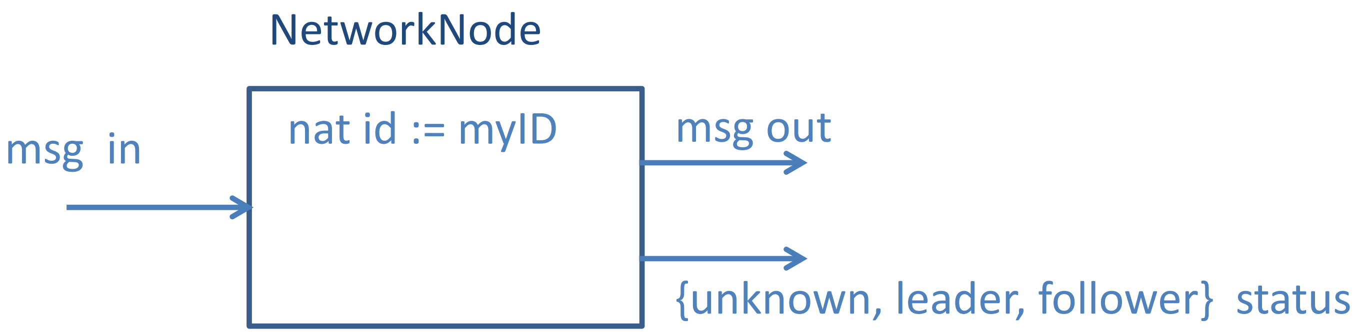
CS:4980
Foundations of Embedded Systems

The Asynchronous Model
Part III

Asynchronous Coordination Protocols

- In the Asynchronous Model, coordination between processes is often necessary
- Algorithms for solving coordination problems cannot assume processes proceed in lock-step rounds (as in the synchronous model)
- This imposes unique design challenges for coordination protocols
- We will see a few next

Leader Election



Recall: Several network nodes elect a unique node as a leader

- Exchange messages to find out which nodes are in network
- Output the decision using the variable status

Requirements:

- Eventually every node sets status to either leader or follower
- Only one node sets status to leader

Asynchronous Leader Election

Asynchronous network

- Channel models directed network link
- If there is a channel/link between nodes M and N, then synchronization on this channel allows M to send a message to N

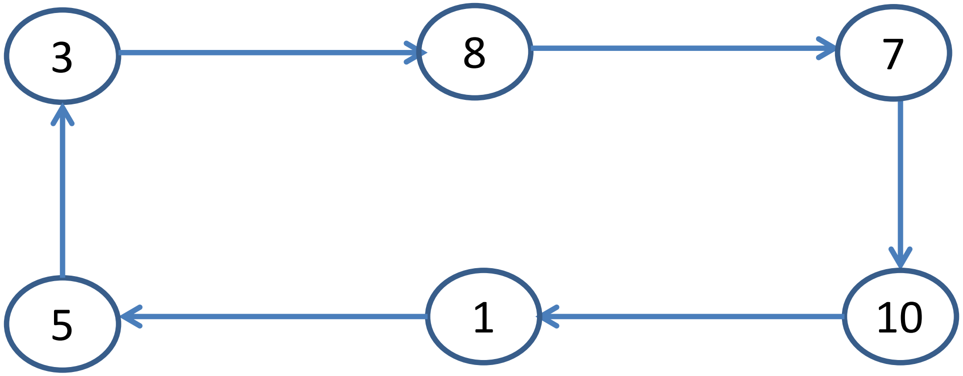
Key challenge (wrt synchronous case)**:** no notion of global round

- Synchronous solution strategy (executing protocol for k rounds implies that message has traveled k hops) does not work here!

