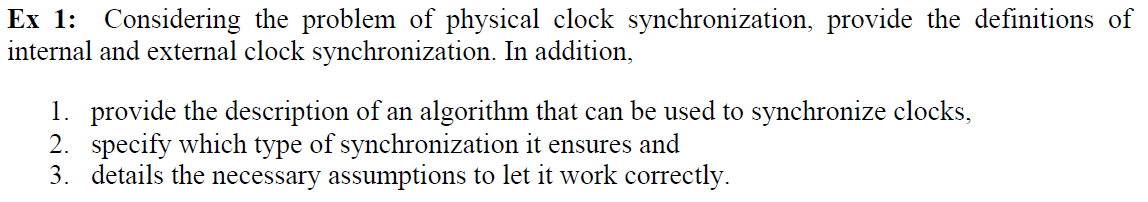
**NOTE: some of those ex might be wrong**

**CLOCK / SYNCHRONIZATION**



4. explain the differences between internal and external clock synchronisation

External clock synchronization:

* Processes synchronize their clock Ci with an authoritative external source S
* Let D>0 be the synchronization bound and S be the source of UTS (Universal Time Coordinated)
* Clocks Ci (for i=1,2,…,N) are externally synchronized with a time source S (UTC) if for each time interval I:

|S(t) - Ci(t)| < D for i=1,2,…,N and for real time t in I

* We say that clocks Ci are accurate within the bound of D

Internal clock synchronization:

* All the processes synchronize their clocks Ci between them
* Let D>0 be the synchronization bound an let Ci and Cj the clocks at processes pi and pj respectively
* Clocks are internally synchronized in a time interval I:

|Ci(t) - Cj(t)| < D for i,j=1,2,…,N and for all time t in I

* We say that clocks Ci, Cj agree within the bound of D

Algorithm to synchronize clocks:

* Christan’s algorithm (external, request-driven):
  + Uses a time server S that receives a signal from an UTS source
  + Works (probabilistically) also in an asynchronous system
  + Based on message round trip (RTT)
  + Synchronization is reached only if RTT is small wrt. required accuracy
  + Process p asks the current time through a message mr and receives t in mt from S
  + Process p sets its clock to t + (RTT/2), RTT is round trip time (RTT) experienced by p
  + Accuracy is ±(RTT/2 - min), min is minimum transmission delay:
    - Case 1: delay Δ = estimate of response – real time

= (RTT/2) – (RTT – min) = -(RTT/2 - min)

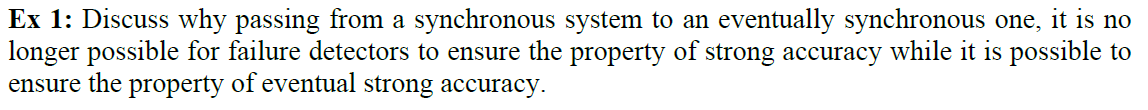
* + - Case 2: delay Δ = estimate of response – real time

= (RTT/2) – (min) = +(RTT/2 - min)

* + Time server is single point of failure (🡪 periods in which synchronization not possible)
  + Time server not crashing or getting attacked (🡪 use cluster of synchronized servers, redundancy, authentication)
* Berkeley’s algorithm (internal, broadcast-based):
  + Master-slave structure, based on gathering the clocks from other processes and computing the differences and respective correction
  + Master process pm sends a message with a timestamp t1 (local clock value) to each process of the system (pm included)
  + When pi receives message from master, it sends back reply with its timestamp t2 (local clock value)
  + When master receives reply message, it reads local clock t3 and computes difference between the clocks: difference Δ = (t1 + t3)/2 – t2
  + Master behaviour:
    - Computes difference Δpi between the master clock and clock of every other process pi (including master)
    - Computes average avg of all Δpi without considering faulty processes (clock differs more than certain threshold)
    - Computes correction of each process (including faulty processes):

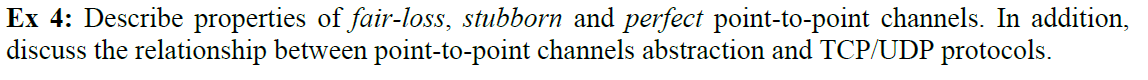
Adgpi = avg – Δpi

* + Slaves behaviour:
    - When process receives correction, it applies it to local clock
    - If correction is negative, don’t adjust the value but slow down the clock by hiding interrupts
  + Accuracy depends on maximum RTT (round trip time), master does not consider clock values associated to RTT greater than the maximum
  + Fault tolerance:
    - If master crashes, new master is elected
    - Tolerant to arbitrary behaviour (eg. wrong values from slaves)



* Synchronous: upper bounded delays wrt. processing, communication and physical clocks or existence of common global clock
* Partial (eventually) synchronous: synchronous most of time with bounded asynchronous periods (there is a unknown time t after which the system becomes synchronous)
* Strong accuracy: If a process p is detected by any process, then p has crashed.
* Eventually strong accuracy: Eventually, no correct process is suspected by any correct process.
* It is no longer possible to ensure strong accuracy, because the system can have an unknown period of time in which the system behaves asynchronous. In this period a correct process might not send its heartbeat reply although it is correct. In this case the perfect failure detector (strong accuracy) detects a crash and will not revise it although the process might be correct.
* It is possible to ensure eventual strong accuracy. In asynchronous period of time a not replying correct process will be flagged as suspect. After the asynchronous period the correct process will deliver its heartbeat reply in bounded time, so that the eventually perfect failure detector (eventually strong accuracy) revises its false judgement and restore the correct process.

**LINKS**



Fair-loss:

* Basic idea that messages might be lost but with probability >0
* Properties:
  + Fair-loss: If correct process p infinitely often sends a message m to a correct process q, then q delivers m an infinite number of times.
  + Finite duplication: If a correct process p sends a message m a finite number of times to process q, then m cannot be delivered an infinite number of times by q.
  + No creation: If some process q delivers a message m with sender p, then m was previously sent to q by process p.

Stubborn:

* After timeout resend each sent message again
* Properties:
  + Stubborn delivery: If a correct process p sends a message m once to a correct process q, then q delivers m an infinite number of times.
  + No creation: If some process q delivers a message m with sender p, then m was previously sent to q by process p.

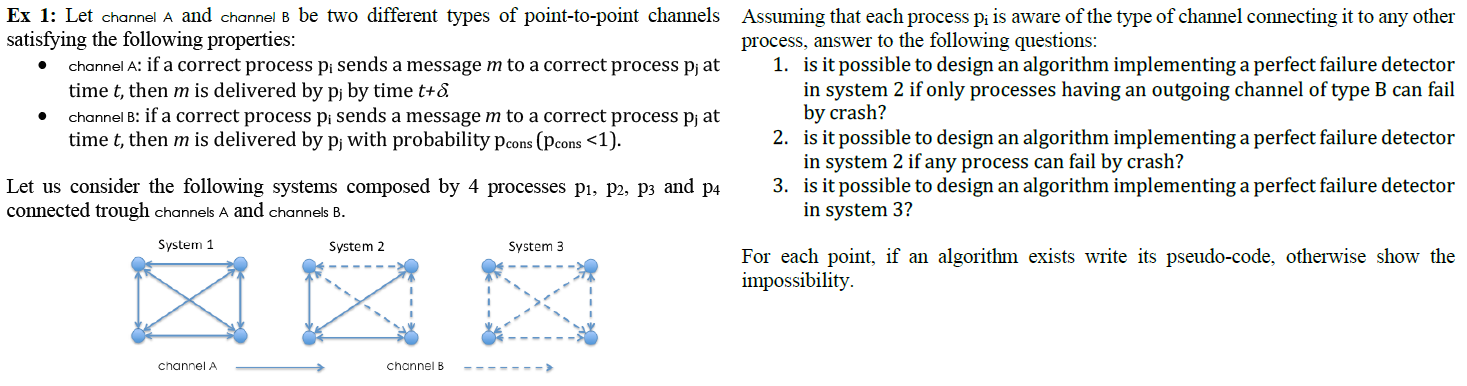
Perfect:

* Deliver each message not more than once
* Properties:
  + Reliable delivery: If a correct process p sends a message m to a correct process q, then q eventually delivers m.
  + No duplication: No message is delivered by a process more than once.
  + No creation: If some process q delivers a message m with sender p, then m was previously sent to q by process p.

TCP/UDP protocols:

* Communication channels providing data exchange between server and client
* Port-to-port instead of point-to-point
* TCP (perfect):
  + Sends data correctly (reliable, ordered and error checked)
  + Higher transmission time
* UDP (fair-loss):
  + Sends data without considering failures
  + Data might be lost
  + Provides checksum function for receiver

**IMPL. PERFECT FAILURE DETECTOR / COMPOSED / FP2P + PP2P**



1. Yes, because crash can be detected by p2 through messages in the type A channels:

pp2p: Perfect Point-to-Point link (channel A)

fp2p: Fair-loss Point-to-Point link (channel B)

Upon event <Init> do

Correct := {P1, P2, P3, P4}

Alive := {};

Δ := 2\*δ; !!!

// Δ := 4\*δ;

Starttimer(Δ);

Upon event <Timeout> do

Forall p ϵ correct do

Trigger xp2pSend(HBRQ);

Alive = {};

Starttimer(Δ);

If myID = P1 do

Update Correct; Correct = Alive; !!!

Trigger xp2pSend(‘Alive’, Alive);

Upon even <xp2pDeliver(‘Alive’, List) do

Forall p ϵ (Correct\List) do

Trigger Crash(p);

// Correct := List;

Upon event <xp2pDeliver(HBRQ)> from pi do

Alive = Alive U {pi}

Trigger xp2pDeliver(HBRL); !!!

// Trigger xp2pSend(HBRL);

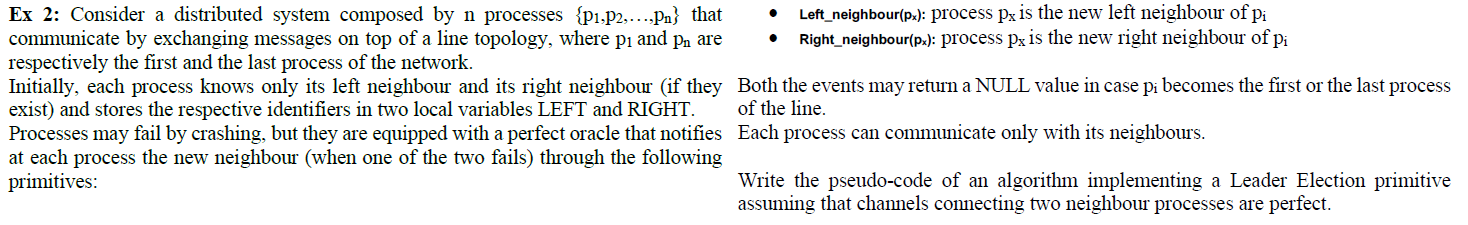
Upon event <xp2pDeliver(HBRL)> from pi do

Alive = Alive U {pi}

1. No, counterexample: If process p1 crashes, the perfect failure detector will detect it due to the type A channels. Then, further messages of p3 might get lost due to the delivering probability pcons < 1 of all remaining type B channels. In this case, a crash of p3 will be detected, although p3 might be alive.

1. No, counterexample: All messages of p3 might get lost due to the delivering probability pcons < 1 of all remaining type B channels. In this case, a crash of p3 will be detected, although p3 might be alive.

**IMPL. LEADER ELECTION / LINE / PP2P**



Upon event <Init> do

RIGHT := myID + 1;

LEFT := myID - 1;

leader := P1;

Upon event <Right\_neighbour(px)> do

RIGHT := px;

Upon event <Left\_neighbour(px)> do

LEFT := px;

If LEFT = ⊥ do

leader = MyID;

Trigger Leader(leader);

Trigger pp2pSend(‘New Leader’, leader);

Upon event <pp2pSend(‘New Leader’, p) do

leader = p;

Trigger Leader(p);

Trigger pp2pSend(‘New Leader’, leader);

**IMPL. Same as Ex 2 but implement PERFECT FAILURE DETECTOR / LINE / PP2P primitive.**

Upon event <Init> do

RIGHT := myID + 1;

LEFT := myID - 1;

Alive := π;

Upon event <Right\_neighbour(px)> do

Alive := Alive \ RIGHT;

Trigger Crash(RIGHT);

Trigger pp2pSend(‘Failed’, RIGHT);

RIGHT := px;

Upon event <Left\_neighbour(px)> do

Alive := Alive \ LEFT;

Trigger Crash(LEFT);

Trigger pp2pSend(‘Failed’, LEFT);

LEFT := px;

