
- Politica de replicare (Rabinovich)
  - nr cereri pentru F < prag stergere **del(Q,F)** ➔ Q sterge fisier F
  - nr cereri pentru F > prag duplicare **rep(Q,F)** ➔ F replicat pe un alt server
  - nr cereri intre cele doua praguri ➔ Q poate decide mutarea lui F pe alt server (de ex. P)



Protocoale de Propagare

Ce se propaga?

- notificari
  - Utilizate de protocoalele de invalidare
    - replicile sunt informate ca datele nu mai sunt valide
  - Reclama largimi de banda reduse
- datele modificate
  - Utila pentru multe citiri: rata read / write este ridicata
  - Pot fi propagate doar log-urile modificarilor
- operatiile de actualizare
  - Replicare activa – re-executa operatiile asupra datelor
  - Cost comunicare redus dar consuma CPU la executie

Protocoale Pull versus Push

Push: actualizari transmise replicilor fara ca acestea sa ceara

- se aplica pentru replici permanente si "server initiated" care necesita un grad inalt de consistenta
- eficiente pentru multe citiri: rata read/write este ridicata

Pull: actualizari transmise la cerere

- folosite pentru cache-uri client
- eficiente pentru putine citiri: rata read/write este redusa

Exemplu - Comparatie intre protocoale push-based si pull-based in cazul sistemelor cu mai multi clienti si un singur server.

Probleme	**Push-based**	**Pull-based**
Starea replicilor pastrata la server	Lista replicilor si a cache-urilor client	Nimic
Mesaje transmise	update sau invalidare plus fetch&update ulterior	poll si fetch&update
Timp de raspuns la client	Imediat (sau timp de fetch&update)	Timp fetch&update

Protocoale de propagare hibride

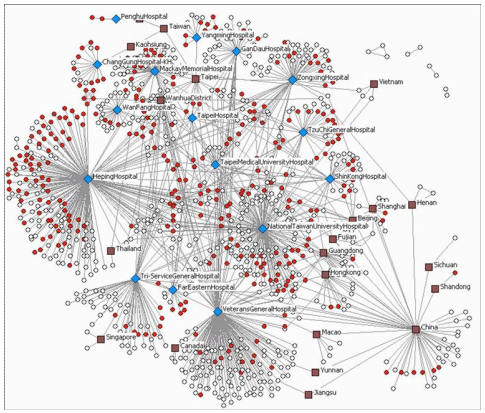
- Bazate pe **Contract** – serverul trebuie sa transmita actualizari la client pe durata specificata in contract

  La expirare contract, actualizarile se trimit la cererea clientului

- Durata poate fi adaptata dinamic in functie de criteriile de contract (lease)
  - –bazat pe vechime
    - contracte de durata pentru elemente cu sansa de a ramane nemodificate (ex. Web)
  - –bazat pe frecventa de innoire
    - contracte de durata pentru clienti ale caror cache-uri trebuie innoite frecvent (pentru celelalte datele se trimit “la cerere”)
  - –overhead-ul la server
    - serverul devine supraincarcat ➔ scade timpul de expirare al noilor contracte (ca urmare, scade numarul de clienti contractuali)

Protocoale de propagare epidemice

- Scop: propagarea actualizarilor la replici cu un numar cat mai mic de mesaje (evita propagarea prin inundare)
- Folosite in depozite de date consistente “in cele din urma”
- Presupunere – propagarea nu este dirijata central, dar
- este initiata la un singur server pentru a evita scrieri conflictuale
- Modelul propagarii se bazeaza pe cel al epidemiilor
- Categorii de servere



- Infectios – detine un update pe care vrea sa-l propage
- Susceptibil – inca ne-actualizat
- Eliminat – detine un update dar nu vrea sa-l propage

- Variante de propagare intre doua servere
- Push based – un server infectios trimite actualizari unui server susceptibil; mai bun pentru putini infectiosi (la inceputul propagarii)
- Pull based – un server susceptibil cere actualizari unui server infectios; mai bun pentru multi infectiosi

Protocoale de propagare epidemice (2)

Varianta gossiping (raspandirea zvonurilor)

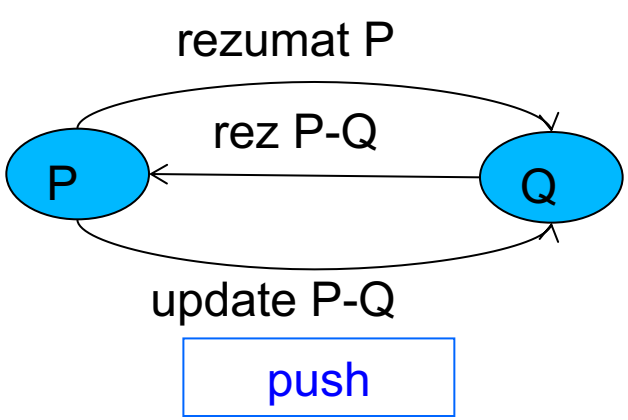
- Vizeaza evitarea propagarii catre noduri deja actualizate
- Serverele infectioase aleg aleator serverele pentru propagare
- Interesul serverului de a propaga actualizarile scade pe masura ce intalneste servere actualizate deja
- Nu garanteaza actualizarea tuturor replicilor
  - se combina cu anti-entropie pentru a completa distributia

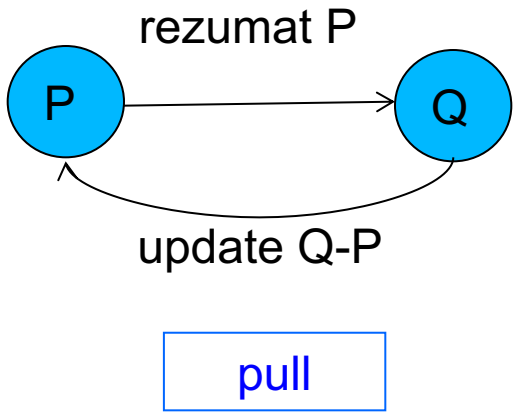



Protocoale de propagare epidemice (3)

Varianta “anti-entropie” (entropie = masura a dezordinii)

- Un server P alege aleator un alt server Q si actualizeaza replicile cu el
- Protocoalele folosite in actualizare
  - **push** - P trimite propriile modificari lui Q
  - **pull** – P preia de la Q noile modificari
  - **push-pull** – P si Q trimit unul altuia modificarile recente



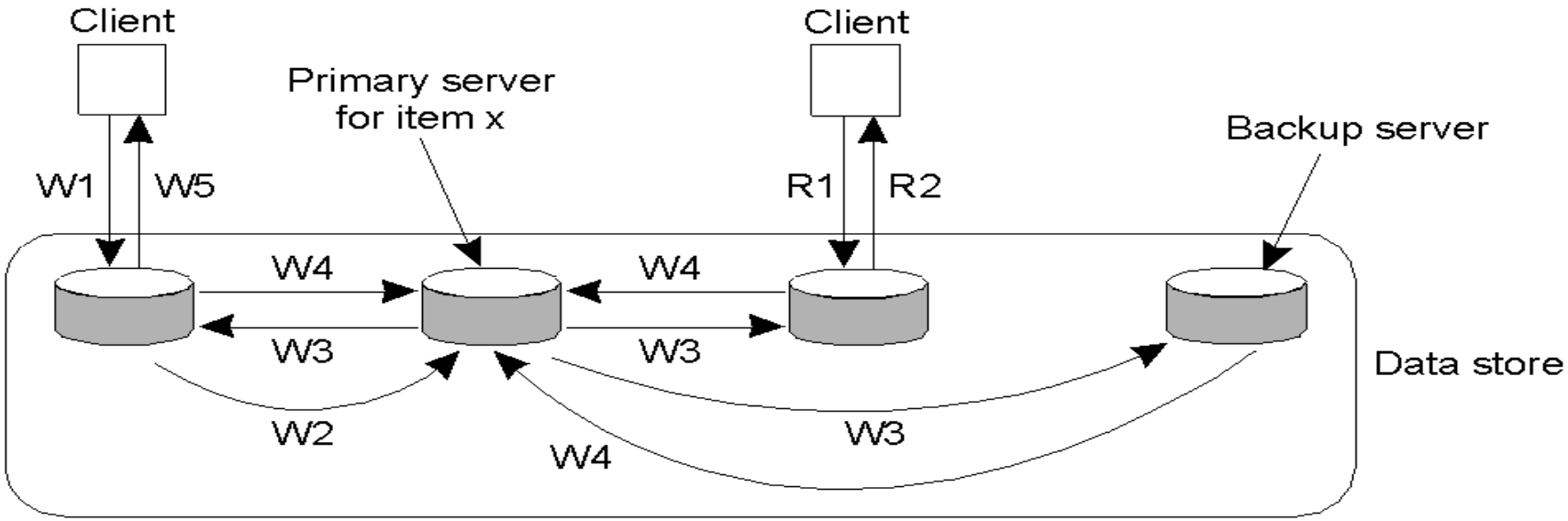


Protocoale de consistenta

- Un protocol de consistenta descrie o implementare a unui model de consistenta
- Protocoale bazate pe o **copie primara**
  - scriere la distanta (Remote-write)
  - scriere locala (Local-write)
- Protocoale cu **scriere** replicata
  - replicare activa
  - protocoale bazate pe cvorum
- Protocoale de coerenta a **cache**-urilor

Protocoale Remote-Write

Protocolul primary-backup: scrierea este blocanta

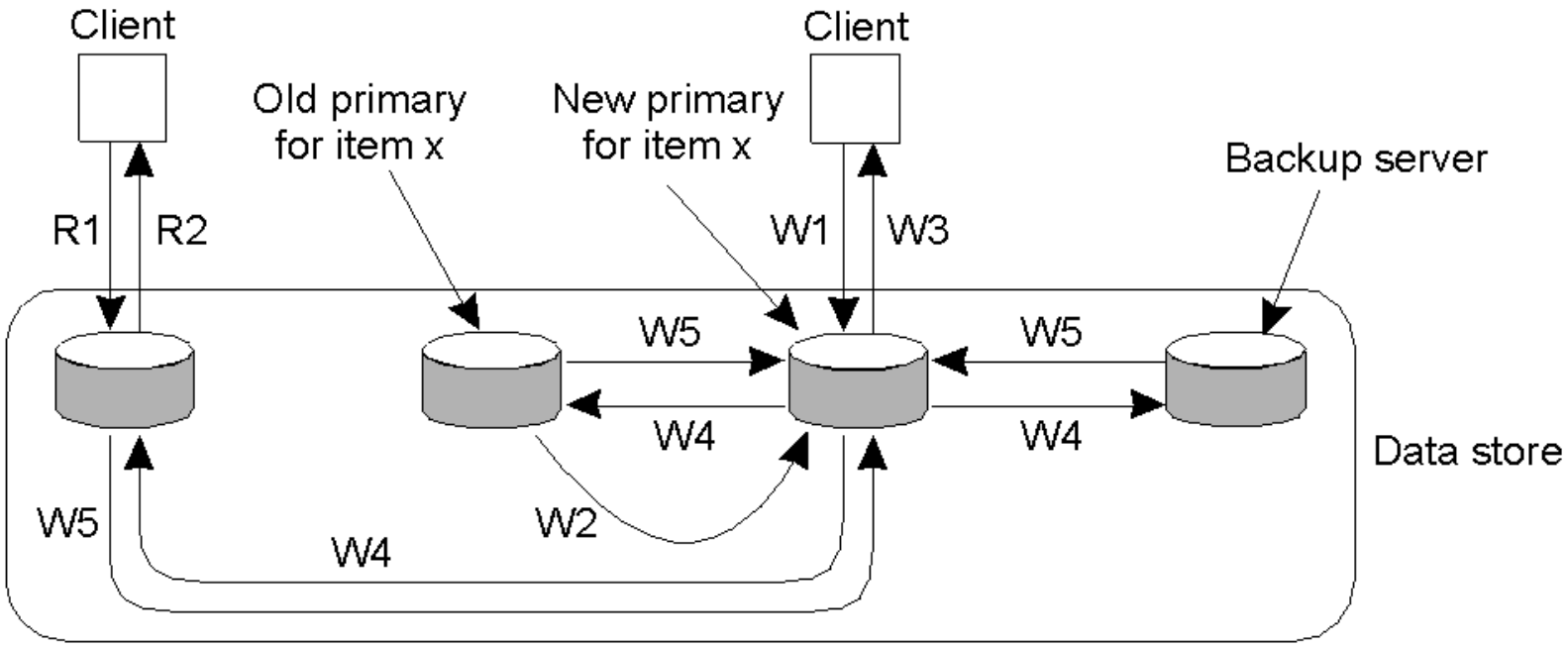


W1. Write request
W2. Forward request to primary
W3. Tell backups to update
W4. Acknowledge update
W5. Acknowledge write completed

