Toleranța la defectări

Concepte generale
Rezilienta proceselor
Multicast fiabil
Recuperarea

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

Concepte de Baza

Increderea (Dependability)

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

Atribute ale sistemelor de incredere

- Disponibilitatea (Availability)
- Fiabilitatea (Reliability)
- Siguranta (Safety)
- Mentenabilitatea (Maintainability)

Terminologie

O *specificatie* descrie comportarea ideala a unui *sistem* la *interfetele* sale.

Esec, incapacitate (Failure): **deviere** de la specificatia interfetei

Eroare (Error): stare interna care poate conduce la esec; nu este vizibila la interfata

Defect (Fault): **cauza** unei erori

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

sistem ventilatie:

praf = cauza; reducere turatie = eroare; mai putin aer = esec

sursa de alimentare

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

Tehnici de tratare a defectelor

Prevenire

- proiectare formala, controlul calitatii
- injectare defecte si testare.

Detectie

Recuperare

Prezicere

Tolerare: furnizarea serviciului chiar daca apar defecte (**mascarea** defectelor)

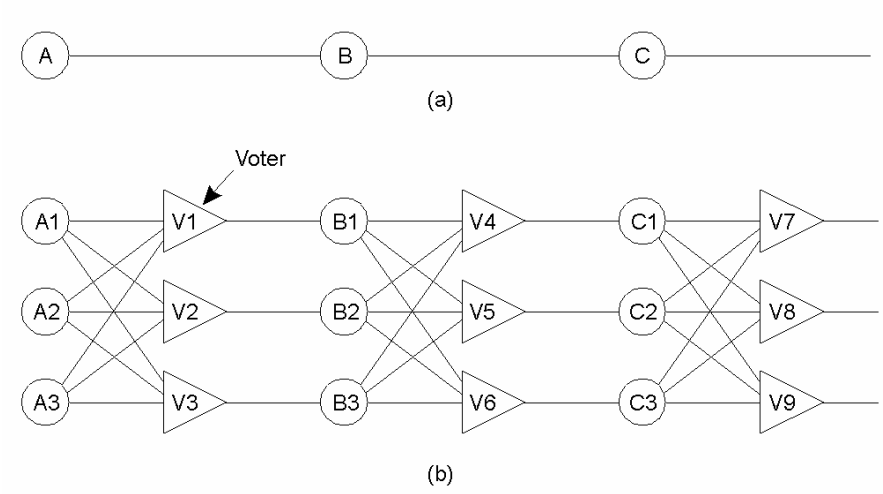
Tipuri de defecte si esecuri

Defecte

- **tranzitorii**
- **intermitente**
- **permanente**

Tip esec	Descriere comportare server
Cadere (Crash)	Un server se opreste dintr-o data
Omisiune *receptie* *transmisie*	Nu raspunde cererilor Nu primeste mesaje Nu trimite mesaje
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

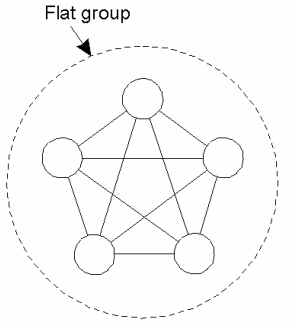
Mascarea esecurilor prin redundanta

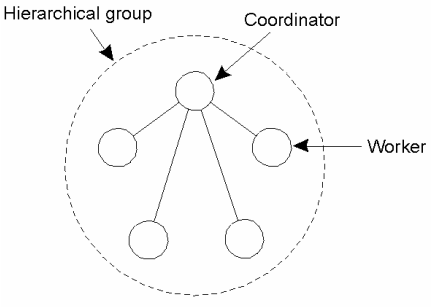


Redundanta modulara tripla.

Rezilienta proceselor – grupuri de procese

Problema: Protectia contra proceselor defecte prin replicare si distribuirea calculelor intr-un grup.





