Toleranţa la defectări

Concepte generale
Comunicarea fiabila client-server
Multicast fiabil si atomic
Tranzactii fiabile
Recuperarea

Bazat pe "Sisteme distribuite" de A.S. Tanenbaum

Terminologie

- O ***specificatie*** descrie comportarea ideala a unui *sistem* la *interfetele* sale.
- **Defect, Greseala** (Fault): este **cauza** unei erori
- **Eroare** (Error): stare interna care poate conduce la esec (nu neaparat!); nu este vizibila la interfata
- **Esec, incapacitate** (Failure): **deviere** de la specificatia interfetei
- Dependenta cauzala

  Defect ➔ Eroare ➔ Esec
- **Exemplu**:

  praful determina reducerea turatiei unui ventilator care va produce mai putin aer si va determina supraincalzirea sursei de alimentare; o componenta arde si sursa se opreste **cum se aplica dependenta cauzala pe acest exemplu ???**
- sistem ventilatie:

  praf = cauza; reducere turatie = eroare; mai putin aer = esec
- sursa de alimentare

  esec ventilatie = cauza; supraincalzire = eroare; oprire functionare = esec

Sisteme de incredere

- **Increderea** (Dependability)
  - Un sistem de incredere (dependable system) respecta specificatia chiar in prezenta defectarilor

- **Atribute** ale sistemelor de incredere ce inseamna ???
  - Disponibilitatea (Availability)
    - gata de a fi folosit imediat (la un moment dat)
  - Fiabilitatea (Reliability)
    - cat timp functioneaza continuu fara esec (failure)
  - Siguranta (Safety)
    - temporar poate functiona incorect fara efecte catastrofice
  - Mentenabilitatea (Maintainability)
    - cat de usor poate fi reparat

Tipuri de defecte si esecuri

Tipuri de defecte

tranzitorii – se produc o data si dispar

intermitente – se produc, dispar si reapar

permanente

Tipuri de esecuri	Descriere comportare server
Cadere (Crash)	Un server se opreste dintr-o data
Omisiune *receptie* *transmisie*	Nu raspunde cererii Nu a primit cererea Nu trimite raspunsul (desi serverul l-a pregatit)
Timing	Raspunde dupa trecerea timpului specificat
Raspuns *valoare* *tranzitie*	Raspunde incorect Valoarea din raspuns este gresita Deviaza de la fluxul de control corect
Arbitrar (Bizantin)	Produce raspunsuri arbitrare la momente arbitrare

Tehnici de tratare a defectelor

- **Prevenire**
  - proiectare formala, controlul calitatii
  - injectare defecte si testare.
- **Detectie**
  - se foloseste schimbul regulat de mesaje intre vecini
  - procesele intreaba vecinii despre starea lor (are you alive?)
  - procesele comunica vecinilor starea lor (I'm alive!)
- **Recuperare**
  - checkpoint-restart
  - log-uri de mesaje
- **Prezicere**
  - folosind date de monitorizare
- **Tolerare**
  - furnizarea serviciului chiar daca apar defecte (**mascarea** defectelor)

Mascarea esecurilor

- Replicare => toleranta la defectari
- Protocoale
  - Primary based
    - o replica primara (leader) coordoneaza toate operatiile de scriere
    - foloseste alegere leader pentru a trata un esec al replicii primare
  - Replicated write
    - replicare activa sau
    - cvorumuri

- Sistem **k-fault tolerant**
  - mascheaza k procese defecte

- Problema: cat de mare trebuie sa fie grupul (n)?
- Solutie:
  - n = k+1 esecuri de tip crash
  - n = 3k+1 esecuri bizantine (solutie Lamport)

Esecuri bizantine

Solutia are la baza generalii bizantini

toti locotenentii loiali «primesc » acelasi ordin de la comandant (loial sau neloial)

Pentru **mesaje orale** (nesemnate), mai mult de 2/3 din generali trebuie să fie loiali.

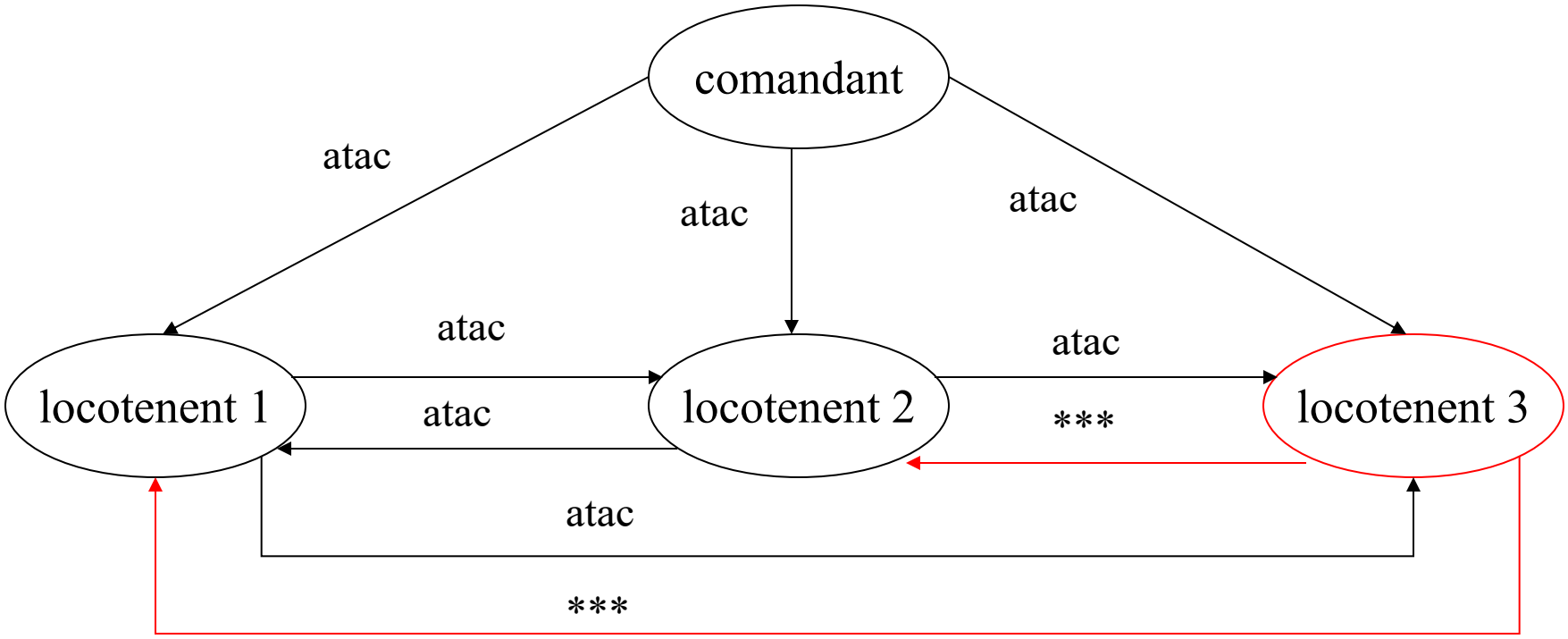
Cu mesaje orale şi un trădător

nu există o soluţie pentru doar trei generali

există soluţie pentru patru generali.

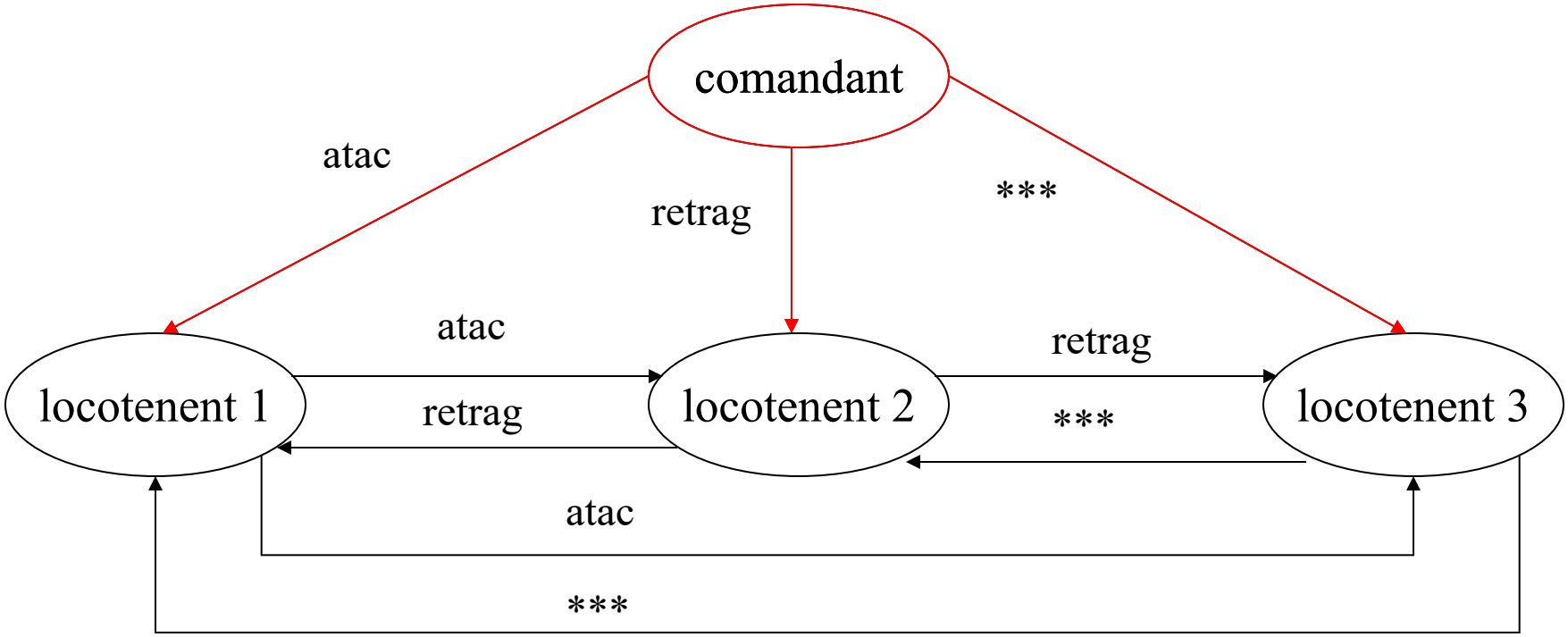
Există soluţie pentru un trădător între 4 generali

Scenariul 1. Comandantul este loial, iar decizia este **atac**.



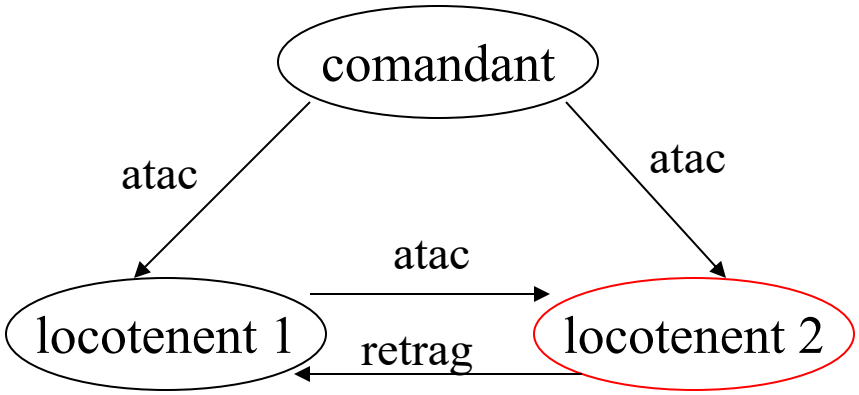
Cei doi locotenenti loiali selecteaza "atac" pe baza principiului majoritatii

Scanariul 2. Comandantul este neloial şi transmite mesaje diferite locotenenţilor.



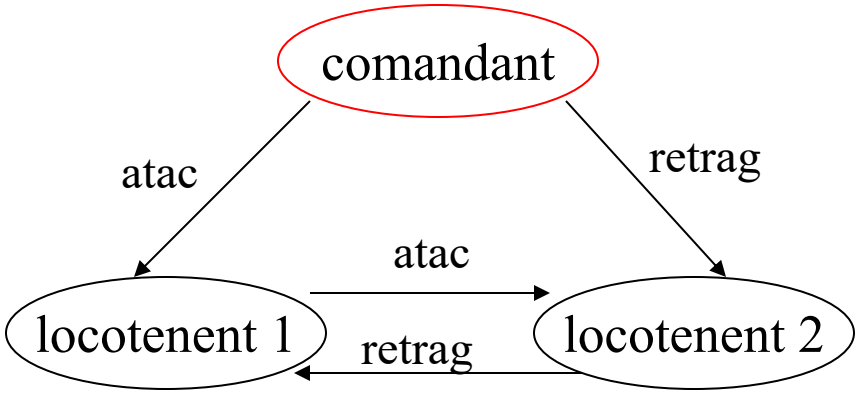
Locotenentii loiali au acelasi set de valori (atac, retrag, ***) si iau aceeasi decizie