R1. Read request
R2. Response to read

- Implementeaza **consistenta secventiala** (copia primara poate ordona toate operatiile de scriere primite) – celelalte replici vad scrierile in aceeasi ordine
- se poate si scriere non-blocanta (acknowledge dupa actualizarea locala)
  - alte procese ar putea sa nu vada noua valoare ci una mai veche

Protocoale Local-Write



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

Protocol "primary-backup" in care copia primara migreaza la procesul care vrea sa faca actualizarea

Replicare activa

Fiecare replica are asociat un proces care executa operatiile de actualizare.

Operatiile trebuie executate in aceeasi ordine in toate replicile
Solutie: mecanism de **multicast total ordonat**

In cazul replicarii **obiectelor**, doar acest mecanism nu este suficient

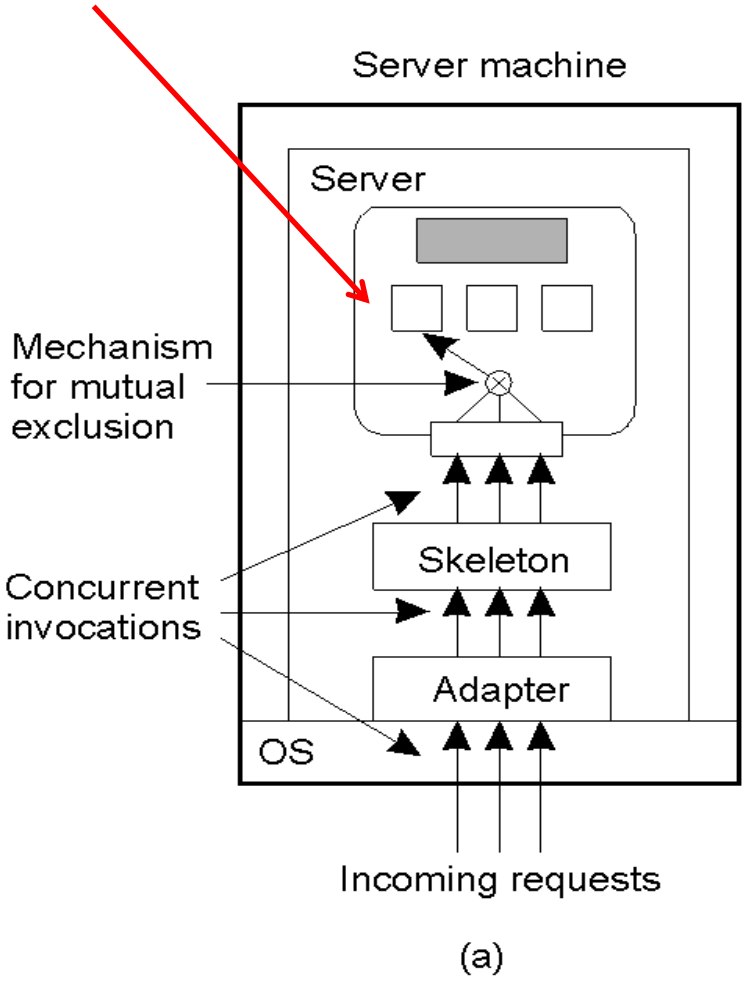
Doua probleme pentru obiecte:

1. Prevenirea executiei concurente a invocarilor aceluiasi obiect
2. Asigurarea executiei in aceeasi ordine a operatiilor in diferite replici

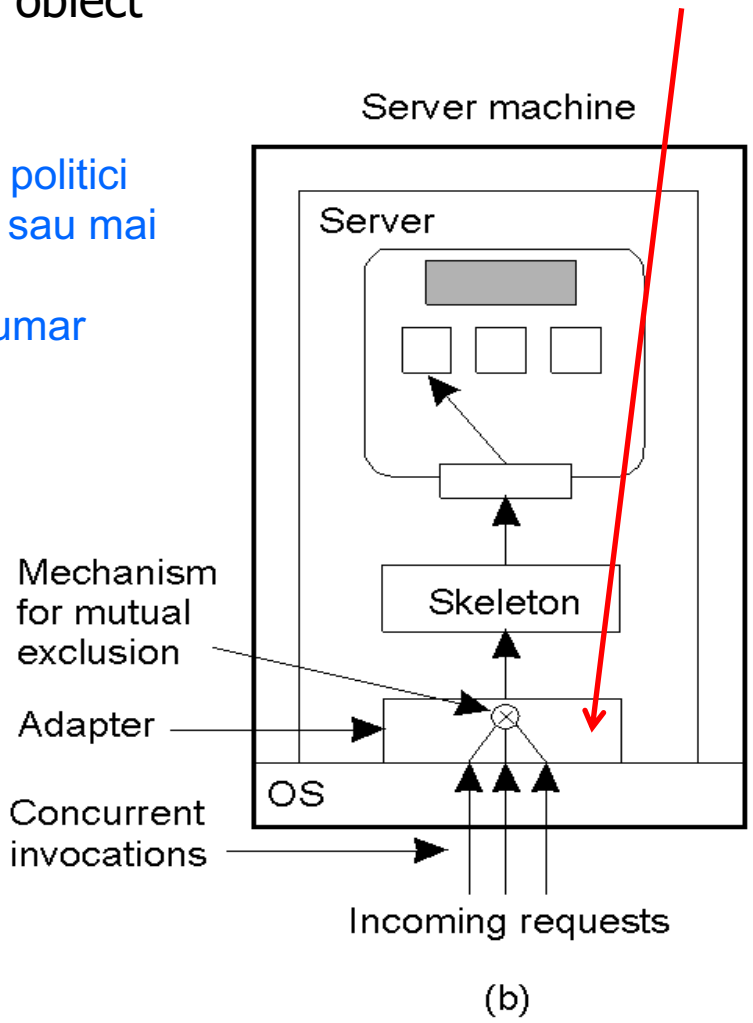
Controlul accesului concurent la un obiect

Obiectul insusi manevreaza accesul concurent – metode *synchronized*

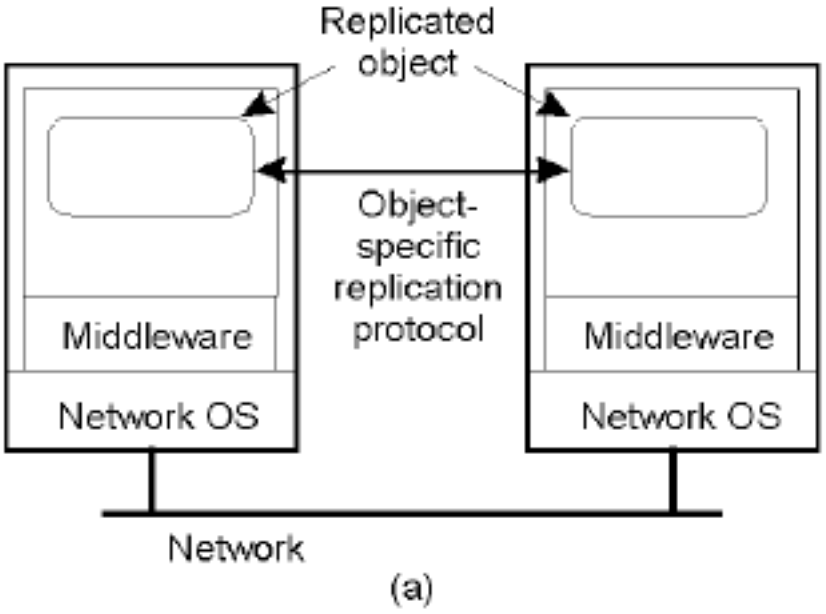
Adaptorul asigura ca invocarile concurente nu lasa obiectul intr-o stare corupta – de ex. asigura un thread per obiect

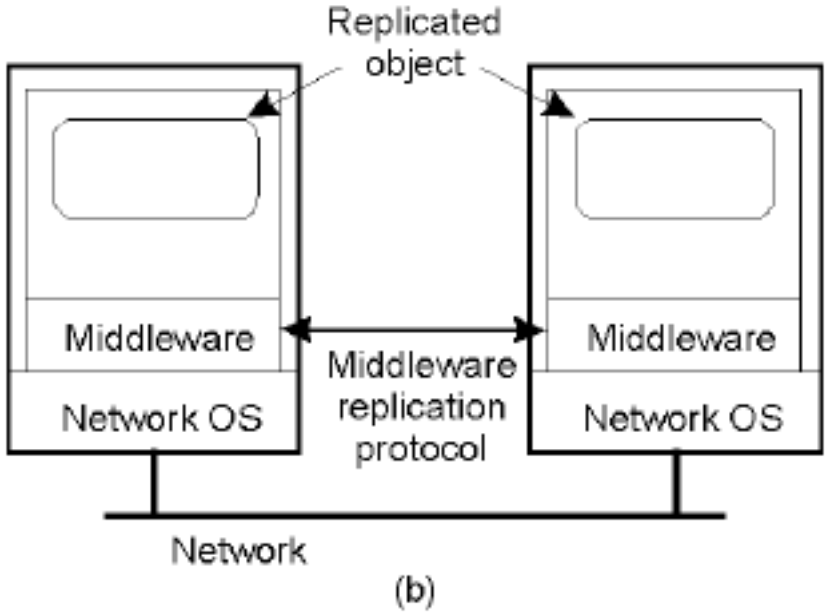


- Adaptor – politici
- un thread sau mai multe
- limitare numar threaduri



Invocari concurente ale obiectelor replicate





- **Problema**: aceleasi modificari de stare sa apara la obiectele replicate
- **Solutia**: invocarile sa se faca in **aceeasi ordine** la toate replicile
- Doua **implementari** posibile:

a) Obiectele sunt "constiente" de replicare (Globe)

b) Sistemul distribuit (middleware) face gestiunea replicilor (Piranha)