Simplification assumption: processes are connected in a unidirectional ring

- Protocols for general topologies exist, but are more complex

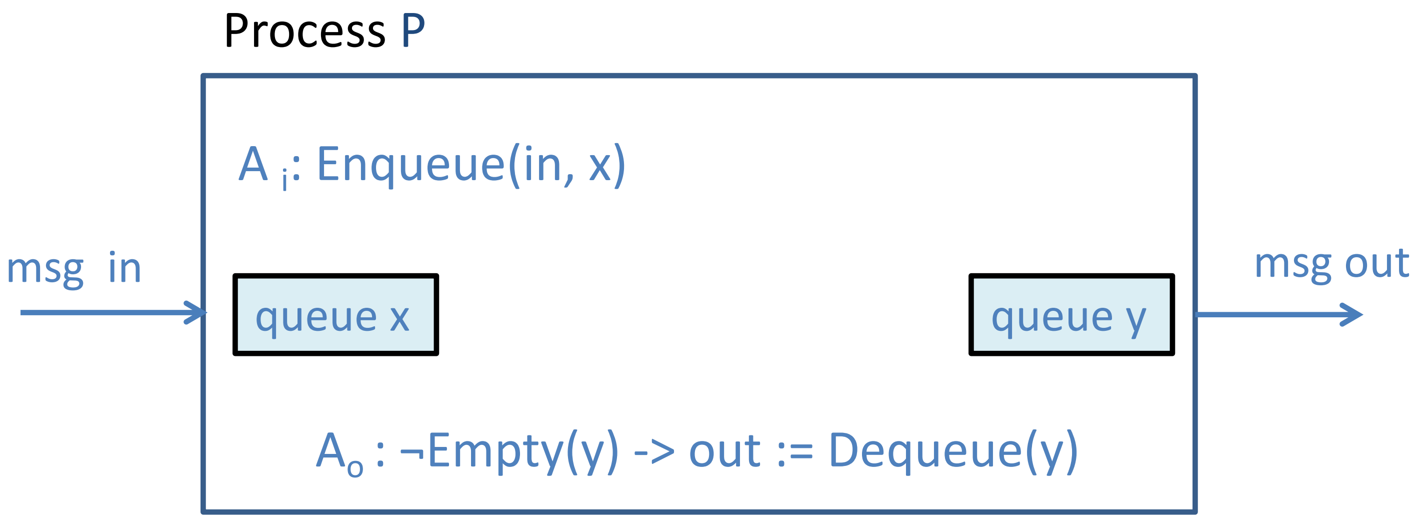
Sample Asynchronous Ring Network



Setting:

- Each process has a unique identifier
- A process does not know the size of the ring (number of processes)
- Execution model is asynchronous
- No failures: each process executes its protocol faithfully

Asynchronous Execution in a Ring



One step in the execution of the system is either

- a step local to one process, or
- a communication step that transfers message
  - from front of output queue y of P
  - to back of input queue x of P's right neighbor

Adapting Synchronous Algorithm

Flooding Algorithm

- Set variable id to MyID, and initialize output queue y to contain id
- Local step/task
  - Remove a value v from queue x
  - If v > id, then change id to v, and enqueue v in queue y
- When should a process stop and decide?
  - If v equals id !
  - This would imply that MyID has traversed the entire ring
- What is an upper bound on the number of messages exchanged?
  - Quadratic, $O(N^2)$, where N is number of processes

Improved Algorithm

1) Set variable id to MyID, and initialize output queue y to contain id, which will be communicated to right neighbor

2) When you receive a value from left neighbor, store it in state variable id1, and also relay it right neighbor (by adding it to output queue y)

3) Receive another value from left neighbor, call it id2
   - id = your value, id1 = left neighbor, id2 = left-left neighbor

