Consensus

DIDA: Class 02

Consensus

- Set of N processess
- Each process proposes an input value
- All correct processes output the same value
- The output value must be one of the values proposed
  - This prevents a trivial solution, where all processes always output some default value
- The output may be any of the proposed values
  - It does not have to be the most frequent value
  - All input values are equally good

Properties of consensus

- Termination: all correct processes eventually terminate
- Uniform agreement: if two porcesses decide, they decide the same value
- Integrity: the vallue decided has been proposed by a correct process

Why is this hard?

- Assume that you have a uniform reliable broadcast (URB) primitive with the following interface and properties:
  - Broadcast (*m*) to send a message
  - Deliver (*m*) tp deliver the message to the application

  - Validity: if a correct process broadcast *m* every correct process delivers *m*
  - No duplication: no message is delivered more than once
  - No creation: if a process delivers *m*, *m* was broadcsat by some process
  - Agreement: if a process delivers *m* every correct process delivers *m*

Now assume the following algorithm

- Bogus consensus

```
When propose (value)
        URB.broadcast (value)
When URB.deliver (v_i)
        setOfValues = setOfValues + v_i
When |setOfValues| == N
        decide (MIN(SetOfValues));
```

Problem with the algorithm above

- If a process fails (and never broadcast its value) all process will block

(still bogus) with a perfect failure detection

Alive = {p1, p2, p3, …}

When fail (px)

    alive = alive – px

Replace

    When |setOfValues| == N

By

    When |setOfValues| == |alive|

Problem with the algorithm above

- Say that the process that proposes the minimum value (*valueMin*) fails while URB is still executing
  - One process may receive *valueMin* before it detects the failure
  - Another process my detect the failure before it receives *valueMin*
  - These processes will output a different value

The problem above is captured by the TRB abstraction

- Terminating reliable broadcast
  - There is a special process that is the sender *s*:
    - Sender initiates the algorithm by issuing *send (m)*
    - Other nodes initiate the algorithm by issuing *wait()*
  - All process eventually execute *output (value)*
    - Where *value = m* if the sender is correct
    - Or *value = null* if the sender has failed

Implementing TRB with consensus and URB and a perfect failure detector

Initially

proposd = false;

When TRB.wait ()

do nothing

When TRB .send (m)

URB.broad*cast (m)*

When URB.deliver (m) and not proposed

consensus.propose (m)

proposed = true

When fail (sender) and not proposed

consensus.propose (null)

proposed = true

When consensus.decide (v)

TRB.output (v)