Grup "plat".
- Nu exista punct de esec unic
- Luarea deciziilor este costisitoare

Grup ierarhizat
- Un singur punct de esec
- Decizii centralizate

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)

Comunicare de grup fiabila

Multicast fiabil: un mesaj trimis unui grup de procese trebuie livrat fiecarui membru al grupului

Probleme

- grup instabil
  - un proces se alatura grupului in timpul comunicarii
  - un proces "crapa" in timpul comunicarii

Simplificare

- grupul este stabil (procesele nu "crapa" si nu se alatura grupului in timpul comunicarii)
- consideram doar defecte de comunicare

Aplicatii si protocoale multicast

White-Board distribuit - **Scalable Reliable Multicast** (SRM)

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)

Conferinte (cu IP multicast)

Jocuri in retea

Evenimente multimedia

Etc.

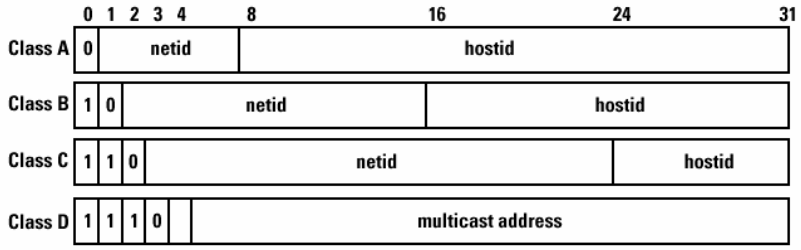
Multicast IP

Foloseste adrese de clasa D pentru transmitere pachete

Gestiune grupuri - Internet Group Management Protocol (IGMP)

Rutere multicast

- **transmit pachete multicast membrilor (best effort)**
- **verifica periodic host-urile ramase in grup (pool – IGMP1)**
- **sunt avertizate cand un host intra in grup**

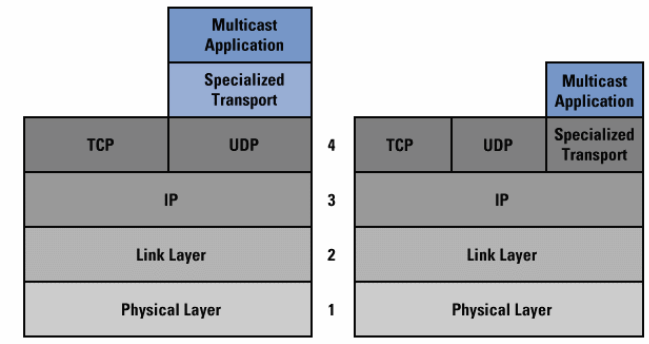


Transport - Multicast peste UDP sau IP

Solutia TCP (corectia la transmitator) nu merge:

- Multe ACK - gatuire
- Evidenta greoaie a setului de receptori
- Conditii diferite la receptori diferiti (round-trip time, delay*bandwidth, legaturi supraincarcate diferite)

Alta solutie: un Transport specific peste UDP sau IP



Scalable Reliable Multicast - SRM

Bazat pe IP multicast (ne-fiabil) care ofera:

- Sursa transmite pe o adresa multicast fara sa cunoasca membrii grupului
- Un receptor intra in- sau iese din grup fara a afecta ceilalti membri

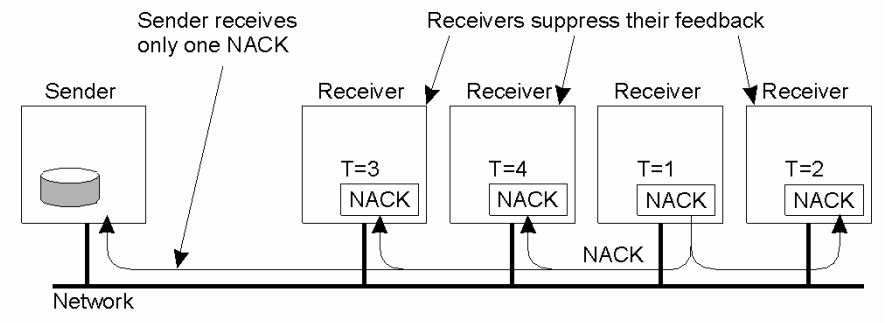
Adauga caracteristici TCP

- Fiabilitate capat la capat
- Adaptare la conditiile retelei (dimensiune, trafic, topologie)

