Master 2 Ubinet

# Distributed Algorithmic


*Françoise Baude*

baude@unice.fr


web site : on the moodle, key M2_19_20
https://lms.univ-cotedazur.fr/course/view.php?id=3535

Chapter : Global State collection - Distributed Transactions



# Course plan,in 2 distinct parts

1. Global state collection
   - Motivation
   - Termination Detection
     - Message counting
     - Active/Passive process states
   - Deadlock Detection
     - Resource deadlock
     - Communication deadlock
2. Distributed Transactions
   - Motivation
   - Atomic distributed commit protocols

## 1. Global state collection

- Why: Detect particular global states
  - Termination: useful to enter next phases of applis.
  - Deadlock: useful to repair the deadlock !

  Or doing special global computations as counting
  - the total number of messages exchanged
  - the total number of processes at a given moment
- Problem: how to collect such states, out of non synchronized/non instantaneous collection of process local state or channel ?
  - Same sort of problem than solving the general purpose distributed consistent snapshot
  - But we need to merge/present the various collected pieces in a way that suits what global state we want to detect/collect

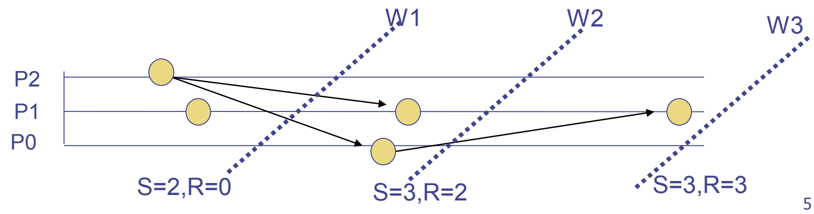


## Termination detection

- 2 different possible formulations
  - All messages sent during the application have been received and treated = no message in transit
    - Just a matter of counting this total number of messages sent/received …
  - Each process involved in the application is passive & no msg in transit
    - When passive, no way to become spontaneously active, except if a message is in transit and will be received later
    - But, if every process is passive, none will magically create a new message
    - Just a matter of being able to collect status of each process in a consistent manner, and able to detect if any msg still in transit
    - E.g.: P1 sent a message towards P2 and became passive
      The observer sees “P2 is passive”, afterwards sees “P1 is passive”. But, in the meantime, the msg reaches P2 which becomes again active

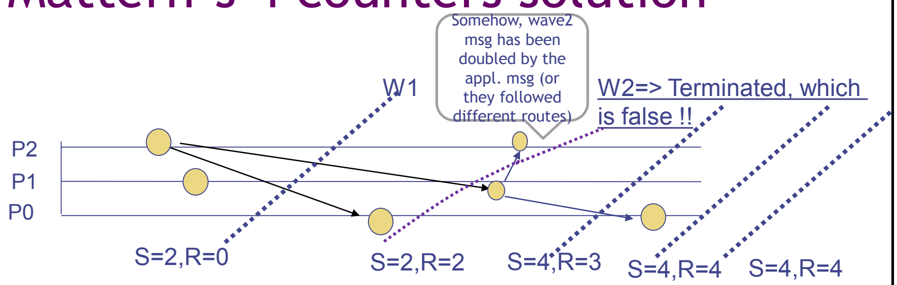
# Counting total number of messages

- Basis of the 4 counters algorithm from F. Mattern: When all messages sent (S) have been received (R) and treated, i.e. S = R, means no message in transit
- Start a collection *wave* from one process (e.g. the leader) e.g. along a ring, after a given timeout.
  - If S=R, proclaim that termination has been detected
  - Otherwise, repeat after the timeout
- OK only if wave corresponds to a consistent cut…






# Mattern’s 4 counters solution




- To avoid this “false termination” detection:
  - Either, forbid to construct non consistent cuts
    - Chandy Lamport algo. with FIFO or extended to NonFiFO channels
  - or, specific Mattern’ solution, easy and efficient to implement: 2 successive waves
    - Thm: **(S1=R1) == (S2=R2), iff it is terminated**
    - These are the 4 counters !
    - See proof on the web site of the course