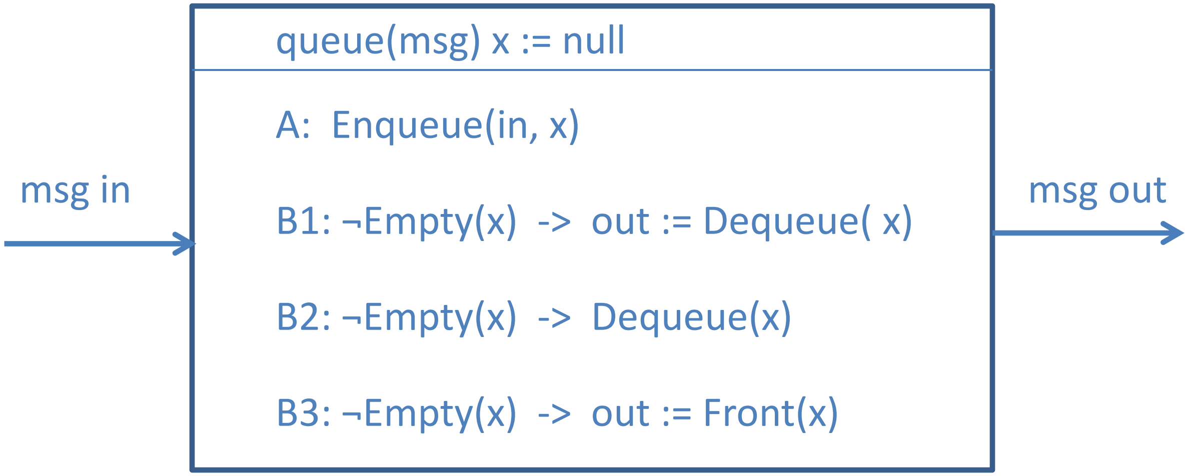
4) If id1 is the max of these three values, set id to id1, and repeat steps 2 and 3 above
   - Continue to next phase as active, but with different identifier

5) If not, then decide to be a follower: continue as a passive participant
   - Do not generate any new messages, just relay messages in input queue to output queue

Algorithm Properties

- Actual execution proceeds asynchronously
  - Messages are processed at arbitrary times
  - Different processes may be executing different phase
- The process that becomes leader need not be the one with the highest original identifier
- In each phase, each process sends only 2 messages
- Among processes active during a phase, if a process continues to next phase as active, then its left neighbor cannot stay active (why?)
- At least one and at most half processes continue to next phase
  - Construct scenarios for these two extremes
  - For a ring of N processes, at most log N phases, so a total of O(N log N) messages
  - Matching lower bound: cannot solve leader election in a ring while exchanging fewer messages

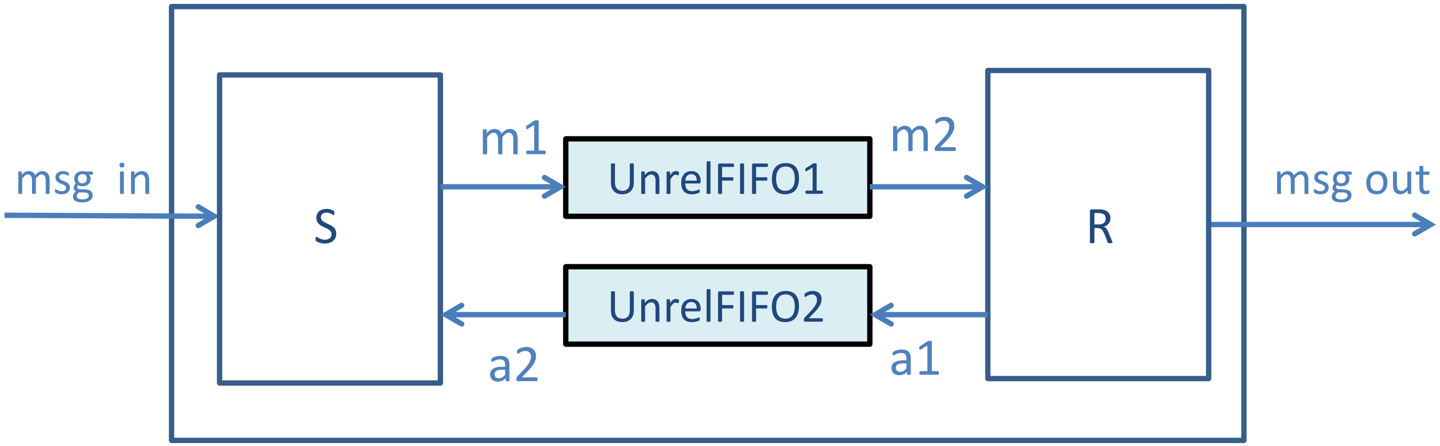
Unreliable FIFO



Models a link that may lose messages and/or duplicate messages

How to implement a reliable FIFO link using unreliable ones?