Ideea

- Fiecare membru al grupului raspunde de receptia corecta a mesajelor care ii sunt adresate (corectia la receptor)

SRM – controlul cererilor



Trei tipuri de mesaje

- *Heartbeat* – permit detectia pachetelor absente (nr. Secv)
- *NACK* – cerere retransmitere
- *Repair* – retransmitere din cache

Optimizari

Principiu: mesajele sunt trimise multicast grupului

Minimizare numar *NAK* si *Repair*

- Transmitere cu o intarziere aleatoare
- Retragere propria transmitere (eventual)
- Problema: planificarea intarzierilor pentru a asigura un singur mesaj

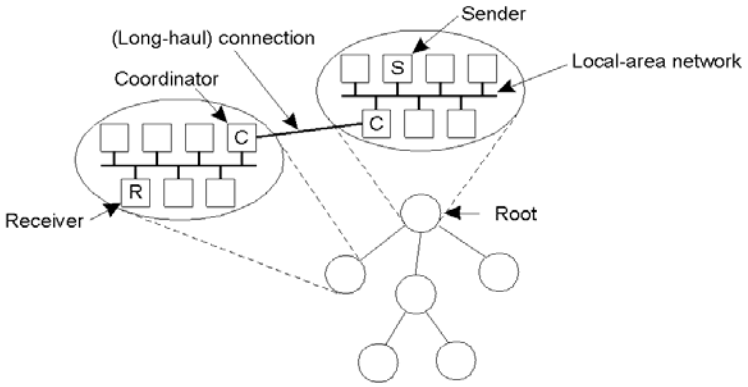
Evitare consum resurse pentru cei care au primit corect

- Retransmitere punct la punct
- Retransmitere multicast doar pentru gruparea de procese care au pierdut acelasi mesaj

Accelerare proces recuperare

- Retransmitere facuta de un membru "local"

Multicast fiabil ierarhic



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

a) retransmit mesajele catre fii.

b) manipuleaza cererile de retransmitere.

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

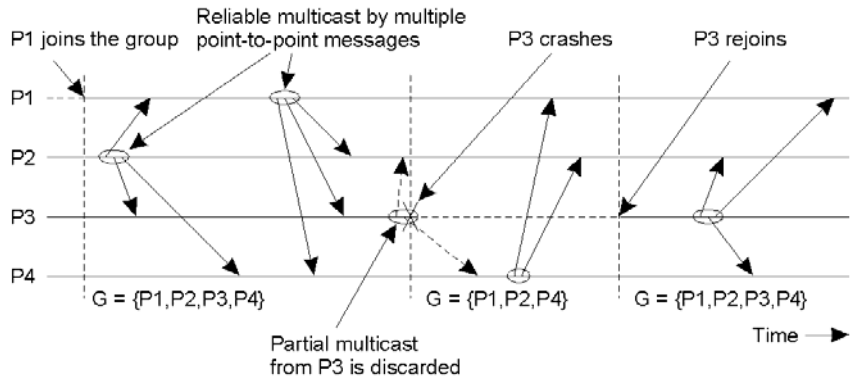
Problema multicast atomic

- 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

Bazat pe notiunea de group view

- setul de procese din grup "vazut" de transmitator cand a trimis mesajul

Virtual Synchrony



Toate procesele din set au acelasi view

Un group view se poate schimba

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

Virtual Synchrony (2)