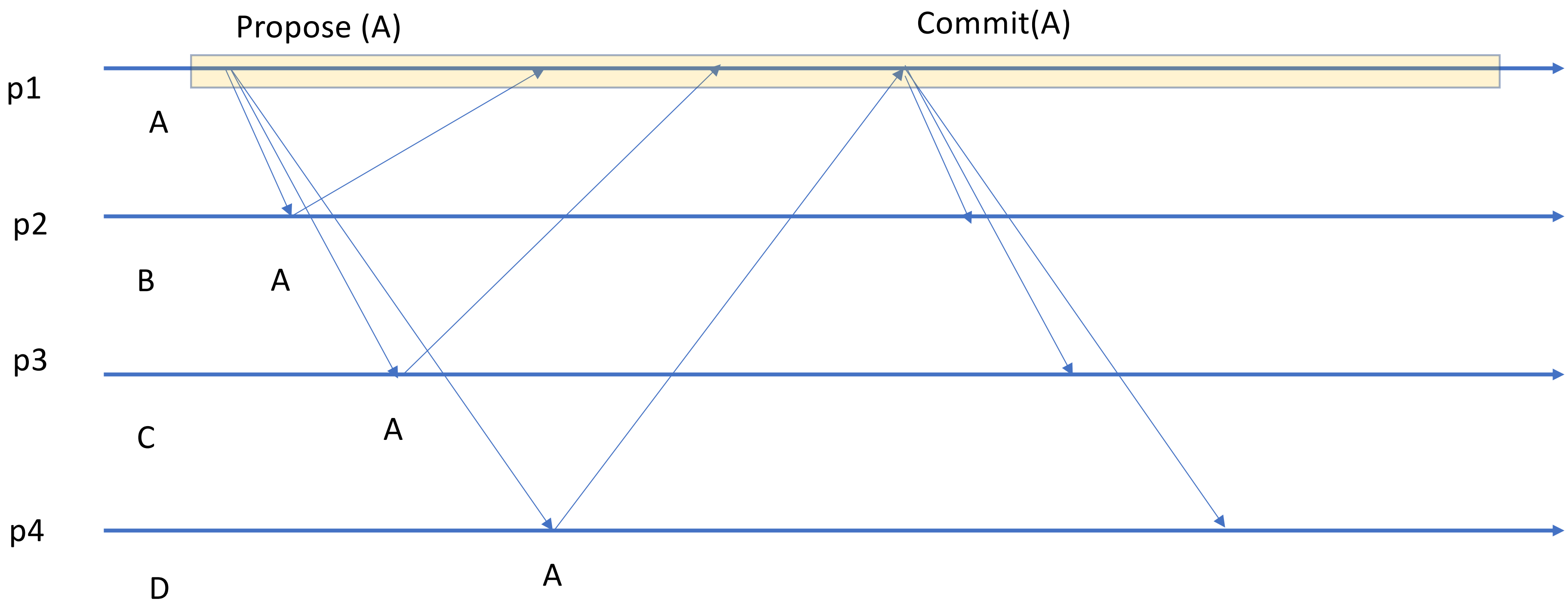
Leader based consensus with P

- The algorithm works in “epochs”, where in each epoch there is a different leader
- The algorithm assumes a perfect failure detector that is able to keep track of which processes are correct and which processes have failed
- In epoch 1 the leader is process p1
- In epoch 2 the leader is process p2
- In epoch n the leader is process pn
- Processes start to execute epoch n only if all leaders from epochs 1 to n-1 have failed.