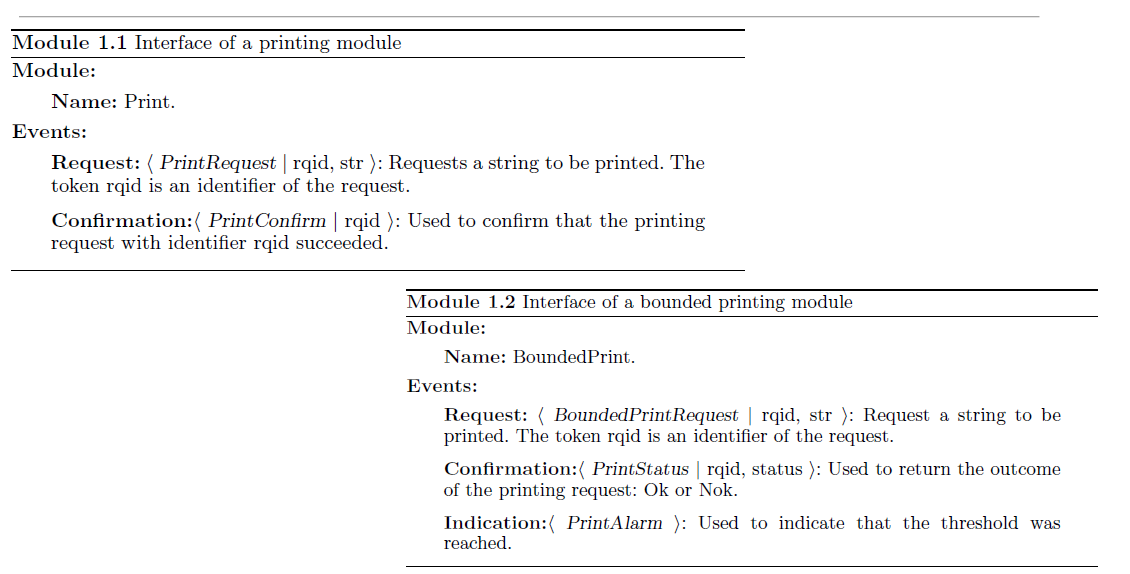
Upon event <pp2pDeliver(‘Failed’, p)> do

If p ϵ Alive do

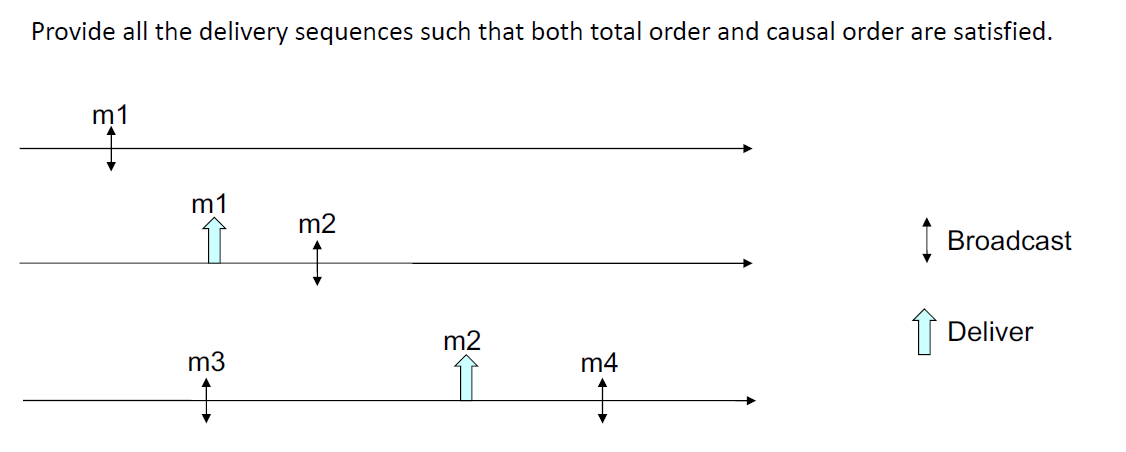
Alive := Alive \ p;

Trigger Crash(p);

Trigger pp2pSend(‘Failed’, p);



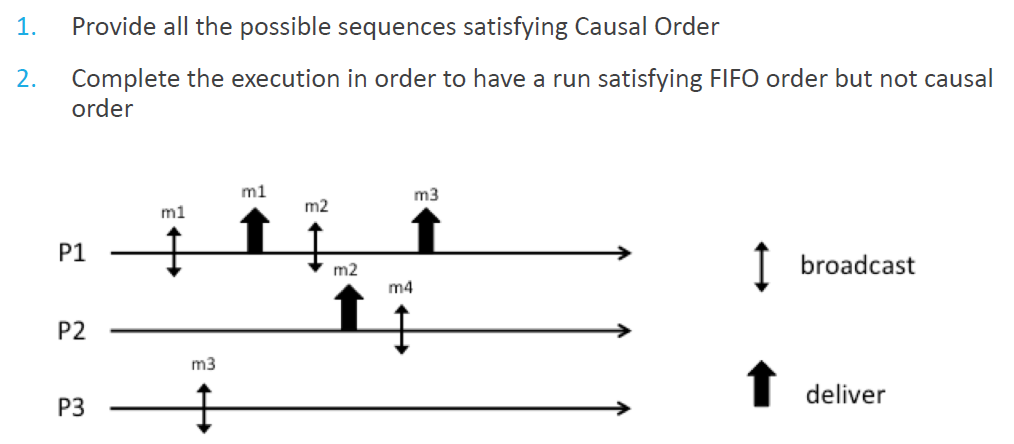
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_



m3, m1, m2, m4

m1, m3, m2, m4

m1, m2, m3, m4



1. m3, m1, m2, m4

m1, m3, m2, m4

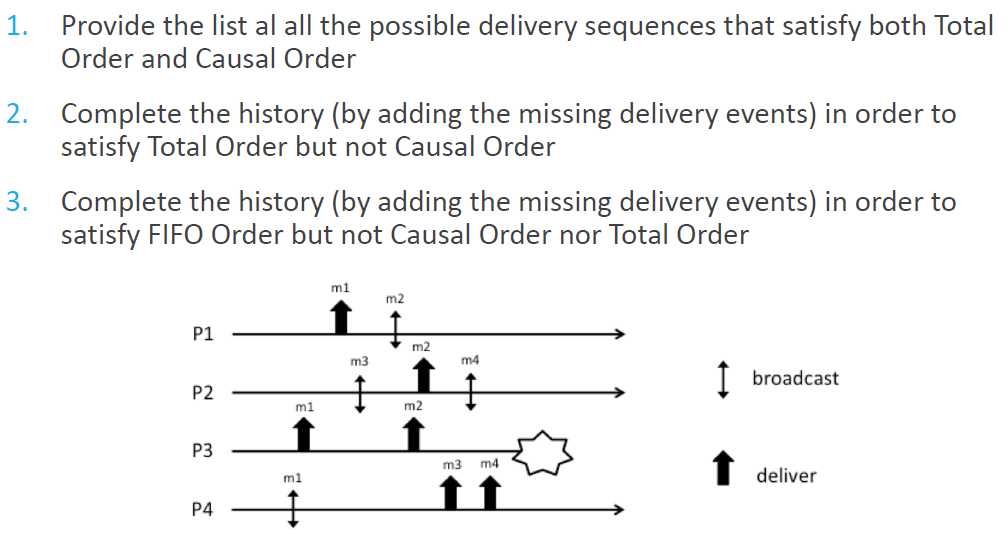
m1, m2, m3, m4

m1, m2, m4, m3

1. p1: m1🡪, 🡪m1, m2🡪, 🡪m3, 🡪m4, 🡪m2

p2: 🡪m1, 🡪m2, m4🡪, 🡪m4, 🡪m3

p3: m3🡪, 🡪m1, 🡪m3, 🡪m4, 🡪m2



1. m1, m2, m3, m4

m1, m3, m2, m4

m3, m1, m2, m4

1. p1: 🡪m1, m2🡪, 🡪m3, 🡪m4, 🡪m2

p2: 🡪m1, m3🡪, 🡪m3, 🡪m2, m4🡪, 🡪m4

p3: 🡪m1, 🡪m3, 🡪m2, 🡪m4, crash

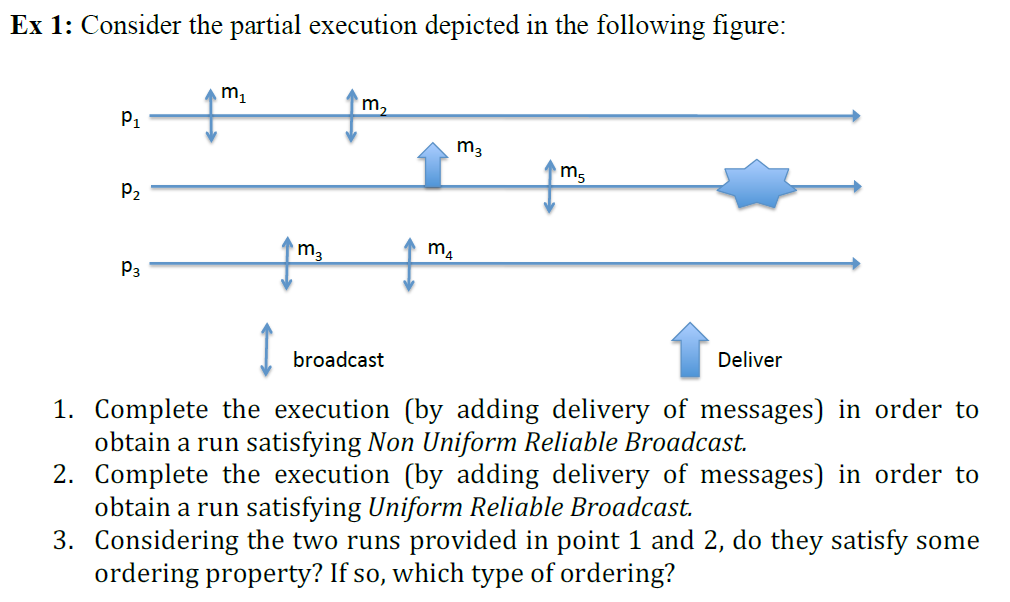
p4: m1🡪, 🡪m1, 🡪m3, 🡪m4, 🡪m2

1. p1: 🡪m1, m2🡪, 🡪m3, 🡪m4, 🡪m2

p2: 🡪m1, m3🡪, 🡪m3, 🡪m2, m4🡪, 🡪m4

p3: 🡪m1, 🡪m3, 🡪m2, 🡪m4, crash

p4: m1🡪, 🡪m3, 🡪m4, 🡪m2, 🡪m1



1. p1: m1🡪, m2🡪, 🡪m1, 🡪m2, 🡪m3, 🡪m4

p2: 🡪m3, m5🡪, crash

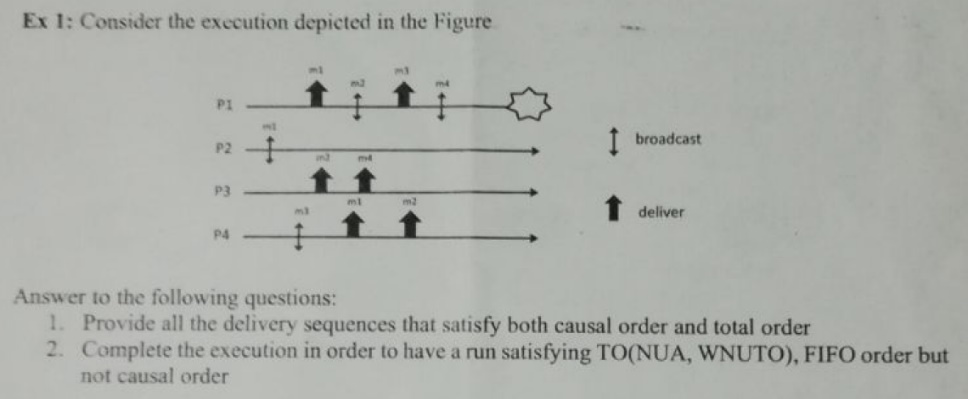
p3: m3🡪, m4🡪, 🡪m1, 🡪m2, 🡪m3, 🡪m4

1. p1: m1🡪, m2🡪, 🡪m1, 🡪m2, 🡪m3, 🡪m4, 🡪m5

p2: 🡪m3, m5🡪, 🡪m5, crash

p3: m3🡪, m4🡪, 🡪m1, 🡪m2, 🡪m3, 🡪m4, 🡪m5

1. Reliable Broadcast does not have any property on ordering deliveries on messages, but only on the set of delivering messages. Only ordering property has to be considered: Deliver after Sending.