- de ex. ordonarea totala a invocarilor la toate replicile

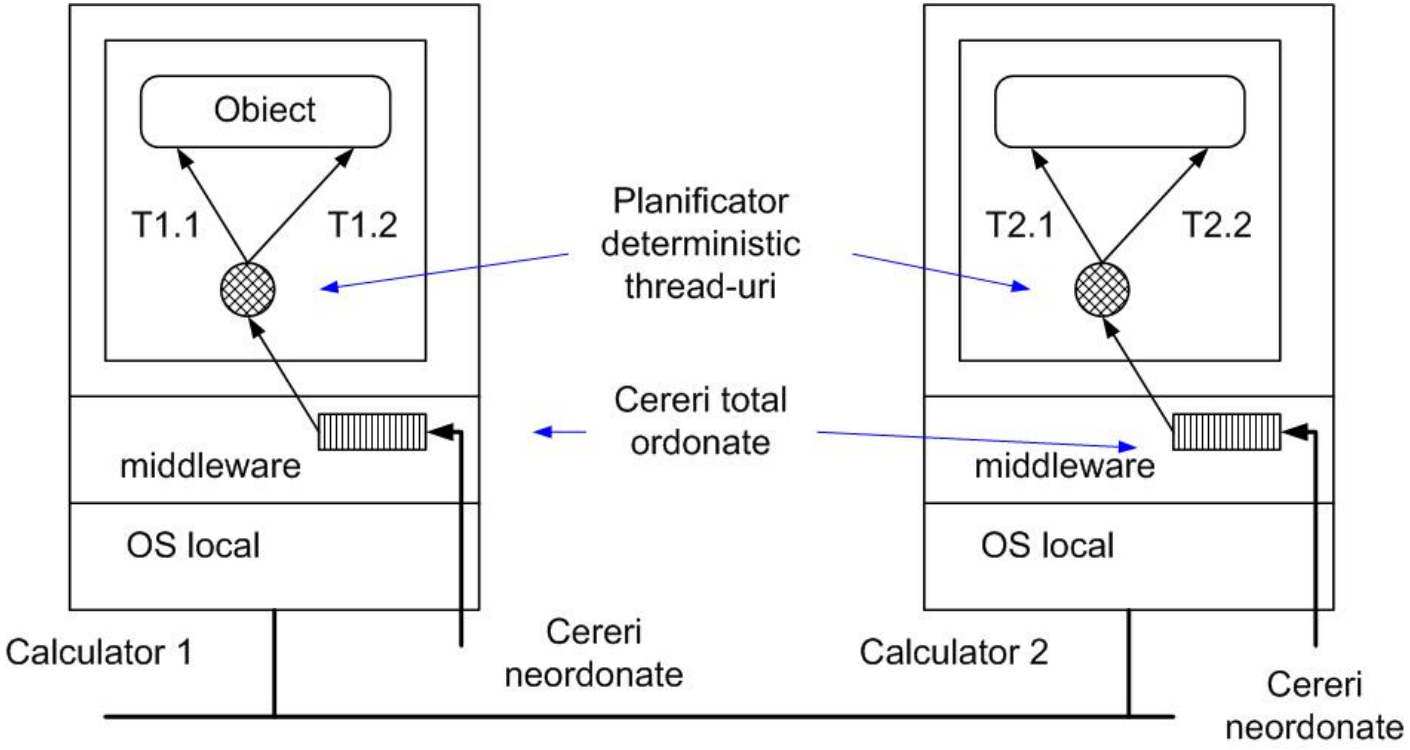
Obiectele sunt "constiente" de replicare

Folosirea unei scheme primary-based la nivel de aplicatie

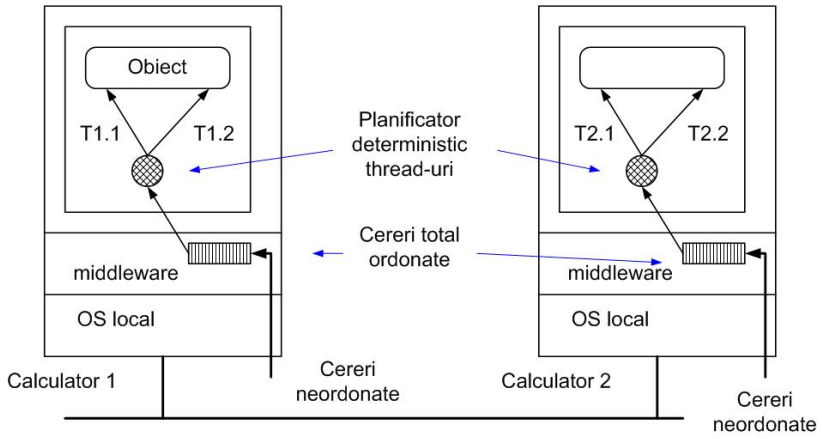
- serializeaza cererile
- neajuns - efort crescut al dezvoltatorului de aplicatii

Implementare consistenta in middelware

- Multicast total ordonat pentru invocari
- In plus - trebuie asigurat si ca threadurile trateaza cererile in ordinea corecta

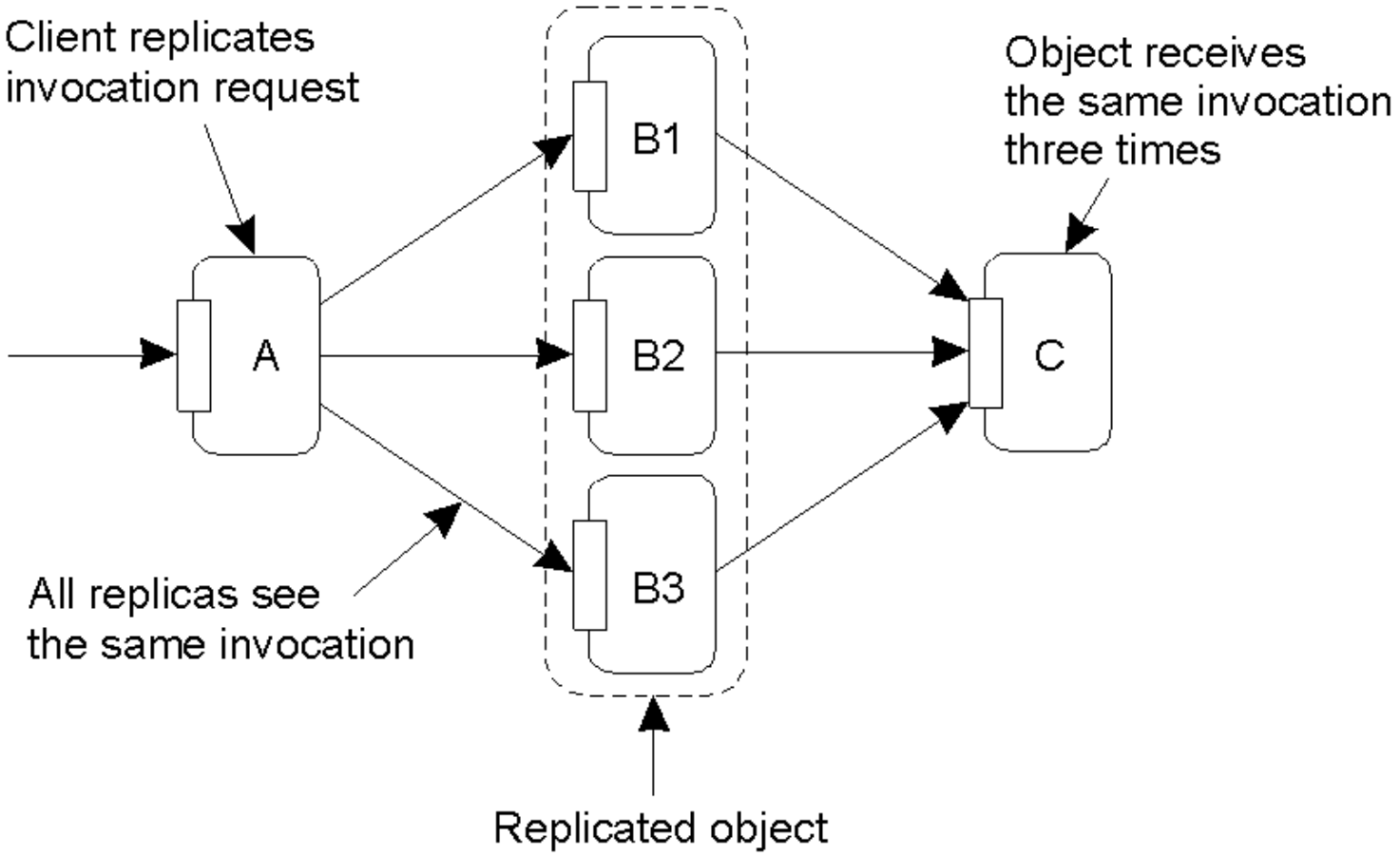


- Functionare servere multithread
  - preiau o cerere si o paseaza unui thread disponibil
  - trec la urmatoarea cerere
  - **planificatorul** de threaduri aloca procesorul threadurilor executabile –ordinea **nu** este aceeasi in toate replicile



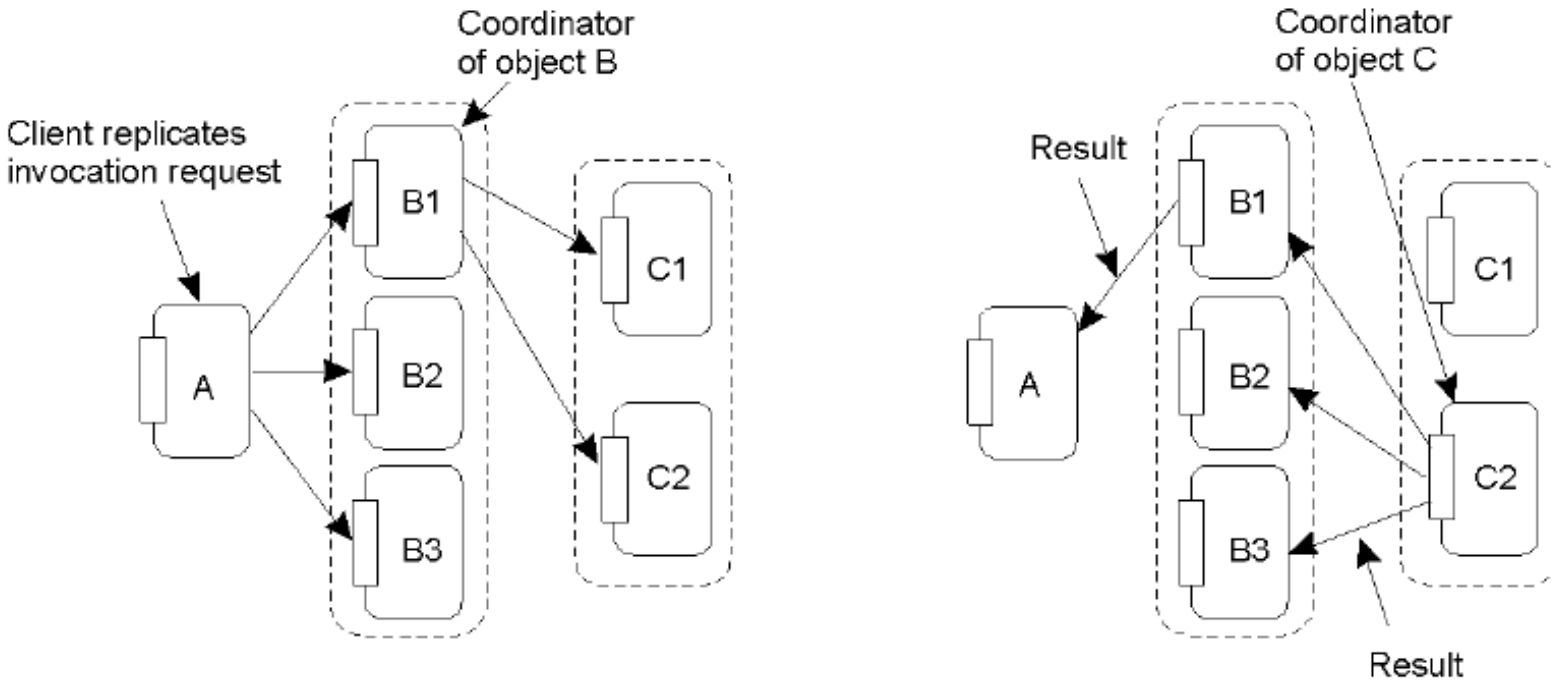
- Solutie: planificator determinist
  - foloseste o varianta de replica primara
    - replica primara determina ordinea threadurilor
    - se sincronizeaza cu celelalte replici prin informatii de context transmise de replica primara
    - se bazeaza pe mecanismul de lock-unlock pentru a porni thread-urile in ordinea stabilita
  - dezavantaj - frecvente comunicari intre replici - ineficient