Reliable Transmission Problem



Design Asynchronous processes S and R so that sequence of messages received on channel in coincides with sequence of messages delivered on channel out

Alternating Bit Protocol

How can the sender S be sure that receiver R got a copy of the message in the presence of message losses?

- S must repeatedly send a message
- R must send back an acknowledgement, and do so repeatedly

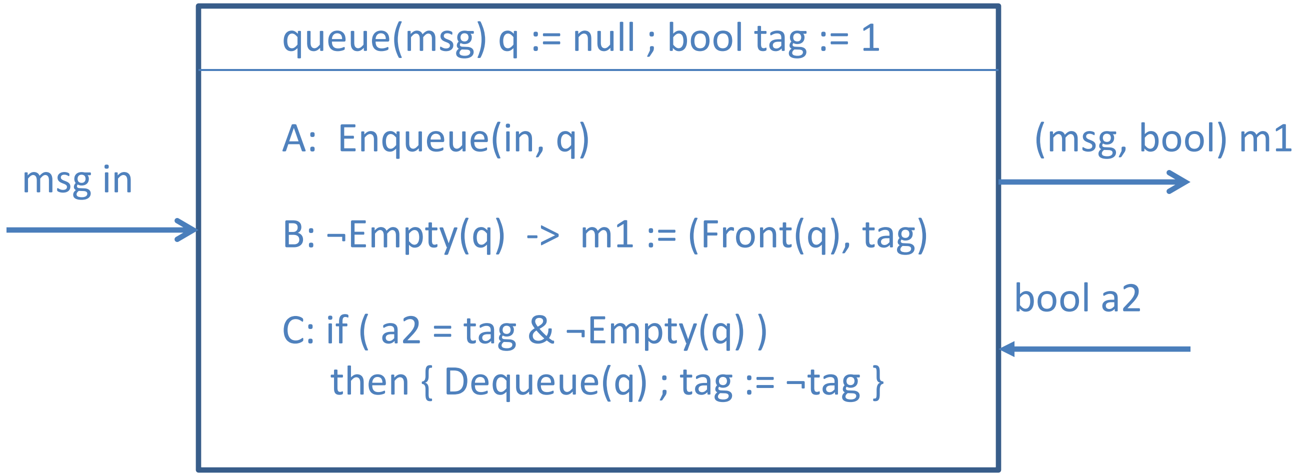
How can the receiver R distinguish between a duplicated/repeated copy and a fresh message?

- Each message must be tagged with extra bits

Alternating bit protocol:

- **Key insight:** tagging each message as well as acknowledgement with a single bit suffices
- Both S and R keep a local tag bit
- if the tag of incoming message matches with the local tag, message is considered fresh, and local tag is toggled

ABP Sender



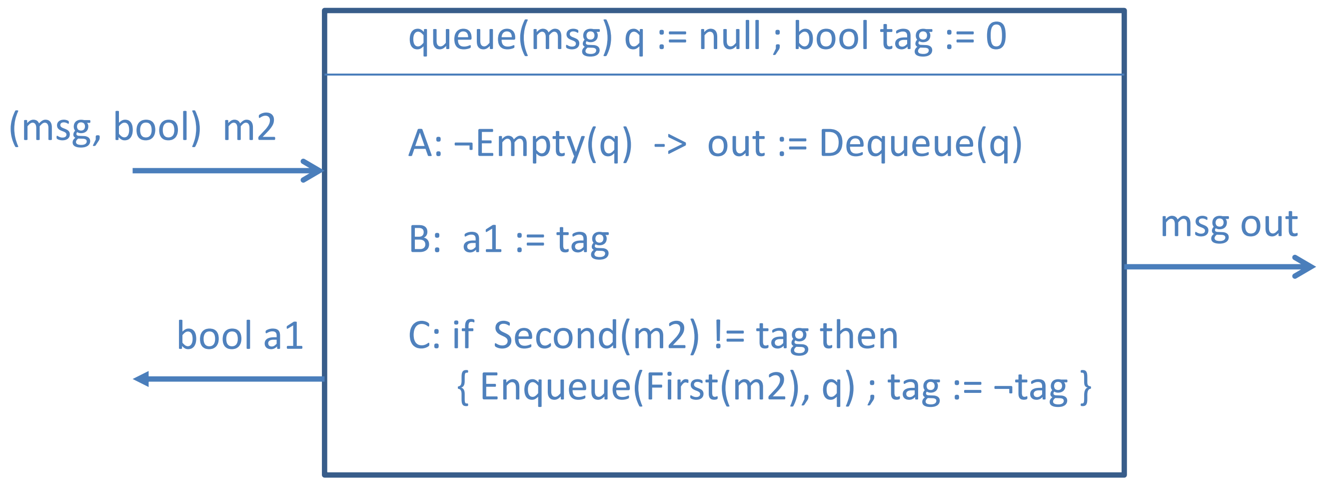
Task A: Store incoming messages in queue q