Nu există soluţie pentru 1 trădător şi 3 generali



Scenariul 3. Grupul conţine trei generali. Alg. locotenenti loiali iau decizie determinista, de ex. dau prioritate comandantului

a) locotenentul 2 este neloial → generalii loiali iau aceeasi decizie



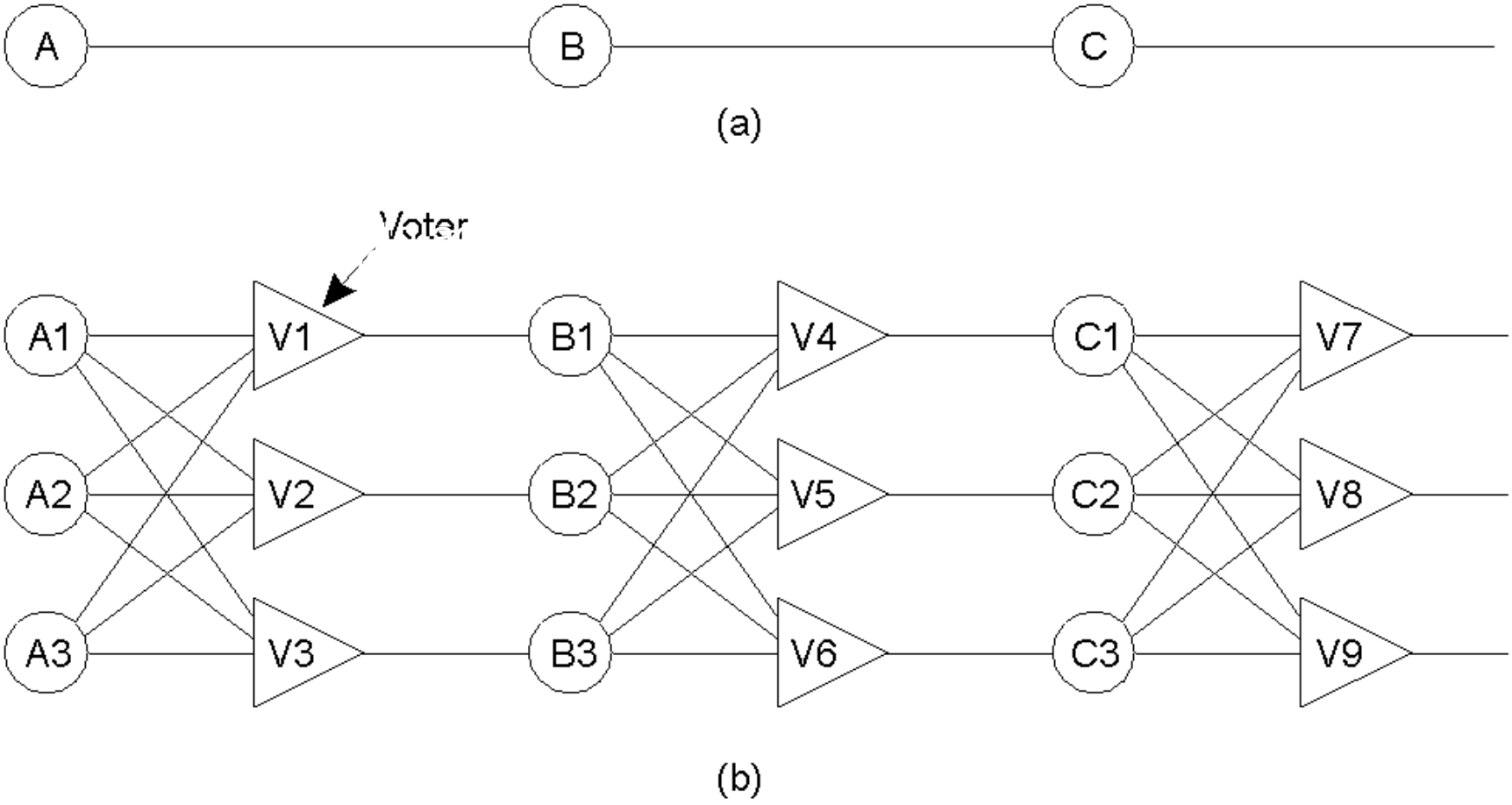
b) comandantul este neloial → generalii loiali iau decizii diferite

Se poate generaliza la n generali (demo constructivă)

Mascarea esecurilor prin redundanta

Redundanta Modulara Tripla.

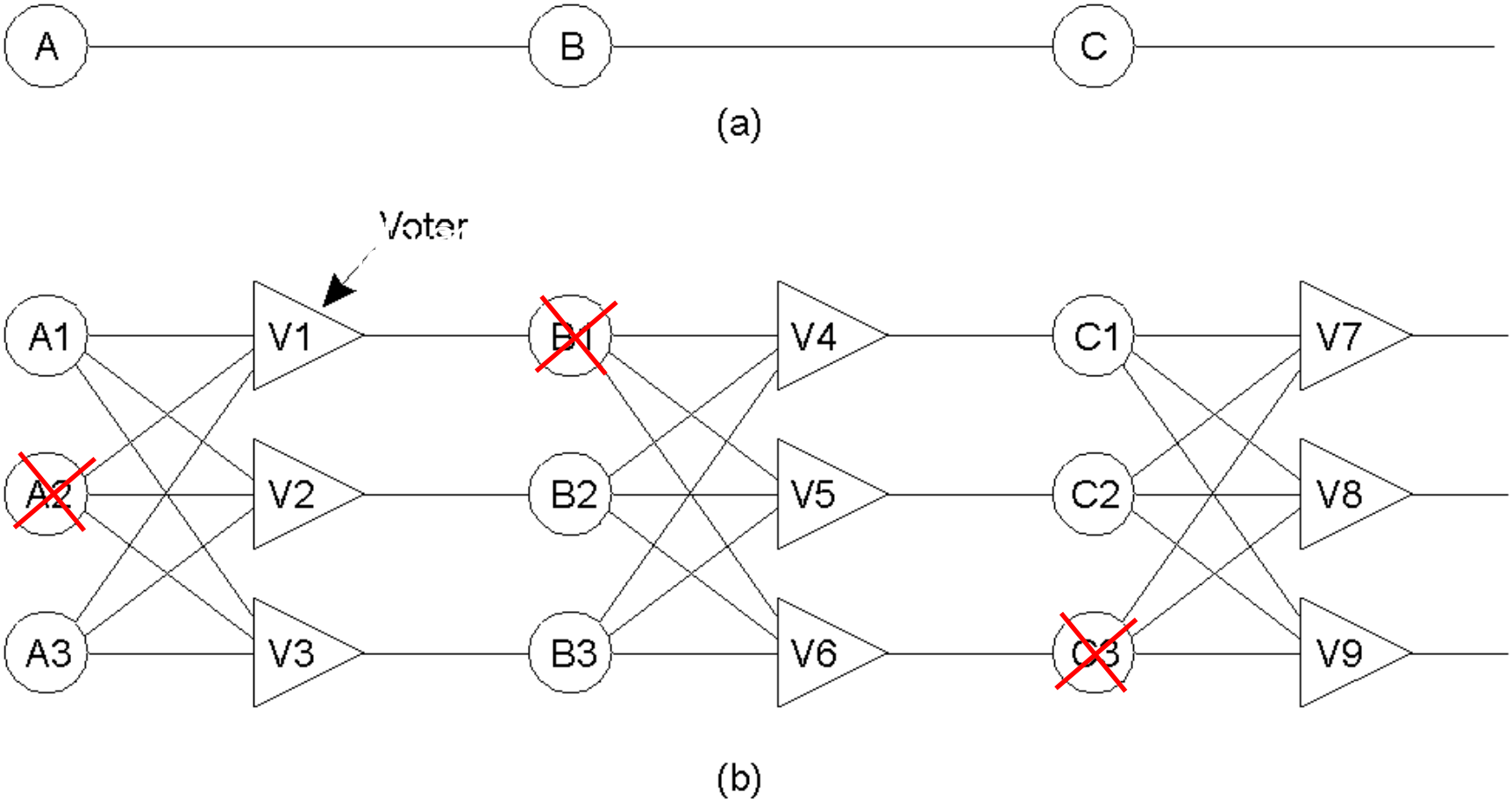
- fiecare echipament este triplat
- dupa fiecare tripleta este inclus un ansamblu de echipamente de votare



Mascarea esecurilor prin redundanta

Redundanta Modulara Tripla.

- suporta defectarea unei componente din trei, la fiecare nivel



Fiabilitatea apelurilor de proceduri la distanta

Solutii in caz de eroare			Semantica invocarii
retransmite *request*	Filtrare duplicate	Re-executa procedura sau Retransmite *reply*	
Nu	Ne-aplicabil	Ne-aplicabil	← Maybe
Da	Nu	Re-executa procedura	← At least once
Da	Da	Retransmite *reply*	← At most once

- Maybe – clientul nu poate spune daca executia s-a facut sau nu
- At least once – rezultate eronate la re-executia procedurii
  - daca operatiile nu sunt idempotente
- At most once – clientul primeste rezultat sau eroare

Semantici RPC in prezenta defectelor

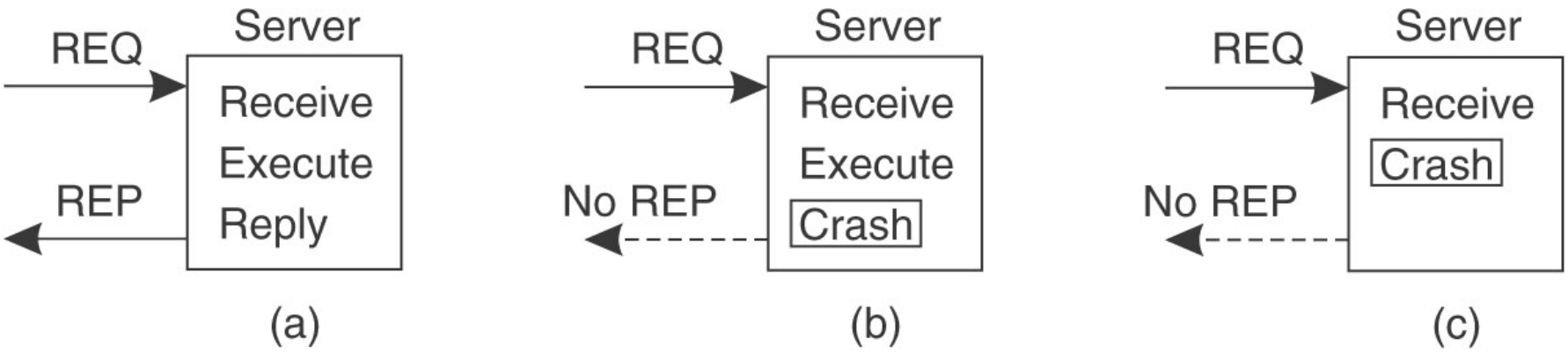
- **Scopul RPC**
  - ascunderea localizarii: apelul la distanta arata ca unul local
  - problema – aparitia defectelor

- Clientul nu poate localiza serverul
  - Server defect sau o versiune noua, necunoscuta de client
  - Solutii
    - Provoaca exceptie (Java) sau foloseste handler de semnalizare (in C)
  - Neajunsuri
    - nu orice limbaj are exceptii sau handler de semnalizare
    - distruge transparenta

Semantici RPC in prezenta defectelor

- Mesaj de cerere pierdut
  - Solutie
    - Timer in OS client
    - Retransmite cererea
    - La mai multe timeout-uri – mesaj de eroare catre client
  - Neajuns
    - Merge daca mesajul a fost cu adevarat pierdut
    - Altfel, serverul trebuie sa faca diferenta intre original si copie

Caderea Serverului



a – normal
b – crash dupa executie
c – crash inainte de executie

cand apare un timeout, clientul nu poate diferentia intre b si c
solutii pentru client ???

- repeta cererea pana reuseste (at least once)
- semnaleaza eroarea (at most once)
- nimic (maybe)
- solutie teoretica? (exactly once)

Exactly once – Strategii server

- Aplicatia - Client-server pentru tiparire text
- Strategii server
  - trimite Mesaj ACK (eveniment M) inainte de a spune Printer sa tipareasca (eveniment P); ordinea este M → P
  - trimite mesaj ACK dupa tiparire text; ordinea este P→ M

- Combinatii posibile (C este eveniment Crash datorita caruia operatiile intre paranteze nu se mai pot executa):

Strategia M → P	Strategia P→ M
M P C	P M C
M C (P)	P C (M)
C (M P)	C (P M)

Exactly once - Strategii client

- *Server anunta ca a cazut si acum este iar operational*
- Strategie client de retransmitere cerere
  - mereu
  - niciodata
  - doar cand nu s-a primit mesaj confirmare cerere de la server
  - doar cand s-a primit mesaj confirmare cerere de tiparire

Combinatii (2)

Client	Server					
	Strategy M → P			Strategy P → M		
Reissue strategy	MPC	MC(P)	C(MP)	PMC	PC(M)	C(PM)
Always	DUP	OK	OK	DUP	DUP	OK
Never	OK	ZERO	ZERO	OK	OK	ZERO
Only when ACKed	DUP	OK	ZERO	DUP	OK	ZERO
Only when not ACKed	OK	ZERO	OK	OK	DUP	OK

OK = Text is printed once
DUP = Text is printed twice
ZERO = Text is not printed at all

pentru nici o perche de strategii client/server nu avem 3 OK-uri
concluzia: nici o combinatie de strategii client si server corecta

Pierdere mesaje reply

Solutie

- timer
- retransmitere cerere

Dezavantaje

- merge la operatii idempotente
- pentru celelalte
  - clientii asociaza cererilor numere de secventa si serverul tine evidenta lor
  - serverul trebuie sa tina evidenta raspunsurilor date clientilor
  - in plus, un bit in mesaj poate distinge cererea originala de copii
    - serverul trateaza imediat originalul (originalul este mai sigur)
    - retransmisia cere mai multa atentie

Cadere Client

- Clientul face o cerere si cade inainte de primirea raspunsului
  → Calcule orfane – produc rezultate pe care nu le asteapta nimeni
- Probleme
  - consum inutil resurse
  - fisiere blocate (lock)
  - sincronizare cu client re-boot-at
- Solutii
  - exterminare
    - fiecare invocare se memoreaza mai intai intr-un log
    - la re-boot-area clientului, orfanii sunt exterminati pe baza informatiilor din log
  - probleme:
    - inregistrarea fiecarui apel este costisitoare
    - descendentii orfanilor (grandorfans) – pot fi greu sau imposibil de localizat
    - retea partitionata – nu se poate face exterminarea

- reincarnare (epoci)
  - timpul este impartit in epoci succesive
  - la rebootare, un client difuzeaza tuturor masinilor un mesaj de epoca noua
  - la receptia lui, toate calculele din epoci anterioare sunt distruse
  - raspunsurile din epoci anterioare care totusi sosesc sunt distruse la client
- reincarnare lina (localizare proprietar)
  - la receptia mesajului de epoca noua, o masina incearca identificarea proprietarului pentru calculele facute pentru client
  - exterminarea se face daca proprietarul nu este gasit
- expirare
  - fiecare RPC are alocata o cuanta de timp T standard
  - daca nu poate termina, cere o alta cuanta
  - la repornire dupa crash, clientul asteapta un timp T (pentru distrugerea orfanilor)