Invocari Replicate



- Problema: la C ajung invocari replicate; doar una este necesara

Communicatii constiente de replicare



Solutie: bazata pe transmiterea multicast a invocarilor/raspunsurilor

- replicile asigneaza acelasi identificator invocarilor
- doar coordonatorul transmite invocarea
- toate replicile primesc copii ale aceluiasi raspuns

Alternativa: identificarea si eliminarea invocarilor duplicate la un obiect

Protocoale bazate pe cvorum

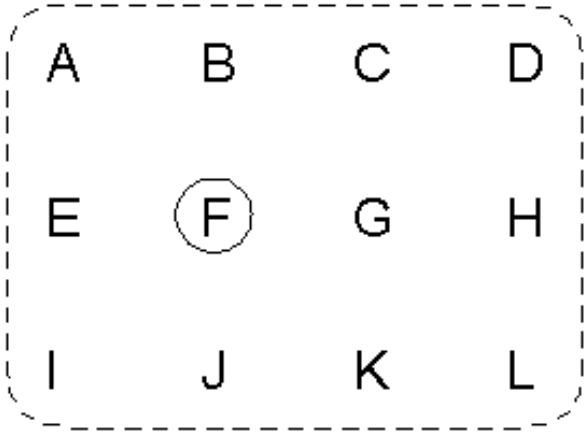
Modelul clasic de gestiune a operatiilor read si write in cazul replicarii datelor este ROWA (Read One, Write All)

In acest caz:

- numarul de replici accesate la read este $N_R = 1$
- numarul de replici accesate la write este $N_W = N$ (egal cu numarul total de replici)

Solutia este convenabila atunci cand frecventa operatiilor read este semnificativ mai mare decat a operatiilor write

In alte cazuri, se pot alege scheme in care numarul de replici citite si numarul de replici scrise sunt ambele mai mari ca 1



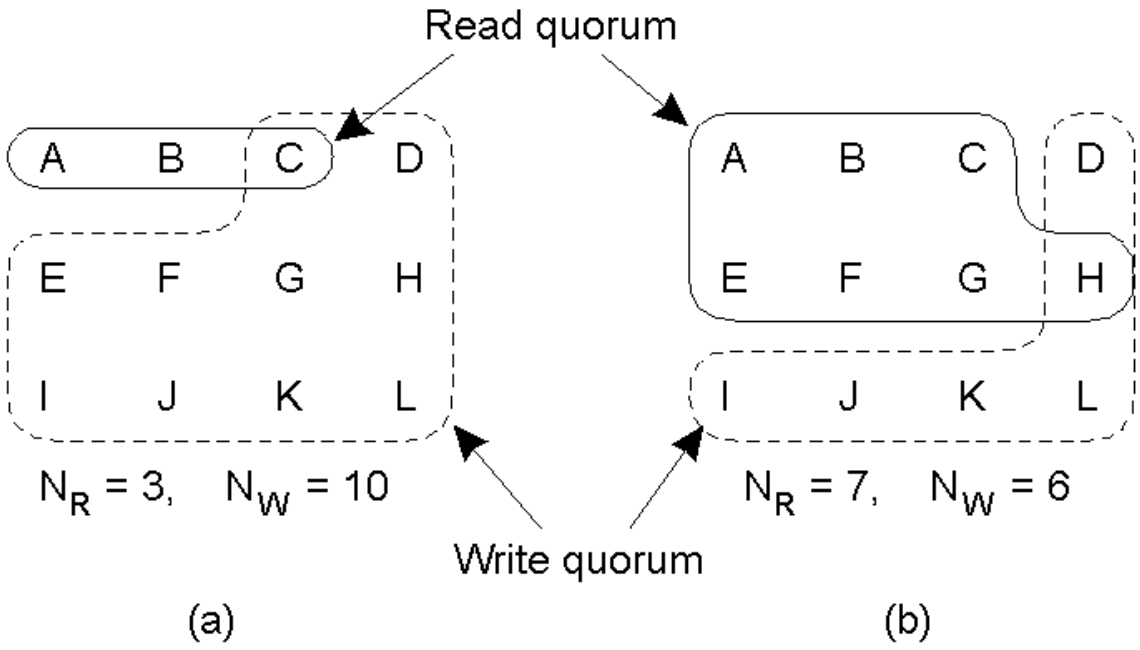
$N_R = 1, \quad N_W = 12$

Protocoale bazate pe cvorum (2)

In alegerile ilustrate aici, N_R si N_W trebuie sa respecte anumite reguli

Se foloseste un numar de versiune pentru fiecare replica

La un write, cele N_W replici capata acelasi numar de versiune.

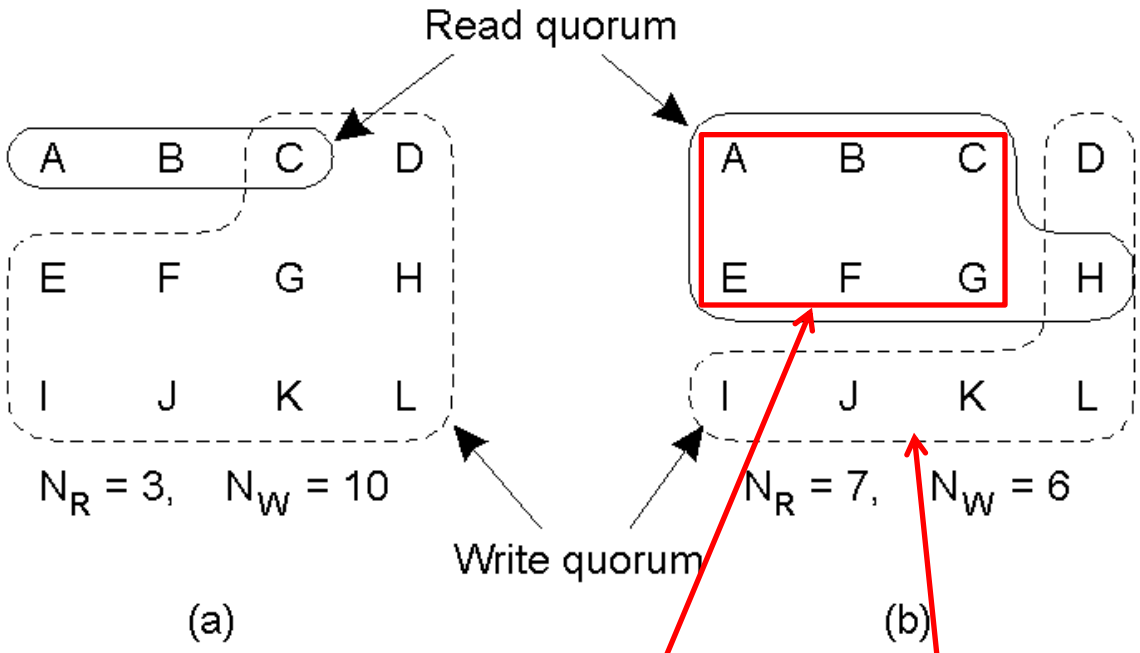


(a) Alegere corecta a seturilor (**forumurilor**) de citire si scriere

$$N_R + N_W > N$$

Oricare ar fi alegerea cititorului, in cvorumul de citire intra cel putin o resursa care are ultima versiune

Protocoale bazate pe cvorum (3)



(b) Alegere ce poate conduce la conflicte write-write

Ex. un proces alege setul de scriere {A,B,C,E,F,G} iar altul {D,H,I,J,K,L} putand rezulta replici cu acelasi numar de versiune dar valori diferite Solutie ???

Eliminarea conflictelor impune $N_W > N/2$

Concluzii ROWA

- (a) Nr+Nw>N – Nu avem scrieri in timpul citirilor
- (b) Nw>N/2 – cel putin un writer

Aplicatie: partajarea fisierelor in CODA

Carnegie Mellon University

Abordarea CODA

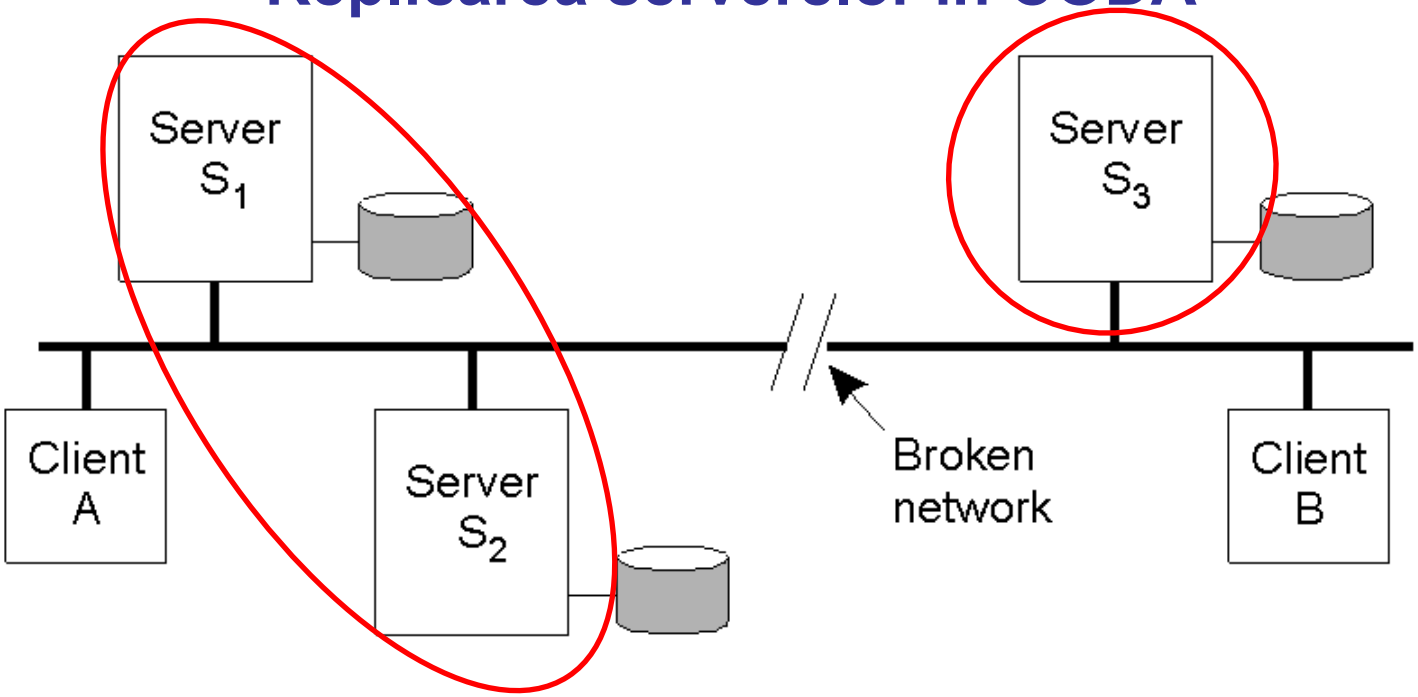
•cand un client deschide un fisier f, o copie a lui f este transferata la client si serverul inregistreaza ca clientul A are o copie a lui f