1. m1, m2, m3, m4

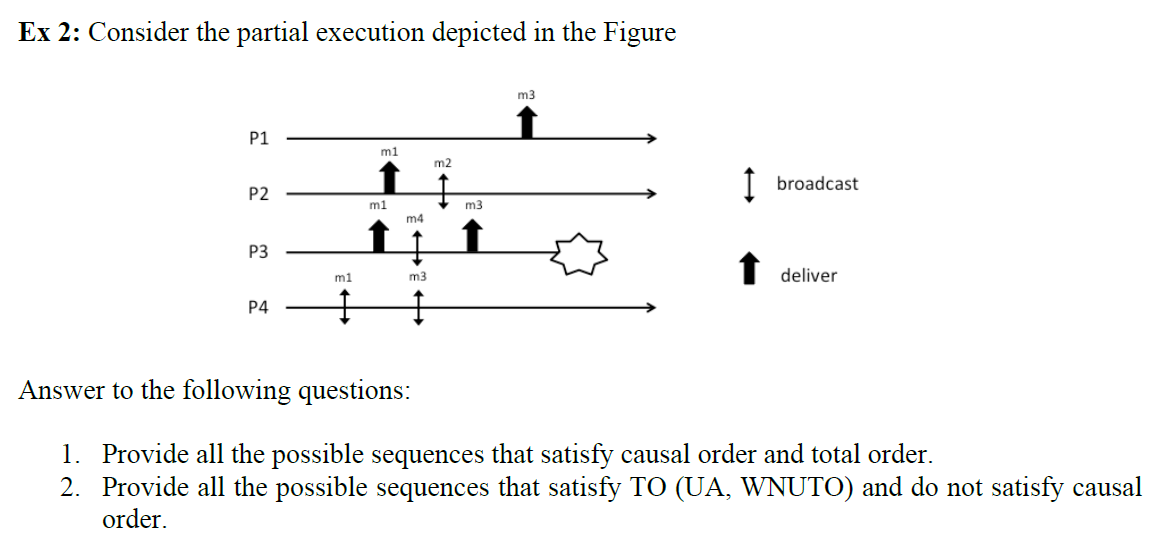
m1, m3, m2, m4

1. p1: 🡪m1, m2🡪, 🡪m3, m4🡪, 🡪m4, m5🡪, 🡪m5, crash

p2: m1🡪, 🡪m1, 🡪m2, 🡪m4, 🡪m3

p3: 🡪m1, 🡪m2, 🡪m4, 🡪m3

p4: m3🡪, 🡪m1, 🡪m2, 🡪m4, 🡪m3



1. m1, m2, m3, m4 Since m4 does not need to be delivered:

m1, m2, m4, m3 m1, m3, m2

m1, m3, m2, m4 m1, m2, m3

m1, m3, m4, m2

m1, m4, m2, m3

m1, m4, m3, m2

1. m4, m3, m1, m2 Since m4 does not need to be delivered:

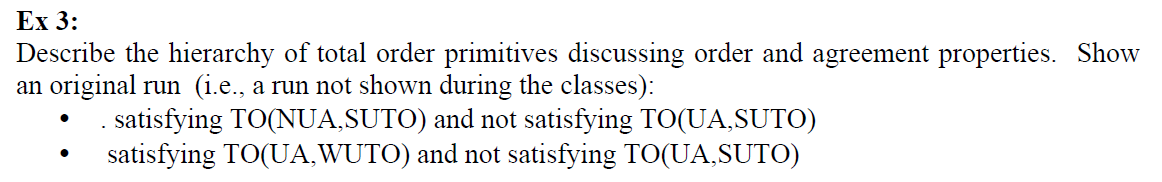
m3, m4, m1, m2 m3, m1, m2

m3, m1, m4, m2

m3, m1, m2, m4

m4, m1, m3, m2

m4, m1, m2, m3

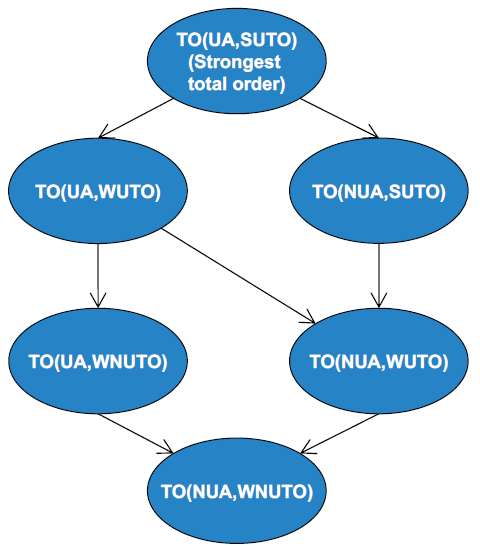


Agreement property:

1. Uniform (UA): If a process (correct or not) TODelivers a message m, then all correct processes will eventually TODeliver m
2. Non-Uniform (NUA): If a correct process TODelivers a message m, then all correct processes will eventually TODeliver m

Order Property:

1. Strong Uniform TO (SUTO): If some process TODelivers some message m1 before message m2, then a process TODelivers m2 only after it has TODelivered m1
2. Weak Uniform TO (WUTO): If process p and q both TODeliver messages m1 and m2, then p TODelivers m1 before m2 if and only if q TODelivers m1 before m2
3. Strong Non-Uniform TO (SNUTO): If some correct process TODelivers some message m1 before message m2, then a correct process TODelivers m2 only after it has TODelivered m1
4. Weak Non-Uniform TO (WNUTO): If correct processes p and q both TODeliver messages m1 and m2, then p TODelivers m1 before m2 if and only if q TODelivers m1 before m2



TO(NUA, SUTO), NOT TO(UA, SUTO):

P1: m1, m2

P2: m1, m2

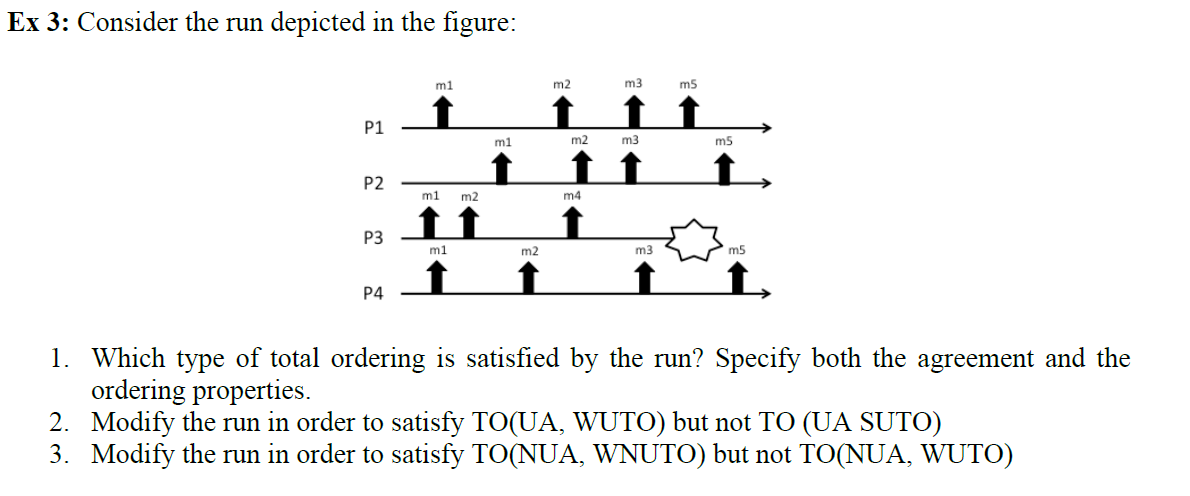
P3: m1, m2, m3, crash

TO(UA, WUTO), NOT TO(UA, SUTO):

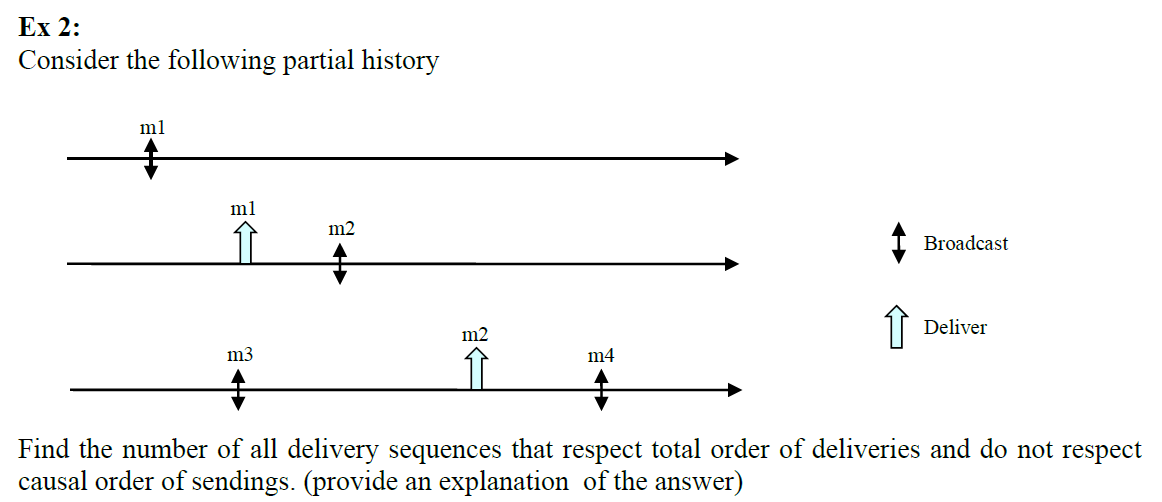
P1: m1, m2, m3

P2: m1, m2, m3

P3: m1, m3, crash



1. TO(NUA, SUTO)
2. P1 & P2 & P4: m1, m2, m3, m4, m5
3. P3: m1, m2, m4, m5, m3



FIFO: m3🡪m4 (change to m4🡪m3 to violate causal order)

LO: m1🡪m2 (not changeable because it would change given history of p2, keep TO)

m2🡪m4 (not changeable because it would change given history of p3, keep TO)

m1, m2, m4, m3 (one possible delivery sequence)

**OR:**

FIFO: m3🡪m4 (change to m4🡪m3 to violate FIFO order)

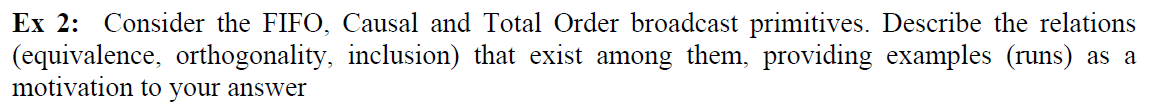
LO: m1🡪m2 (change to m2🡪m1)

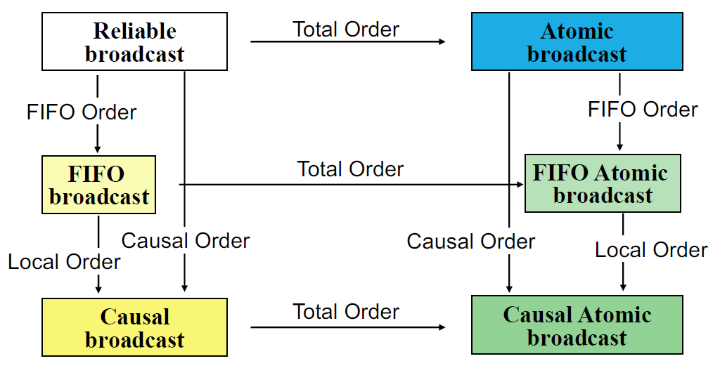
m2🡪m4 (change to m4🡪m2)

m4, m2, m1, m3

m4, m2, m3, m1

m4, m3, m2, m1 (3 possible delivery sequences)





FIFO: If some process broadcasts message m1 before it broadcasts message m2, then no correct process delivers m2 unless it has already delivered m1.

Causal: For any message m1 that potentially caused a message m2, no process delivers m2 unless it has already delivered m1 (see potentially causes).

Total: Total order orders all messages, even those from different senders and those that are not causally related. The message is delivered to all or none of the processes and, if the message is delivered, every other message is ordered either before or after this message.

Causal Order 🡪 FIFO Order

FIFO Order !🡪 Causal Order

Causal Order 🡨🡪 (=) FIFO Order + Local Order (if process delivers m before sending m’, then no correct process deliver m’ if it not already delivered m)

Total order is orthogonal to FIFO and Causal order.

TO: p1: m1🡪, m2🡪, 🡪m2, 🡪m1

p2: 🡪m2, 🡪m1

FIFO: p1: m1🡪, m2🡪, 🡪m3, 🡪m1, 🡪m2

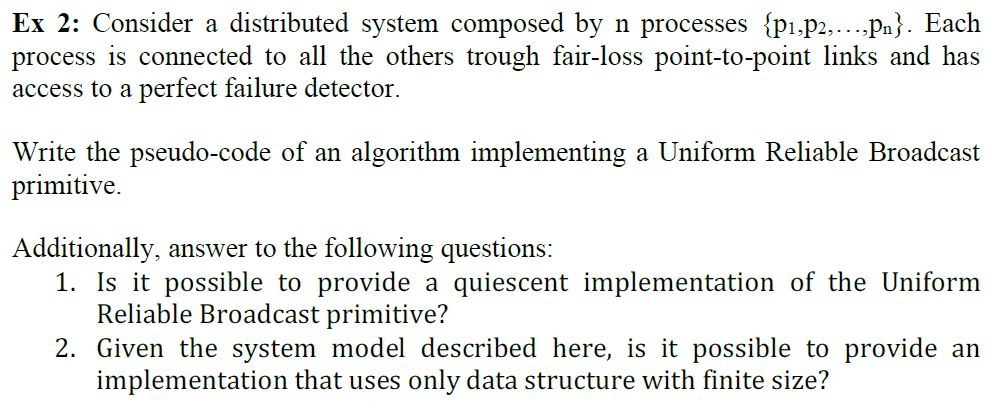
p2: 🡪m1, m3🡪, 🡪m2, 🡪m3

Causal: p1: m1🡪, 🡪m3, 🡪m1, 🡪m2, 🡪m4

p2: 🡪m1, m2🡪, 🡪m3, 🡪m2, m4🡪, 🡪m4

p3: m3🡪, 🡪m1, 🡪m2, 🡪m3, 🡪m4

**IMPL. UNIFORM RELIABLE BROADCAST / COMPOSED / FP2P**



Upon event <Init> do

Sent = {};

Delivered = {};

Pending = {}; !!!

Correct = π;

DEL[] = [{}];

ACK[] = [{}];

Starttimer(Δ);

Upon event <Crash(p)> do

Correct = Correct \ {p};

// Broadcast to all correct processes and save msg in Sent

Upon event <URB\_Broadcast(m)> do

Forall p ϵ Correct do

Trigger fp2pSend(m) to p;

Sent = Sent U {<’MSG’, m>};

// Save arrival of msg and retransmit an ACK

Upon event <fp2pDeliver(‘MSG’, m) from p do

Pending = Pending U {<’MSG’, m, p>}; !!!