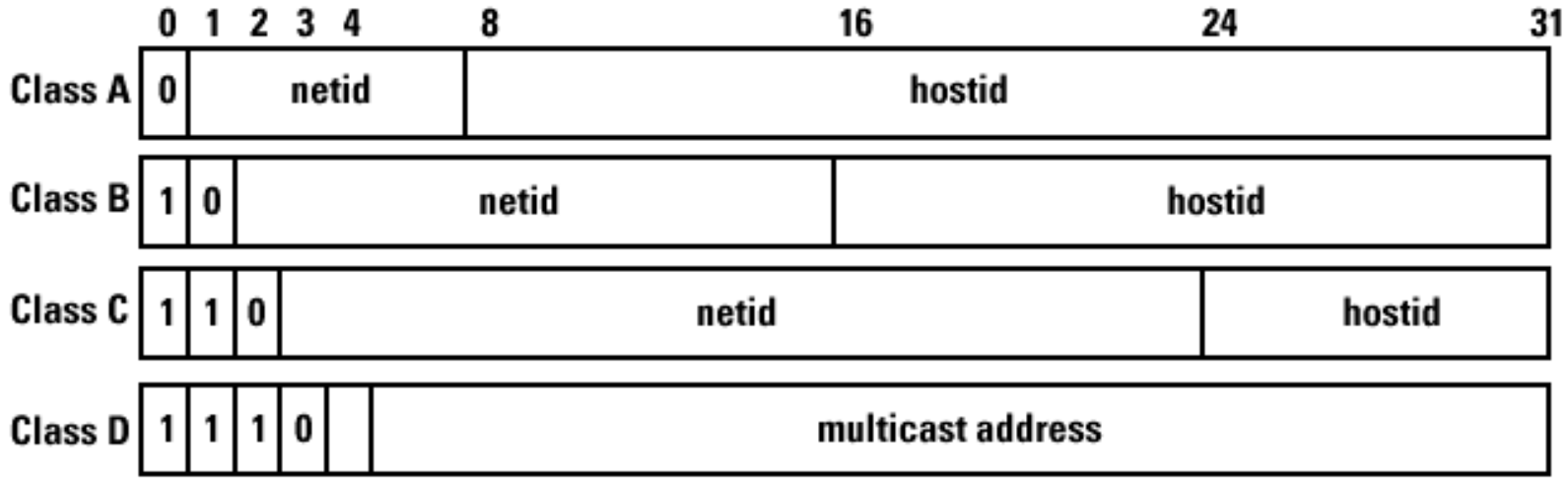
Comunicare de grup fiabila

- Multicast fiabil: un mesaj trimis unui grup de procese trebuie livrat fiecarui membru al grupului
- Simplificare
  - **grupul este stabil** (procesele nu «cad» (nu crash) si nu se alatura grupului in timpul comunicarii)
  - consideram doar **defecte de comunicare**
- Exemple de utilizare
  - Tranzactii fiabile si ordonate - **Uniform Reliable Group Communication Protocol** (URGC)
  - Multicast articole noi pe Mbone – **Muse**
  - Transmitere fisiere - **Multicast File Transfer Protocol** (MFTP)
  - Evidenta pachetelor transmise intr-un server de logging – **Log-Based Receiver-reliable Multicast** (LBRM)

Multicast IP

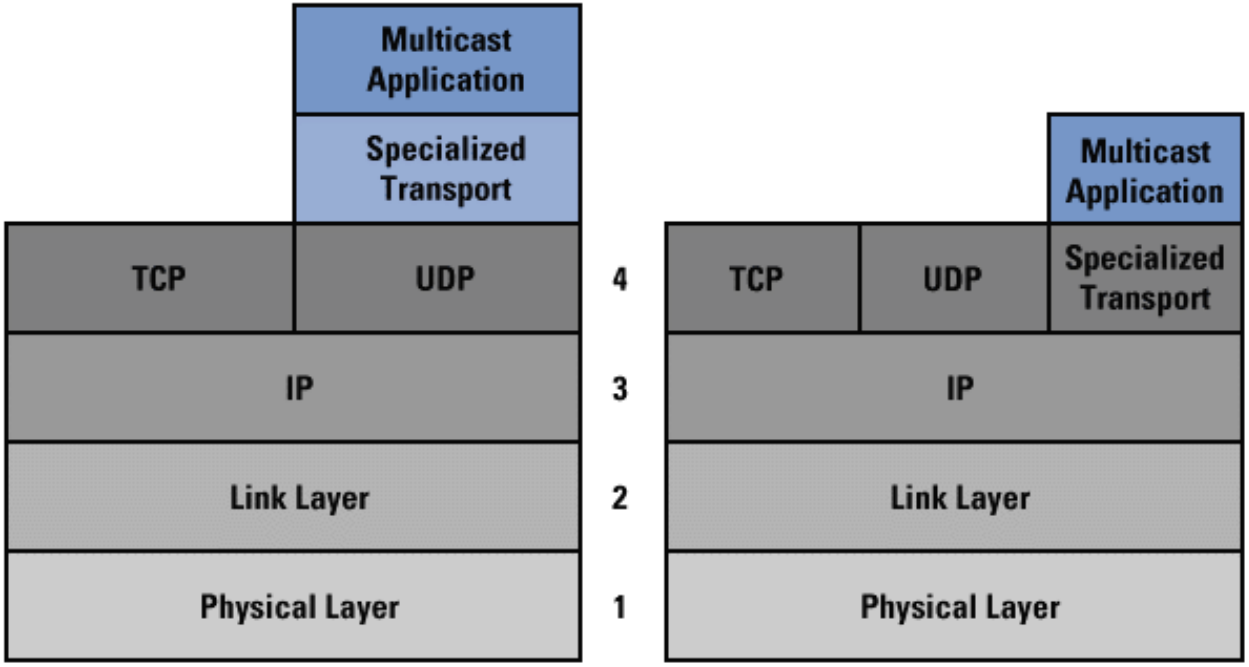
Se bazeaza pe

- Adrese de clasa D pentru transmitere pachete
- Internet Group Management Protocol (IGMP) pentru gestiunea grupurilor
- Rutere multicast
  - **transmit pachete multicast membrilor (best effort)**
  - **un nou host notifica ruterul la intrarea in grup**
  - **ruterul verifica periodic host-urile ramase in grup**



Transport - Multicast peste UDP sau IP

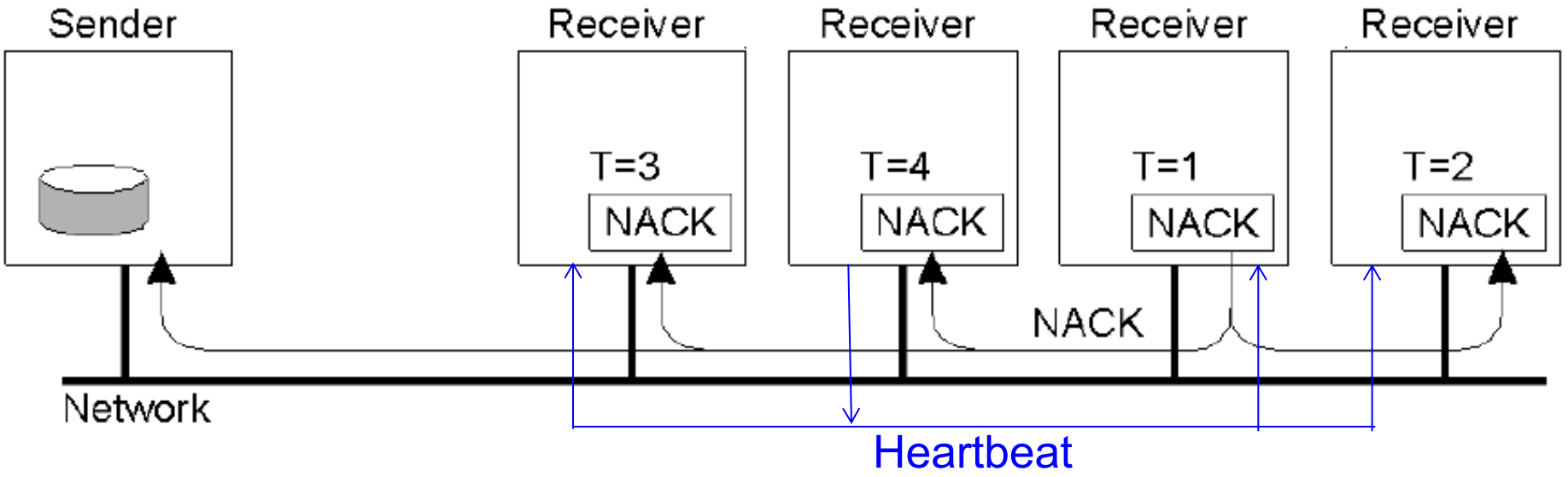
- Solutia TCP (corectia la transmitator) nu merge:
  - Multe ACK la transmitator - gatuire
  - Evidenta greoaie a setului de receptori
  - Conditii diferite la receptori diferiti (round-trip time, delay*bandwidth, legaturi supraincarcate diferit)
- Alta solutie: un Transport specializat peste UDP sau IP



Scalable Reliable Multicast - SRM

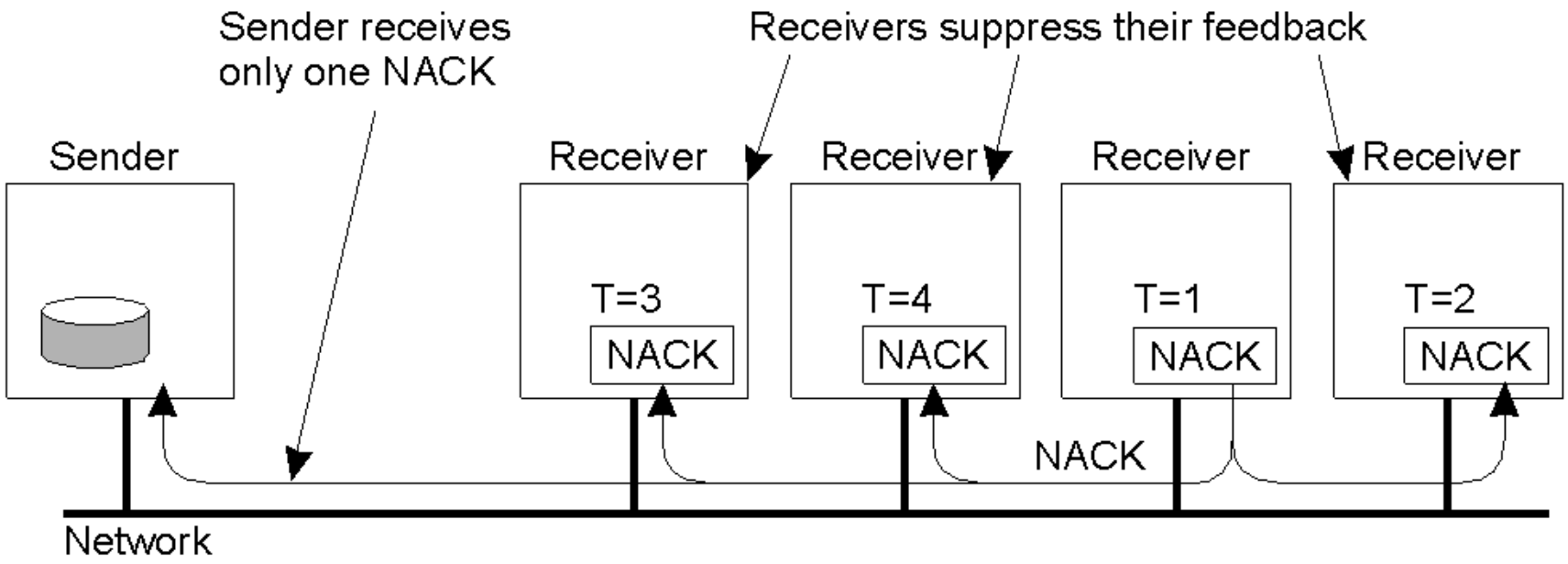
- Bazat pe IP multicast (ne-fiabil) la care:
  - Sursa transmite pe o adresa multicast fara sa cunoasca membrii grupului
  - Un receptor intra in grup sau iese din grup fara a afecta ceilalti membri
- Adauga caracteristici TCP
  - Fiabilitate capat la capat
  - Adaptare la conditiile retelei (dimensiune, trafic, topologie)
- Ideea - corectia la receptor
  - Fiecare membru al grupului raspunde de receptia corecta a mesajelor care ii sunt adresate

SRM – controlul cererilor



- Mesajele transmise de Sender poarta numere de secventa
- Se mai folosesc mesaje:
  - *Heartbeat* (intre receptori) – informeaza despre numerele de secventa ale mesajelor primite
  - *NACK* (spre Sender) – cerere retransmitere
  - *Repair* (de la Sender) – retransmitere din cache

Optimizari



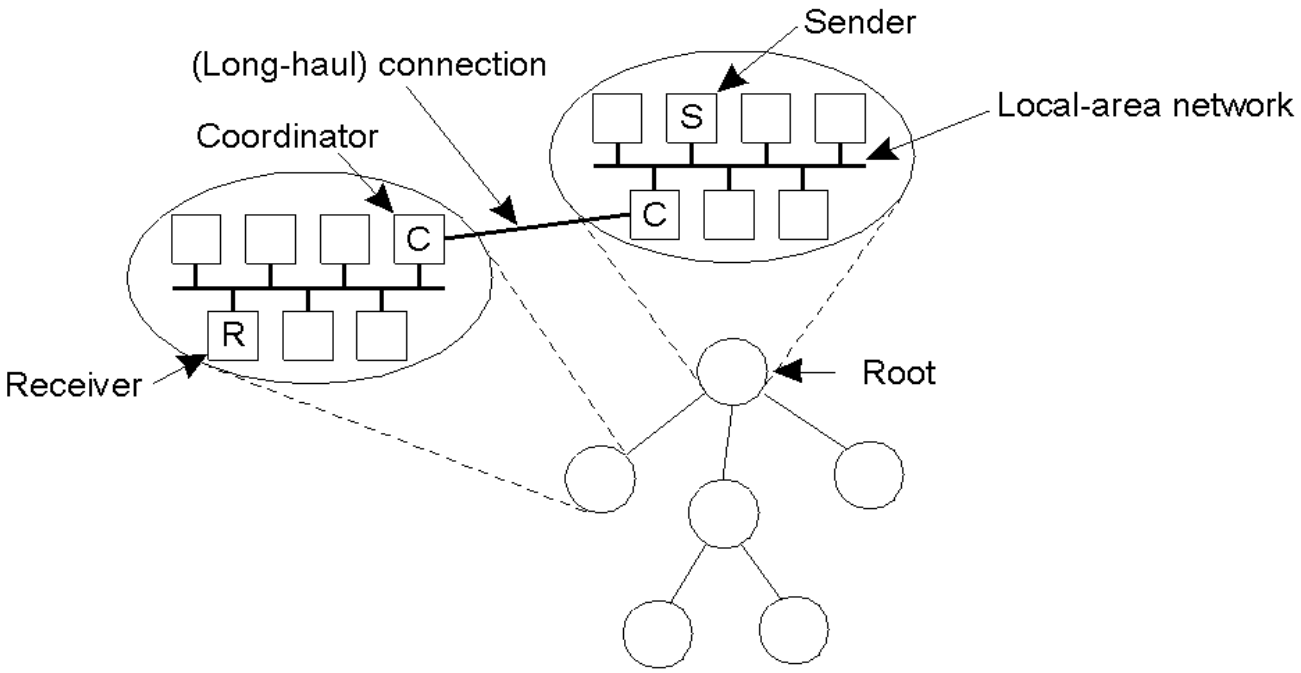
Minimizare numar *NAK – un singur NAK ajunge la transmitator*

- Transmitere multicast NAK cu intarzieri aleatoare (diferite de la un receptor la altul)
- Procesele care primesc un NAK si nu au transmis anuleaza propriul NAK
- Problema: planificarea intarzierilor pentru a asigura un singur NAK

Optimizari

- Evitare consum resurse pentru mesaje *Repair*
  - Retransmitere punct la punct doar pentru procese care nu au mesajul
  - Retransmitere multicast pe un alt grup, doar pentru procesele care au pierdut acelasi mesaj m
    - procesele care nu au primit m se alatura acestui grup
    - evita retransmiterea catre procese care au primit corect mesajul
    - dezavantaj - solutia cere un management foarte eficient al grupurilor, ceeace este greu de realizat
- Accelerare proces – recuperare locala
  - Retransmitere facuta de un membru "local"