Task B: Transmit message at front of queue q tagged with local tag

Do not remove the message: this ensures it is transmitted repeatedly

Task C: If ack a2 matches tag, then message successfully delivered; so remove it from q, and flip tag

ABP Receiver



Task A: Transmit outgoing messages from queue q to output channel out

Task B: Transmit local tag as acknowledgement on channel a1

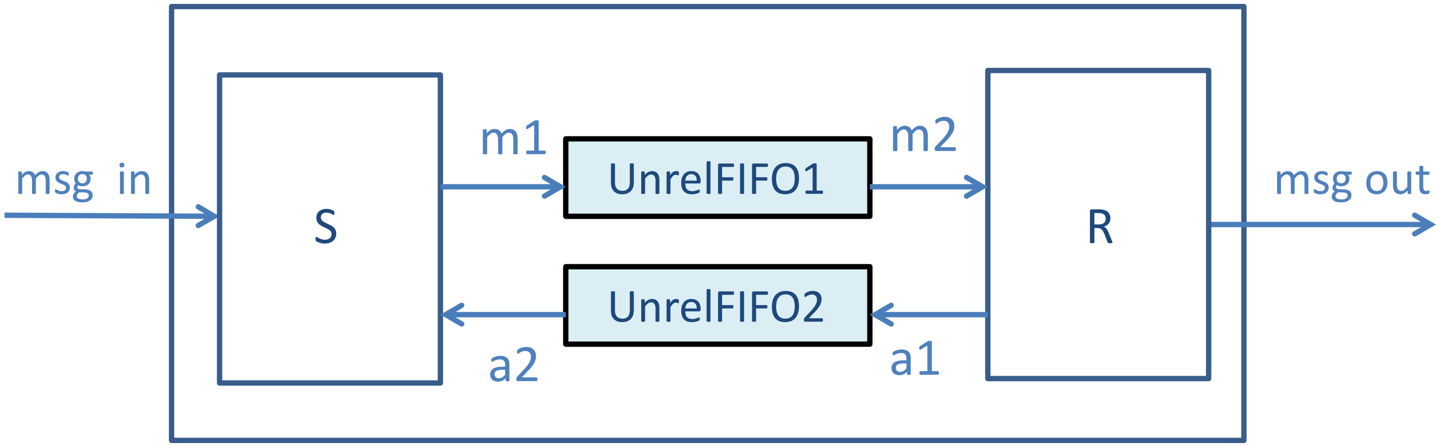
Note: Same acknowledgement is potentially transmitted repeatedly

Task C: If tag of incoming message (Second(m2)) differs from local tag, then message is new; so add message to q and flip tag

ABP Sample Execution

- Initially S.tag = 1 and R.tag = 0
- Suppose S receives a message m to be delivered
- S repeatedly sends (m,1) over unreliable link
- Eventually, R gets at least one, maybe multiple, copies of (m,1)
- Meanwhile, R is sending 0, possibly multiple times, but all these acknowledgements are ignored by S for a while
- When R gets (m,1) the first time, it stores m in its queue q (and this message will then eventually be transmitted on out), and sets tag to 1
- Duplicate versions of (m,1) are ignored by R
- R repeatedly sends the acknowledgment 1 over unreliable link
- Eventually, S gets at least one ack = 1, and then, it removes m from input queue, and sets its tag to 0
- Duplicate versions of ack = 1 are ignored by S
- Input messages received by S are queued up in S.q
  S repeats the cycle by sending next message m' along with tag 0

ABP Variations



- Suppose unreliable link can lose messages, but is guaranteed not to duplicate a message, can we simplify the protocol?
- Suppose unreliable link can also reorder messages (in addition to losing and duplicating messages), how should we modify the protocol to ensure reliable transmission?