Anunt schimbare group view - prin transmiterea multicast a unui vc – view change

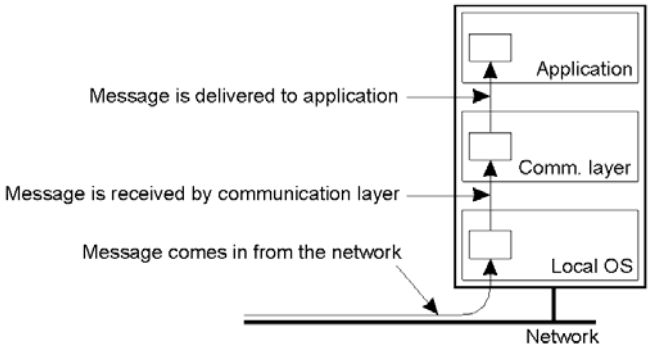
Principiul **multicast-ului sincron virtual** (G = group view):

- 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)

Ipoteze

- utilizeaza comunicarea fiabila punct-la-punct (TCP)
- multicast – trimitere mesaj **m** fiecarui membru din grup
- mesajele de la aceeasi sursa sunt primite de nivelul de comunicare in ordinea trimiterii
- arhitectura sistemului (diferenta intre receptie mesaj si livrare mesaj)



Implementare Virtual Synchrony (2)

Cerinta

- un mesaj trimis unui group view G trebuie livrat tuturor proceselor din G inainte de urmatoarea schimbare de view

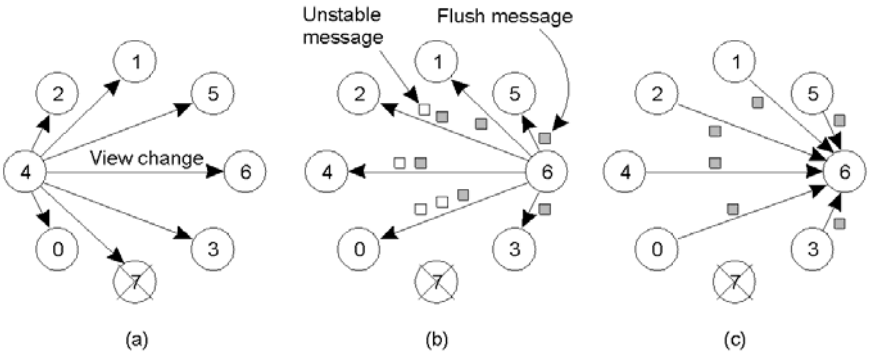
Probleme

- cv 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

- mesaje **stabile** (primite de toate procesele): sunt livrate
- mesaje **instabile**: pastrate de procesele care l-au primit; unul din ele il va trimite tuturor celorlalte procese

Implementare Virtual Synchrony (3)



a) Procesul 4 observa ca 7 a cazut si trimite un mesaj view change

b) Procesul 6 primeste view change si trimite toate mesajele instabile (pe care le schimba in stabile), urmate de un mesaj flush (nu mai are mesaje instabile)
Celelalte procese procedeaza la fel

c) Procesul 6 instaleaza noul view cand primieste un flush de la toti ceilalti (fiecare proces valid procedeaza ca 6)

Generalizare – distributed commit

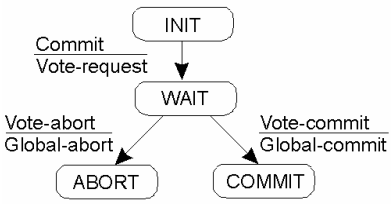
Definitie: data fiind o operatie distribuita proceselor unui grup, se asigura ca