Trigger fp2pSend(‘ACK’, m, myID) to p;

// Save the arrival of the msg at target process

Upon event <fp2pDeliver(‘ACK’, m, pi)> from p do

ACK[m] = ACK[m] U {p};

// If msg is arrived in all correct target processes, send delivering order to all correct processes

Upon exists m | Correct ⊆ ACK[m] do

Forall p ϵ Correct do

Trigger fp2pSend(‘DEL’, m);

Sent = Sent U {<’DEL’, m>};

// Resend all msg after every timeout

Upon event <timeout> do

Forall <\*, m> ϵ Sent do // Don’t care if ‘MSG’ or ‘DEL’

Trigger fp2pSend(\*, m);

// Starttimer(Δ);

Upon event <fp2pDeliver(‘DEL’, m)> from p do

Forall p ϵ Correct do

Trigger fp2pSend(‘DEL’, m);

Sent = Sent U {<’DEL’, m>};

DEL[m] = DEL[m] U {<m, p>}

Upon exists m | Correct ⊆ DEL[m] do

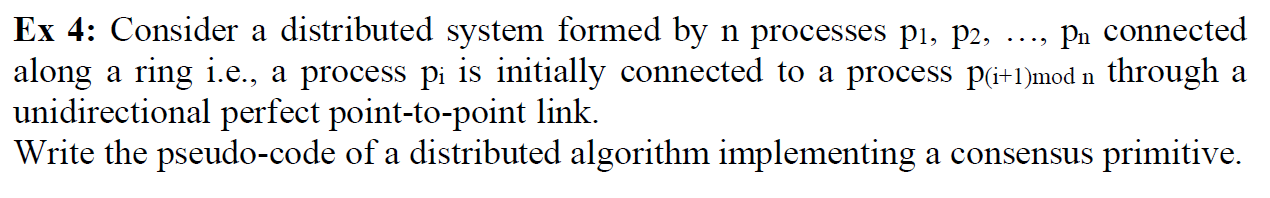
If m ∉ delivered

Delivered = delivered U {m}

Trigger URB\_Deliver(m);

1. Not in the case of fair-loss point-to-point links, since messages have to be transmitted infinitely often to be sure that it will be delivered due to the finite duplication property.
2. No, since every message gets an entry in the DEL[] and ACK[] arrays and an infinite number of messages are possible.

**IMPL. CONSENSUS / RING / PP2P**



Upon event <Init> do

U = I;

Decision = ⊥;

Decided = False; !!!

If (myID = P1) do

T = generate\_token();

Trigger pp2pSend(‘Token’, T) to next process P(myID+1)mod n;

T = {};

Else do

T = ⊥; !!!

// T = {};

Upon event <Propose(v)> do

U = val;

If T ≠ {} do !!! Only takes proposals if token here

T = T U {<R, U>}; !!!

T = T U {<U>};

Upon event <Deliver(‘Token‘, Ti)> do

T = Ti;

Starttimer(1);

Trigger pp2pSend(‘Token’, T) to next process P(myID+1)mod n;

T = {};

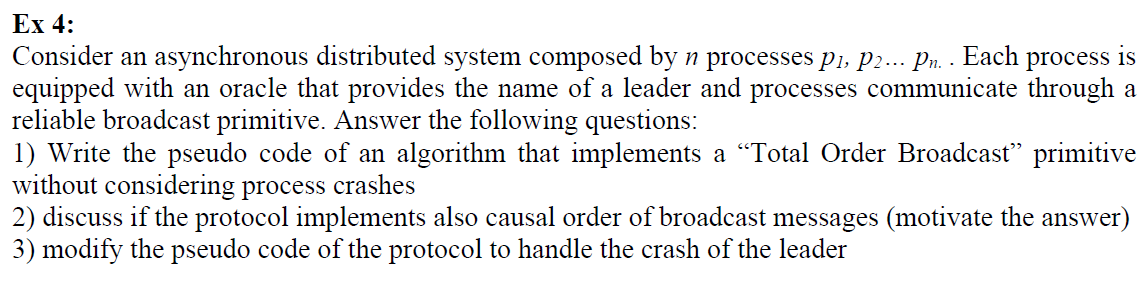
Upon T ≠ {} AND |T| = n *(// AND Decision = ⊥)* do !!! What if token n+1 comes?

Decision = min(T);

Trigger Decide(Decision);

Decided = T; !!!

**IMPL. TOTAL ORDER BROADCAST / COMPOSED / WITH LEADER AND RELIABLE BROADCAST**



1.

Upon event <Init> do

Leader = p1;

Delivered = {};

Pending = {};

Upon event <leader(p)> do

Leader = p;

Upon event <TO\_B(m)> do

Trigger RB\_B(m);

// Save locally all incoming messages if you are leader

Upon event <RB\_D(m)> do

If m ∉ delivered AND myID = Leader do

Pending = Pending U {m};

Upon Pending ≠ {} AND myID = Leader do

List = sort(Pending);

Pending = Pending \ List;

Trigger RB\_B(‘DEL’, List);

Upon event <RB\_D(‘DEL’, List)> do

For all (m) ϵ List do

Delivered = Delivered U {m};

Trigger TO\_D(m);

2.

No, since only the delivery sequence of the leader is considered, which is not depending on causal ordering but the delivery delays.

3.

Upon event <Init> do

…

Correct = PI;

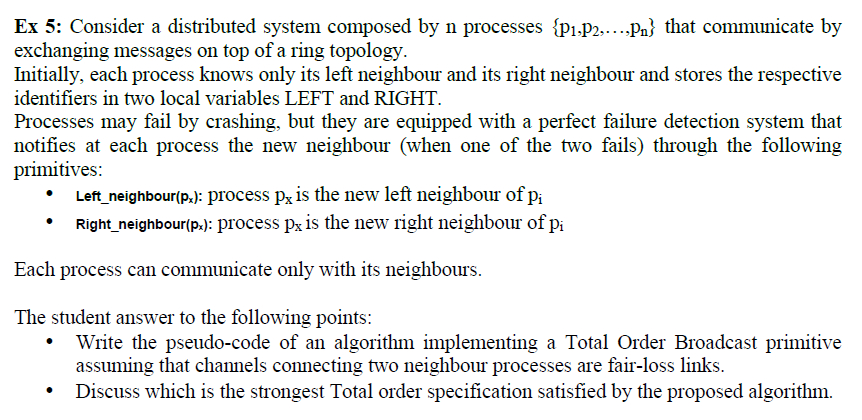
Upon event <crash(p)> do

Correct = Correct \ p;

If Leader = p do

Trigger leader(maxrank(Correct));

**IMPL. TOTAL ORDER BROADCAST / RING / FP2P & PERFECT FAILURE DETECTOR**



Upon event <Init> do

If myID = 1 do

LEFT = N;

Else do

LEFT = myID – 1;

If myID = N do

RIGHT = 1;

Else do

RIGHT = myID + 1;

Sent = {};

Starttimer(Δ);

Upon event <timeout> do

For all (m) ϵ Sent do

Trigger fp2pSend(m) to right neighbour;

Starttimer(Δ);

Upon event <TO\_B(m)> do

Trigger fp2pSend(m) to right neighbour;

Sent = Sent U {m};

Upon event <fp2pDeliver(m)> do

Trigger fp2pSend(m) to right neighbour;

Sent = Sent U {m};

Upon event <right\_neighbour(px)> do

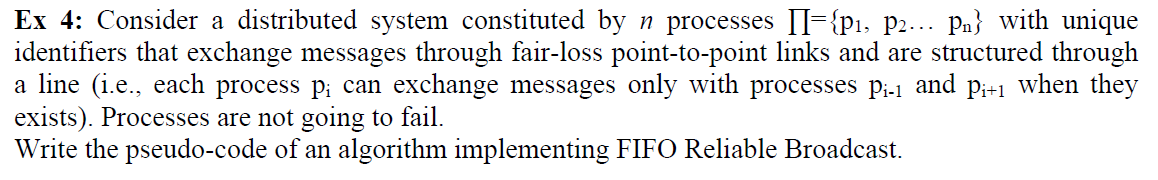
RIGHT = px;

Upon event <left\_neighbour(px)> do

LEFT = px;

**CONSENSUS & TO MISSING**

**IMPL. FIFO RELIABLE BROADCAST / COMPOSED / FP2P**



Upon event <Init> do

Sn = 0;

Send = {};

Pending = {};

Next = [1]N; // Save current round for each process

Starttimer(Δ);

Upon event <FIFO\_Broadcast(m)> do

Sn = Sn + 1;

// Trigger fp2pSend(m, self, Sn) to PmyID-1 & PmyID+1;

Send = Send U {(m, self, Sn)};

Upon event <timeout> do

Forall (m, p, Sn) ϵ Send do

Trigger fp2pSend(m, p, Sn) to PmyID-1 & PmyID+1;

Starttimer(Δ);

Upon event <fp2pDeliver(m, p, Sn)> do

// Trigger fp2pSend(m, p, Sn) to PmyID-1 & PmyID+1; // Resending to next process

Send = Send U {(m, p, Sn)};

Pending = Pending U {(m, p, Sn)};

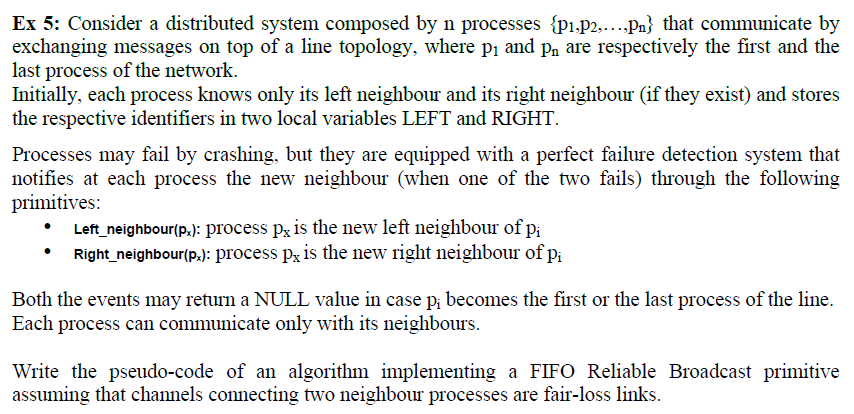
While exists (m’, p, Sn’) ϵ Pending, s.t. Sn’ = next[p] do

next[p] = next[p] + 1;

// pending = pending \ {(m’, p, Sn’)};

trigger FIFO\_Deliver(m);

**IMPL. FIFO RELIABLE BROADCAST / LINE / FP2P**



Upon event <Init> do

If myID = 1 do

LEFT = NULL;

Else do

LEFT = myID – 1;

If myID = N do

RIGHT = NULL;

Else do

RIGHT = myID + 1;

Sent = {};

Sn = 0;

Pending = {};

Next = [1]N; // Save current round for each process

Starttimer(Δ);

Upon event <timeout> do

Forall (m, p, Sn) ϵ Send do

Trigger fp2pSend(m, p, Sn) to PmyID-1 & PmyID+1;

Starttimer(Δ);

Upon event <FIFO\_Broadcast(m)> do

Sn = Sn + 1;

// Trigger fp2pSend(m, self, Sn) to PmyID-1 & PmyID+1;

Send = Send U {(m, self, Sn)};

Upon event <right\_neighbour(px)> do

RIGHT = px;

Upon event <left\_neighbour(px)> do

LEFT = px;

Upon event <fp2pDeliver(m, p, Sn)> do

// Trigger fp2pSend(m, p, Sn) to PmyID-1 & PmyID+1; // Resending to next process

Send = Send U {(m, p, Sn)};

Pending = Pending U {(m, p, Sn)};

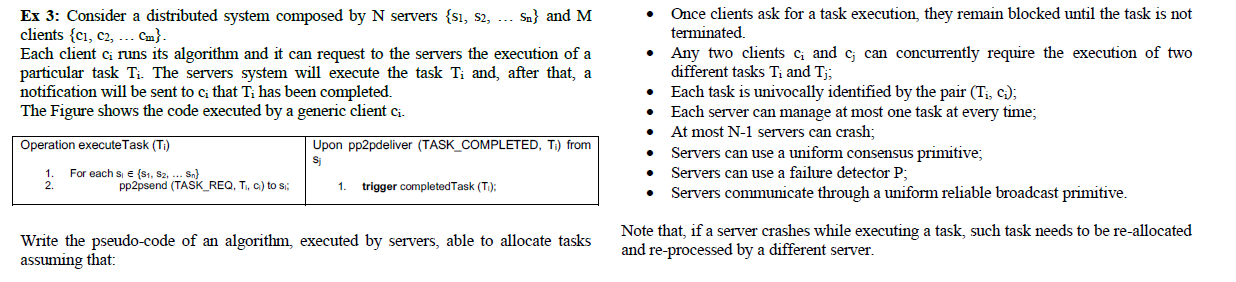
While exists (m’, p, Sn’) ϵ Pending, s.t. Sn’ = next[p] do

next[p] = next[p] + 1;

// pending = pending \ {(m’, p, Sn’)};

trigger FIFO\_Deliver(m);

**IMPL. SERVER / COMPOSED / CONSENSUS & PERFECT FAILURE DETECTOR & UNIFORM RELIABLE BC**



Upon event <Init> do

Busy = False;

Pending = {};

Correct = S;

Cons\_Run = False; // Consensus running

Assignment = {}; // Store (server, task)

Upon event <pp2pDeliver(TASK\_REQ, Ti, sni, ci)> do // sni = sequence number

Pending = Pending U {(Ti, sni, ci)}

When (Pending ≠ {}) AND (NOT Cons\_Run) AND (NOT Busy) do

Task = select\_Task(Pending);

Cons\_Run = True;

Trigger Propose(pi, Task); // propose itself for task

Upon event <decide(pi, Task)> do

Cons\_Run = False;

Assignment = Assignment U {(pi, Task)};

Pending = Pending \ Task;

If myID = pi do

Busy = True;

EXECUTE\_TASK();

Trigger pp2pSend(TASK\_COMPLETED, Task) to client;

Busy = False;

Trigger URB\_Broadcast(COMPLETE, Task, myID);

Upon event <URB\_Deliver(COMPLETE, Task, p)> do

Assignment = Assignment \ {(pi, Task)};

Upon event <crash(p)> do

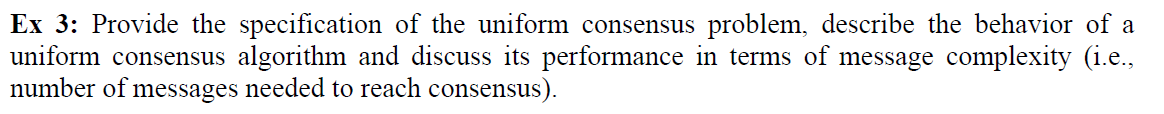
Correct = Correct \ {p};

When exists (p, Task) ϵ Assignment do // Reassign task to other server

Assignment = Assignment \ {(p, Task)};

Pending = Pending U {(p, Task)};

**UNIFORM CONSENSUS**



Specification:

Module: UniformConsensus, instance uc

Events: Request <uc, Propose | v>: Proposes value v for consensus

Indication <uc, Decide | v>: Outputs a decided v of consensus

Properties: Termination: Every correct process eventually decides some value.