# Detecting passive states: general principles

- As for Mattern', detection done in successive waves **(wave algorithm)**
- A Control msg visits each process in turn
  - Because no other way to observe the global status!
  - Is treated only once no more applicative msg pending, i.e. when the process has become passive
    - process was still passive since last visit => aggregate "passive" to the global information transported by the control msg, and forward it to the next process
    - If not, aggregate the "active" information, forward it to the next process
  - On initiator: initiate a new wave if control msg="active", otherwise, proclaim termination
  - How to ensure "no msg in transit" ? -> different algos



# Ring-based application topology

- Restriction about routing path used by application msgs
- The control msg also transmitted along this unidirectional ring
  - A way to ensure that the control msg "empties" the comm. channels ! OK only if channels are FIFO
- Misra algorithm based on a "counting" passive or active processes token:
  - initially each of the n process state is white, Initiator is process 0
  - On msg reception on any Process: state=black;
  - (On each Pi, i in [1..n-1]) On token reception:
    - if state= black token:=1 else token:=token+1
    - state = white; forward token
  - (On P0) On token reception:
    - if (state=white & token=n) "terminated", else state=white; forward token=1

## Ring-based topology for control only

- Application messages can use any topology ;-)
- Token transmitted along unidirectional ring
- Risk: token=n, but appl. msg still in transit …
- Sol:
  - Synchronous communications… -> no in-transit msg
  - Ring can be built by connecting all processes according to the ID's ascendant order
    - Appl. Msg sent forward or backward w.r.t ring
    - Whenever the control msg has already detected a passive process, we still need an additional mechanism: to claim that an appl. msg for this process has possibly reactivated it



## Dijkstra-Feijen-van Gasteren algorithm

- Ring is 0->n-1->n-2->… -> 1 ->0
- Communications are synchronous (or bounded with D)
  - Messages are never in transit !!
- R1: When a non-initiator process sends a msg to a higher numbered process (=backwards in the ring), it turns black
  - Means that Pi reactivates some process possibly already visited by the token
- Token only treated when no app. msg in receive queue
- R2: When a black process sends a token, the token turns black. If a white process forwards a token, then it retains (keeps the color of the token as it)
- R3: When a black process sends a token to its successor, the process turns white
- Initiator treats a white token and itself is white => terminated, otherwise, new white token is created

## Resource-based Deadlock

- Resource based Deadlock:
  - Non shareable, i.e. used in mutual exclusion
  - Each process requires at least 2 resources at once
  - No preemption: a process owns a resource, and only it can relinquish it. A process "waits" for an other until this one relinquishes the requested resource
  - A set of processes is deadlocked whenever there is a directed cycle in the associated "waiting for resource" graph
- Detection: exhibit the <u>distributed</u> graph
- The graph corresponds to a snapshot of the global state of the application, but it does not mandatorily contain <u>all</u> processes



## Wait-for Graph (WFG)

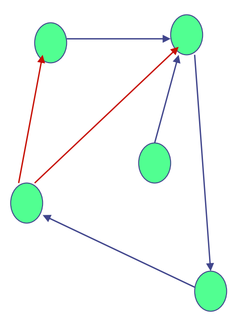
- Represents who waits for whom.
- No single process can see the WFG.
- **Resource deadlock**

  [R1 AND R2 AND R3 …]

  also known as AND deadlock, because a process can not progress until it has acquired ALL resources it waits for
- **Communication deadlock**

  [R1 OR R2 OR R3 …]

  also known as OR deadlock
- Eg: [R1 OR (R2 AND R3)]: ReceiveM1, or (Receive both M2 and M3)



# Detection of resource deadlock [Chandy-Misra-Haas]

**Notations**

**w(j) = true** ≡ (j is waiting)

**depend [j,i] = true** ⇒

$j \in succ^n(i)$ $(n>0)$

"Pi is blocked directly or indirectly due to the fact that Pj is also blocked"