Multicast fiabil ierarhic



Grupul este impartit in subgrupuri cu **coordonatori locali** care

- primesc mesaj de la Sender si retransmit mesajele catre receptori.
- primesc cererile de retransmitere de la receptori si transmit un singur mesaj la Sender

Problema: constructia dinamica a arborelui

- utilizare arbori multicast din reteaua suport
- solutie la nivelul aplicatiei (overlay network)

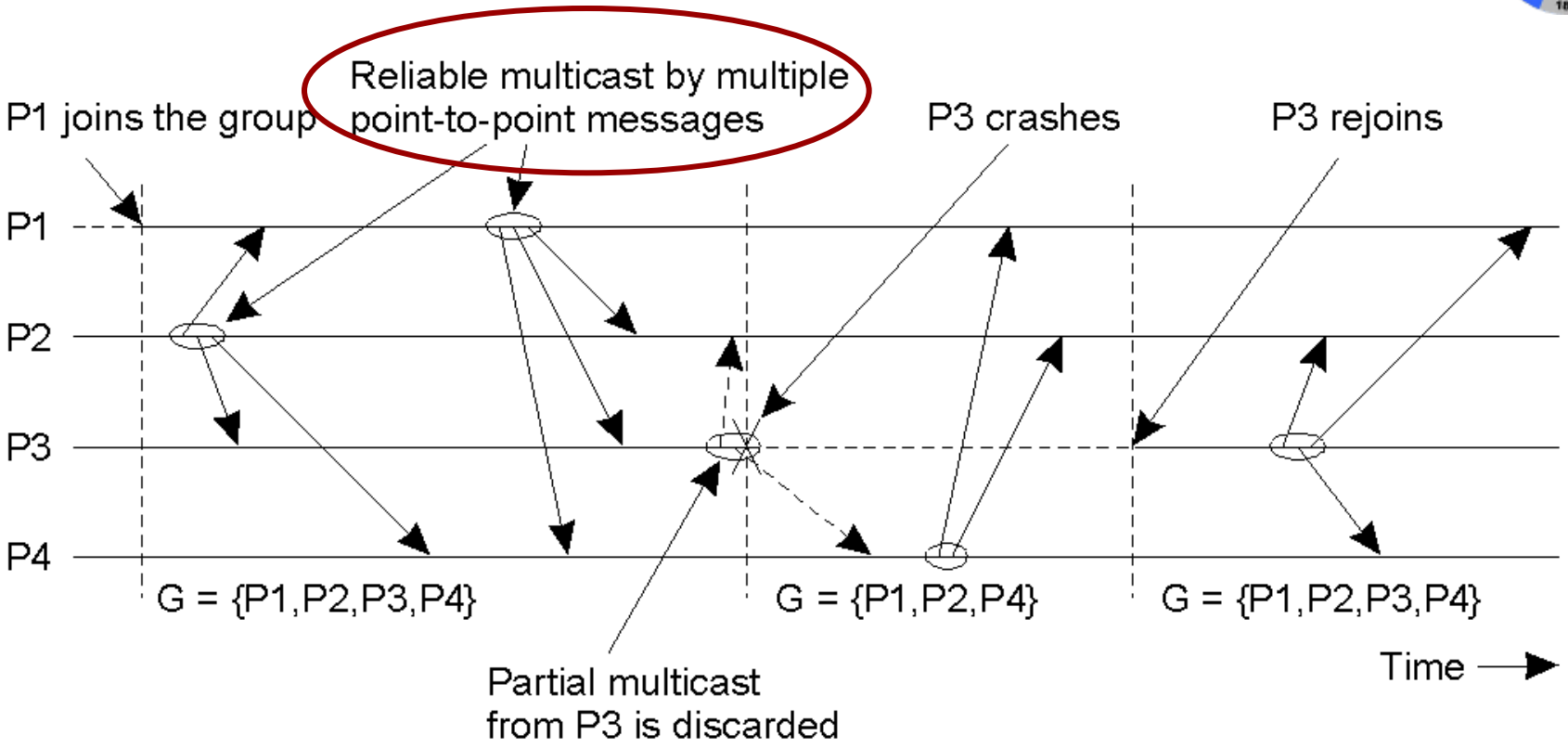
Multicast Atomic

Multicast fiabil chiar daca **unele procese se defecteaza**

Problema multicast atomic

- un mesaj este:
  - fie livrat tuturor proceselor din grup
  - fie nelivrat
    - acceptabil cand transmitatorul cade
    - similar cu caderea transmitatorului inaintea transmiterii mesajului
  - in plus, mesajele sunt livrate in **aceeasi ordine** tuturor proceselor
- **Solutia** tine cont de instabilitatea grupului, pentru care:
  - foloseste setul de procese din grup "vazut" de transmitator cand a trimis mesajul - **group view**

Virtual Synchrony



Toate procesele din set au acelasi view

Un group view se poate schimba

Toate operatiile multicast **au loc intre schimbarile de view.**

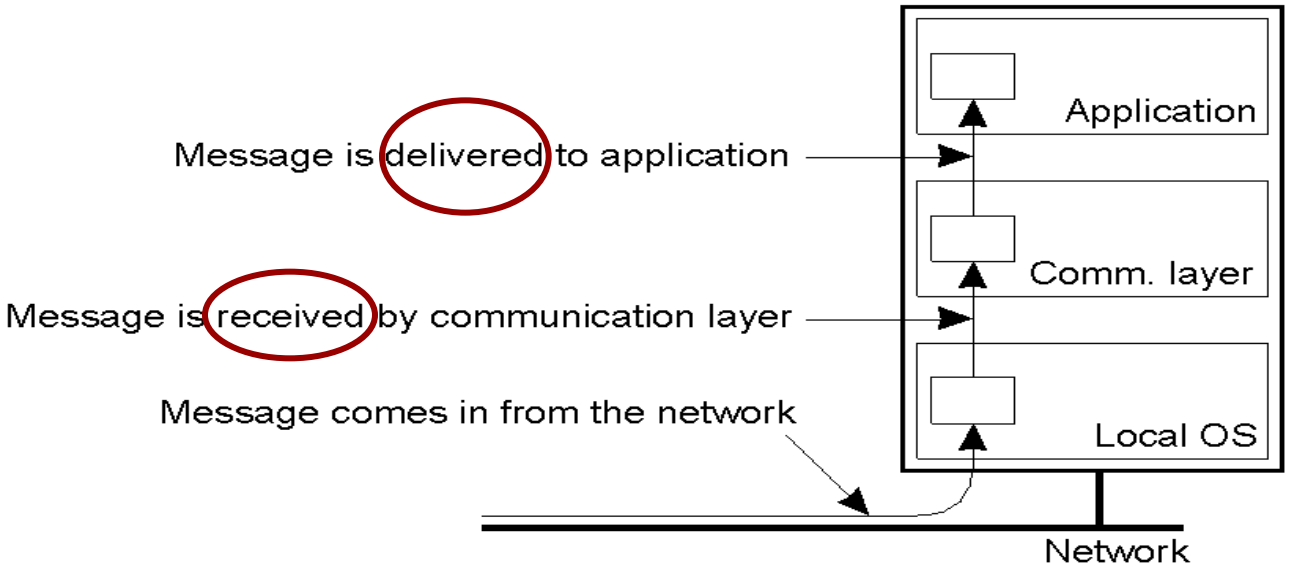
Principiul Virtual Synchrony

- Anunt schimbare group view (**G**)
  - se face prin transmiterea multicast a unui mesaj **vc – view change**

- Principiul **multicast-ului sincron virtual**
  - Daca m este transmis proceselor din G si, inainte de terminare, se transmite **vc** atunci
    - toate procesele din G primesc **m** inainte de **vc**,
    - sau nici un proces nu primeste **m**
  - Mesajul **m** transmis lui G nu poate fi livrat **decat** proceselor din G.

Implementare Virtual Synchrony (1)

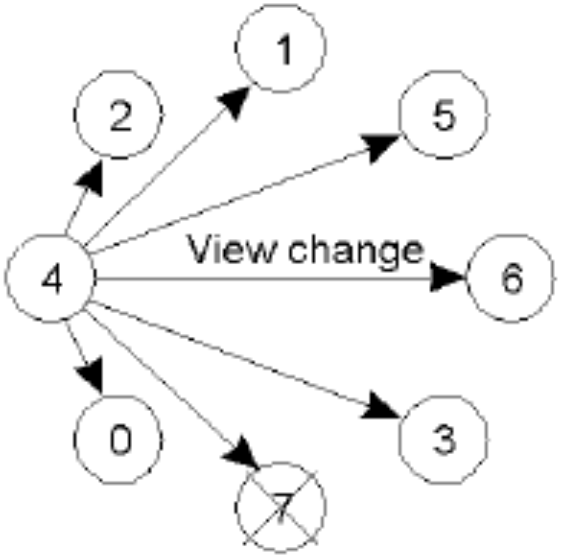
- Utilizeaza **comunicarea fiabila punct-la-punct** (TCP)
  - multicast = trimitere secventiala mesaj **m** fiecarui membru din grup
- Mesajele sunt **receptionate** de **nivelul comunicare** in ordinea primirii de la Local OS
- si sunt **livrate** aplicatiei intr-o ordine diferita (FIFO, cauzala), aceeasi pentru toate procesele



Implementare Virtual Synchrony (2)

- Obiectivul adoptat la implementare
  - un mesaj trimis unui group view G trebuie livrat tuturor proceselor valide din G inainte de urmatoarea schimbare de view
- Probleme
  - **vc** apare **inainte** ca mesajul **m** sa fie livrat tuturor proceselor din G
  - daca **transmitatorul cade**, cum iau mesajul procesele care nu l-au primit?
- Solutia foloseste doua categorii de mesaje:
  - mesaj **stabil** (a fost **primit** de toate procesele din grup): poate fi **livrat**
  - mesaj **instabil**: pastrat de procesele care l-au primit;
    - cand transmitatorul cade - unul din procese il va trimite celorlalte

Implementare Virtual Synchrony (3)

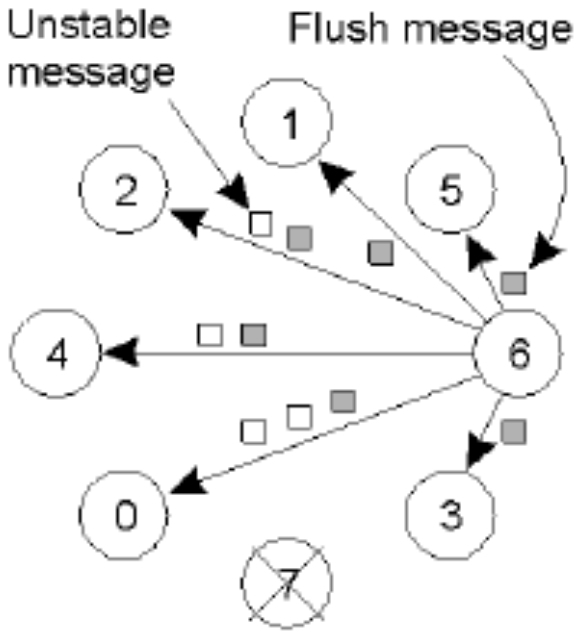


Un proces initiaza un view change cand

- primeste un mesaj provenind de la un alt proces care vrea sa se alature grupului,
- sa paraseasca grupul
- sau detecteaza defectarea unui proces in grupul curent

In figura, procesul 4 observa ca 7 a cazut si trimite un mesaj view change celorlalte procese, urmand ca 7 sa fie eliminat din group view

Implementare Virtual Synchrony (4)



Inainte de trecerea la noul group view G_{i+1}, mesajele instabile din grupul curent G_i trebuie facute stabile (adica transmise tuturor proceselor valide din G_i)

Ex, procesul 6 (care a primit view change) trimite proceselor valide toate mesajele instabile si le schimba starea in stabile

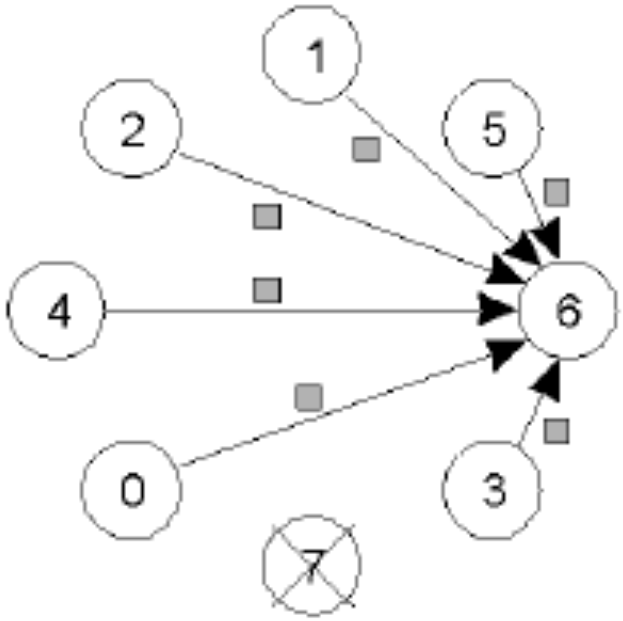
Apoi, procesul trimite un mesaj flush (nu mai are mesaje instabile)

Celelalte procese procedeaza la fel

Se asigura ca fiecare mesaj care a fost primit de cel putin un proces din G_i va fi primit de oricare alt proces valid din G_i

•alternativa – doar procesul 6 (coordonator) difuzeaza mesajele

Implementare Virtual Synchrony (5)



Procesul 6 **instaleaza** noul **view** cand primeste mesaje flush de la toate celelalte procese

Fiecare proces valid procedeaza ca 6

Ordinea livrarii mesajelor multicast

a) nici una

b) **FIFO** – mesajele de la un **acelasi proces** sunt livrate in ordinea transmiterii

Process P1	Process P2	Process P3	Process P4
sends m1	receives m1	receives m3	sends m3
sends m2	receives m3	receives m1	sends m4
	receives m2	receives m2	
	receives m4	receives m4	

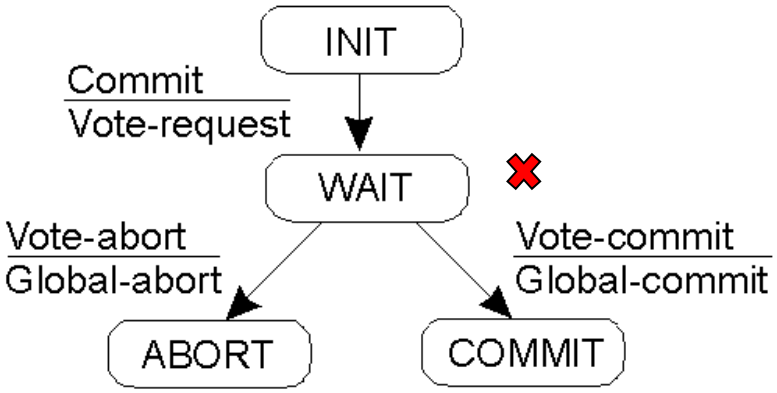
c) **cauzala** – daca **m1** precede cauzal **m3**, **m1** este primit inaintea lui **m3**