- sau fiecare proces din grup executa operatia
- sau nici un proces nu o executa

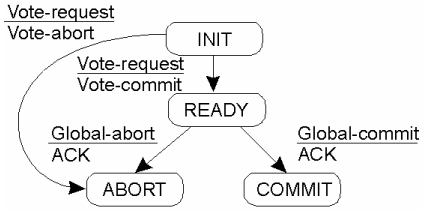
Solutie: folosirea unui coordonator

- 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 pentru **coordonator**



Masina de stari pentru un **participant**

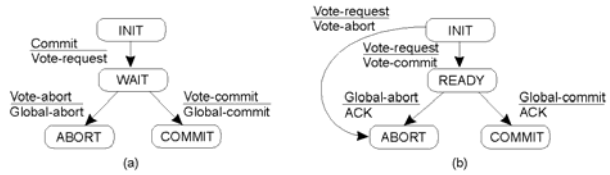
Coordonator este procesul care initiaza operatia

Exista stari in care coord (WAIT) sau participant (INIT, READY) asteapta

Pentru evitarea blocarilor se folosesc timeout-uri

Comportare la timeout in INIT (participant) si in WAIT (coordonator)

Comportare la timeout in READY



Participant *P* in starea *READY* contacteaza 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
READY	Contacteaza alt participant	Daca toti READY atunci asteapta recuperare coordonator

Implementare: logare stare ptr refacere la recuperare

Log 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;
}
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;
}
```

Operatii participant pentru tratarea cererilor de decizie de la alti participanti.

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

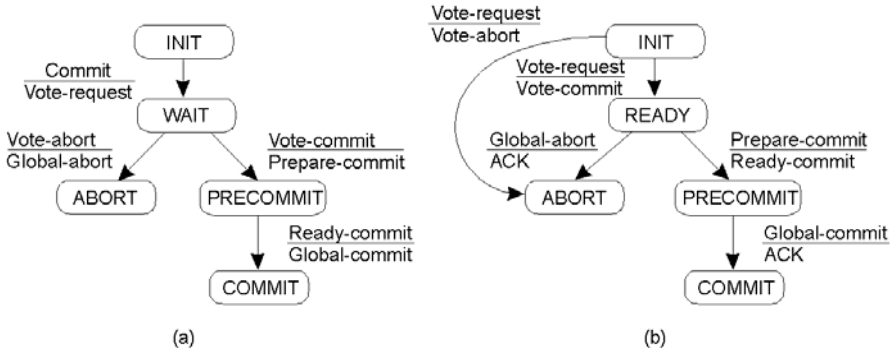
```
while true {
    wait until any incoming DECISION_REQUEST is received; /* remain blocked */
    read most recently recorded STATE from the local log;
    if STATE == GLOBAL_COMMIT
        send GLOBAL_COMMIT to requesting participant;
    else if STATE == INIT or STATE == GLOBAL_ABORT
        send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

Participantii nu pot decide cand:

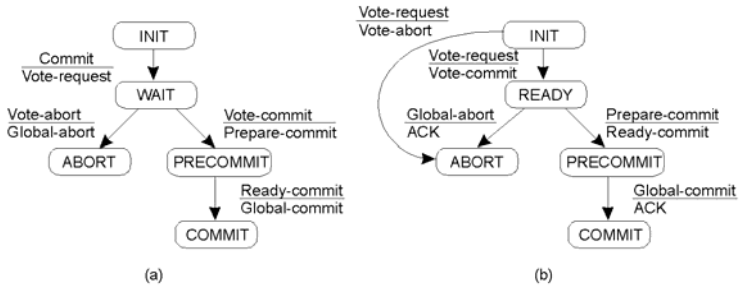
- toti participantii au primit si procesat VOTE_REQUEST
- intre timp coordonatorul a crapat

Corespunde lui **skip** din actiunile participantilor

Three-Phase Commit



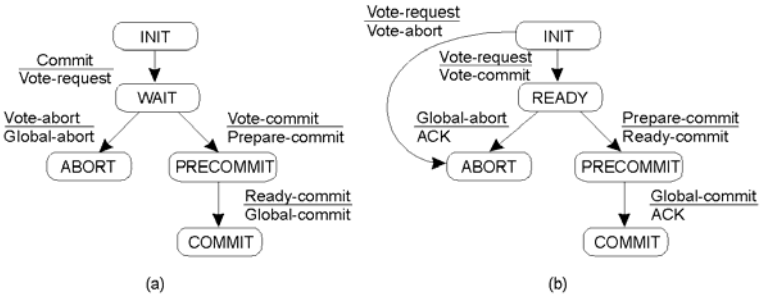
a) Masina de stari pentru coordonator in 3PC
b) Masina de stari pentru un participant