•mai multi clienti pot deschide f pentru read

•un singur client poate deschide f pentru write

- cand termina sesiunea de actualizare, clientul va trimite serverului fisierul actualizat
- serverul va trimite un mesaj de invalidare clientilor care citesc fisierul
- acestia pot ignora mesajele si pot continua sa lucreze pe copiile locale, neactualizate – justificare ???
  - o sesiune este tratata ca o tranzactie – procesul care citeste poate considera ca sesiunea lui s-a terminat inainte de modificarea lui **f**

Replicarea serverelor in CODA



Unitatea de replicare este **volumul** (colectie de fisiere)

- **VSG** (Volume Storage Group) = grupul de servere care au copia volumului
- **AVSG** (Accessible Volume Storage Group) = serverele la care un client are acces

  In figura {S1, S2} este AVSG pentru clientul A
  {S3} este AVSG pentru clientul B

CODA foloseste un protocol **ROWA** ptr. consistenta unui volum replicat

- la read, clientul foloseste un fisier dintr-un volum accesibil
- la write, un singur client modifica un fisier dintr-un volum accesibil; la sfarsitul sesiunii, el transfera fisierul, in paralel, celorlalte volume accesibile

La partitionare retea (defectare)

- doi clienti pot lucra pe AVSG-uri diferite si pot modifica replici diferite ale aceluiasi fisier
- la refacerea retelei, se propaga modificarile la toate replicile din WSG
- problema: modificari inconsistente

Solutie detectie inconsistente

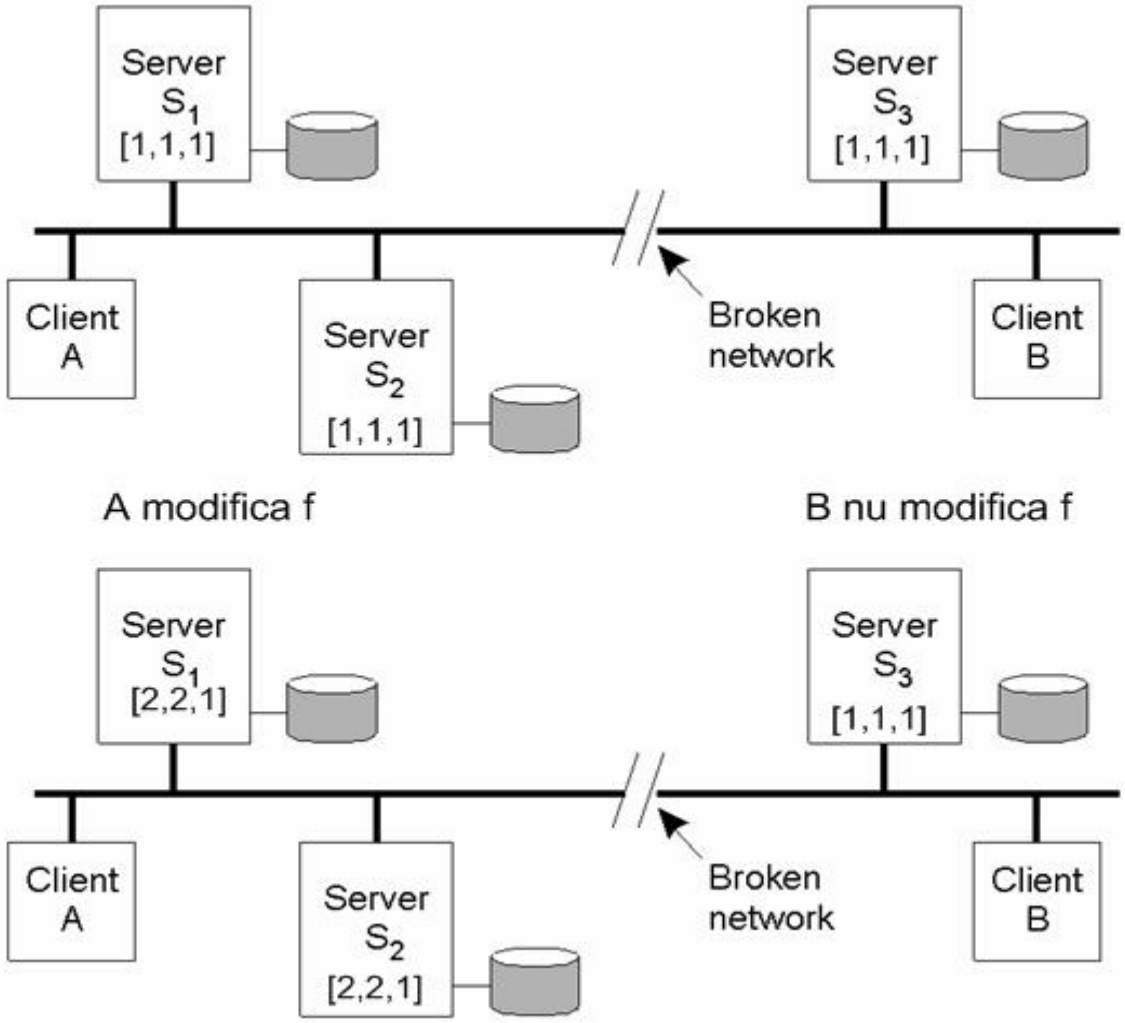
- CODA pastreaza pentru fiecare fisier **f** si server **k** un vector de versiune (Coda Version Vector) $\mathbf{CVV_k(f)}$
- fiecare element $\mathbf{CVV_k(f)[j]}$ este numarul, pastrat de **k**, al versiunii curente a lui **f** pe serverul **Sj**
- dupa rezolvarea defectului de partitionare a retelei, se compara CVV-urile si se detecteaza conflictele

Fara conflict

[1,1,1] = numerele de versiune ale unui fisier, corespunzatoare serverelor S1, S2, S3 in aceasta ordine

doua CVV nu sunt in conflict daca sunt egale sau unul este <= decat celalalt

Modificarile facute de Clientul A si inregistrate de S1 si S2 vor fi acceptate de S3 dupa refacerea retelei

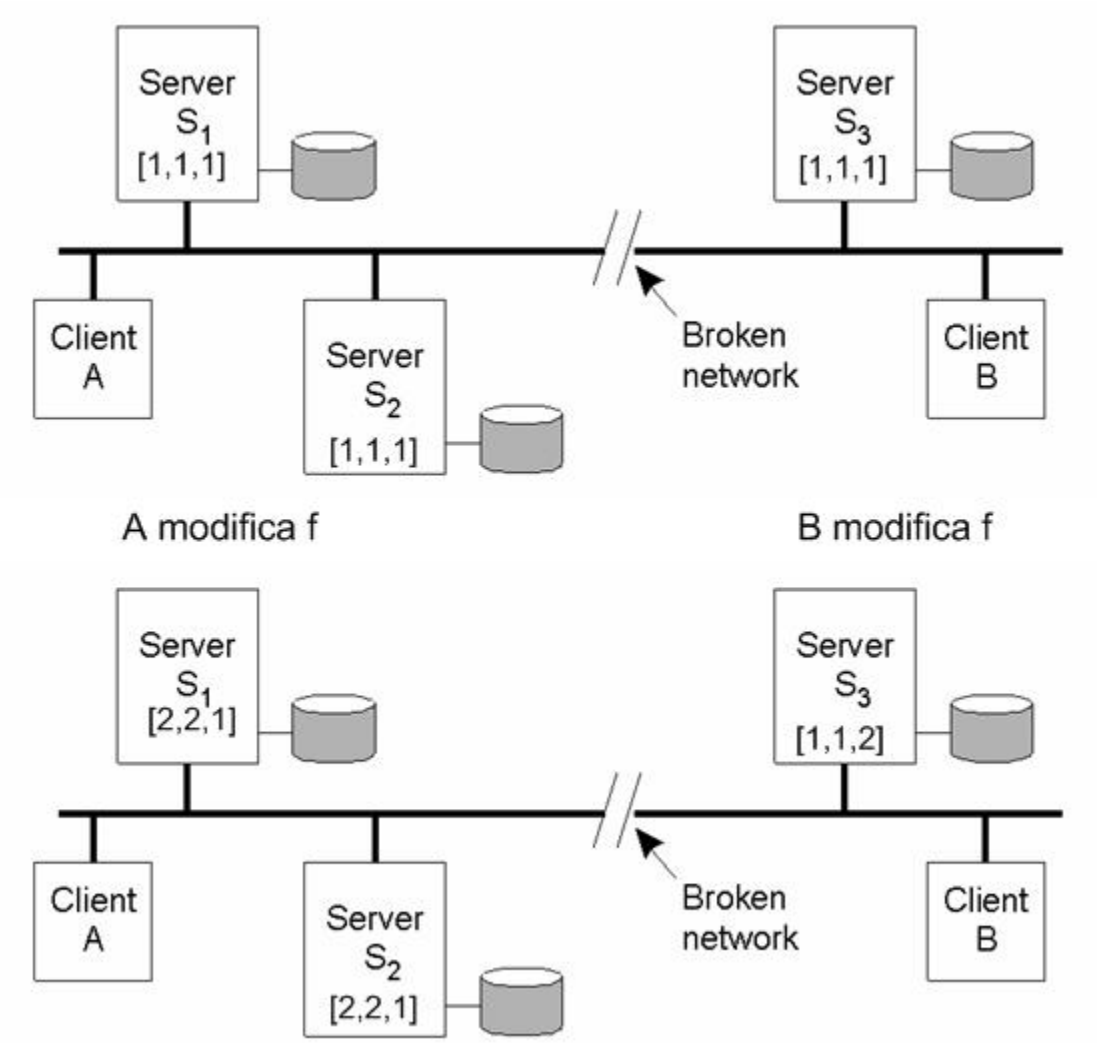


Conflict

- f este modificat de ambii clienti, A si B
- cele doua CVV, [2,2,1] si [1,1,2], sunt in conflict