Process P1	Process P2	Process P3	Process P4
sends m1	receives m1	receives m1	sends m3
sends m2	receives m3	receives m3	sends m4
	receives m2	receives m4	
	receives m4	receives m2	

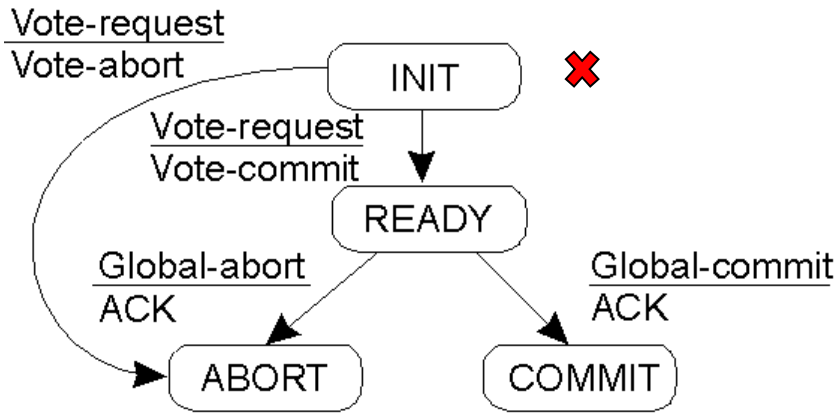
Distributed commit

- Definitie: data fiind o operatie distribuita proceselor unui grup, se asigura ca
  - sau fiecare proces din grup executa operatia
  - sau nici un proces nu o executa
- Solutie: folosirea unui coordonator
- Trei variante
  - one-phase commit – nu poate trata caderea unui proces
  - two-phase commit – nu poate trata caderea coordonatorului
  - three-phase commit

Two-Phase Commit



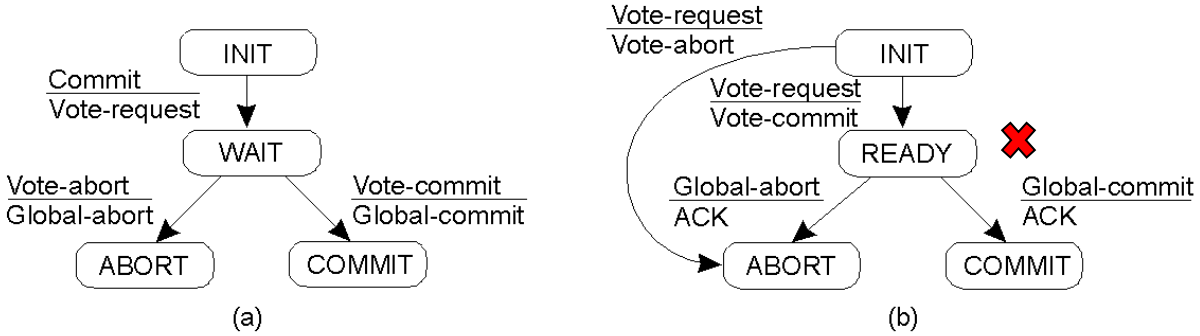
Masina de stari ptr **coordonator**



Masina de stari ptr un **participant**

- Coordonator este procesul care initiaza operatia
- Exista stari in care coord (WAIT) sau participant (INIT, READY) asteapta
- Pentru evitarea blocarilor se folosesc timeout-uri
- Comportare participant la timeout in INIT ???
  - trimite VOTE_ABORT
- si coordonator la timeout in WAIT ???
  - trimite GLOBAL_ABORT

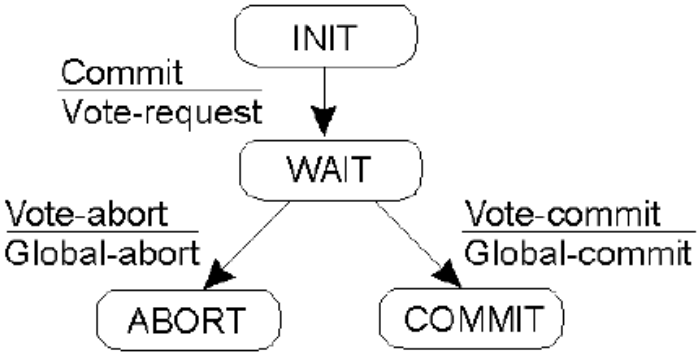
Comportare participant la timeout in READY



Participant *P* in starea *READY* contacteaza, la timeout, un alt participant Q.

Starea lui Q	**Actiunile lui P**	**Justificare**
COMMIT	Trece in COMMIT	P a pierdut GLOBAL_COMMIT
ABORT	Trece in ABORT	P a pierdut GLOBAL_ABORT
INIT	Trece in ABORT	Coord a crapat inainte de trimitere VOTE_REQUEST tuturor participantilor. La timeout, Q va trece si el in ABORT
READY	Contacteaza alt participant	Daca toti READY atunci asteapta recuperare coordonator

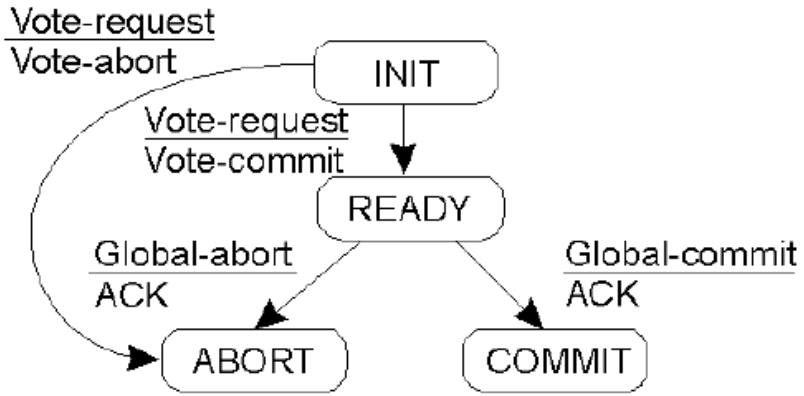
Logare stare la Coordonator




```
write START_2PC to local log;
multicast VOTE_REQUEST to all participants;
while not all votes have been collected {
    wait for any incoming vote;
    if timeout {
        write GLOBAL_ABORT to local log;
        multicast  GLOBAL_ABORT to all participants;
        exit;}
    record vote;                    /*inregistreaza un vot*/
}
if all participants sent VOTE_COMMIT and coordinator votes COMMIT{
    write GLOBAL_COMMIT to local log;
    multicast GLOBAL_COMMIT to all participants;}
else {
    write GLOBAL_ABORT  to local log;
    multicast GLOBAL_ABORT to all participants;
}
```

Logare stare la participant

```
write INIT to local log;
wait for VOTE_REQUEST from coordinator;
if timeout {
    write VOTE_ABORT to local log;
    exit;
}
if participant votes COMMIT {
    write VOTE_COMMIT to local log;
    send VOTE_COMMIT to coordinator;
    wait for DECISION from coordinator;
    if timeout {
        multicast DECISION_REQUEST to other participants;
        wait until DECISION is received; /* remain blocked */
        write DECISION to local log;
    }
    if DECISION == GLOBAL_COMMIT
        write GLOBAL_COMMIT to local log;
    else if DECISION == GLOBAL_ABORT
        write GLOBAL_ABORT to local log;
} else {
    write VOTE_ABORT to local log;
    send  VOTE_ABORT to coordinator;
}
```



Participant: tratarea cererilor de la alti participanti.

actions for handling decision requests: /* executed by separate thread */

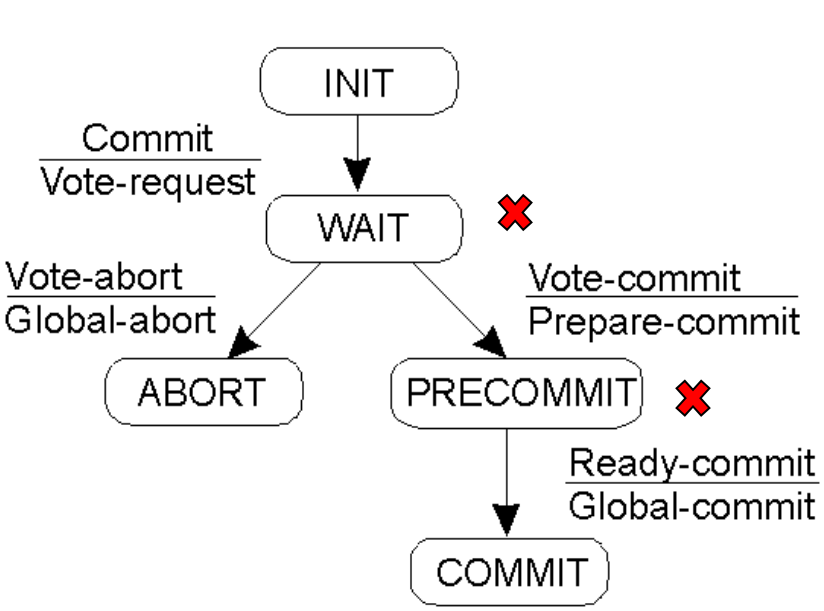
```
while true {
   wait until any incoming DECISION_REQUEST is received; /* remain blocked */
   read most recently recorded STATE from the local log;
   if STATE == GLOBAL_COMMIT
      send GLOBAL_COMMIT to requesting participant;
   else if STATE == INIT or STATE == GLOBAL_ABORT
      send GLOBAL_ABORT to requesting participant;
   else       /* STATE == READY */
      skip;  /* participant remains blocked */

}
```

Participantii nu pot decide cand:

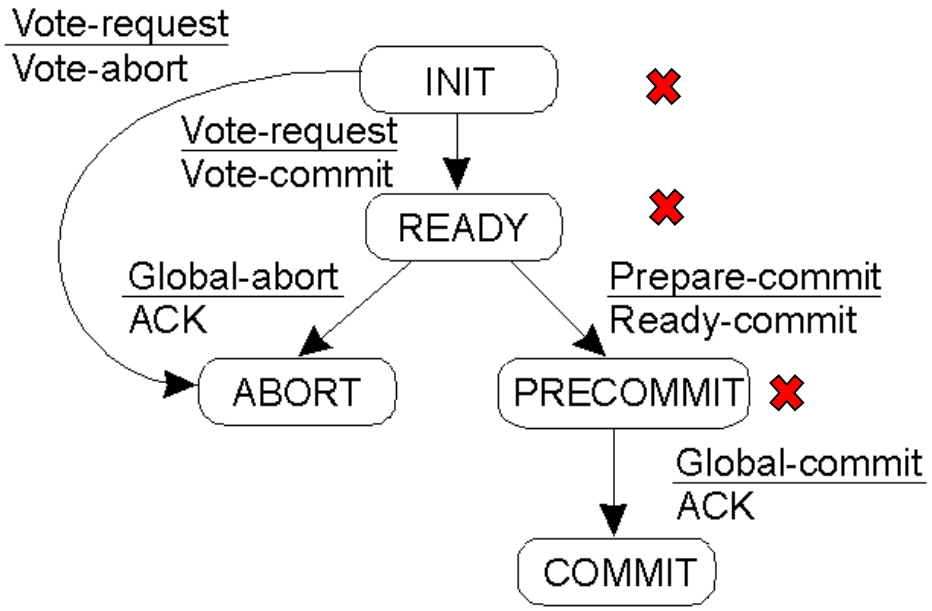
- toti participantii au primit si procesat VOTE_REQUEST
  deci STATE == READY
- intre timp coordonatorul a crapat si nu trimite decizia

Corespunde alternativei **skip** din actiunile participantilor

Three-Phase Commit

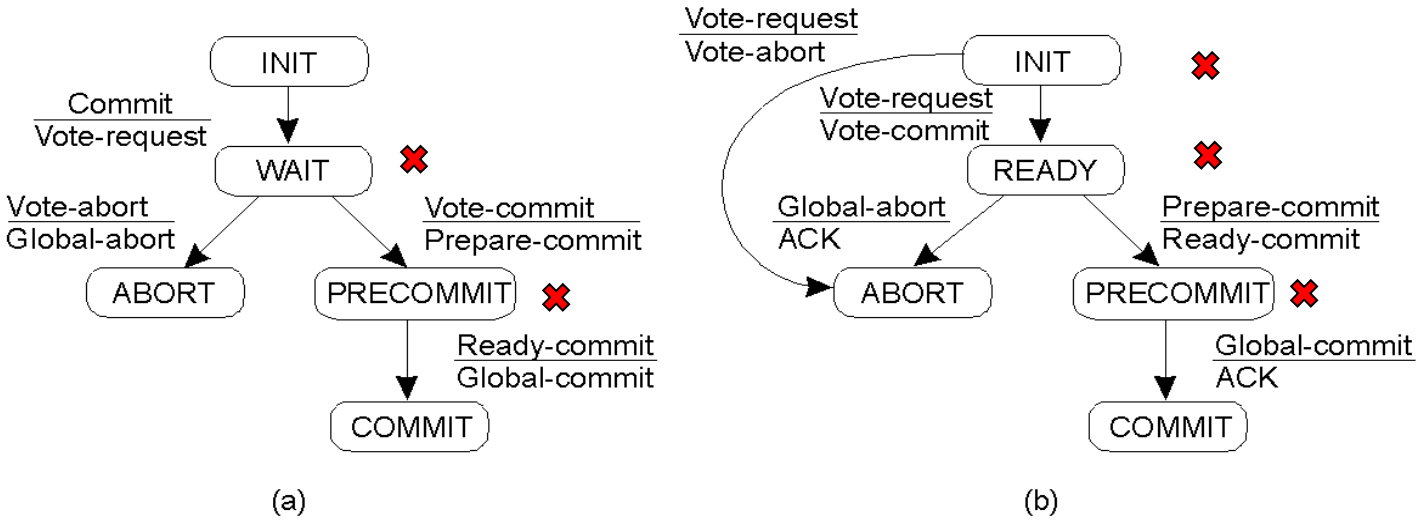


(a)



(b)

a) Masina de stari pentru coordonator in 3PC
b) Masina de stari pentru un participant