Validity: If a process decides v, then v was proposed by some process.

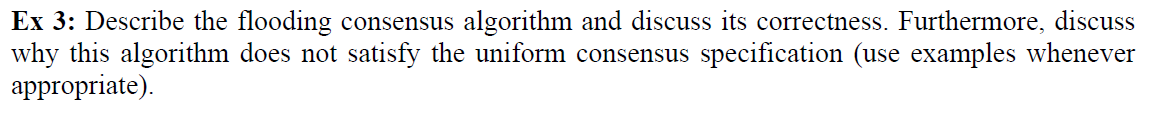
Integrity: No process decides twice.

Uniform agreement: No two processes decide differently.

If a process wants to propose a value, it broadcasts the value to all other processes and saves it locally. Each delivering process saves the proposed value and the broadcasting process number locally. In each round each process broadcasts the current list of proposed values. When all correct processes proposed and no value is decided yet a new round starts. After N (number of processes) rounds each process decides a value from the list by choosing the minimum. A perfect failure detector gets crashing processes.

The algorithm takes N rounds, so N communication steps. Moreover, in each round each process sends it proposal set to all other processes, that is O(N3) number of messages.

**FLOODING CONSENSUS**



If a process wants to propose a value, it broadcasts the value to all other processes and saves it locally. Each delivering process saves the proposed value and the broadcasting process number locally and its current round. When all correct processes proposed in a round and no value is decided yet a new round starts. In each round each process broadcasts the current list of proposed values. A decision is made if in the first round all correct processes proposed or in two rounds in a row the same correct processes proposed. Then, each process decides a value from the current round list by choosing the minimum and broadcasts it to all other processes. Each process delivering a decision from a correct process decides and rebroadcasts its value if no decision is made yet. A perfect failure detector gets crashing processes.

Properties: Termination: Every correct process eventually decides some value.

Validity: If a process decides v, then v as proposed by some process.

Integrity: No process decides twice.

Agreement: No two correct processes decides differently.

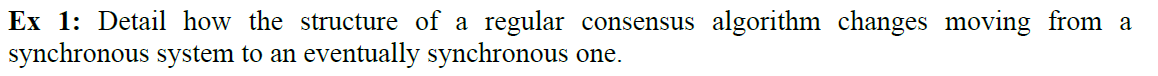
Validity and Integrity: Follow from the properties on the communication channels.

Termination: At most after N rounds processes decide.

Agreement: The same deterministic function is applied to the same values by correct processes.

No, the flooding consensus algorithm does not satisfy the uniform consensus specification (Uniform Agreement: No two processes decide differently). In case a process decides a value and then crashes. The decision will not be broadcasted to all the other processes. All other processes get notified by the PFD and start the next round. Another proposed value might be decided by the remaining correct processes.

**REGULAR CONSENSUS & SYNCHRONIZATION**



Regular Consensus algorithm:

Basic Idea: Processes exchange their values. When all proposals from correct processes are available a one value can be chosen.

Problem: Due to failures, some values can be lost

Solution: A value can be selected only when no failures happen during the communication

Algorithm: Each process saves locally in each round the delivered proposals of the other processes. When no decision is made yet and all correct processes proposed, a decision is triggered. Otherwise a new round is started.

In asynchronous system, no reliable prediction for faulty processes can be made (no perfect failure detector). The Paxos algorithms make some progress (liveness) only when network behaves in a ‘good way’ for long enough periods of time.

Paxos Consensus algorithm:

Basic Idea: Use actors (Proposer proposing values, multiple Acceptors committing on a final decided value, Learners passively assisting to decision and obtaining final decided value).

Majority is needed to guarantee only one value is accepted

If proposal with value v and number n is issued, then there is a set S consisting of majority of acceptors such that either (a) no acceptor in S has accepted any proposal numbered less than n, or (b) v is the value highest-number proposal among all proposals numbered less than n accepted by the acceptors in S.

Algorithm: Phase 1 (Prepare request), Phase 2 (Accept request)

Phase 1: - proposer sends prepare request (PREPARE, n) to majority of acceptors

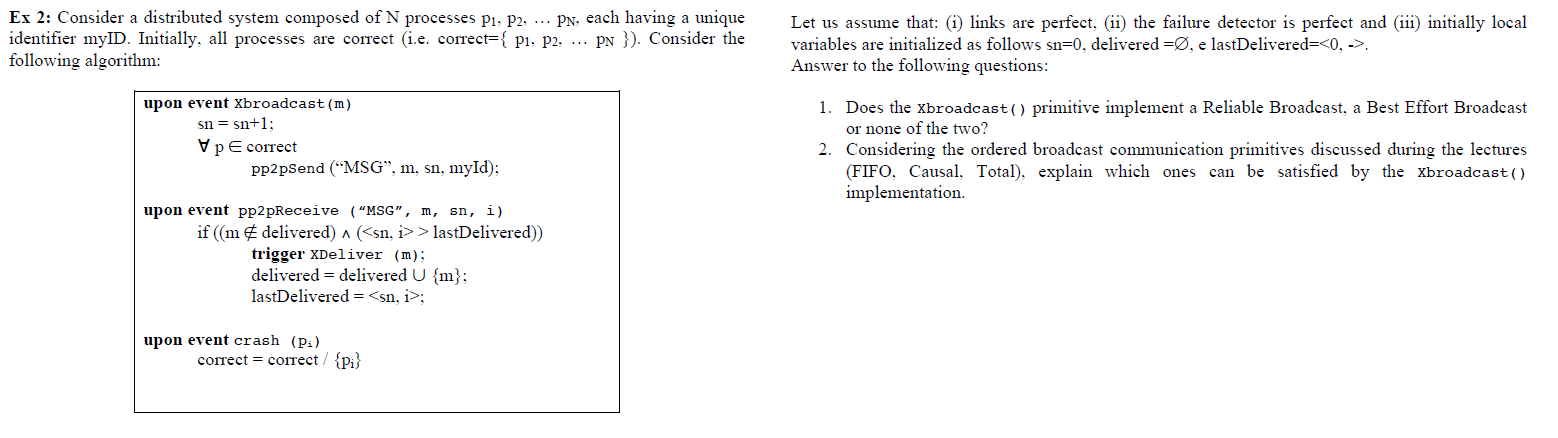
- acceptors reply with highest number accepted less than n, else ⊥ (or denial if n too old)

Phase 2: - proposer receives from majority and sends accept request with n from prepare request and v of highest-numbered proposal of responses

- acceptor accepts request, if not already responded having number greater than n

- learners receive from majority of acceptors, decide and resend to other learners

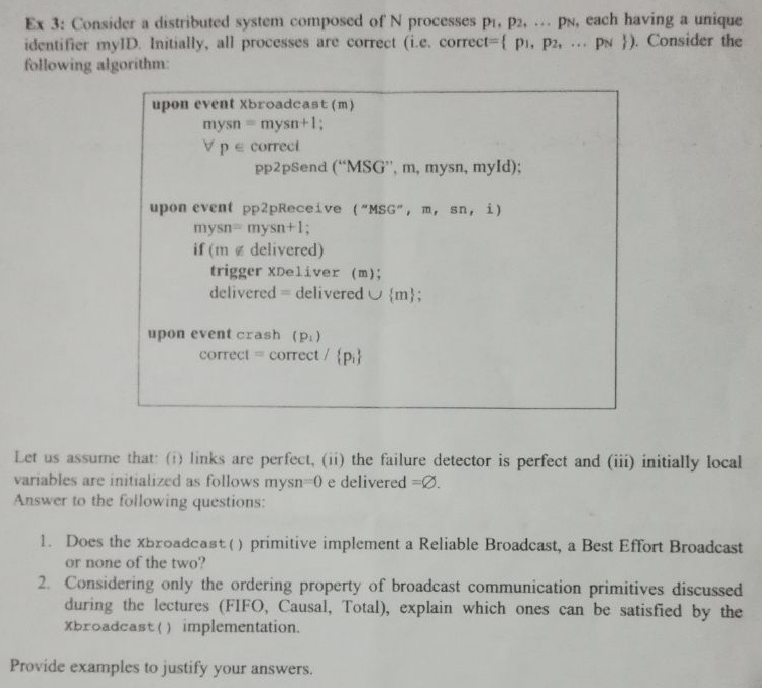
Properties: Liveness is not guaranteed (due to FLP), Omissions delay protocol but don’t block it, Proposers are elected using leader election protocol



1. None, due to Validity property: If correct process p broadcasts a message m, then every correct process (process p) eventually delivers m. Counterexample: process p broadcasts message m with sn=1, then broadcasts message m’ with sn=2. If m’ is delivered before m, m cant be delivered anymore since <sn=1, p> < lastDelivered=2.
2. FIFO is not satisfied, due to FIFO delivery property: If some process broadcasts message m1 before it broadcasts message m2, then no correct process delivers m2 unless it has already delivered m1. Counterexample: process p broadcasts message m with sn=1, then broadcasts message m’ with sn=2. m’ can be delivered before m, since <sn=2, p> > lastDelivered=0.

If FIFO is violated, Causal is not ensured, since: Causal = FIFO + Local Order.

Total order is not satisfied. Counterexample: process p1 broadcasts m1 with sn=1 then broadcasts m2 with sn=2. Process p2 receives m1 before m2, so it delivers m1 before m2. Process p1 receives m2 before m1, so only delivers m2, since <sn=1, p> < lastDelivered=2.



1. Xbroadcast implements a Best Effort Broadcast, but no Reliable Broadcast.

Best Effort: All properties are satisfied:

* 1. Validity is ensured due to the reliable delivery property of the perfect p2p links and the fact that the sender sends the message to every correct process in the system.
  2. No Duplication is ensured due the No Duplication property of perfect p2p links and the assumption of uniqueness of messages (different IDs of messages).
  3. No Creation is ensured due to the No creation property of perfect p2p links.

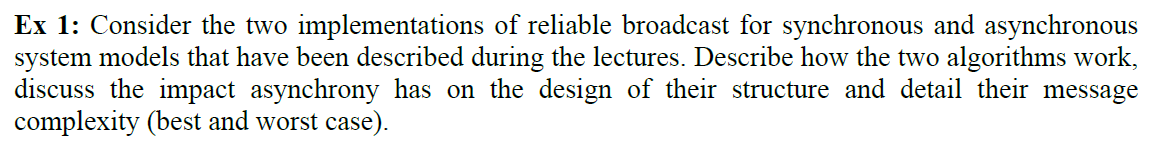
Not Reliable, since a message sent by a faulty process and delivered by only one correct process will not be retransmitted to the remaining correct processes. Therefore, the RB4 Agreement property is not satisfied.

1. FIFO is not satisfied. Counterexample: Correct process p broadcasts m1, then broadcasts m2. The algorithm does not make any ordering specifications. So delivering m2 before m1 is allowed, since both are delivered for the first time.

If FIFO is violated, Causal is not ensured, since: Causal = FIFO + Local Order.

Total order is not satisfied. Counterexample: Correct process p1 broadcasts m1, then broadcasts m2. The algorithm does not make any ordering specifications. So delivering m2 before m1 is allowed, while correct process p2 is delivering m1 before m2. This is possible, since each message is delivered for the first time.

**RELIABLE BROADCAST & SYNCHRONIZATION**



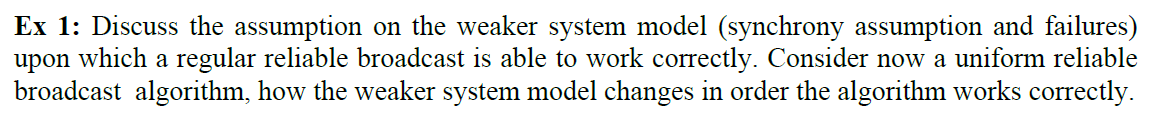
The Reliable Broadcast satisfy the following properties:

1. Validity: If a correct process p broadcasts m, then p eventually delivers m.
2. No duplication: No message is delivered more than once.
3. No creation: If a process delivers a message m with sender s, then m was previously broadcast by process s.
4. Agreement: If a message m is delivered by some correct process, then m is eventually delivered by every correct process.