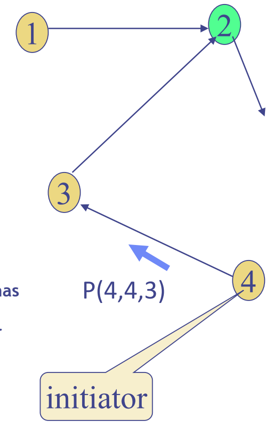
**P(i,s,r) is a probe**

**(i=initiator, s= sender, r=receiver)**

- **i=initiator : after a given timeout, Pi tries to figure out why it has not progressed**
- **r=receiver and s=sender: Ps blocked because it is waiting for Pr**

**Idea**

- P(i, x , i) back to Pi : Pi is member of a circuit (oriented cycle) in the WFG, so it is deadlocked, because Pi is blocked due to the fact that Pi is also blocked !






# Detection of resource deadlock

{Program for process k}

- P(i,s,k) received ∧

  $w[k] \wedge (k \neq i) \wedge \neg$ depend[k, i] →

  send P(i,k,j) to each successor j;

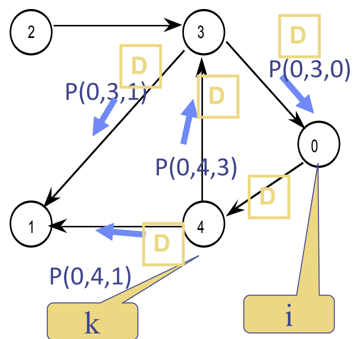
  depend[k, i]:= true

  //Pi is blocked due to me (Pk) also blocked

- P(i,s, k) received ∧ w[k] ∧ (k = i) →

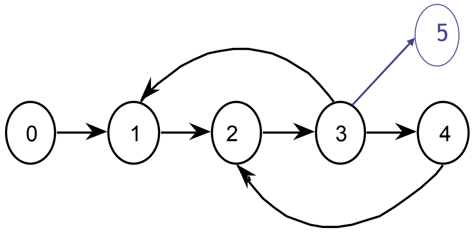
  process k is deadlocked

**(edge-chasing algorithm)**




- **The algorithm can be triggered by each process in a waiting situation in //**
  - **E.g if we continue the simulation with i=3, P3 will not detect deadlock, but as it is in the same circuit as P0, the deadlock will eventually be repaired**
- **To detect deadlock, the initiator must be in a circuit**
- **No-deadlocked situation is not proclaimed, but resource release will happen**

# Communication deadlock



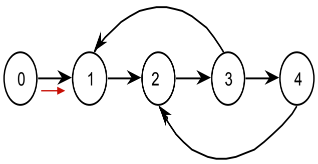

- This WFG has a resource deadlock:
  - 1,2,3,4 all belong to oriented cycles
- but it has no communication deadlock
  - 3 is waiting a message from either 1,5,4
    - As 5 is not deadlocked, it will eventually unblock 3
    - Then 3 can send a msg to e.g. 2, which will send a msg to 1, etc… until 1 gets unblocked, and so send a msg to 0 which unblocks 0
- If 5 were not part of the WFG = the WFG would contain an OR deadlock.
  - 0 can know it is OR-deadlocked
  - Rem : in this WFG, 0 is blocked, but not deadlocked => it is not itself part of a circuit (no risk that 0 gets killed to repair the deadlock !)



# Detection of communication deadlock [Chandy-Misra-Haas]

A process ignores a probe, if it is not waiting for any process. Otherwise,

**(probe-echo algorithm)**




- *first* probe →
  mark the sender as *parent*;
  forwards the probe to successors
- Not the first probe →
  send ack to that sender
- ack received from every successor →
  send ack to the parent

Communication deadlock is detected

if the initiator receives ack from all its successors (implying none of its successors can unblock it =>it is deadlocked)

On the example, 0 will detect that it is OR deadlocked (whereas from a resource-deadlock viewpoint, 0 is not in a cycle, so not "AND-deadlocked", but can not however make progress !)