Rezolvare

- solutie dependenta de aplicatie, posibil automata
- solutie manuala

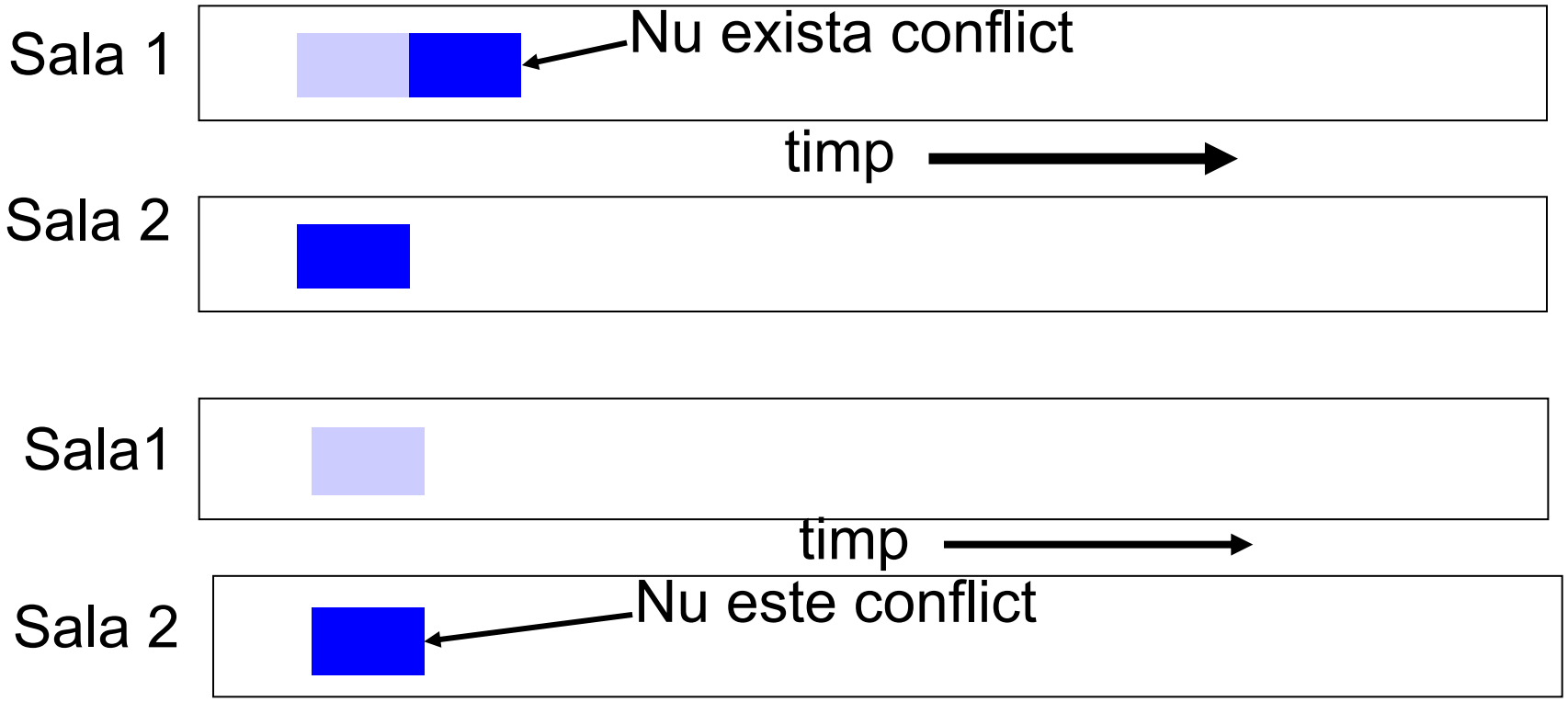


Replicare si Consistenta in Bayou

- Modelele de consistenta centrate pe client isi au originea in Bayou
  - Bayou = Baza de date distribuita peste o retea cu conectivitate nesigura
- Scopul Bayou: gestiunea bazelor de date total replicate in orice numar de site-uri
- Modelul de consistenta este relaxat
  - accesul si actualizarea datelor se poate face oriunde (access-update-anywhere)
- Replicarea este ne-transparenta
  - Aplicatiile sunt implicate in detectia si rezolvarea conflictelor
    - aplicatiile stiu mai bine cum sa trateze inconsistentele
    - Dezavantaj – implementare mai complicata a aplicatiei

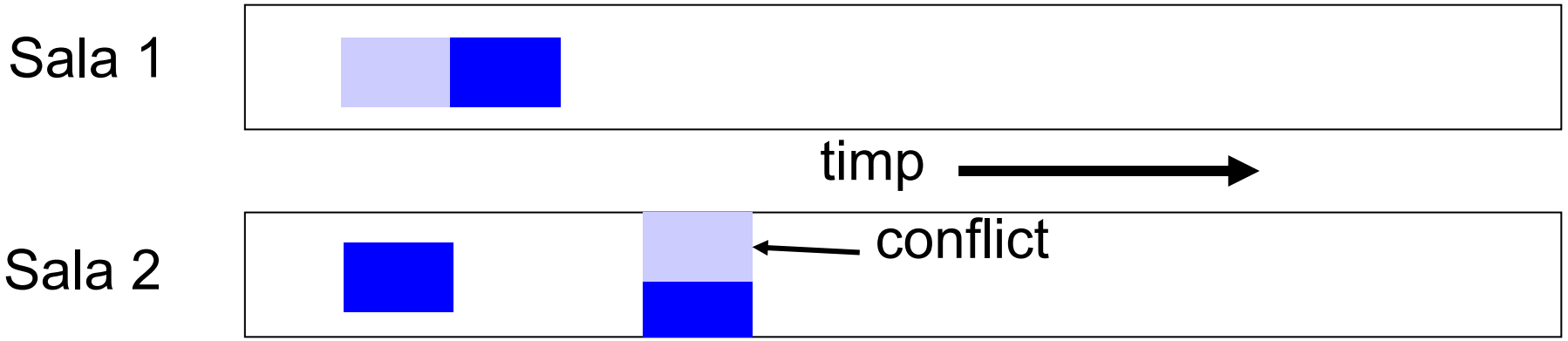
Aplicatie: Planificarea unei sali de conferinte

- Presupunem ca sunt disponibile doua sali de conferinte de aceeasi capacitate.
- Daca doua persoane rezerva sala la ore diferite sau rezerva sali diferite la aceeasi ora nu exista conflict.



Planificarea unei sali de conferinte

- Daca doua persoane rezerva aceeasi sala la o aceeasi ora exista un conflict



Implicarea aplicatiilor

- Pentru fiecare rezervare, se poate face lock pe toata baza de date
  - Nu este necesar daca nu exista conflict
  - In caz de conflict, exista mai multe alternative
    - Se muta o rezervare in cealalta sala
    - Daca cealalta sala este rezervata, se muta rezervarea la o alta ora acceptabila

Concluzii

- Aplicatiile stiu mai bine cum sa rezolve conflictele
- Trebuie oferita interfata care suporta cooperarea intre aplicatii si managerii datelor

Detectia Conflictelor: Controlul dependentelor

Detectia conflictelor dependente de aplicatii se face in Bayou la fiecare write, prin controlul dependentelor (dependency_check)

- controlul consta dintr-un query furnizat de aplicatie si rezultatul asteptat
- controlul se face de serverul invocat, pe replica curenta de date
- se detecteaza un conflict daca rezultatul produs nu este cel asteptat

In caz de conflict, operatia write nu se executa,

iar serverul apeleaza o procedura de rezolvare a conflictului (mergeproc)

Exemplu (descris in continuare)

- operatia write incearca sa rezerve o sala de intalniri pentru un interval de o ora

```
Bayou_Write(
update = {insert, Meetings, 12/18/95, 10:00am, 60min, “Project Meeting: Kevin”},
dependency_check = {
   query = “SELECT key FROM Meetings WHERE day = 12/18/95
      AND start < 11:00am AND end > 10:00am”,
   expected_result = EMPTY},
mergeproc = {
   alternates = {{12/18/95, 12:00pm},{12/19/95, 11:30am}};
   newupdate = {};
   FOREACH a IN alternates {
     # check if there would be a conflict
     IF (NOT EMPTY ( SELECT key FROM Meetings WHERE day = a.date
                  AND start < a.time + 60min AND end > a.time))
          CONTINUE;
     # no conflict, can schedule meeting at that time
     newupdate = {insert, Meetings, a.date, a.time, 60min, “Project Meeting: Kevin”};
     BREAK;
}
IF (newupdate = {}) # no alternate is acceptable
   newupdate = {insert, ErrorLog, 12/18/95, 10:00am, 60min, “Project Meeting: Kevin”};
RETURN newupdate;}
)
```

Comentarii

Procesul este executat automat de server ca parte a operatiei write

Forma: <update><dependency check><mergeproc>

Operatia write: insert este o actualizare a rezervarilor salilor de conferinte

update = {insert, Meetings, 12/18/95, 10:00am, 60min, “Project Meeting: Kevin”},

Preconditia de executie: daca e satisfacuta se aplica update

dependency_check = {
query = “SELECT key FROM Meetings WHERE day = 12/18/95
AND start < 11:00am AND end > 10:00am”,
expected_result = EMPTY},

Comentarii (2)

Procedura de rezolvare: aplicata daca preconditia nu este satisfacuta

- cod interpretat (stil SQL) care genereaza un update
- verifica mai multe rezolvari alternative

```
mergeproc = {
    alternates = {{12/18/95, 12:00pm},{12/19/95, 11:30am}};
    newupdate = {};
    FOREACH a IN alternates {
        # check if there would be a conflict
        IF (NOT EMPTY ( ...
```

Rezolvare automata Imposibila: conflictul este semnalat, urmand a fi rezolvat de utilizator.

```
IF (newupdate = {})      # no alternate is acceptable
    newupdate = {insert, ErrorLog, 12/18/95, 10:00am, 60min, “Project
        Meeting: Kevin”};
```

Executia op. write si inconsistenta replicilor

O operatie write (actualizare) poate fi adresata de un client oricarui server care contine o replica a datelor corespunzatoare

- ea este executata local
- si este propagata altor replici, care o executa imediat ce o primesc
- intarzierile de propagare difera de la o replica la alta → doua replici pot avea valori diferite datorita unor op. write diferite sau executate in ordini diferite

Bayou foloseste un protocol anti-entropie de reconciliere intre replici care asigura ca, in cele din urma

- serverele adopta aceeasi ordine a operatiilor write
- ca urmare, toate replicile vor avea acelasi continut de date
- Bayou **nu poate garanta** o limita a intarzierii de reconciliere a operatiilor write

Sistemul Bayou – log-uri si baza de date

Sistemul de stocare (server) pastreaza pentru fiecare replica

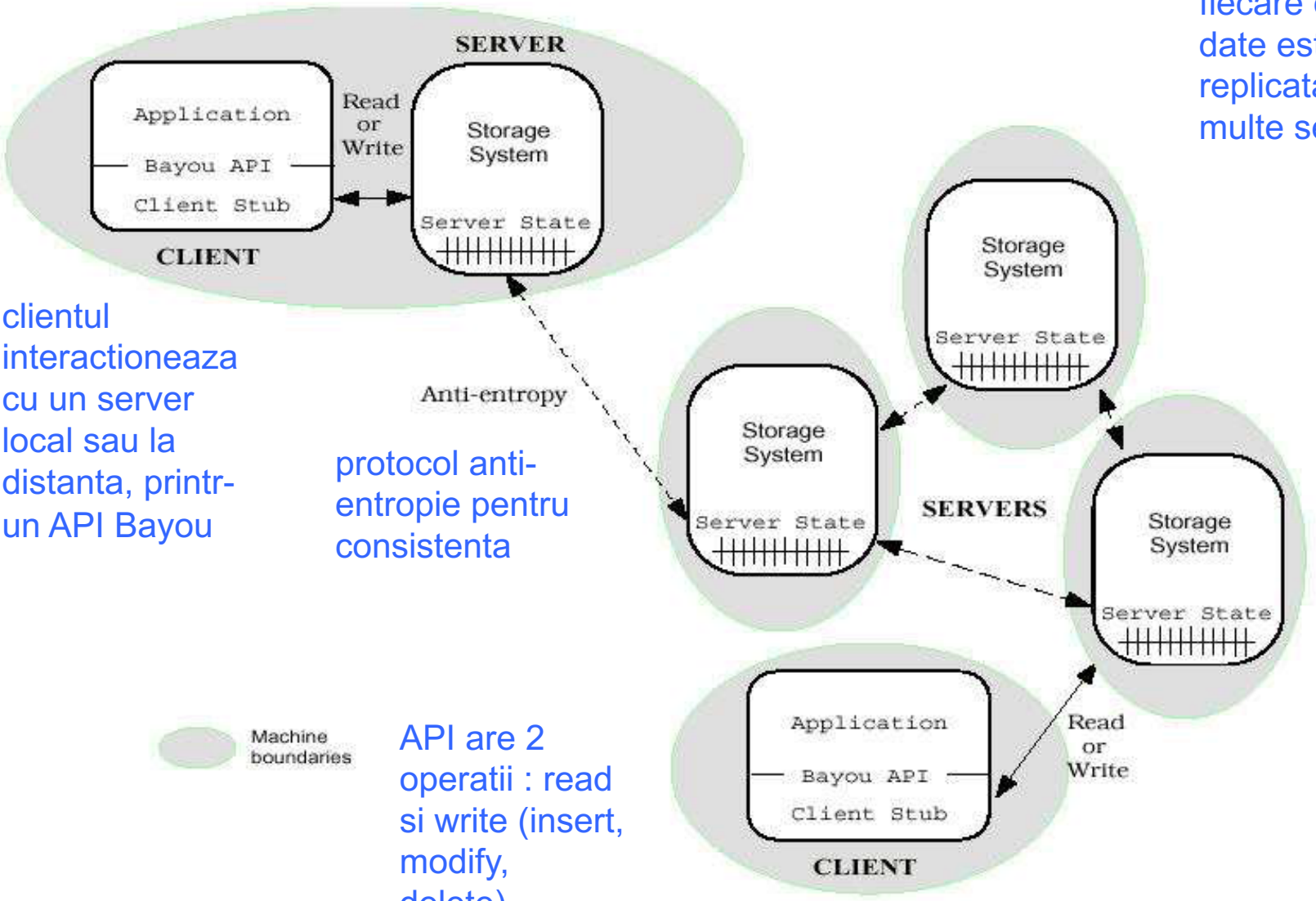
- un log ordonat de operatii write (scrieri)
  - care include toate operatiile write primite de la aplicatie (clienti) sau de la alte servere
- o baza de date al carei continut rezulta din aplicarea ordonata a scrierilor