In synchronous and in asynchronous mode the Best-Effort Broadcast is used. The synchronous model uses the perfect failure detector in addition, since in asynchronous mode no reliable prediction of faulty processes are possible. Therefore, in asynchronous mode, each message will be broadcasted to all other processes. Each delivering message triggers a retransmitting to all other processes. The algorithm is called eager, in sense that it retransmit every message. Due to that fact the complexity is in best case equal to the complexity in worst case. N (number of processes) BEB messages are created for each RB message. For each BEB delivery N retransmitting broadcast messages are created.

In comparison the synchronous RB saves locally each receiving message and its sending process. In case of a crash or receiving a message from a faulty process, each message from the respective crashing process is rebroadcast (at least) to all correct processes. In the best case, 1 BEB message is created per 1 RB message. In the worst case of N-1 failures, N-1 BEB messages are created per 1 RB message.

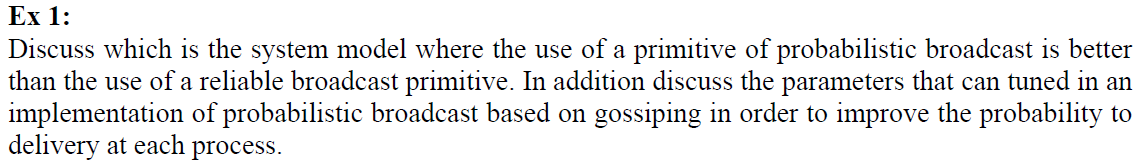
**RELIABLE BROADCAST & SYNCHRONIZATION**



The weaker model gives a PFD to detect crashes. Moreover, synchrony implies upper boundes delays wrt. processing, communication and physical clocks or existence of common global clock. Therefore, the RB implementation in synchronous systems are lazy in the sense that it retransmits only when necessary. Each delivered message and its sending process is saved locally. In case of crashing process or receiving a message from a already crashes process, the messages of the respective process is re broadcasted. In the best case, 1 BEB message per 1 RB message. In the worst case, n-1 BEB messages per 1 TB message (when n-1 failures).

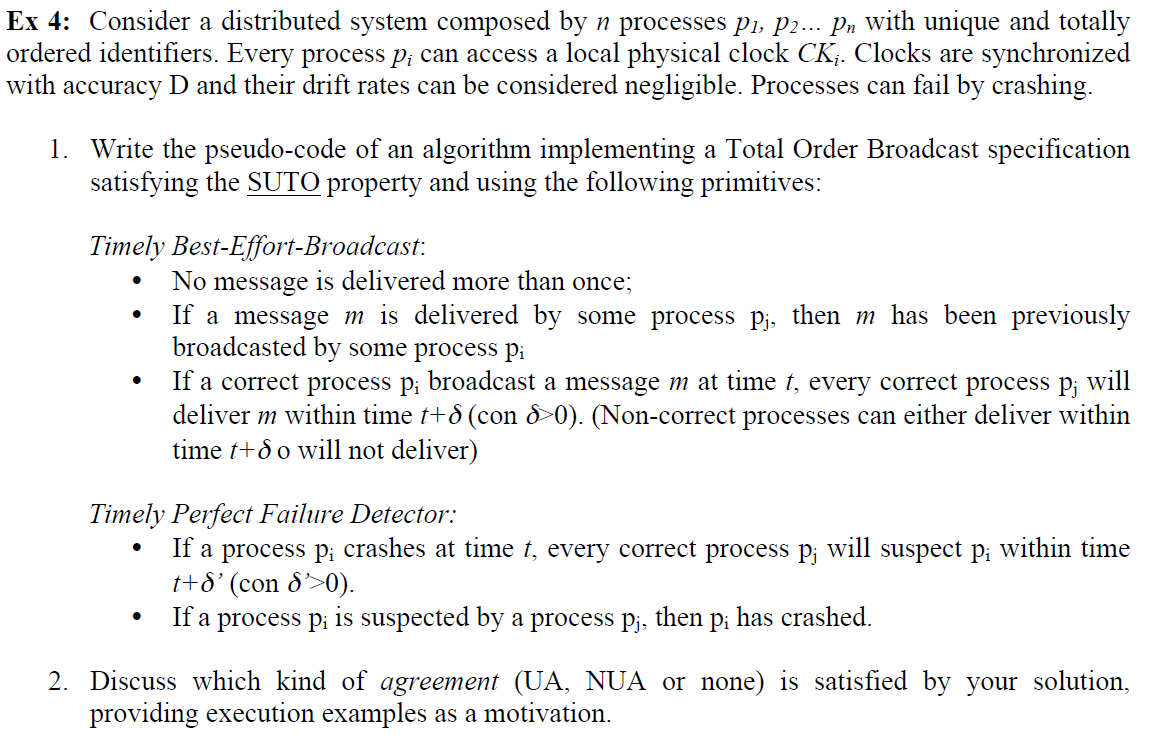
The uniform RB in synchronous systems saves receiving messages and its sending process until a process receives the message m from all correct processes. Then m is delivered, if message m is not yet delivered. There exists an algorithm for asynchronous system when assuming a ‘majority of correct processes’. Without the assumption of majority but partially synchrony, a probabilistic broadcast can be used.

**PROBABILISTIC BROADCAST**

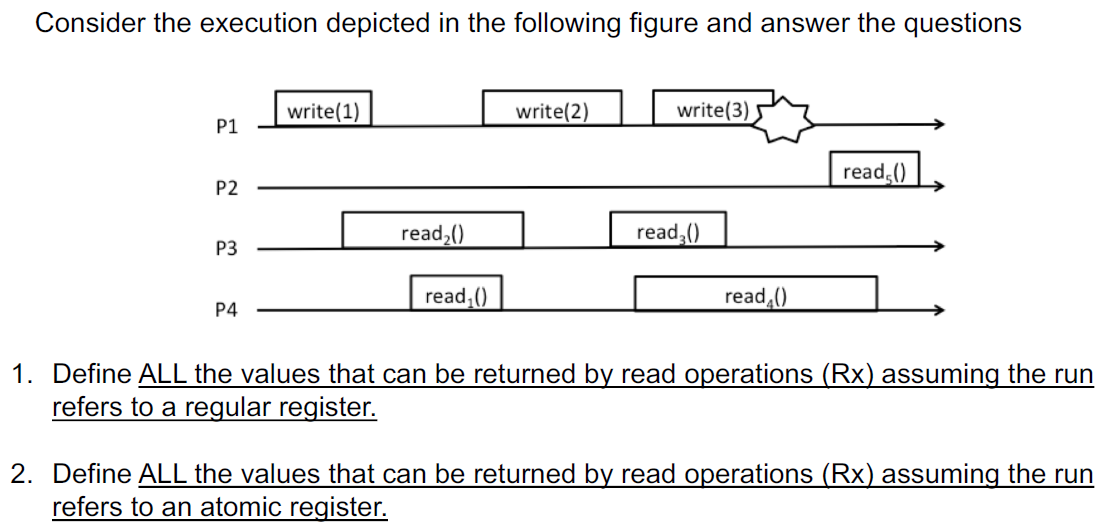


Probabilistic broadcast is better when a uniform reliable broadcast for partially synchronous system (using an eventually perfect failure detector) but without the assumption of a majority of correct processes is needed.

A process sends a message to a set of randomly chosen k processes. With higher k, more processes can be reached in one round to achieve higher reliability (higher probability of delivering each process). The algorithm performs a maximum number of r rounds. With higher r (more rounds), the probability of not reaching a process is reduced, so a higher reliability of the broadcast (higher probability of delivering each process) is achieved.