=> This algorithm is thus more general

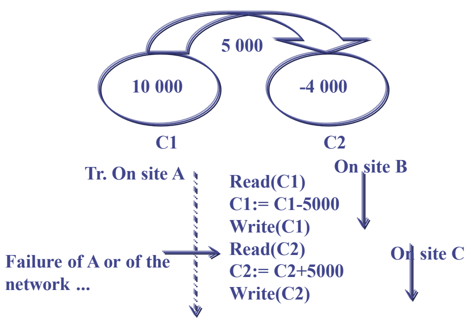
*Has many similarities with Dijkstra-Scholten's termination detection algorithmn, also of a probe-echo type*

# 2. Distributed Transactions

- Why: Very important pattern !
- Goal: ensure ACID properties, including in the case where the transaction executes in a distributed manner
- ACID:
  - Atomicity (all or nothing)
  - Consistency (w.r.t. integrity rules)
  - Isolation (no side effects in case transactions concurrently access same objects)
  - Durability (effects of any transaction is permanent)



# Example and objectives




Result : C1 = 5 000 et C2 = -4 000

Lost money !

<u>Transaction</u>: delimits a sequence of instructions
=> To be run atomically (either all, or none)
=> Delimited by pseudo-instructions (or API)

<u>Distributed Transaction</u>: e.g. C1 et C2 are objects located on different machines. The transaction must still execute atomically and ensure ACID properties
=> Sites must cooperate in a distributed manner
= COMMIT PROTOCOL

<u>Concurrent Transactions</u>: when an other transaction concurrently runs, and also accesses in read or write mode C1 and/or C2
For performance purposes, still enable that the two transactions execute in parallel
= CONCURRENCY CONTROL
=> Ensure that the result (last values of C1 and C2) is the same had the two transactions run <u>serially</u>.
Can require to abort some already started transactions in case one aborts

## Transactions Concurrency control

- Pertains to isolation property
- As if each transaction is run serially, but in //:
  - A serial schedule of the set of transac. executions
- Serializability: strongest consistency property
- Pessimistic concurrency control: can block
  - Lock (or time-sort) data access in potential conflict: R/W mode
  - Detect if cycle (WFG) & Break cycle by aborting one transaction (before it reaches its normal end)
- Optimistic concurrency control: non blocking runs
  - Just remember which data is accessed in R/W mode
  - Build a precedence graph & detect cycle when a Tr reaches its normal end, to decide of a Tr to abort

19

## (Non) serialisable transactions runs

| T1 | T2 |
|---|---|
| R(A) | |
| | R(B) |
| W(A) | |
| | W(B) |
| | Commit |
| R(B) | |
| W(B) | |
| Commit | |

- This run is serializable:
  - Equivalent to T2;T1 serial schedule

| T1 | T2 |
|---|---|
| R(A) | |
| W(A) | |
| | R(A) |
| | W(A) |
| | R(B) |
| | W(B) |
| | Commit |
| R(B) | |
| W(B) | |
| Commit | |

- Dirty reads:
  - eg if T2 is allowed to R(A), what if later T1 has to abort?
- This run of T1&T2 non serializable

20

## Distributed transactions: Commit protocol needed

- Coordinator for any transaction
  - The site the client contacted first to begin the transaction
- Participants
  - All sites that the transaction has accessed, and on which it has accessed some objects
- Risk of failure (sites or networks) must be accounted
- Abortion may be decided locally to fulfill serializability
- On reaching the *end-transaction* pseudo-instruction (means the client wants to validate the transaction)
  - Rem: In case transaction abortion has been triggered earlier by the client, the coordinator has already cancelled all sub-parts of the transaction;
  - The coordinator must initiate a commit protocol
  - End result of the commit protocol: either <u>commit all</u> sub-parts of the transaction, or <u>abort all</u> (Important: all still-alive must execute the decision that has been agreed together)