Asigurarea consistentei “in cele din urma” (eventual consistency) se bazeaza pe

- ordonarea **globala** a tuturor operatiilor write
- folosirea unor proceduri deterministe pentru detectia si rezolvarea conflictelor la write

Modelul sistemului Bayou

fiecare colectie de date este total replicata pe mai multe servere



clientul interactioneaza cu un server local sau la distanta, printr-un API Bayou

protocol anti-entropie pentru consistenta

API are 2 operatii : read si write (insert, modify, delete)

Ordonarea partiala a operatiilor write

- Cand un server primeste un write de la un client, el ii asigneaza un numar de ordine / amprenta de timp – accept-stamp
  - poate fi un contor a carui valoare creste de la un write la altul
- Operatiile write acceptate de la client se transmit altor servere
  - impreuna cu accept-stamp si cu identificatorul serverului care i-a asignat numarul de ordine - (accept-stamp, server_ID)
- Accept-stamps ordoneaza partial setul operatiilor write primite de un server, de la client sau de la alte servere
  - se poate spune ca write A precede write B cand ambele au fost acceptate de acelasi server si write A a fost acceptat inaintea lui write B

Ordonarea totala – operatii stabile si tentative

- Ordinea partiala nu este suficienta pentru asigurarea consistentei
- Pentru a obtine consistenta “eventual” se foloseste un proces anti-entropie care stabileste ordinea corecta a operatiilor write, pe care trebuie sa o respecte toate replicile
  - procesul stabileste operatiile stabile (committed) - a caror pozitie in log nu se mai modifica in procesul anti-entropie
  - si operatiile a caror ordine se poate modifica, numite tentative
  - logul operatiilor write pastrat pentru fiecare replica contine operatiile stabile in ordinea numerelor de secventa comise, urmat de operatiile tentative in ordinea amprentelor accept-stamp
- Operatiile tentative trebuie sa poata fi anulate si re-executate
  - deoarece serverul executa imediat operatiile write primite, ordonarea totala poate schimba ordinea operatiilor tentative si cere anularea efectelor unora din operatii si re-executia lor in ordinea corecta

Protocolul anti-entropie de baza

Protocolul permite ca doua severe sa puna de acord (compatibilizeze) continutul log-urilor lor folosind numerele de ordine accept-stamps

- se bazeaza pe schimbul de operatii write intre replici

Propagarea operatiilor write este constransa de proprietatea prefix

- un server R care detine o scriere Wi initial acceptata de serverul X, detine toate scrierile acceptate de X inainte de Wi
- simplifica evidenta operatiilor de scriere cunoscute de R, folosind un vector de versiune R.V avand o intrare pentru fiecare server
- astfel, intrarea din R.V pentru serverul X, notata R.V(X) contine doar accept-stamp maxim al operatiilor write acceptate de X despre care R are cunostinta

Algoritmul anti-entropie

Algoritmul actualizeaza serverul R cu operatiile write detinute de S si necunoscute de R, folosind vectorul R.V de versiuni al lui R

```
anti-entropy(S,R) {
   Preia R.V de la serverul R
    #trimite toate op write necunoscute de R
   w = primul write in S.write-log
   while (w) do
     if R.V(w.server-id) < w.accept-stamp
        then
           # w este nou pentru R
            SendWrite(R,w)
        w = urmatorul write in S.write-log
}
```

Procesul de actualizare a serverului S de catre R este identic

Stabilizarea operatiilor - comiterea

Obtinerea operatiilor write stabile in loguri permite executia lor in aceeasi ordine in toate replicile (ops. stabile nu trebuie re-executate!)

Bayou foloseste un primary-commit protocol pentru a determina cand un write devine stabil

- o replica este desemnata primara
- ea “stabilizeaza” (commit) operatiile write
  - asociaza operatiei un numar de secventa comis CSN (Commit Sequence Number)
- operatiile write stabile sunt total ordonate

Log-ul ordonat al operatiilor

Cu CSN, informatia asociata unei operatii write devine

(CSN, accept-stamp, server_ID)

Prin conventie, numarul CSN asociat unei operatii tentative este infinit ∞

Log-ul ordonat al operatiilor (dupa comitere) are doua parti

- lista scrierilor stabile total ordonata crescator dupa CSN
- lista scrierilor tentative partial ordonata dupa accept-stamps

CSN este propagat celorlalte servere folosind o extensie a algoritmului anti-entropie

La propagare se trimit in ordine operatiile write stabile, apoi operatiile write tentative

Extensia algoritm anti-entropie

```
anti-entropy(S,R) {
   Preia R.V de la serverul R
    #Trimite toate op write comise pe care R nu le stie
    if R.CSN < S.CSN then
       w = primul write comis despre care R nu stie
       while (w) do
           if w.accept-stamp < R.V(w.server-id) then
                # R are op write, dar nu stie ca ea este comisa
                SendCommitNotification(R, w.accept-stamp,w.server-id, w.CSN)
           else SendWrite(R,w)
           end
           w = urmatorul write comis din S.write-log
       end
    end
```

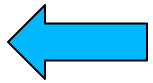
Algoritmul este executat de serverul S pentru a actualiza serverul R

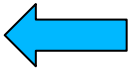
S.CSN este cel mai mare numar de secventa comis cunoscut de S

El reprezinta concis toata partea comisa a log-ului deoarece

- operatiile comise sunt ordonate dupa numerele de secventa comise
- propagarea lor se face in aceasta ordine

```
anti-entropy(S,R) {
   Preia R.V de la serverul R
    #Trimite toate op write comise pe care R nu le stie
    if R.CSN < S.CSN then
       w = primul write comis despre care R nu stie
       while (w) do
           if w.accept-stamp < R.V(w.server-id) then
               # R are op write, dar nu stie ca ea este comisa
               SendCommitNotification(R, w.accept-stamp,w.server-id, w.CSN)
           else SendWrite(R,w)
           end
           w = urmatorul write comis din S.write-log
       end
    end
```





In prima parte a algoritmului, S trimite lui R operatiiile write comise pe care R nu le stie, in doua feluri

- notificari de comitere pentru operatiile w pe care R le are dar nu stie ca sunt comse (w.accept-stamp < R.V(w.server-id)
- operatiile in intregime, pentru writes pe care R nu le are

```
anti-entropy(S,R) {
   Preia R.V de la serverul R
    #Trimite toate op write comise pe care R nu le stie
    if R.CSN < S.CSN then
       w = primul write comis despre care R nu stie
       while (w) do
            if w.accept-stamp < R.V(w.server-id) then
                 # R are op write, dar nu stie ca ea este comisa
                 SendCommitNotification(R, w.accept-stamp,w.server-id, w.CSN)
            else SendWrite(R,w)
            end
            w = urmatorul write comis din S.write-log
       end
    end
    w = primul write tentativ
    #acum trimite writes tentative
    while (w) do
          if R.V(w.server-id) < w.accept-stamp then
                SendWrite(R,w)
          w = urmatorul write din S.write-log
    end
}
```

In partea a doua a algoritmului, se trimit operatiile tentative pe care R nu le are

Proprietatile solutiei

Anti-entropie incrementala

- reconcilierea intre doua replici poate fi intrerupta (defecte de retea)
- reluarea nu reclama re-transmiterea operatiilor deja actualizate

Replicile pot trunchia orice prefix din partea stabila (operatii comise)

- Implicatie: o replica nu mai are toate operatiile pentru reconciliere incrementala cu o alta replica
- Rezolvare: se mentin informatii de identificare a partii trunchiate
- daca este cazul, se transfera intreaga baza de date pentru actualizarea altei replici

Exemplu cu trei servere

Trei servere cu vectorii de versiune initiali

P	A	B
P.V = [0, 0, 0]	A.V = [0, 0, 0]	B.V = [0, 0, 0]

Un vector de versiuni contine accept-stamp-urile maxime ale replicilor

Elementele vectorilor corespund, in ordine, replicilor P, A, B

Serverele scriu independent

P adauga log-ului operatiile acceptate
Fiecare intrare contine <CSN, accept-stamp, server_ID>

inf = ∞

P	A	B
<inf, 1, P>	<inf, 2, A>	<inf, 1, B>
<inf, 4, P>	<inf, 3, A>	<inf, 5, B>
<inf, 8, P>	<inf, 10, A>	<inf, 9, B>
[8, 0, 0]	[0, 10, 0]	[0, 0, 9]

vectorul de versiuni al replicii P
dupa acceptarea unor operatii write

P si A fac un schimb anti-entropie

P	A	B
<inf, 1, P>	<inf, 1, P>	<inf, 1, B>
<inf, 2, A>	<inf, 2, A>	<inf, 5, B>
<inf, 3, A>	<inf, 3, A>	<inf, 9, B>
<inf, 4, P>	<inf, 4, P>	
<inf, 8, P>	<inf, 8, P>	[0, 0, 9]
<inf, 10, A>	<inf, 10, A>	
[8, 10, 0]	[8, 10, 0]	dupa schimb

<inf, 1, P>	<inf, 2, A>	
<inf, 4, P>	<inf, 3, A>	
<inf, 8, P>	<inf, 10, A>	
[8, 0, 0]	[0, 10, 0]	inainte de schimb

P comite unele scrieri

P comite 3 operatii

P	A	B
<1, 1, P>	<inf, 1, P>	<inf, 1, B>
<2, 2, A>	<inf, 2, A>	<inf, 5, B>
<3, 3, A>	<inf, 3, A>	<inf, 9, B>
<inf, 4, P>	<inf, 4, P>	
<inf, 8, P>	<inf, 8, P>	[0, 0, 9]
<inf, 10, A>	<inf, 10, A>	
[8, 10, 0]	[8, 10, 0]	

< inf, 1, P>
< inf, 2, A>
< inf, 3, A>
<inf, 4, P>
<inf, 8, P>
<inf, 10, A>

[8, 10, 0]