Actiuni la **timeout**

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

Coordonator in WAIT –> Global-abort (un participant a cazut)
Coordonator in PRECOMMIT –> Global-commit (un participant a cazut dar el a votat pentru comitere si va recupera)
Participant in INIT –> Vote-abort
Participant P in READY sau PRECOMMIT -> contacteaza alti participanti Q (coord a cazut)
un Q in COMMIT sau ABORT -> P trece in starea respectiva
un Q in INIT (obligatoriu coord si nici un participant nu sunt in PRECOMMIT) -> P aborteaza
toti in PRECOMMIT -> P trece in COMMIT



Daca doar unii participanti pot fi contactati

toti ce pot fi contactati sunt in PRECOMMIT -> P trece in COMMIT (un proces cazut va recupera in READY, PRECOMMIT sau COMMIT – oricum, va ajunge in final la COMMIT)

toti ce pot fi contactati sunt in READY -> P aborteaza (un participant a cazut si va recupera in INIT, ABORT sau PRECOMMIT; in toate cazurile el va ajunge sa aborteze)

Concluzie: procesele supravietuitoare pot lua intotdeauna o decizie

Recuperarea

Sistemul este adus intr-o stare corecta

Tipuri

- inpoi (backward) – aduce sistemul intr-o stare anterioara
- inainte (forward) – gaseste o stare noua

Recuperare inapoi

- mai folosita
- mai complicata in sist distribuite (procesele trebuie sa identifice o stare consistenta)
- tehnici
  - checkpointing
  - message logging (folosita in combinatie cu checkpointing)

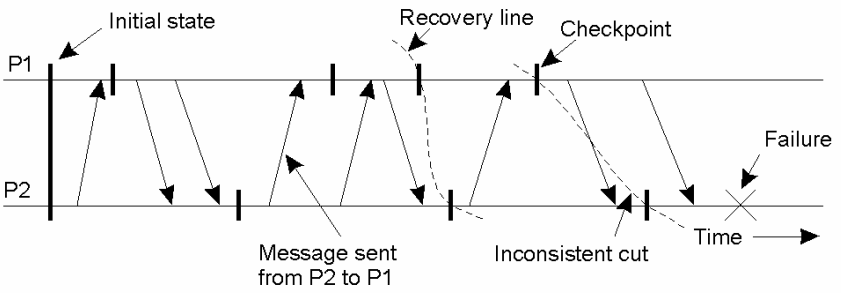
Checkpointing

Stare globala consistenta = instantaneu distribuit

- constituita din starile proceselor salvate in **memorii stabile** locale
- daca P1 a inregistrat o receptie de mesaj starea trebuie sa includa o transmitere la un alt proces P2

Linie de recuperare = cel mai recent instantaneu distribuit

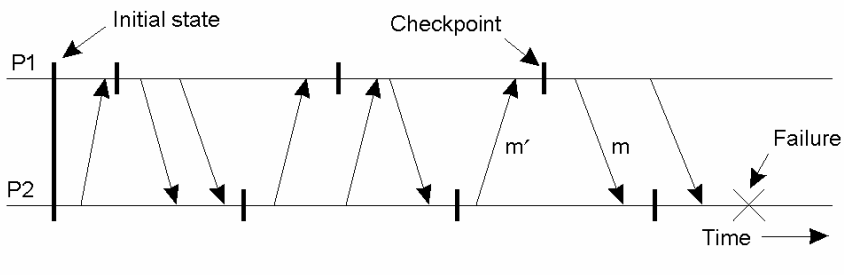
(cea mai recenta taietura consistenta)



Checkpointing Independent

Linie de recuperare greu de gasit:

- Efectul de domino



- Solutia: coordonare globala a checkpointing

Implementare

Notatii:

CP[i](m) denota checkpoint *m* al procesului *Pi*

INT[i](m) interval intre *CP[i](m-1) si CP[i](m)*.