## Naïve one phase commit protocol

- The coordinator loops, requesting each participant to commit, until all acknowledge they have done it.
- But, the decision to abort or commit may be not uniform on all involved servers
  - A server may abort a transaction due to concurrency control, or it may have crashed and has been restored using its checkpointed state
    - Even if the sub-transaction is restarted later, it can run / leave the database in an inconsistent state

# Two-phase commit protocol [Gray]

Coordinator (transaction manager)
On ‘ EndTransaction ’ from client =>

Participant(s)

Phase 1

- Write « Preparation » in its log
- Send « Preparation » msg to all
- Collect all votes

All replies

Not all replies: *after timeout*, decide « Abort »

Preparation commit

... Once transaction locally terminated, wait for this msg (abort if *after a timeout*, no such msg arrives)
- Write « Ready Y/N» in the log (at the end of logs of actions done by the transaction, in stable memory) then Send the message « Ready Y or N» (send “N” if decided to abort if abnormal transaction termination or conc control’s abort)

... This msg is not sent if participant failed

Ready to commit
Yes or No

Undecided state

Phase 2

- Write result of the vote on the log
- Send voted msg « Commit » or « Abort » to all
- Collect all replies to reach protocol end (release all transaction information)

Uniform& global decision

Write « Commit » or « Abort » in the log

... Commit... or ... abort... (use R,W ops log)
Send msg « Done »

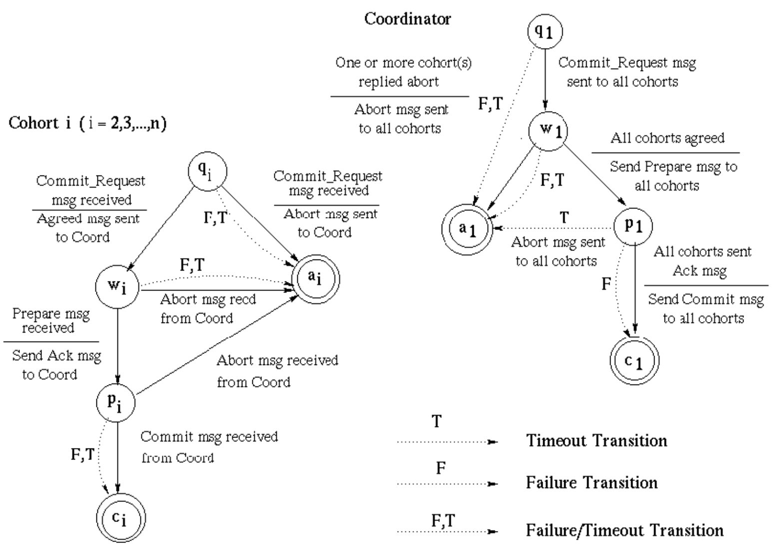
Done!

Rem: if any participant fails during the protocol, the log enables to restart it at the right same point.
A protocol is said non-blocking if the failure of any participant does not avoid the others to make a decision
There exists an undecided state if the coordinator fails just between the 2 phases
⇒2PC is blocking while the coordinator has not been restarted :=(
⇒See http://en.wikipedia.org/wiki/Three-phase_commit_protocol for a non-blocking extension

23

# Three-phases commit [Skeen]

- http://ei.cs.vt.edu/~cs5204/sp99/distributedDBMS/sreenu/3pc.html




24

# Commit protocols seek properties

- Agreement: All participants must agree to the same decision
  - here if any participant wants to abort, consensus will be value 'abort'; On the contrary, for the general consensus problem, the decision could be any replica' value; and must loop for each replica value to reach successive total ordered consensus
- Termination: All non-faulty servers must eventually reach an irrevocable decision
- Validity: if all servers vote commit and there is no failure, then all servers must commit
- Commit protocols are solutions to reach consensus in asynchronous with failures (crash&comm.) systems
  - But…Consensus in asynchronous systems without taking real actions to face failure is known to be unsolvable ! (see FLP theorem)
  - =>Like for the general consensus pbm, commit protocols thus include *fault suspicion* & *handling*. Suspected failed processes are always acting according to reached agreement thanks to permanent storage stored information used at recovery time