Consensus

Each process starts with an initial preference value, known only to itself

Goal of coordination: exchange information and arrive at a common decision value

Classical example: Byzantine Generals Problem communicating by messengers to decide on whether or not to attack

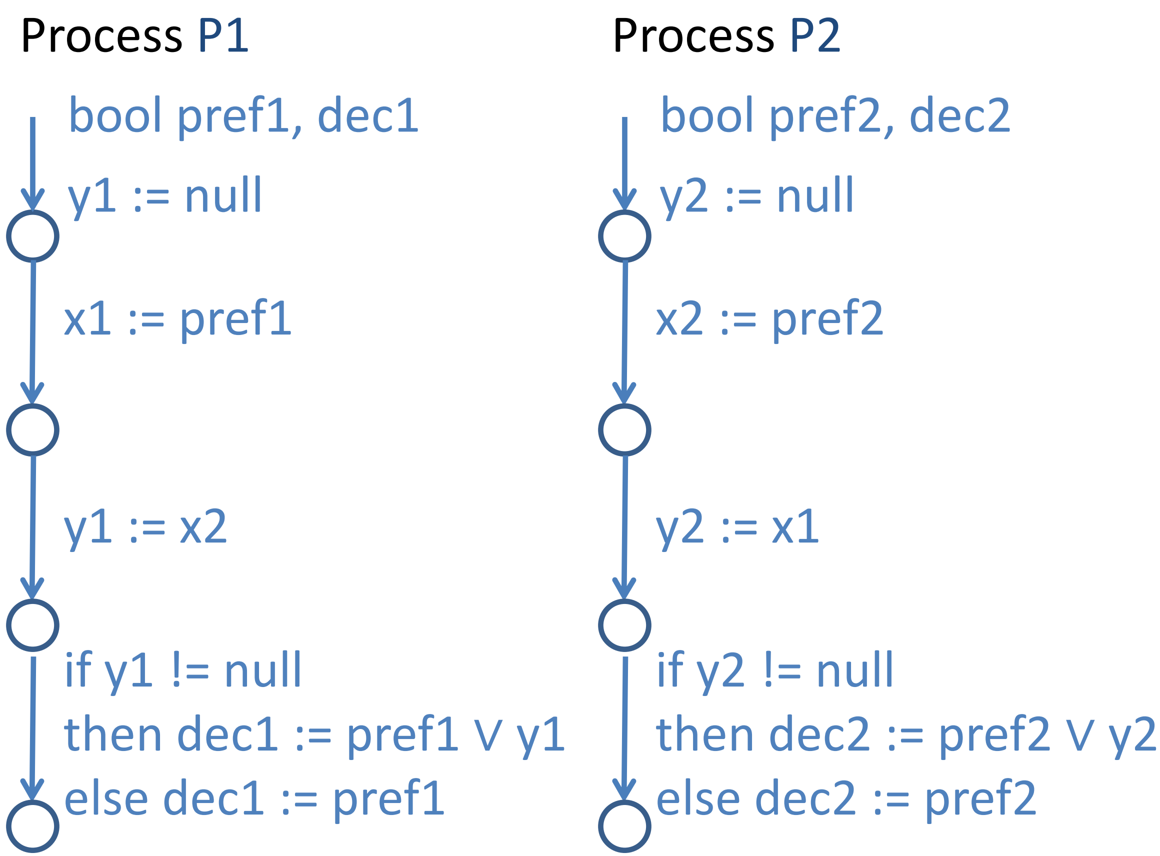
Our focus: Two processes P1 and P2 with Boolean preferences, and communicating by shared memory