Ideea:

Inregistreaza dependenta intervalelor

Algoritm:

Pi trimite mesaj in *INT[i](m)* si adauga (i,m) ptr receptor

Pj primeste mesaj in *INT[j](n)*

inregistreaza dependenta *INT[i](m) -> INT[j](n)*

la checkpoint CP[j](n): salveaza (*INT[i](m) -> INT[j](n)*)

in memoria stabila

Daca *Pi* revine la *CP[i](m-1)* => *Pj* revine la CP[j](n-1)

OBS: Daca starile nu sunt consistente sunt necesare alte reveniri

Checkpointing Coordonat

Solutie simpla: protocol cu blocare in doua faze:

Coordonator:	trimite multicast cerere *checkpoint*
Participant:	primeste o cerere *checkpoint* face checkpoint opreste trimiterea mesajelor raporteaza ca a facut checkpoint
Coordonator:	asteapta toate checkpoints confirmate difuzeaza *checkpoint done*
Participant:	primeste *checkpoint done* continua

Logarea Mesajelor

Starea consistenta globala poate fi atinsa si prin "repetarea" mesajelor transmise

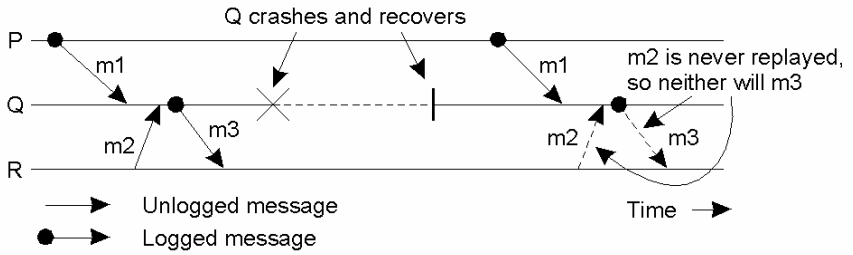
Ideea: model de executie determinist pe bucati

- executie = secventa de intervale
- fiecare eveniment nedeterminist incepe un nou interval (ex. Receptia unui mesaj)
- in interval executia este determinista

Concluzie: inregistram evenimentele nedeterministe

Cand trebuie inregistrate mesajele?

- 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
- Ex. Q a primit si livrat m1 si m2; 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
- m2 (nelogat) nu este repetat ➔ m3 nu este transmis ➔ stari inconsistente pentru Q si R ➔ R devine orfan.



Scheme de logging

HDR[m]: antet mesaj *m* = sursa, destinatia, numar secventa, numar livrare
- Antetul contine toate info necesare retransmiterii mesajului si livrarii sale in ordinea corecta

Mesaj *m* este **stabil** cand nu mai poate fi pierdut (a fost pus in memoria stabila).
- pot fi "rejucate" pentru pentru recuperare.

DEP[m]: include procesele la care s-a livrat *m* si procesele 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).

C colectia proceselor cazute

Q este **orfan** daca
depinde de un **m** (exista *m* a.i. Q este in DEP[m]) si
m nu poate fi retransmis – procesele care au o copie a lui *m* au cazut (COPY[*m*] este inclus in C).

Scheme de logging (2)

Scop: evitare orfani
- Daca procesele din COPY(m) au cazut, nu ramane nici un proces in DEP(m)

Se poate asigura daca orice proces dependent de *m* pastreaza o copie al lui *m*:
- pentru fiecare *m*, DEP[m] este inclus in COPY[m]

Solutii

Protocol pesimist: pentru fiecare mesaj *nonstabil 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 instabil *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 Q este intors la o stare in care Q nu este in DEP[m].
- complicat de implementat