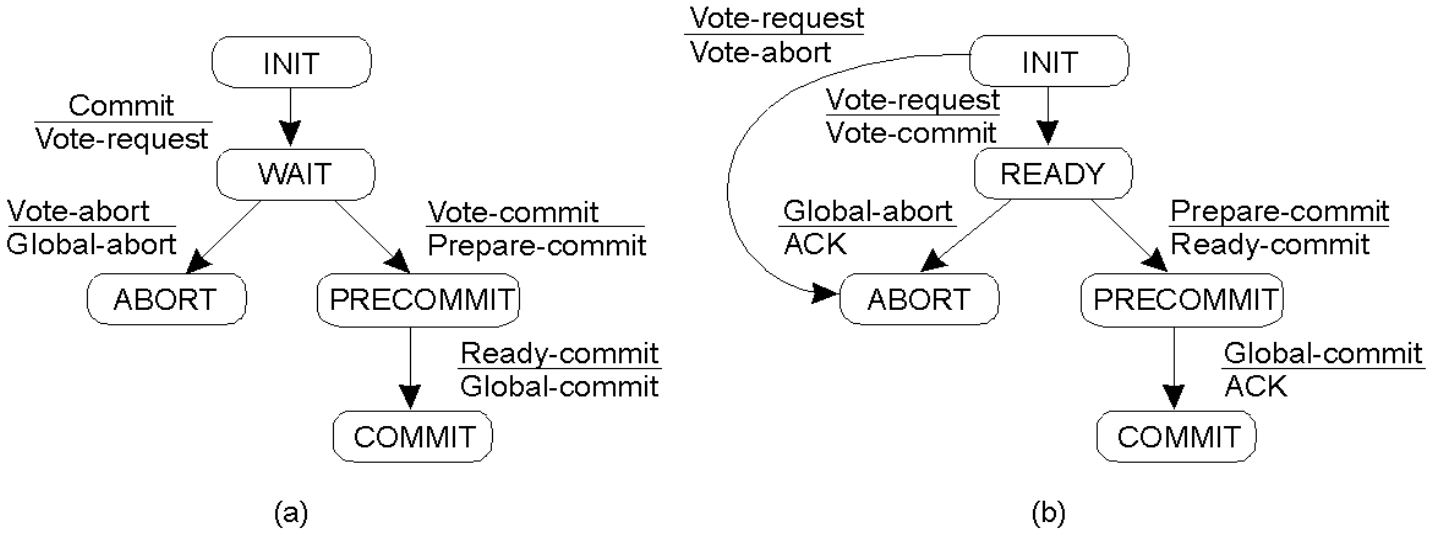
Actiuni la **timeout**

Principiul: pe calea spre COMMIT, coordonatorul si participantii **nu difera prin mai mult de o tranzitie**

Coordonator in WAIT –> la timeout, Global-abort (un participant a cazut)

Coordonator in PRECOMMIT –> Global-commit (un participant a cazut dar el a votat pentru comitere si, la recuperare va comite)

Participant in INIT –> Vote-abort (coordonatorul a cazut)



Actiuni la **timeout**

Participant P in READY sau PRECOMMIT

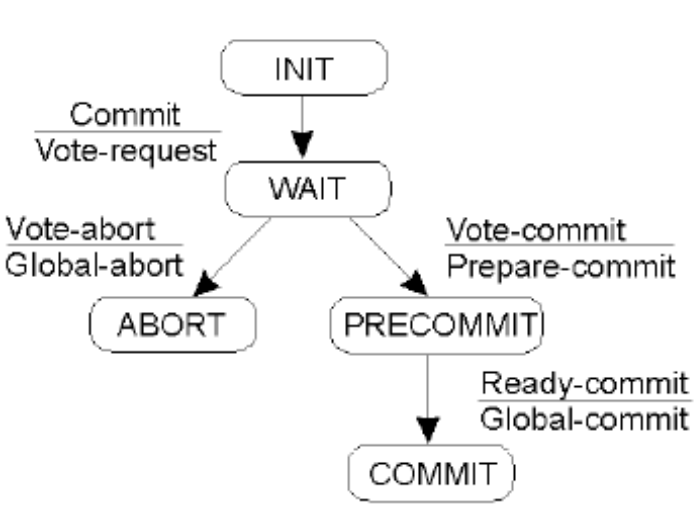
P presupune coord. a cazut → contacteaza alti participanti Q

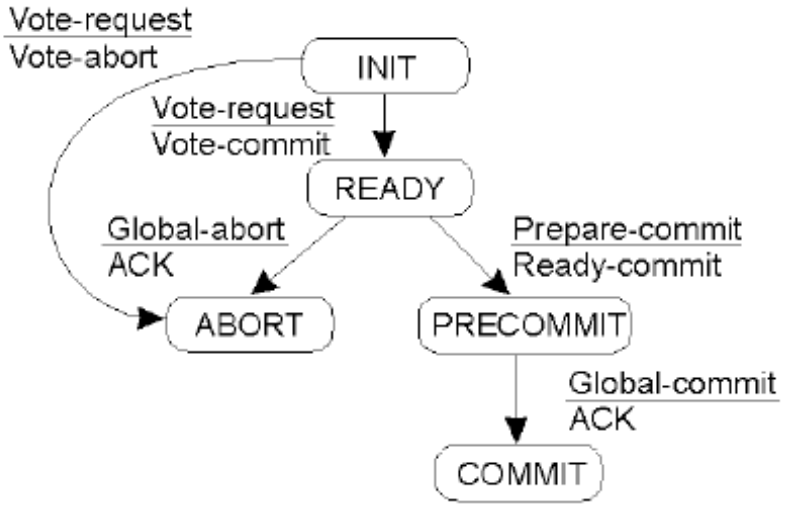
un Q in COMMIT (P in PRECOMMIT) → P trece in COMMIT

un Q in ABORT → P trece in ABORT

toti in PRECOMMIT (inclusiv P) → P trece in COMMIT

un Q in INIT (coord si nici un participant nu sunt in PRECOMMIT) → P aborteaza





Daca doar unii participanti pot fi contactati

- *toti contactatii* sunt in PRECOMMIT si majoritari → P trece in COMMIT
  - un proces cazut va recupera in READY, PRECOMMIT sau COMMIT; procedura de recuperare il va conduce in COMMIT
- *toti contactatii* sunt in READY si formeaza majoritatea → P aborteaza
  - un participant cazut va recupera in INIT, ABORT si va aborta
  - sau in PRECOMMIT (NU in COMMIT) si la recuperare, va aborta, neprimind Global-commit si majoritatea participantilor abortand

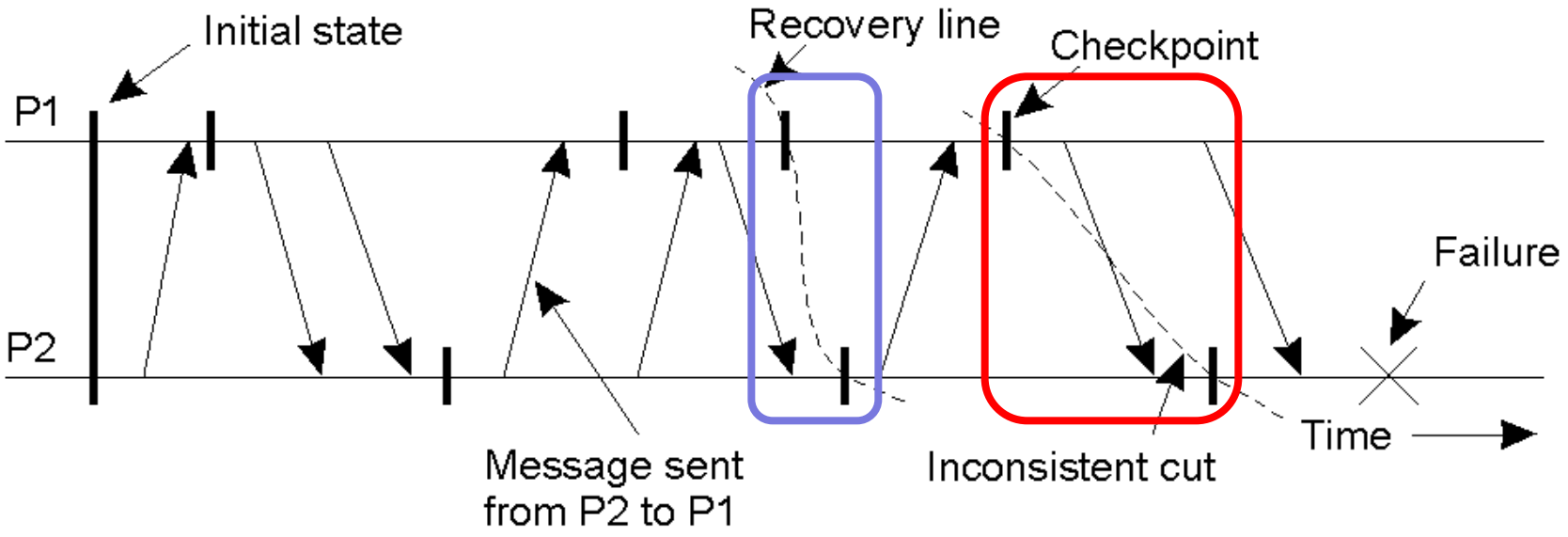
Concluzie: procesele supravietuitoare pot lua intotdeauna o decizie (spre deosebire de two-phase commit)

Recuperarea

- Sistemul este adus intr-o stare corecta
- Tipuri
  - inpoi (backward) – aduce sistemul intr-o stare anterioara
  - inainte (forward) – gaseste o stare noua din care sistemul poate continua operarea
- Recuperare inapoi
  - mai folosita
  - tehnici
    - checkpointing
    - message logging (folosita in combinatie cu checkpointing)
  - Recuperarea este mai complicata in sist distribuite
    - procesele trebuie sa identifice o stare consistenta

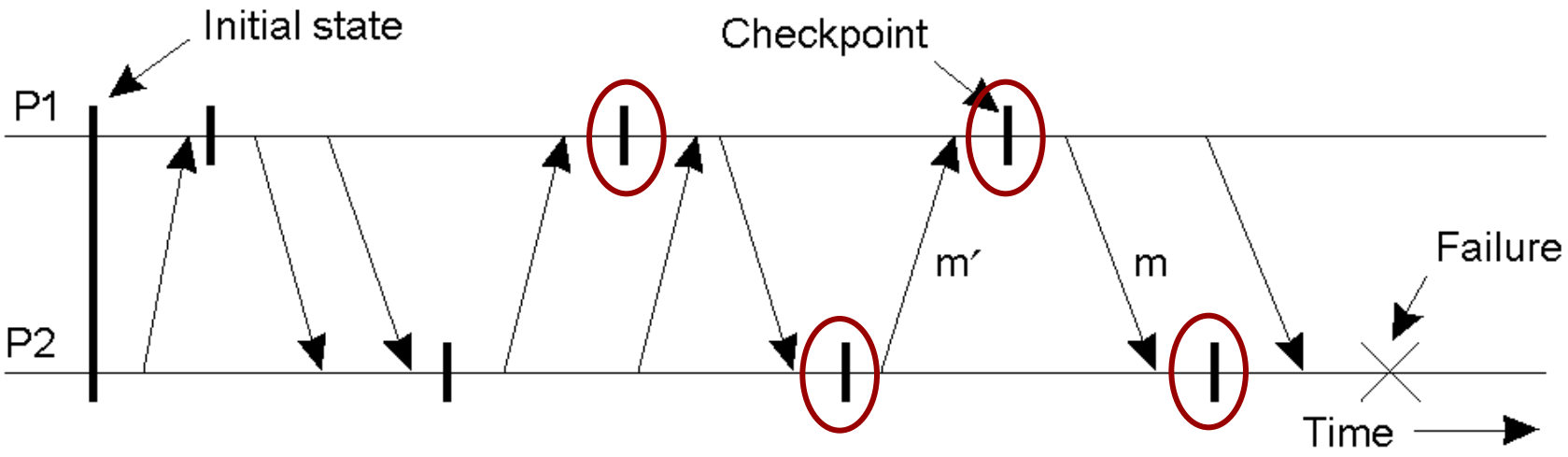
Checkpointing

- **Stare globala consistenta** = instantaneu distribuit
  - constituita din starile proceselor salvate in **memorii stabile** locale
  - constangere dificila pentru **mesaje**
    - daca P1 a inregistrat o receptie de mesaj starea trebuie sa includa o transmitere anterioara intr-un alt proces P2
- Linie de recuperare = cel mai recent instantaneu distribuit
  (cea mai recenta taietura consistenta)



Checkpointing Independent

- Fiecare proces salveaza starea periodic, independent de alte procese
- Linie de recuperare greu de gasit:
  - Efectul de domino

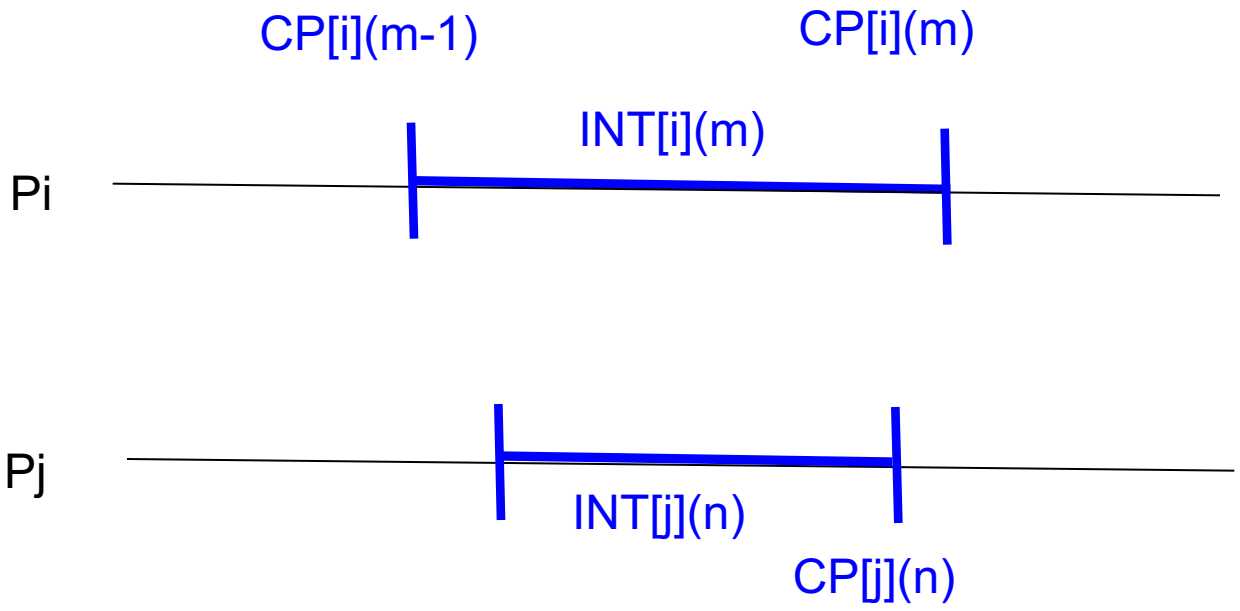


Pentru recuperare se inregistreaza dependentelor intre checkpoint-urile luate de diferite procese

Dependente intre checkpoints

Tine cont de mesajele trimise in intervalele dintre checkpoint-uri:

- *CPi(m)* denota checkpoint *m* al procesului *Pi*
- *INTi(m)* intervalul intre *CPi(m-1) si CPi(m)*.
- Ideea: Inregistreaza, la CP, dependenta intervalelor