P1 and P2 start with initial Boolean preferences v1 and v2, and arrive at Boolean decisions d1 and d2 so that

1. *Agreement*: d1 must equal d2
2. *Validity*: The decision value must equal either v1 or v2
3. *Wait-freedom*: At any time, if only one process is executed repeatedly, it eventually reaches a decision (does not have to wait for the other, and thus, is fault-tolerant)

First Attempt at Solving Consensus

AtomicReg { 0, 1, null } x1 := null ; x2 := null



Write your value in a shared var, read other's value, decide on OR of the values; but if the other has not written yet, choose your own initial value

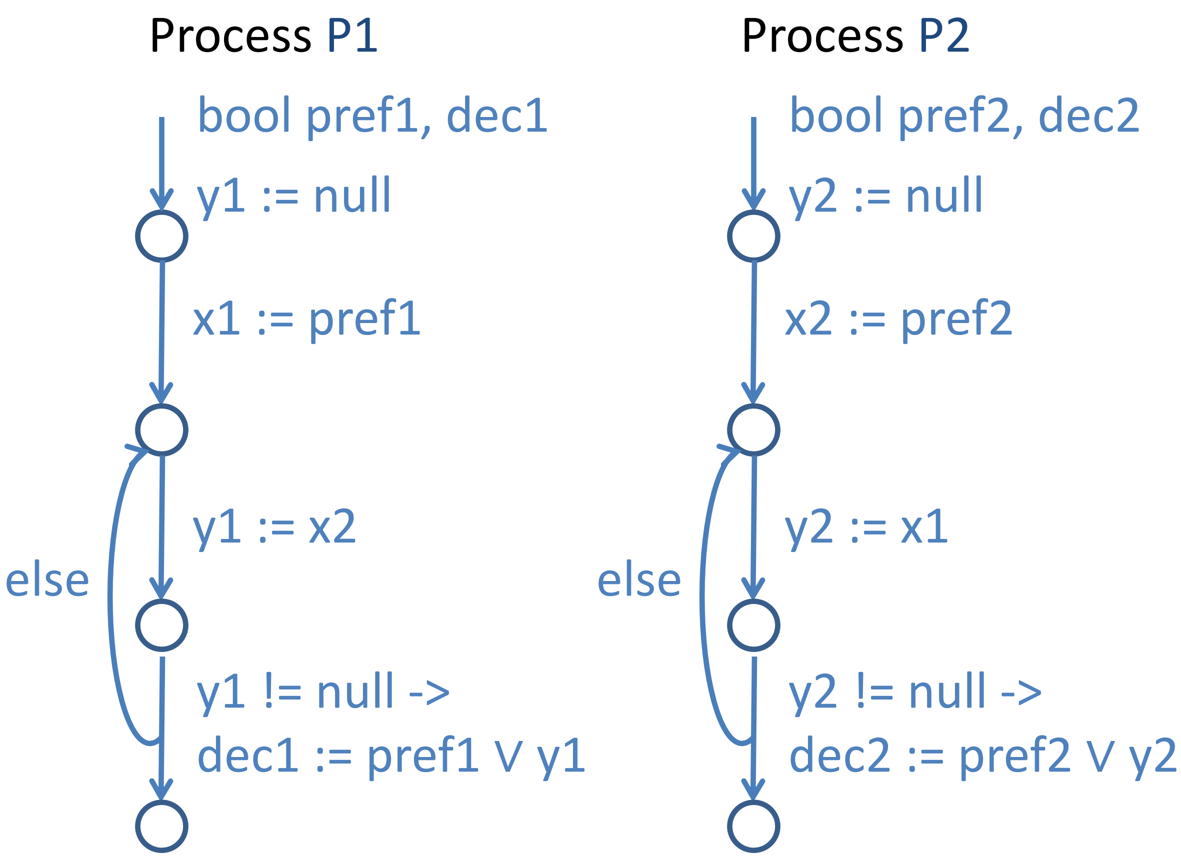
Agreement?

Validity?

Wait-freedom?

Second Attempt at Solving Consensus

AtomicReg { 0, 1, null } x1 := null ; x2 := null



Write your value in a shared var, read other's value, decide on OR of the values; but if the other has not written yet, read again

Agreement?