\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_



1. R1: 1, 2

R2: {}, 1, 2

R3: 1, 2, 3

R4: 2, 3

R5: 2, 3

1. R1: 1, 2

R2: {}, 1, 2

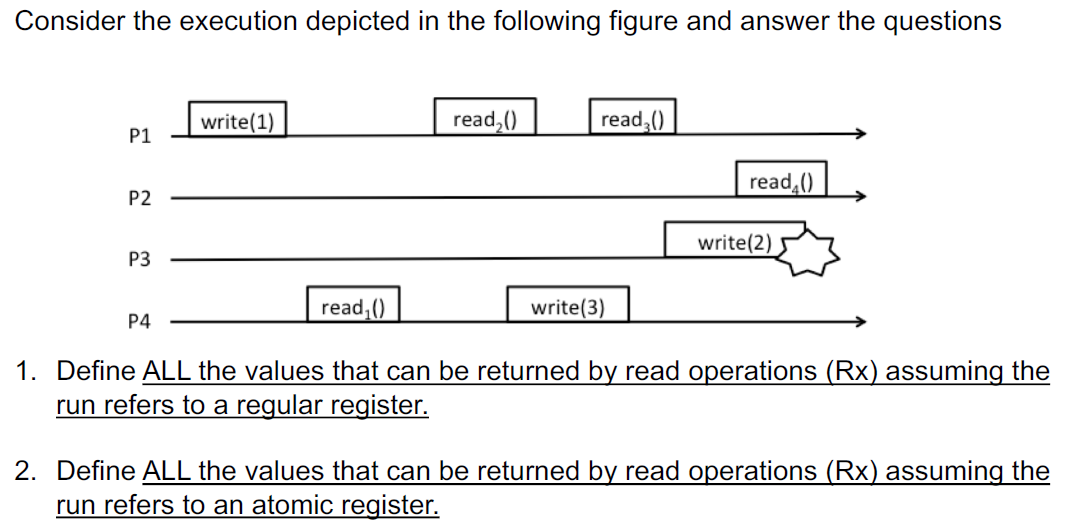
R3: If R1 = 2 OR R2 = 2 🡪 R3 = 2, 3

Else R3 = 1, 2, 3

R4: 2, 3

R5: If R3 = 3 🡪 R5 = 3

Else R5 = 2, 3



1. R1: 1,

R2: 1, 3

R3: 1, 3, 2

R4: 3, 2

1. R1: 1

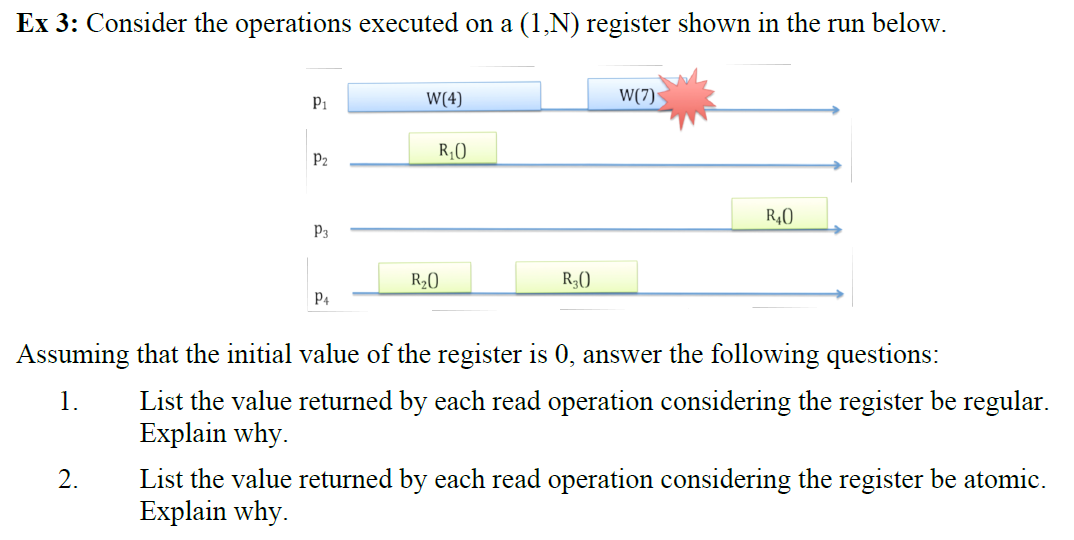
R2: 1, 3

R3: If R2 = 3 🡪 R3 = 3, 2

Else R3 = 1, 3, 2

R4: If R3 = 2 🡪 R4 = 2

Else R4 = 3, 2



1. R1 = 0, 4

R2 = 0, 4

R3 = 0, 4, 7

R4 = 4, 7

1. R1 = 0, 4

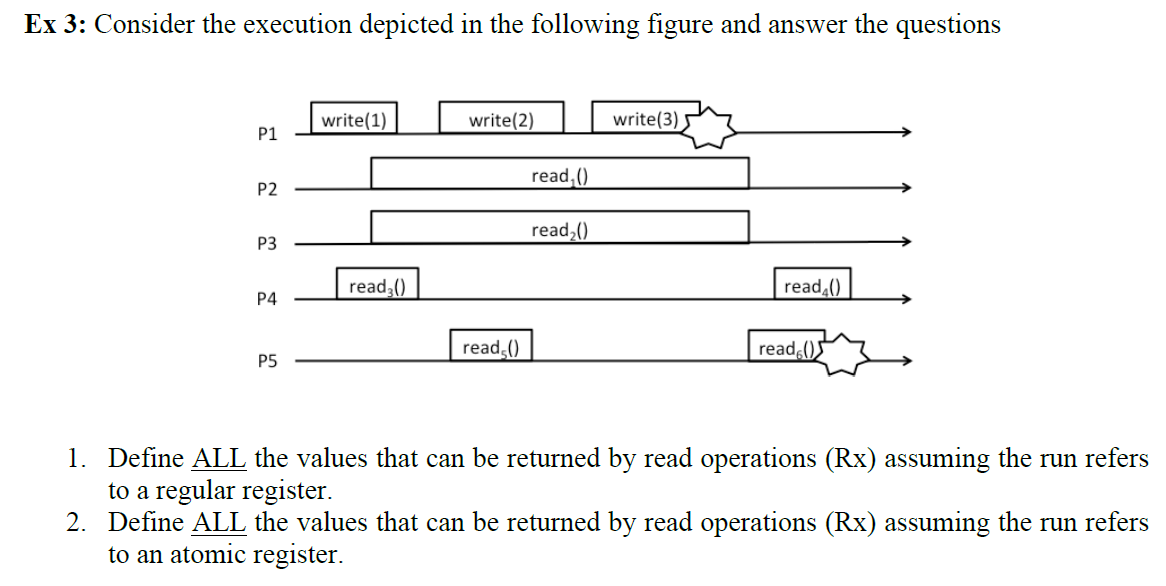
R2 = 0, 4

R3: If R1 = 4 OR R2 = 4 🡪 R3 = 4, 7

Else R3 = 0, 4, 7

R4: If R3 = 7 🡪 R4 = 7

Else R4 = 4, 7



1. R1 = 0, 1, 2, 3

R2 = 0, 1, 2, 3

R3 = 0, 1

R4 = 2, 3

R5 = 1, 2

R6 = {}, 2, 3

1. R1 = 0, 1, 2, 3

R2 = 0, 1, 2, 3

R3 = 0, 1

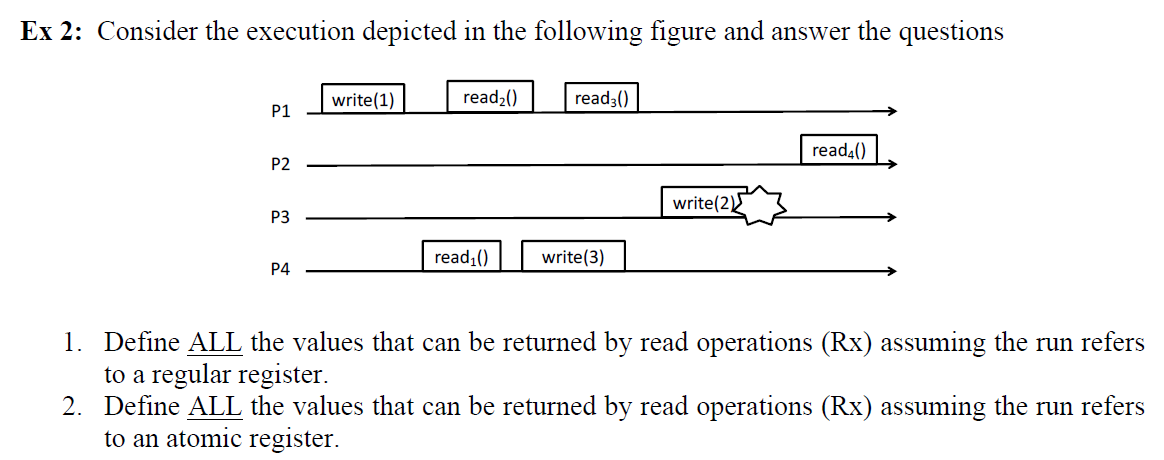
R4: If R1 = 3 OR R2 = 3 🡪 R4 = 4

Else 🡪 R4 = 2, 3

R5 = 1, 2

R6: If R1 = 3 OR R2= 3 🡪 R6 = {}, 3

Else R6 = {}, 2, 3



1. R1 = 1

R2 = 1, 3

R3 = 1, 3

R4 = 3, 2

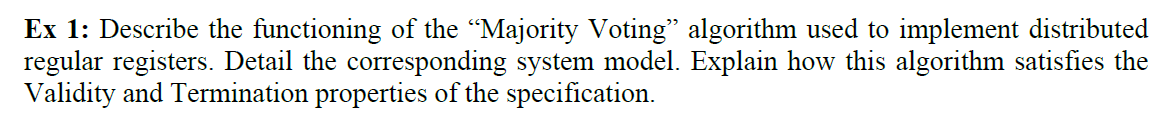
1. R1 = 1

R2 = 1, 3

R3: If R2 = 3 🡪 R3 = 3

Else R3 = 1, 3

R4 = 3, 2



Also called Fail-silent algorithm: “process crashes can never be reliably detected.” (No perfect failure detector).

Assumptions: - N processes whose 1 writer and N readers

- A majority of correct processes

Communication: - PP2P

- Best Effort Broadcast

Idea: - Each process locally stores copy of current value of register

- Each written value is univocally associated to a timestamp

- Writer and reader processes use a set of witness processes, to track last value written

- Quorum: intersection of any two sets of witness processes is not empty

- Majority Voting: each set is constituted by majority of processes

Write: - Broadcast value to be written with new timestamp to all processes

- Each delivering process writes new value if it is latest and resends ACK

- Return / confirm if majority (>N/2) of processes responded with respective timestamp

Read: - Broadcast reading request with reading id

- Each delivering process resends local register value with reading id

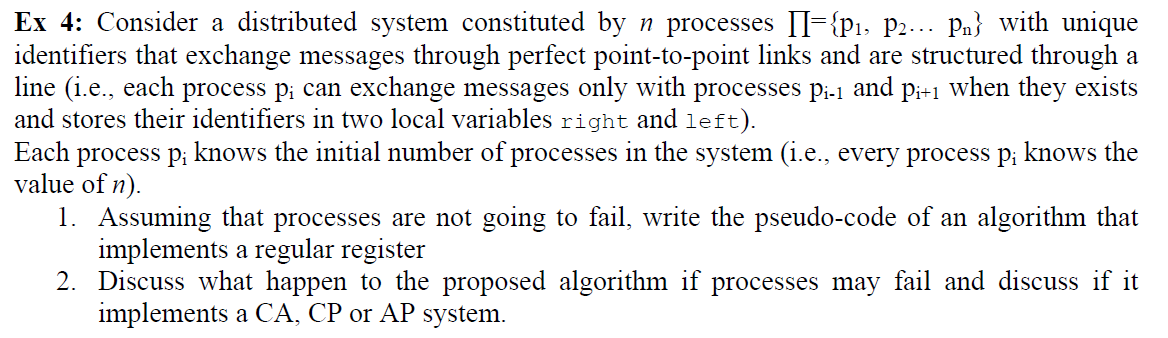
- Save all incoming values in list

- If majority (>N/2) of processes responded, take value of latest timestamp and confirm / return

Correctness: - Termination: from properties of communication primitives and assumption of majority of correct processes

- Validity: from intersection property of quorums

**IMPL. REGULAR REGISTER / LINE / PP2P**



Upon event <Init> do

R = GET\_RIGHT();

L = GET\_LEFT();

Correct = π;

ACK = {};

Val = ⊥;

Writing = False;

Upon event <Write(v)> do

Val = v; // Set own local register value

Trigger pp2pSend(‘WRITE’, v) to L;

Trigger pp2pSend(‘WRITE’, v) to R;

ACK = {}; // Collect ACKs of updated registers

Writing = True;

Upon event <pp2pDeliver(‘WRITE’, v)> from pi do

Val = v;

If pi = R do // If WRITE comes from the right

Trigger pp2pSend(‘ACK’, myID) to R;

Trigger pp2pSend(‘WRITE’, v) to L;

Else do // If WRITE comes from the left

Trigger pp2pSend(‘ACK’, myID) to L;

Trigger pp2pSend(‘WRITE’, v) to R;

Upon event <pp2pDeliver(‘ACK’, id)> from pi do

If Writing do // If you are the process that sent the write order

ACK = ACK U {id};

Else if pi = R do

Trigger pp2pSend(‘ACK’, id) to L;

Else

Trigger pp2pSend(‘ACK’, id) to R;

When correct = ACK do

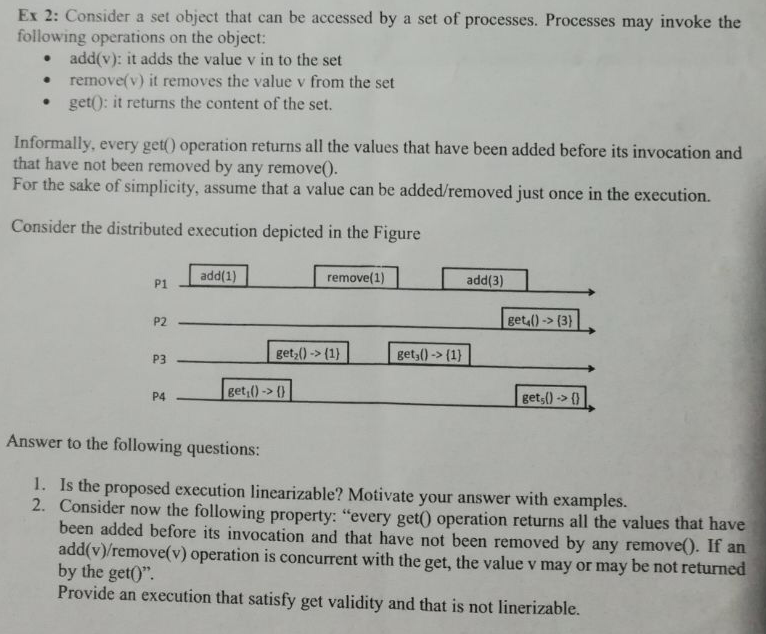
// Writing = False;

Trigger WriteReturn();

No crashes: CA system, since all processes connected and available & consistent

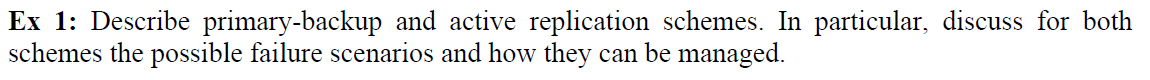
A crash disconnects system: AP system

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_



1. S = {get2(), add(1), get2(), get3(), rm(1), get5(), add(3), get4}
2. P1: <--------------ADD(1)---------------->

P2: <--get(1)--> <--get({})--->



Primary-backup:

Primary: Receives invocations from clients and sends back the answers.

Backup: Interacts with primary. Is used to guarantee fault tolerance by replacing a primary when crashes

1. When update messages are received by backups, they update their state and send back the ack to primary
2. Primary waits for ack message from each correct backup and then sends back the answer, res, to the client

Guarantee of linearizability due to the order in which primary receive clients’ invocations define the order of operation on the object.

Failure scenarios (in all cases new leader need to be elected):

1. Primary fails after client receives answer.
2. Client does not receive response due to pp2p. If response is lost, client retransmits request after timeout. New primary will recognize the request re-issued by client and sends back the result without updating the replicas.
3. Client receives answer and everything is all right, but a new leader is elected.
4. Primary fails before sending update messages.
5. Client does not get answer and resends requests after timeout. New Primary will handle request as new.
6. Primary fails after sending update messages and before receiving all ack messages.
7. When primary fails, elect new leader among all correct replicas (guarantee atomicity due to update received either by all or by no one.

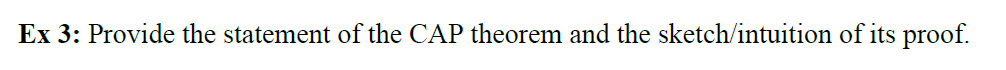
Active replication:

There is no coordinator. All replicas have same role. Each replica is deterministic, i.e. if any replica starts from same state and receives same input, they produce same output. As matter of fact clients will receive same response one from each replica. Active replication does not need recovery action upon failure of a replica.

Guarantee linearizability by:

1. Atomicity: If a replica executes an invocation, all correct replicas execute same invocation.
2. Ordering: (At least) no two correct replicas have to execute two invocations in different order.

* TOTAL ORDER Broadcast is needed (including the clients)



and present an example of a system which is CA, CP and AP.

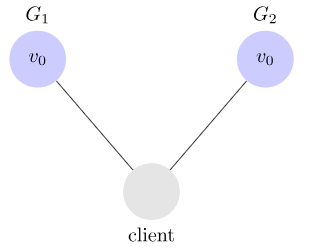
The CAP theorem states that a distributed system cannot simultaneously be consistent, available, and partition tolerant.

Consistency: Any read operation that begins after a write operation completes must return that value, or the result of a later write operation.

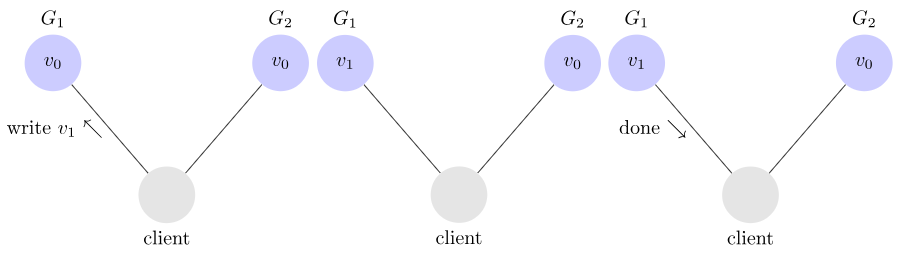
Availability: Every request received by a non-failing node in the system must result in a response.

Partition Tolerance: The network will be allowed to lose arbitrarily many messages sent from one node to another.