Protocolul

Protocol:

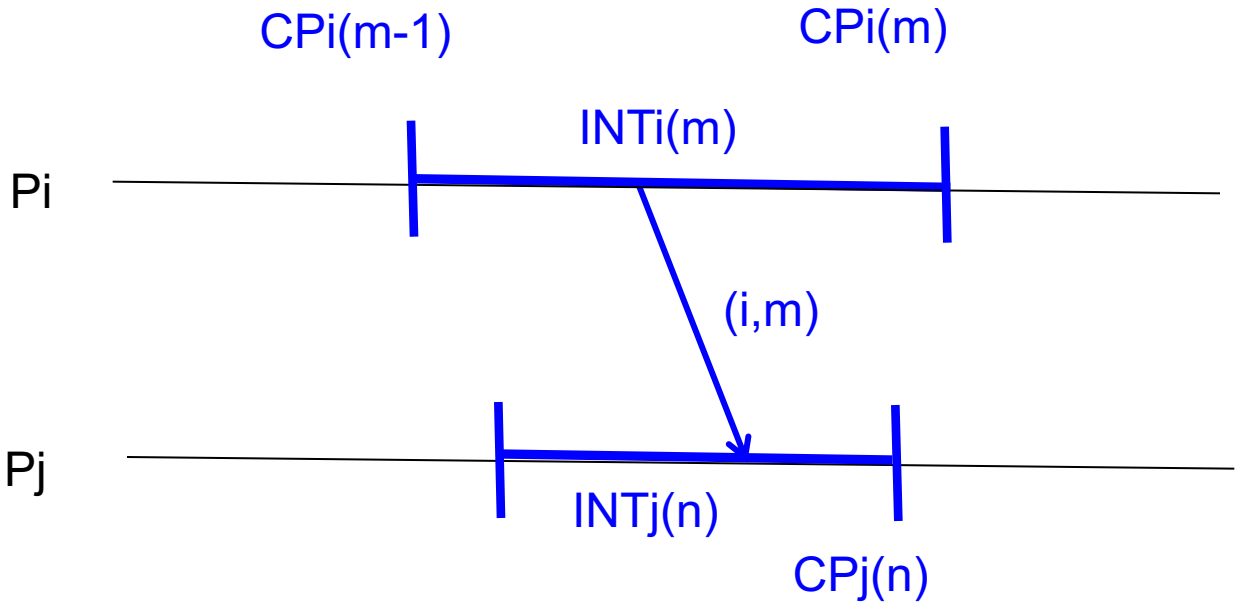
Pi trimite mesaj in *INTi(m)* si adauga (i,m) in mesaj

Pj primeste mesaj in *INTj(n)*

inregistreaza dependenta *INTi(m)* → *INTj(n)*

la checkpoint CPj(n): salveaza dependenta (*INTi(m)* → *INTj(n)*)

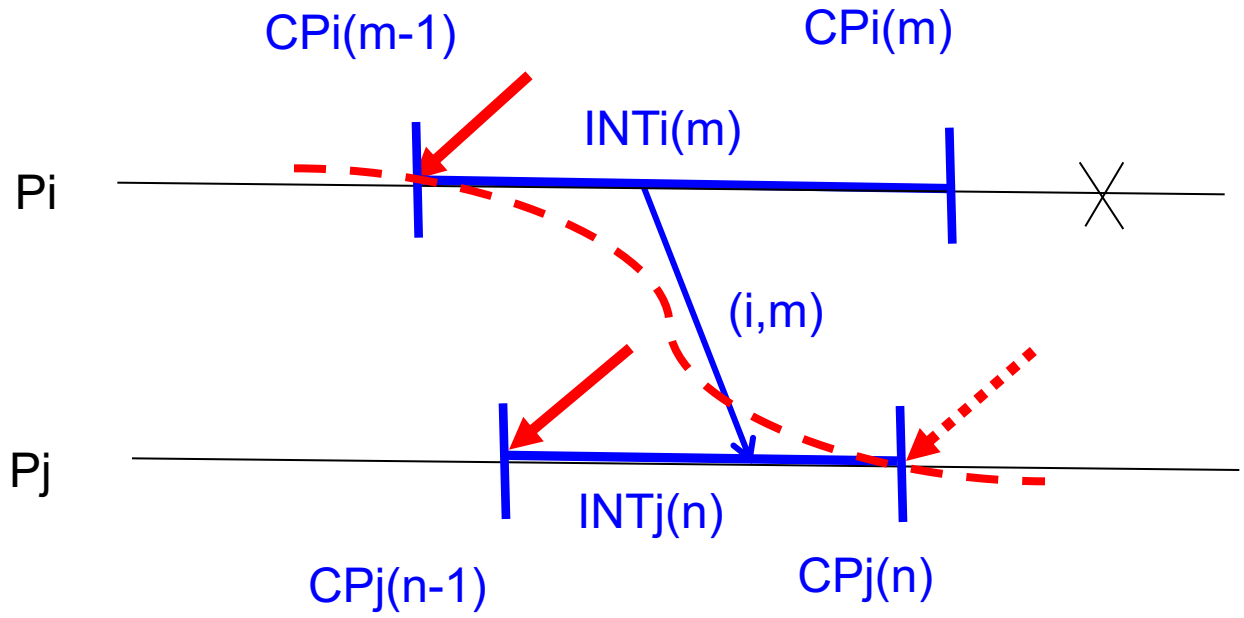
in memoria stabila



Consistenta

Daca *Pi* revine la *CPi(m-1)* => *Pj* revine la un CP anterior primirii mesajelor trimise de Pi in intervalul INTi(m), adica la CPj(n-1)

Daca prin aceasta starile nu sunt consistente, sunt necesarealte reveniri



Checkpointing Coordonat

Solutie simpla: protocol cu blocare in doua faze:

Coordonator: trimite multicast cerere *checkpoint*
Participant: primeste o cerere *checkpoint*
face checkpoint
pune intr-o coada mesajele care ii sunt pasate de aplicatia
pe care o executa, pentru a fi trimise
raporteaza ca a facut checkpoint
Coordonator: asteapta toate confirmarile de checkpoint
difuzeaza *checkpoint done*
Participant: primeste *checkpoint done*
continua

Solutia conduce la o stare globala consistenta:

- mesajele pasate de aplicatii pentru trimitere sunt pastrate in coada pana dupa checkpoint done
- in checkpoint-ul local nu vor fi inregistrate mesaje primite dupa cererea de checkpoint

Logarea Mesajelor

- Salvarea starii proceselor este costisitoare
  - Checkpoint se foloseste la intervale mari de timp
    - toate calculele de la ultimul checkpoint trebuie refacute – nu este convenabil
  - starea consistenta globala se poate reface prin "rejucarea" mesajelor logate de la ultimul checkpoint
- Protocol
  - periodic, fiecare proces salveaza starea sa locala
  - procesul logheaza mesajele pe care le-a primt dupa integistrarea starii
  - cand procesul cade, se creeaza un nou proces in locul lui
  - noului proces i se da starea locala si
  - i se trimit mesajele logate in ordinea originala

Procese orfane

- Cerinta pentru protocoalele de recuperare:
  - dupa recuperare, starea procesului sa fie consistenta cu a celorlalte procese
  - Consistenta = sa nu existe **procese orfane**
  - Def. Un proces R este orfan daca rezista caderii unui alt proces Q dar a carui stare este inconsistenta cu starea lui Q dupa recuperare

- Solutia: model de executie determinist pe bucati
  - executie = secventa de intervale
  - fiecare interval incepe cu un eveniment nedeterminist (ex. receptia unui mesaj)
  - in interval executia este determinista si poate fi rejucata de un proces
- Concluzie: in vederea recuperarii, inregistram doar evenimentele nedeterministe

Cand trebuie inregistrate mesajele?

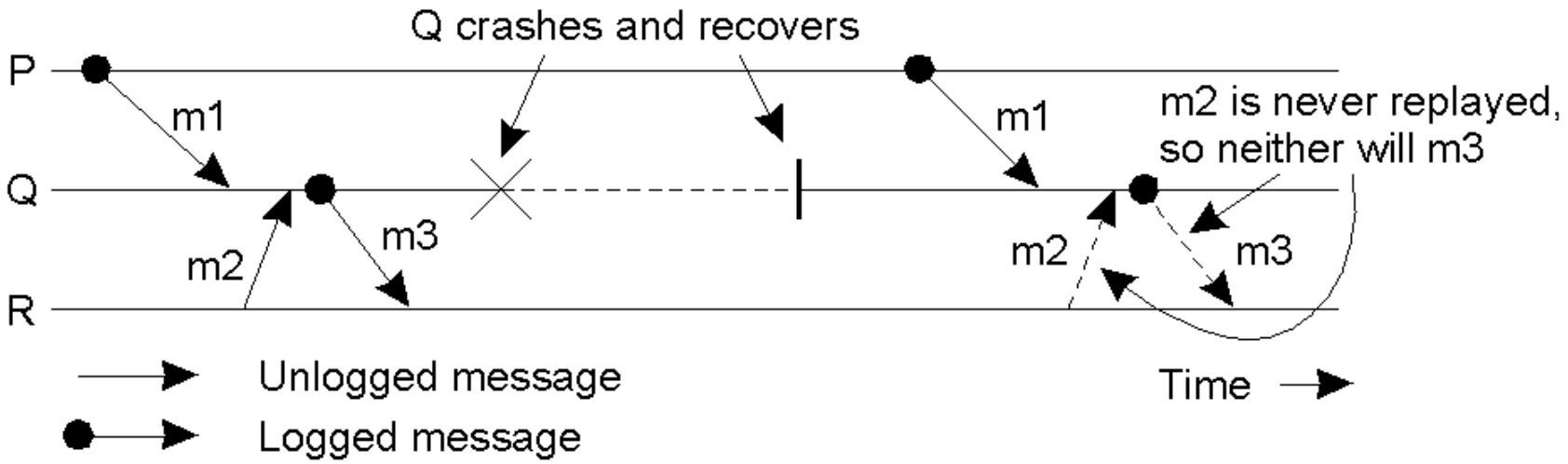
Ex. Q a primit si livrat m1 si m2; m1 este logat; m2 nu este logat

Q transmite m3 lui R; R primeste si livreaza m3

Q cade

La reluare se rejoaca doar mesajele pentru recuperarea lui Q

- Se rejoaca m1 (logat); m2 (nelogat) nu este rejucat ➔ m3 de asemena
  - Pentru R: m3 este primit de la Q;
  - pentru Q recuperat: m3 (ne-rejucat) nu apare ca transmis
- ➔ stari inconsistente pentru Q si R ➔ R devine orfan - are un mesaj care (aparent) nu a fost transmis



Informatii logate

- **HDR[m]**: antet mesaj *m* = *(*sursa, destinatia, numar secventa, numar livrare)
  - Antetul contine toate info necesare retransmiterii mesajului si livrarii sale in ordinea corecta;
  - datele sunt reproduse de aplicatie
- Mesaj *m* este **stabil** cand nu mai poate fi pierdut (a fost pus in memoria stabila).
  - poate fi "rejucat" pentru recuperare.
- **DEP[m]**: multimea proceselor la care s-a livrat *m* si proceselor la care s-a livrat *m'* dependent cauzal de livrarea lui *m*
- **COPY[m]**: procesele care au o copie a lui *m* in memoria volatila (inca nu in memoria stabila locala).
  - Cand Q livreaza m el devine membru al lui COPY[m].
  - Un proces din COPY[m] poate oferi o copie a lui m pentru a fi utilizata in rejucarea transmisiei lui m (daca procesul nu a cazut!)
- ***C*** colectia proceselor cazute

Scheme de logging (2)

Ipoteza: intr-un sistem distribuit, unele procese s-au defectat, dar R supravietuieste defectelor

R este **orfan** daca este dependent de un mesaj **m** si nu se poate rejuca transmisia lui **m**

adica – R este in DEP[m] si COPY[*m*] este inclus in C.

Problema: evitare orfani

Daca procesele din COPY(m) au cazut, sa nu existe procese in DEP(m)

Se poate asigura daca, atunci cand un proces devine dependent de **m**, el sa aiba o copie a lui **m**

altfel spus, daca procesul devine membru al lui DEP[m] atunci devine membru si al lui COPY[m]

Solutii

Protocol pesimist: pentru fiecare mesaj *non-stabil m*, exista cel mult un proces dependent de *m*, adica |DEP[m]|<=1

Consecinta: daca P primeste *m*, el il face stabil (scrie in memoria stabila) inainte de a trimite mesajul urmator

Protocol optimist: pentru fiecare mesaj non-stabil *m*, daca COPY[m] este inclus in C, atunci DEP[m] este inclus de asemena in C

Consecinta: actioneaza dupa o "cadere" - fiecare proces orfan R este intors la o stare in care R nu este in DEP[m].

complicat de implementat

Studiu de caz: Dynamo (*)

Modul de operare

- infrastructura Amazon cuprinde milioane de componente
- in orice moment numarul de componente defecte este semnificativ
- tolerarea defectelor este modul normal de operare

Solutia Dynamo – depozit de date cu disponibilitate ridicata

- model simplu de date, cheie-valoare
- replicarea datelor conform model eventual consistency
  - propagarea schimbarilor la replici se face in background
  - suporta deconectari de retea si actualizari concurente ale unor replici diferite
    - creste disponibilitatea la scriere a depozitelor de date (always writeable)
  - permite tratarea conflictelor de catre aplicatie

(*) G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall and W. Vogels. Dynamo: Amazon's Highly Available Key-value Store

Topici discutate

- Interfata depozitului de date
- Partitionarea datelor
- Replicarea
- Versionarea datelor
- Executia operatiilor
- Tratarea defectelor tranzitorii
- Tratarea defectelor permanente

Interfata

Interfata are doua operatii

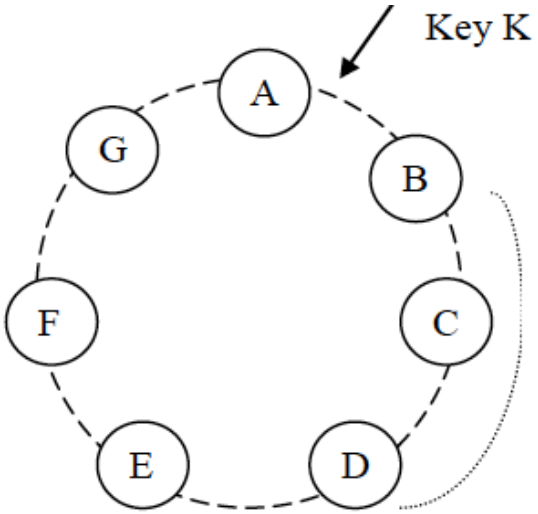
- put (key, context, object)
  - determina nodurile (virtuale) unde trebuie stocate replicile obiectului si scrie in depozit obiectul si contextul
  - nodurile sunt gasite printr-un algoritm de complexitate logaritmica
  - contextul include, printre altele, versiunea obiectului
- get (key)
  - localizeaza replicile (atentie, diferit de modelul clasic de citire a unei singure replici!) asociate cu key si intoarce un singur obiect sau o lista de obiecte conflictuale plus un context