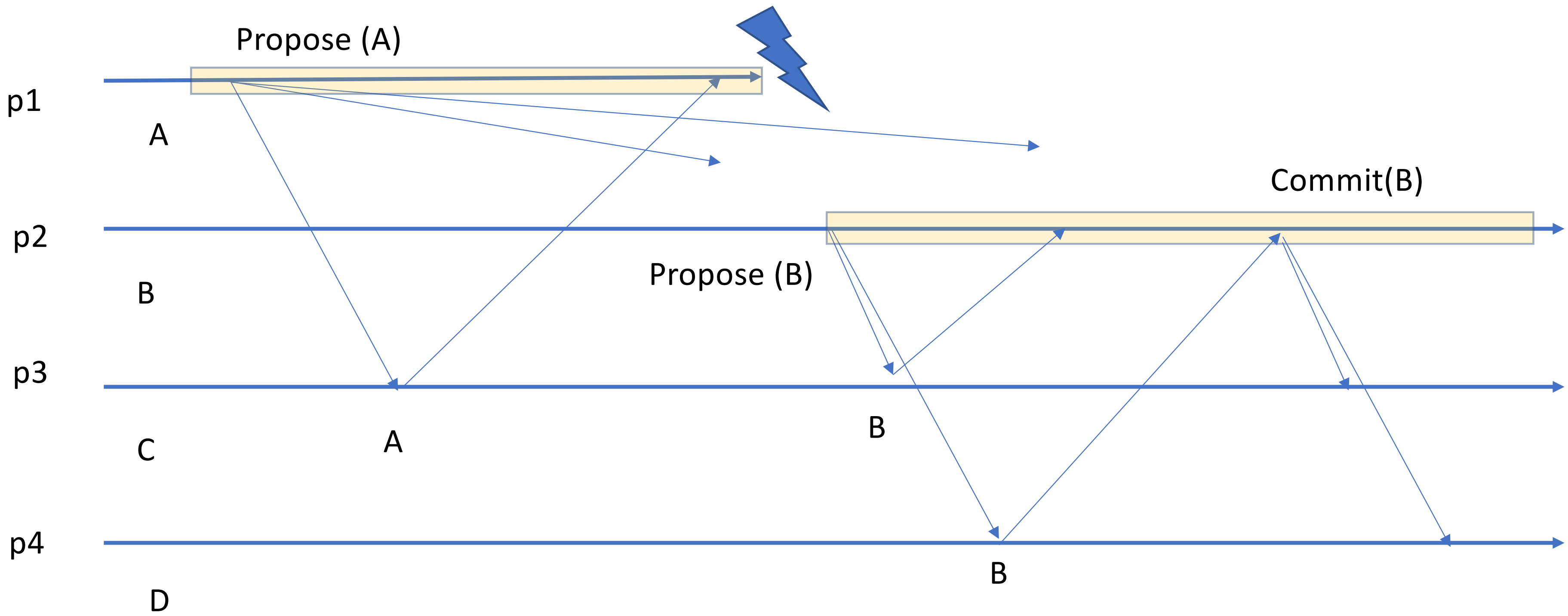
Execution of the leader in epoch n

- The leader sends its value to all other processes
- When another process receives the value form the leader it adopts that value (and forgets any old value it had adopted in the past). It then sends back an acknowledgement to the leader
- The leader waits untill it receives an acknowledgement from every correct process. At this point, all correct processes have adopted the same value (the value proposed by leader n).
- The leader then sends a second message that commits its proposed value
- When the commit message is received, a process outputs the that value as the valued decided by consensus

Example 1



Example 2



Perfect failure detector

- A perfect failure detector is very hard to implement in the general setting
  - Tipically requires the use of real-time OS and real-time network plus bound and predictable loads
  - Thes is to make sure process always respond on time
- If processes may suffer arbitrary delays, it may be impossible to distinguish a failed process fom a slow process

Impossibility of consensus (FLP)

- There is no **deterministic** protocol that solves consensus in an asynchronous system where even a single process may suffer a crash fault
  - Fisher, Lynch, and Paterson. Impossibility of distributed consensus with one faulty process. JACM, Vol. 32, no. 2, April 1985, pp. 374-382
- We will present a simple and elegant proof for consensus among two processes
  - The main result applies to an arbitrary number of processes

Proof of the impossibility of consensus

- By contradiction, let's consider that there exists an algorithm that solves consensus
- We consider three different executions of that algorithm, with varying network conditions
  - Note that any behavior from the network is possible in an asynchronous system
- The two processes executing consensus are called A and B

Execution #1

- Both processes propose 0 initially
- Process B crashes as soon as the execution starts
- By the validity condition of the specification, process A must decide 0
- And by the termination property it must eventually decide 0 let's say it decides at some instant t1

Execution #2

- Both processes propose 1 initially
- Process A crashes as soon as the execution starts
- By the validity condition of the specification, process B must decide 1
- And by the termination property it must eventually decide 1, let's say it decides at some instant t2

Execution #3

- Process A proposes 0 and process B proposes 1 initially
- Messages between A and B (in both directions) are delayed such that they are never delivered before max(t1,t2)
- Process A decides 0 by t1, since its execution is indistinguishable from execution #1
- Process B decides 1 by t2, since its execution is indistinguishable from execution #2
- We found a contradiction (which?)

Unreliable failure detector

- It “suspects” that a process may have failed; however it may be wrong
- Can we solve consensus in such case?
- Or at least prevent processes from deciding different values when the failured detector makes mistakes (and suspects a process that is still active)?

From P to Paxos

- How to change the leader based protocol above to avoid using P?

From P to Paxos

- Challenge 1: Leader cannot wait an acknowledgement from every correct process

    - Solution: Leader should be able to make progress with a majority of replies

From P to Paxos

- Challenge 2: When a leadar fails, and a new leader starts, the new leader may have been excluded from the majority used by the previous leader
  - Solution: New leader must enquire the other processes to check what the previous leader ou leaders have done; this should be aso possible to do using majorities

From P to Paxos

- Challenge 3: Two leaders may start working in parallel; this may happen if a new leader suspects another leader but that leader is still working
  - Solution: ???