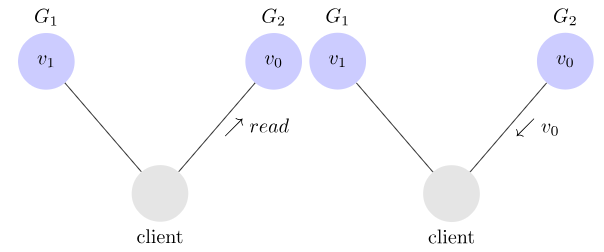
Proof: Assume for contradiction that there does exist a system that is consistent, available, and partition tolerant. A sketch of simple partitioned system:



Next, we have our client request that v1 be written to G1. Since our system is available, G1 must respond. Since the network is partitioned, however, G1 cannot replicate its data to G2.



Next, we have our client issue a read request to G2. Again, since our system is available, G2 must respond. And since the network is partitioned, G2 cannot update its value from G1. It returns v0.



G2 returns v0 to our client after the client had already written v1 to G1. This is inconsistent.

CA: Examples: Single-site databases, Cluster databases, Fiefdoms

Traits: 2-phase commit, Cache validation protocols

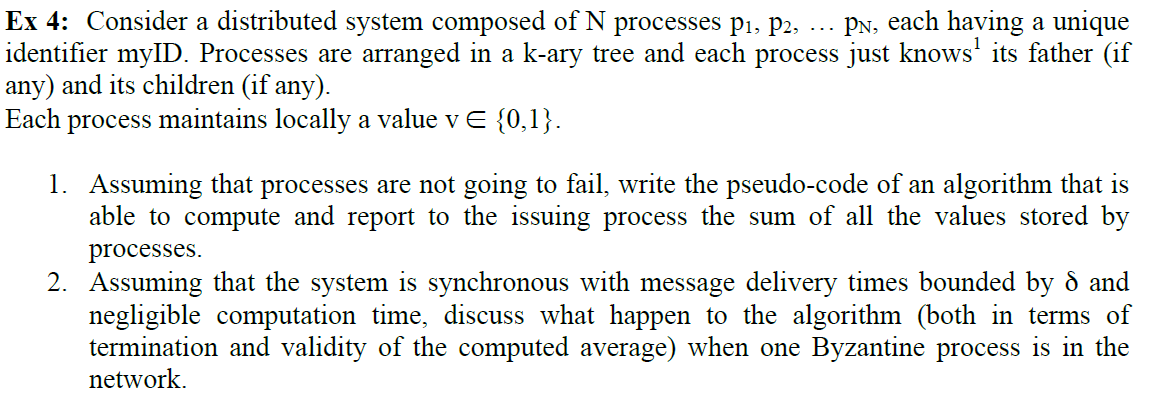
CP: Examples: Distributed databases, Distributed locking, Majority protocols

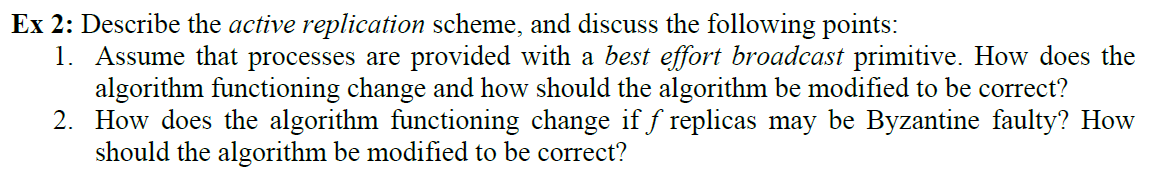
Traits: Pessimistic locking, Make minority partitions unavailable

AP: Examples: Coda, Web caching, DNS

Traits: expirations/leases, Conflict resolution

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_





Active replication:

There is no coordinator. All replicas have same role. Each replica is deterministic, i.e. if any replica starts from same state and receives same input, they produce same output. As matter of fact clients will receive same response one from each replica. Active replication does not need recovery action upon failure of a replica.

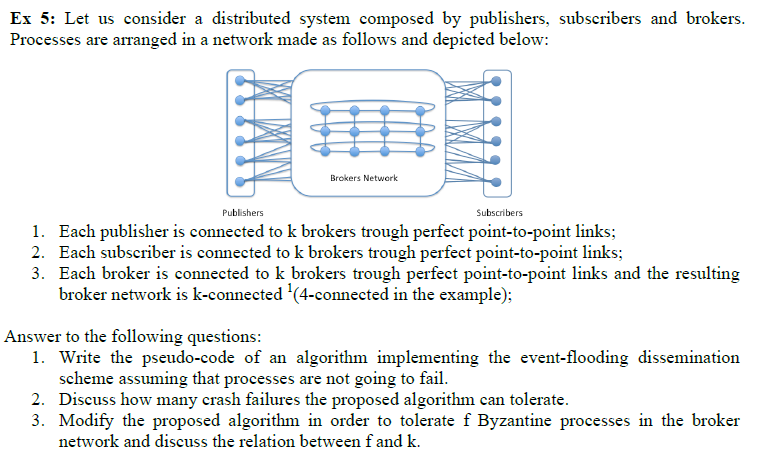
Guarantee linearizability by:

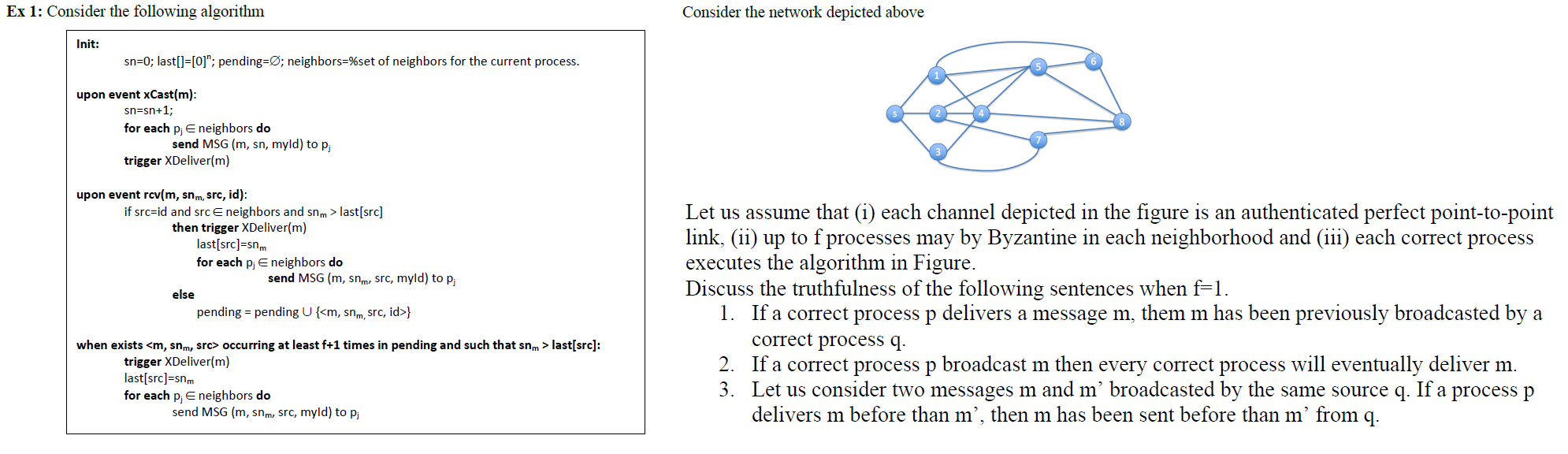
1. Atomicity: If a replica executes an invocation, all correct replicas execute same invocation.
2. Ordering: (At least) no two correct replicas have to execute two invocations in different order.

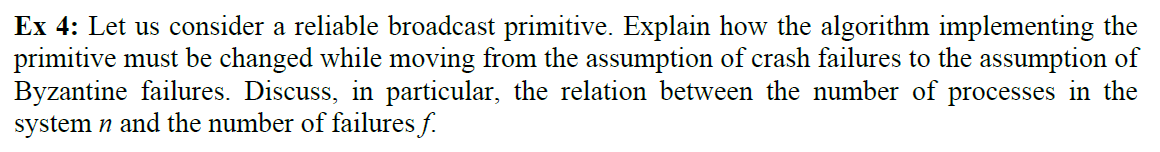
* TOTAL ORDER Broadcast is needed (including the clients)

A BEB can be used to broadcast the invocations to all replicas. The delivering event of each replica has to be extended with the execution of the operation sent by the process. After successful execution the replicas send back an ACK to the process to signalize their response.

Consider N = #replicas. Process can return its response when (N+f)>2 ACKs are received. So a majority over f Byzantines are achieved.







The Reliable Broadcast algorithm saves locally each message and the respective sending process. In case of a crash of or of incoming message from a crashing process, each message is rebroadcasted to all processes to achieve the Agreement property. The algorithm in synchronous is lazy in the sense that it retransmits only when necessary.

The Byzantine Reliable Broadcast uses Authenticated PP2P Links to ensure the authenticity and correctness of a sending process. Each delivering process sends an echo to all processes to signalize the delivering of a message. Each delivering echo is also saved locally. As soon as the number of processes which sent message m to a process (majority over Byzantines) or a process delivered more than f READY messages, the process rebroadcasts a READY message to all other processes. As soon as a process delivers more than 2f READY deliveries the message can be finally delivered.

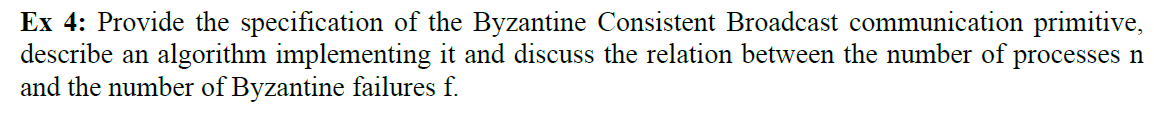
In total each message broadcasting is followed by an SEND, ECHO and READY phase. In comparison to the RB without Byzantines the messages are double checked and messages have to be delivered by a threshold of number of processes to guarantee the Totality property.

The goal is to achieve majority voting over f Byzantine processes. The number of correct processes is given by N – f (N: #processes in system):

N – f > 2f Threshold to achieve majority over Byzantine processes

N > 3f #Number of processes in system more than 3 times number of Byzantines

🡪 Number of votes must be greater than (N+f)/2: If N > 3f is guaranteed, threshold (N+f)/2 > (3f+f)/2 = 2f (more than twice the number of number of Byzantines)



Byzantine Consistent Broadcast Specification:

- Uses Authenticated Perfect Links

- Events: 1. Request: Broadcast message m to all processes. Executed only by process s.

2. Indication: Deliver message m broadcast by process p.

- Properties: 1. Validity: If a process p broadcasts a message m, then every correct process eventually delivers m.

2. No duplication: Every correct process delivers at most one message.

3. Integrity: If some correct process delivers a message m with sender p and process p is correct, then m was previously broadcast by p.

4. Consistency: If some correct process delivers a message m and another correct process delivers a message m’, then m = m’.

A message is broadcast to each process in the system. Each delivered message triggers an rebroadcast of an ECHO signal. As soon as more than (N+f)/2 ECHOs of the same message from different processes are delivered, the actually message m is delivered.

The goal is to achieve majority voting over f Byzantine processes. The number of correct processes is given by N – f (N: #processes in system):

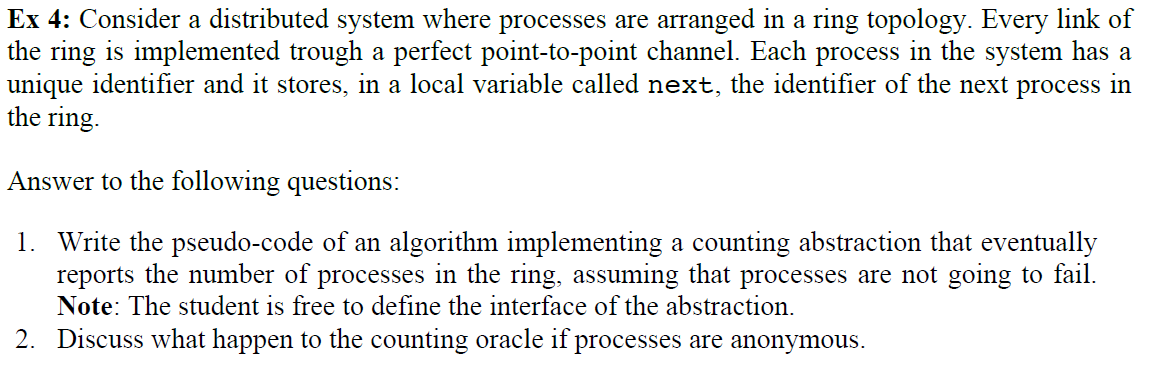
N – f > 2f Threshold to achieve majority over Byzantine processes

N > 3f #Number of processes in system more than 3 times number of Byzantines

🡪 Number of votes must be greater than (N+f)/2: If N > 3f is guaranteed, threshold (N+f)/2 > (3f+f)/2 = 2f (more than twice the number of number of Byzantines)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**IMPL. COUNTER / RING / PP2P**



Upon event <Init> do

Next = P(i+1) mod n

Count = 0;

Upon event <Start()> do

If Count = 0 do

Count = 1;

Trigger pp2pSend(‘COUNT’, myID, Count) to Next;

Else do // If already counted

Trigger Count(Count);

Upon event <pp2pDeliver(‘COUNT’, id, c)> do

If id = myID do

Count = c;

Trigger Count(Count);

Else do

c = c +1;

Trigger pp2pSend(‘COUNT’, id, c);

🡪 Infinite loop

!!! Synchronous 🡪 time-bounded