Partitionarea datelor

Depozitul de date consta din mai multe noduri fizice, fiecare incluzand mai multe noduri virtuale

Datele sunt repartizate unui set de noduri virtuale (de stocare)

- foloseste o schema de hashing consistent cu spatiul valorilor organizat logic in inel
- fiecare nod virtual este identificat printr-o valoare din acest spatiu, reprezentand pozitia nodului pe inel
- fiecare data este identificata printr-o cheie; hash-ul acestei chei reprezinta pozitia datei pe inel
- data este stocata in nodul cu
  cea mai mica pozitie >= pozitia datei,
  numit nod coordonator al cheii

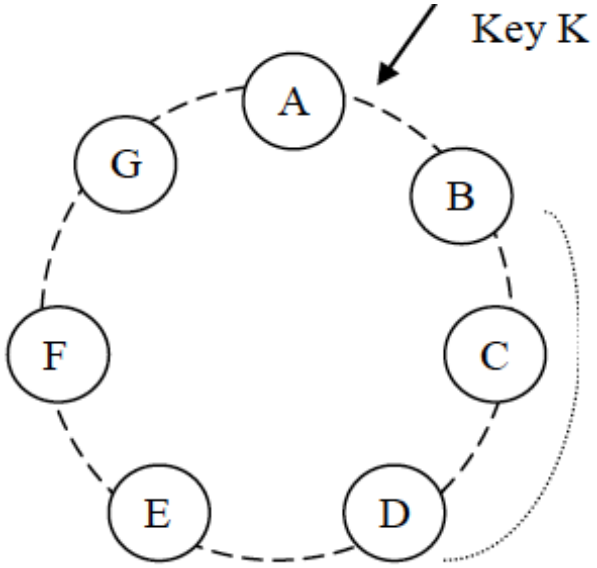


In figura, nodul virtual B depoziteaza cheile K din domeniul (A,B]

Replicarea

- o data este replicata in N noduri virtuale (N configurabil)
- schema: fiecare cheie este asignata coordonatorului care memoreaza datele corespunzatoare cheii si face replicarea lor la alte N-1 noduri virtuale; de ex.
  - daca N = 3, nodul coordonator B face replicarea cheii K la C si D
  - (B, C, D) alcatuiesc lista de preferinte a cheii K
- pentru toleranta la defecte
  - lista de preferinte include peste N noduri
  - nodurile virtuale trebuie sa fie situate

  in noduri fizice distincte (se “sar” nodurile virtuale care nu respecta conditia)



Versionarea

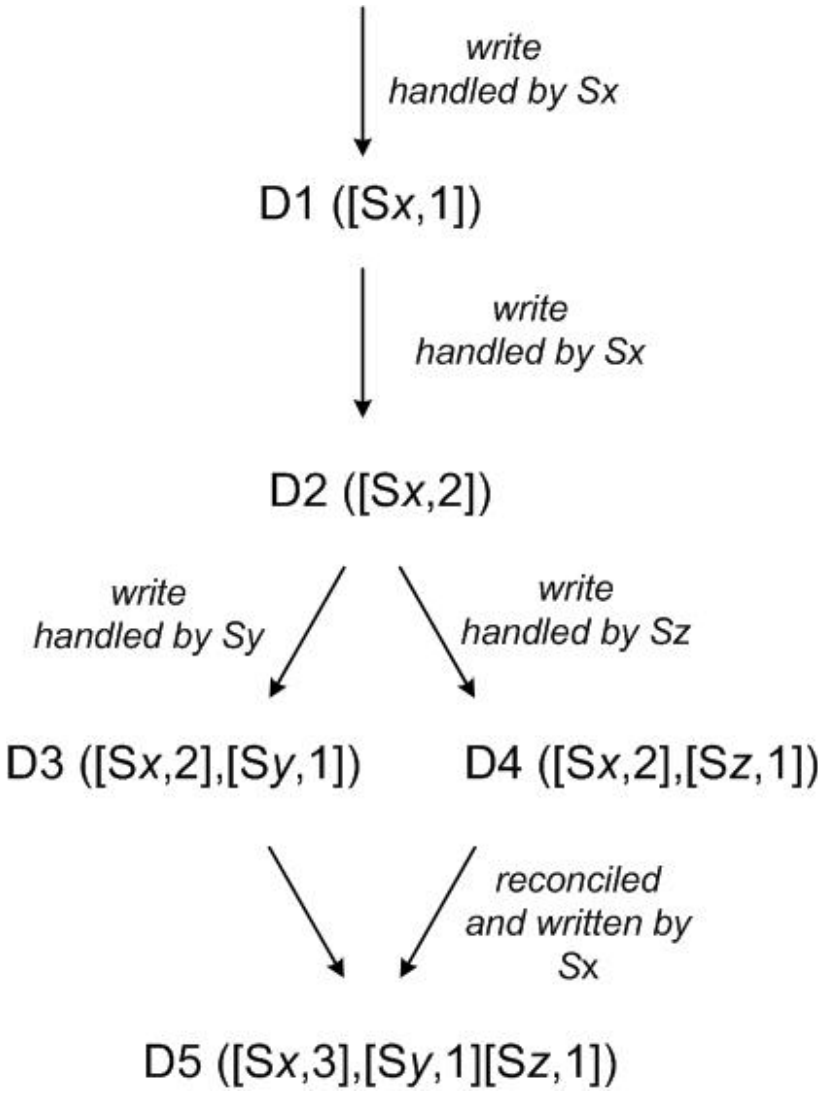
- Dynamo foloseste “eventual consistency” in care actualizarile sunt propagate asincron replicilor
  - o operatie **put** reda controlul apelantului inainte ca actualizarea sa se fi aplicat tuturor replicilor
  - consecinta: o operatie **get** ulterioara poate returna un obiect care nu are ultimele actualizari
- Ex. Amazon cere ca orice operatie “add to Cart” sa nu fie uitata sau rejectata
  - daca cea mai recenta replica a cosului nu este disponibila, operatia se face pe o replica mai veche → pot aparea conflicte intre replici
  - reconcilierea urmeaza sa se faca la o citire (get) ulterioara
- Amazon trateaza rezultatul fiecarei modificari (put) a unei date ca o noua versiune

Vectori de timp

- Pentru a reprezenta versiunile, se folosesc vectori de timp
  - vector de timp = lista de perechi [nod, contor]
    - nodul care a generat versiunea
    - numarul versiunii (local nodului)
  - la o scriere, clientul specifica versiunea pe care o actualizeaza, pe care o cunoaste din contextul primit la o citire anterioara
    - obiectul actualizat capata o noua versiune
- versiuni dependente cauzal
  - daca vectorul de timp al versiunii unui obiect, A1 este mai mic sau egal cu cel al versiunii A2 atunci A1 este un stramos al lui A2 si poate fi ignorat
  - altfel, versiunile se considera in conflict si trebuie reconciliate

Un exemplu

- versiunea D1 produsa de nodul Sx este stramos al versiunii D2 (a aceluiasi obiect)
- scrieri concurente conduc la doua versiuni diferite D3 si D4, ambele dependente de D2 dar conflictuale (sunt pe ramuri diferite ale arborelui de dependente)
- la citire, ambele versiuni sunt oferite ca rezultat
- clientul reconciliaza D3 si D4 (coordonat de Sx) si produce o singura versiunea D5 (reducand cele doua ramuri la una singura)



Executia operatiilor get si put

Oricare nod este eligibil pentru a primi de la client invocarile operatiilor get si put pentru orice cheie

De regula, o invocare este gestionata de nodul coordonator pentru cheia k specificata in invocare

In cazul in care coordonatorul este inaccesibil, (partitionari ale retelei) invocarea (get sau put) este pasata unui alt nod din lista de preferinte

Executia operatiei invocate implica primele N noduri din lista de preferinte (ocolind nodurile defecte sau inaccesibile)

Consistenta operatiilor get si put

Pentru a pastra consistenta replicilor, executia operatiilor put si get se face conform modelului de cvorum

- operatia presupune ca
  - minimum N_W noduri participa la operatia de scriere (put) sau
  - minimum N_R noduri participa la operatia de citire (get)
  - cu conditia suplimentara $N_R + N_W > N$

La put (key, context, object), coordonatorul (sau inlocuitorul)

- genereaza vectorul de timp al noii versiuni si scrie local noua versiune
- trimite versiunea catre primele N noduri tangibile din lista de preferinte
- operatia se termina cand raspund N_W-1 noduri

Consistenta operatiilor get si put (2)

La get (key), coordonatorul

- cere toate versiunile de date pentru key de la primele N noduri tangibile din lista de preferinte
- asteapta N_R-1 raspunsuri pentru a trimite rezultatul la client
- trimite toate versiunile necorelate cauzal
- versiunile divergente sunt reconciliate
- versiunea rezultata din reconciliere inlocuieste versiunile curente in depozit

Tratarea defectelor tranzitorii

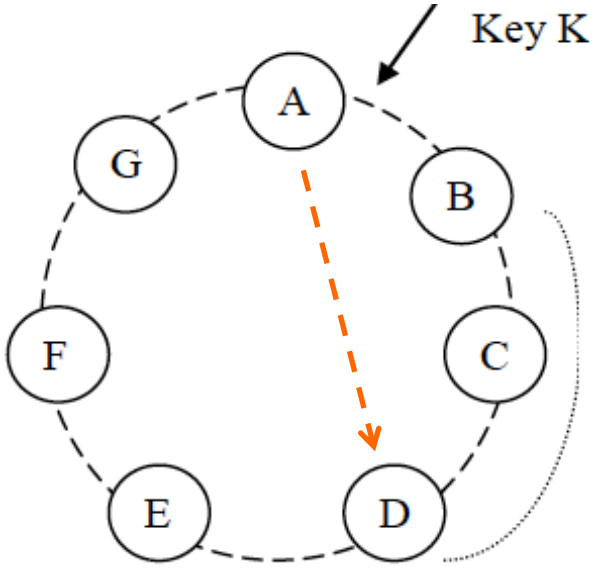
Abordarea cvorum nu functioneaza la defectarea nodurilor sau la partitionarea retelei

Solutia: model cvorum “relaxat”

- operatiile se executa pe primele N noduri “sanatoase” din lista de preferinte
- pentru N=3 si nod A cazut temporar, o replica destinata lui A este trimisa lui D
- D asociaza un hint – “locul replicii este A”
- pastreaza replica intr-o BD separata
- cand A revine, D ii transmite replica

Pentru defecte la nivel data center

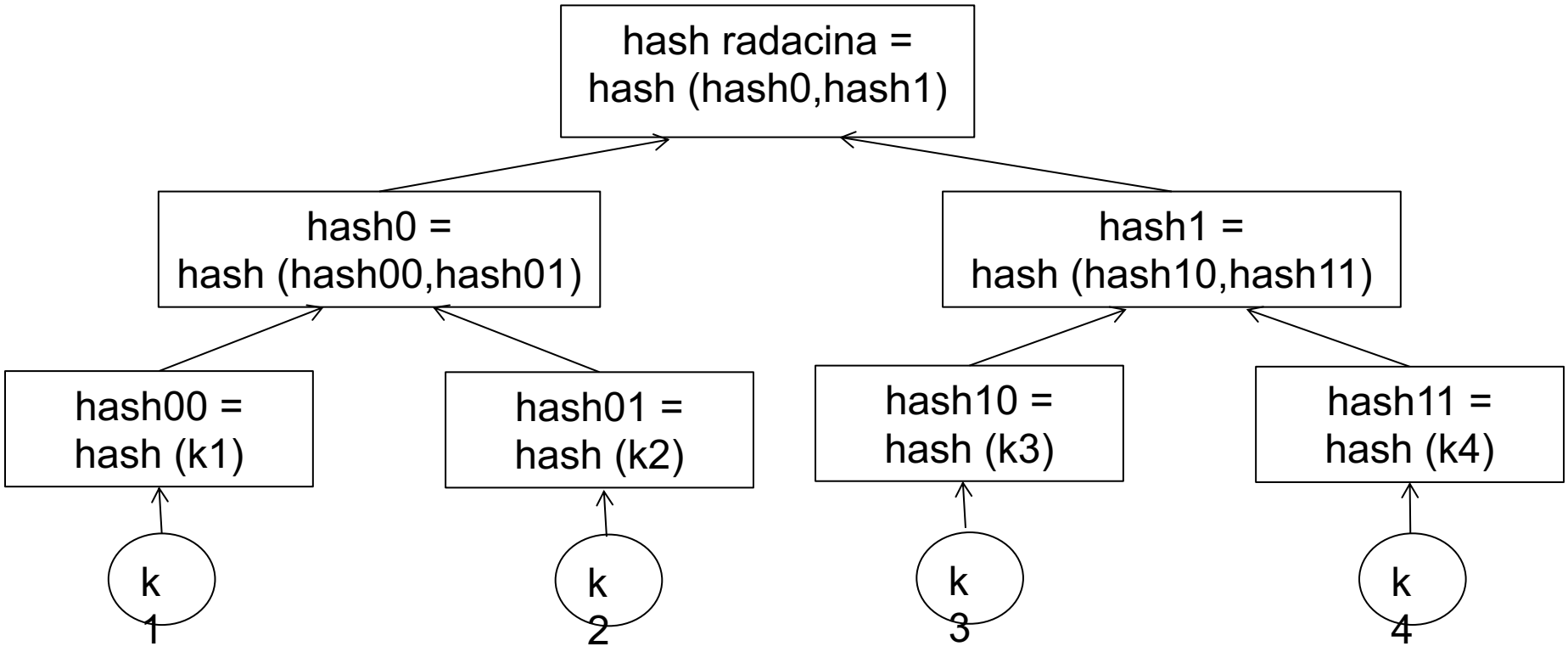
- lista de preferinte include noduri din diferite data centere
- data centere sunt conectate prin legaturi de mare capacitate



Tratarea defectarilor permanente

Protocol de sincronizare descentralizata a replicilor bazat pe Merkle trees

- frunzele sunt hash-uri de chei
- celelalte noduri sunt hash-uri ale fiilor



Algoritm anti-entropie pentru sincronizarea replicilor

Fiecare nod pastreaza un arbore Merkle separat pentru fiecare set de chei acoperite de un nod virtual

1. doua noduri schimba intre ele hash-urile din radacina arborilor Merkle corespunzatoare unui set de chei comune
2. daca nu sunt diferite → STOP (nu au nevoie de sincronizare)
3. daca sunt diferite → listele hash de la nivelul inferior sunt schimbate intre noduri
4. pentru fiecare pereche de hash-uri din cele doua liste ale nodurilor se repeta operatiile 2 si 3
5. operatiile continua pana se ajunge la frunzele arborelui Merkle, cand se identifica cheile desincronizate