Validity?

Wait-freedom?

Solving Consensus

Solving consensus using only atomic registers is impossible!

- Primitives of read and write are too weak to achieve desired coordination while satisfying all 3 requirements

Intuitive difficulty:

- When a process writes a shared variable, it does not know whether the other process has read this value, so cannot decide right away
- When a process reads a shared variable, it needs to communicate to other process that it has seen this value, so needs to continue

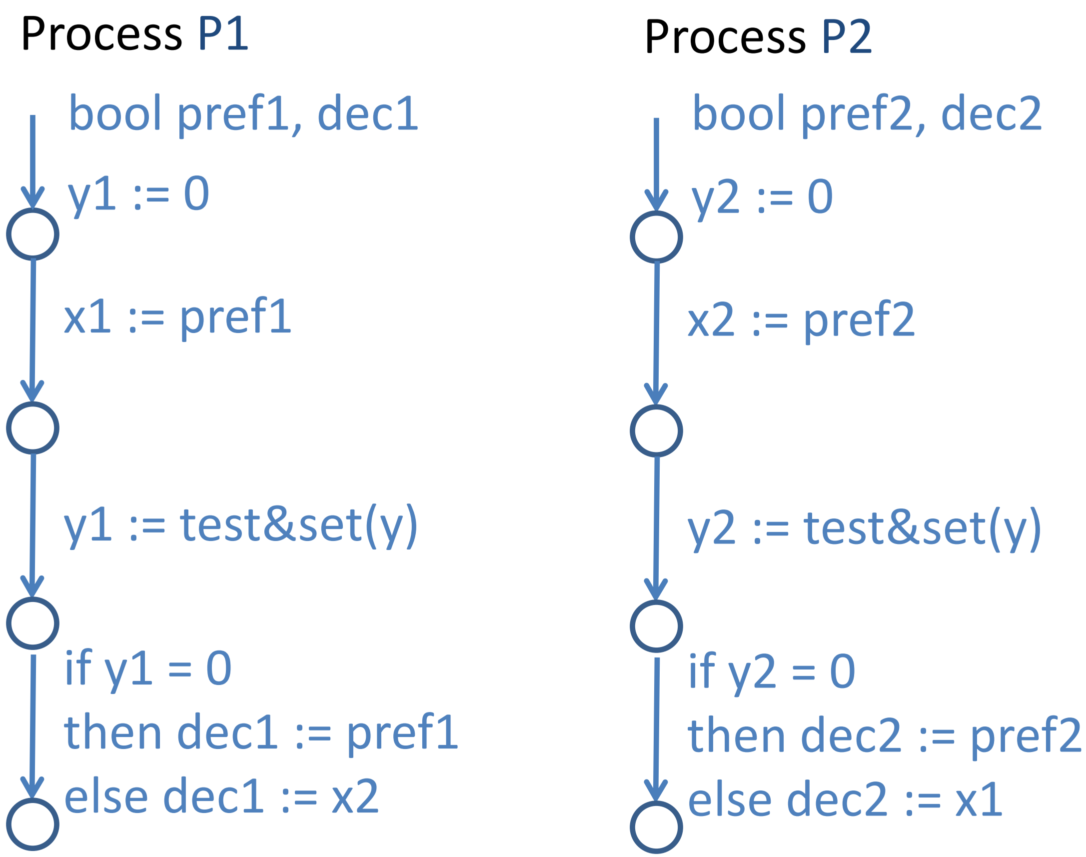
Byzantine Generals Problem: Coordination is impossible

- Sending a message, and receiving a message are similar to write and read operations

Solution: Use stronger primitives such as Test&Set registers

Consensus using Test&Set Register

AtomicReg bool x1, x2 ; Test&SetReg y := 0



Write your value in a shared var; execute test&set; if you win, choose your own initial value, else read other's preference as decision value

Agreement?

Validity?

Wait-freedom?

Credits

Notes based on Chapter 4 of

Principles of Cyber-Physical Systems
by Rajeev Alur
MIT Press, 2015