P si B fac un schimb anti-entropie

P	A	B
<1, 1, P>	<inf, 1, P>	<1, 1, P>
<2, 2, A>	<inf, 2, A>	<2, 2, A>
<3, 3, A>	<inf, 3, A>	<3, 3, A>
<inf, 1, B>	<inf, 4, P>	<inf, 1, B>
<inf, 4, P>	<inf, 8, P>	<inf, 4, P>
<inf, 5, B>	<inf, 10, A>	<inf, 5, B>
<inf, 8, P>		<inf, 8, P>
<inf, 9, B>	[8, 10, 0]	<inf, 9, B>
<inf, 10, A>		<inf, 10, A>
[8, 10, 9]		[8, 10, 9]
↑		↑
<1, 1, P>		<inf, 1, B>
<2, 2, A>		<inf, 5, B>
<3, 3, A>		<inf, 9, B>
<inf, 4, P>		
<inf, 8, P>		[0, 0, 9]
<inf, 10, A>		
[8, 10, 0]		

P comite mai multe operatii write

P

<1, 1, P>
<2, 2, A>
<3, 3, A>
<inf, 1, B>
<inf, 4, P>
<inf, 5, B>
<inf, 8, P>
<inf, 9, B>
<inf, 10, A>

[8, 10, 9]

P

<1, 1, P>
<2, 2, A>
<3, 3, A>
<4, 1, B>
<5, 4, P>
<6, 5, B>
<7, 8, P>
<inf, 9, B>
<inf, 10, A>

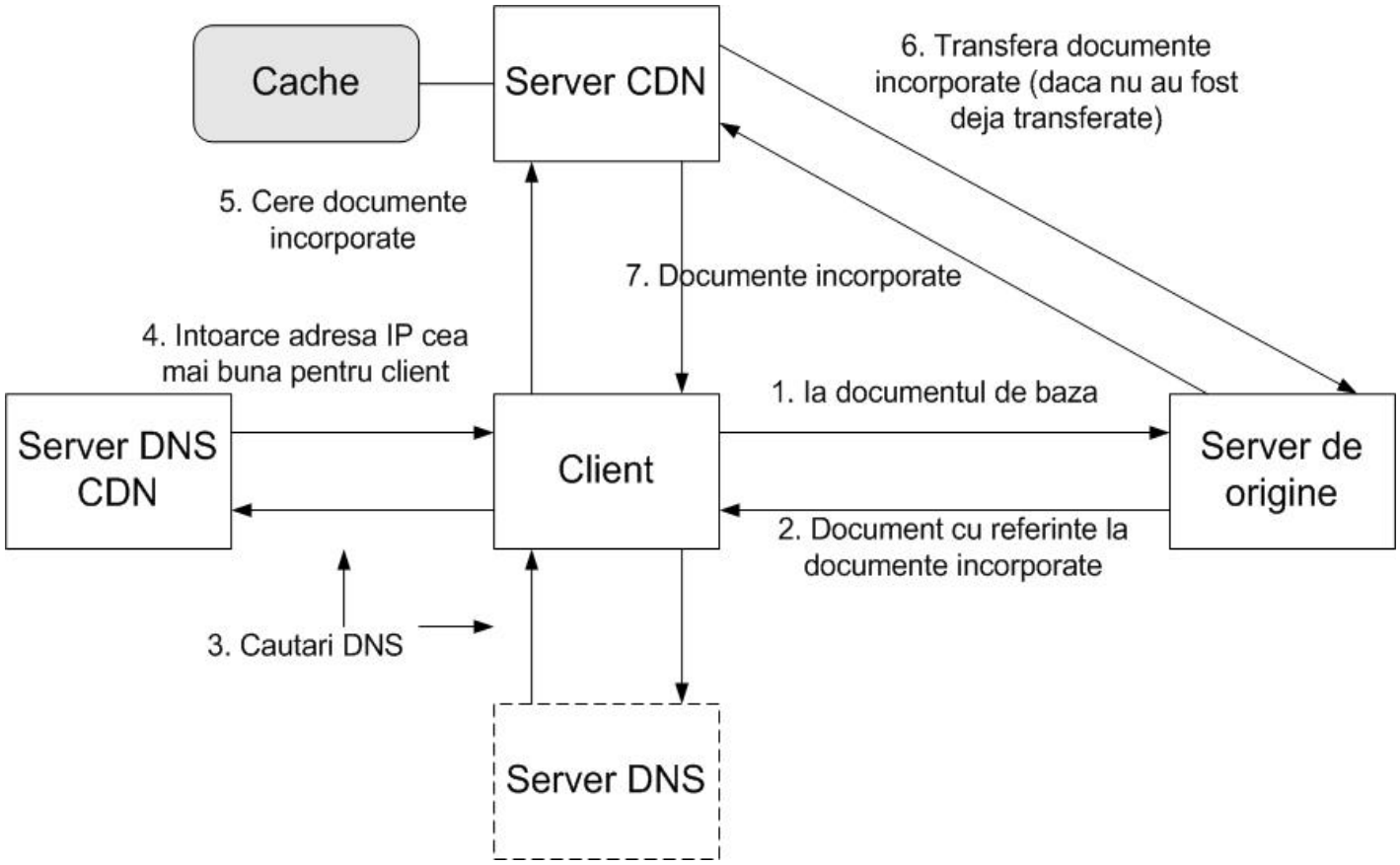
[8, 10, 9]

Replicare – CDN – Content Delivery Network

- Un document Web poate consta dintr-o pagina care include referinte la alte documente (imagini, video etc.)
- Pentru afisarea intregului document, browser-ul trebuie sa descarce documentele incluse
- Daca aceste documente se schimba rar in timp, are sens replicarea lor
- O solutie: servere Akamai CDN
  - foloseste peste 12 000 servere CDN
- Functionare
  - fiecare document inclus este referit, de regula, printr-un URL
  - in Akamai se foloseste un URL modificat → virtual ghost care
    - este o referinta la un server in CDN
    - contine, in plus, numele DNS al serverului de origine

Replicare – CDN – Content Delivery Network

pas 4 - DNS CDN returneaza adresa unui server CDN care:
- este apropiat geografic de client
- are o «incarcare» redusa (nu este aglomerat)

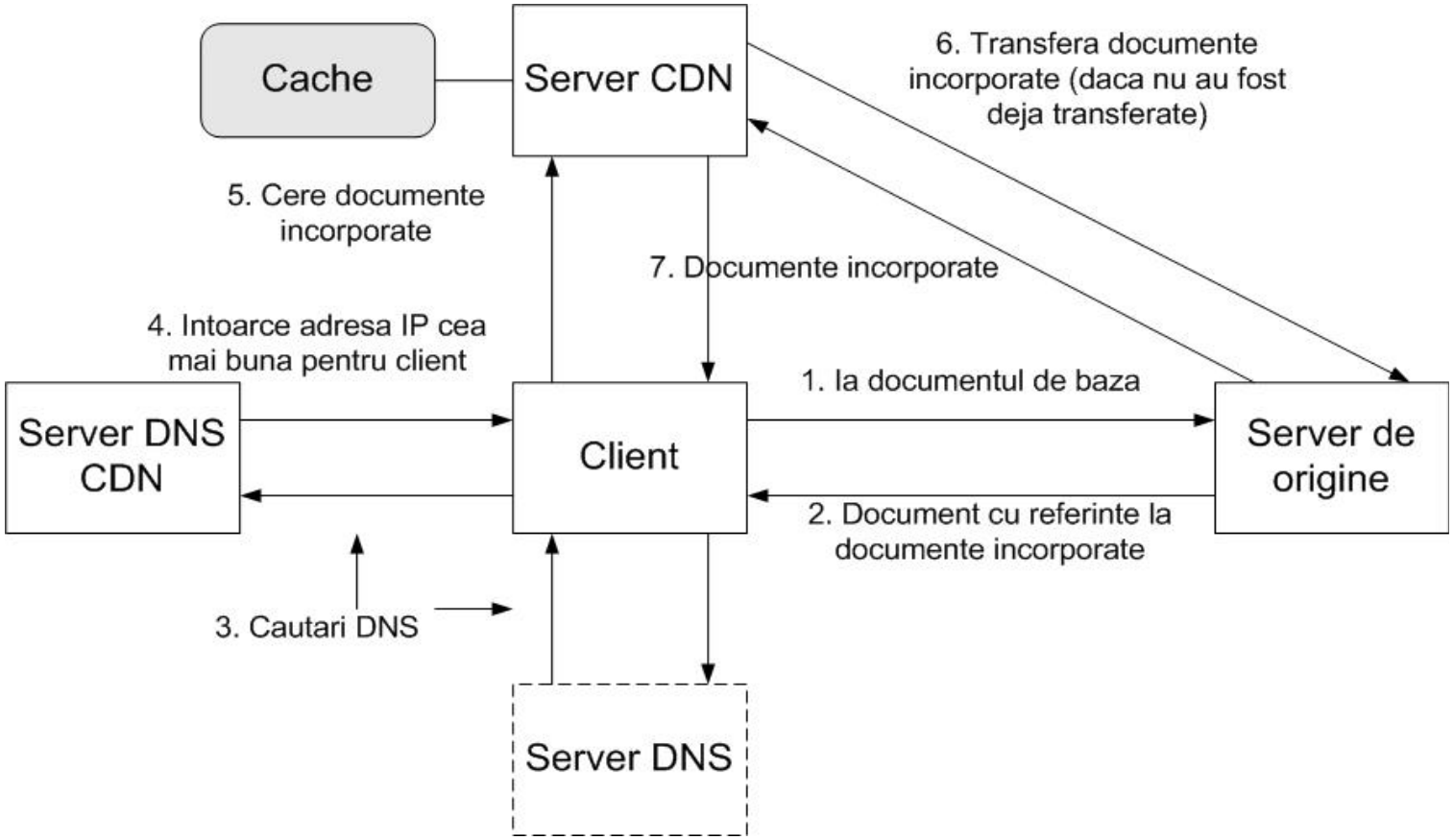


pas 3 – numele DNS este rezolvat de un server DNS la un server DNS CDN

pas 1 – cere o pagina
pas 2 - Referinta virtual ghost include un nume DNS (ex. ghosting.com)

Solutia Akamai

- pas 5 - clientul cere documentul
- pas 6 - daca serverul CDN nu are inca resursele, le descarca folosind adresa serverului de origine
- pas 7 - returneaza documentul cerut



Solutia Akamai

Rezolvarea legaturii la CDN apropiat

- se cauta dupa adresa IP a clientului intr-o baza de date care mentine o harta a Internet-ului
- alternativa: se da aceeasi adresa IP mai multor servere CDN, urmand ca cel mai apropiat sa fie gasit prin politica de rutare "calea cea mai scurta »

Consistenta

- identificatorul documentului incorporat se schimba odata cu continutul
- noul document nu este gasit in serverul CDN si este descarcat de la serverul de origine

Bibliografie

- A.S. Tanenbaum, M. van Steen (2007). Distributed Systems. Principles and paradigms, Ed.2, Prentice Hall
- L. Shklar, R. Rosen (2003) Web Application Architecture. John Wiley & Sons
- Fielding, R. (2000). Architectural styles and the design of network-based software architectures. Ph.D. thesis, University of California, 2000, Retrieved April 12, 2009 from http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm.
- Karin Petersen, Mike J. Spreitzer, Douglas B. Terry, Marvin M. Theimer and Alan J. Demers (1997) Flexible Update Propagation for Weakly Consistent Replication, Computer Science Laboratory, Xerox Palo Alto Research Center, Palo Alto, California 94304 U.S.A.
- Ion Stoica, CS 268 Classic Distributed Systems: Bayou and BFT, Retrieved October 2015 from http://www.cs.berkeley.edu/~istoica/classes/cs268/06/notes/20-BFTx2.pdf