CS:4980
Foundations of Embedded Systems

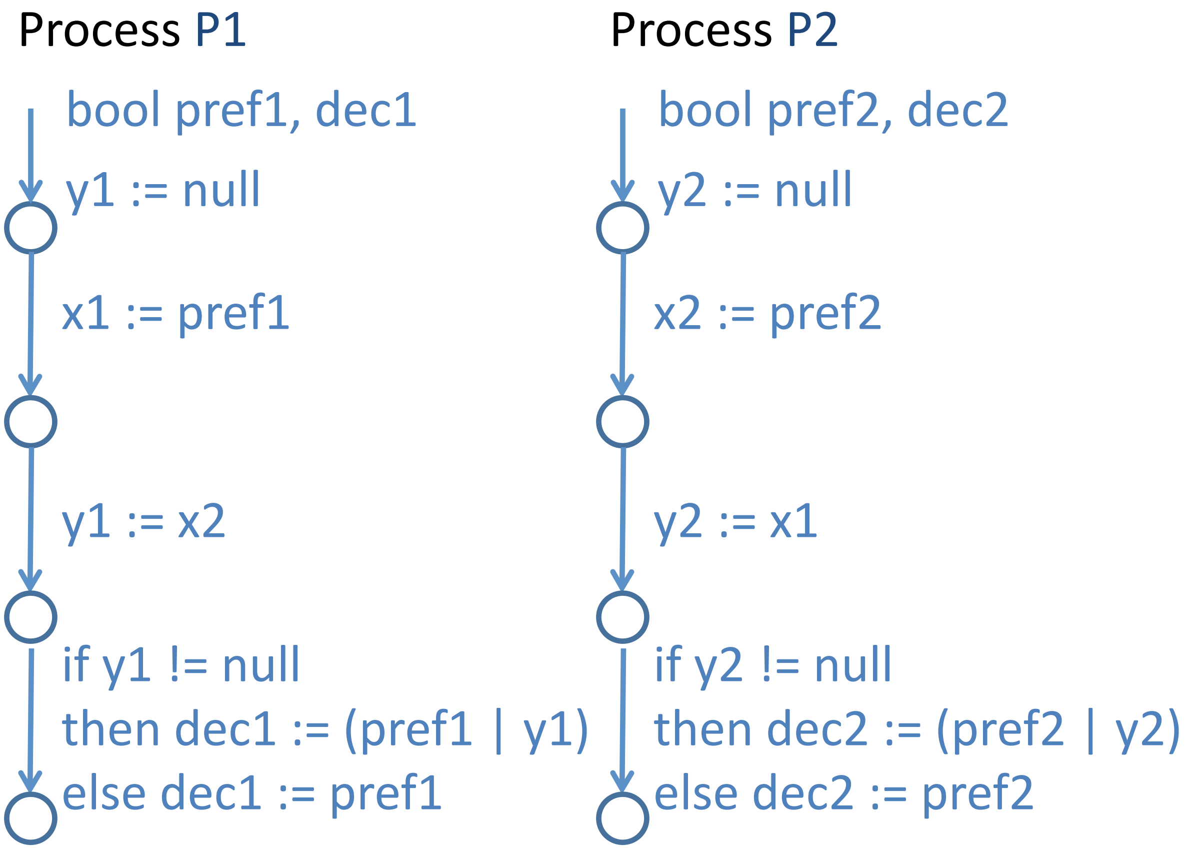
The Asynchronous Model
Part III

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 with Boolean preferences, and communicating by shared memory
- ☐ Processes 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, tolerant to failures)

First Attempt at Solving Consensus

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



Process P1

```
bool pref1, dec1
y1 := null
x1 := pref1
y1 := x2
if y1 != null
then dec1 := (pref1 | y1)
else dec1 := pref1
```

Process P2

```
bool pref2, dec2
y2 := null
x2 := pref2
y2 := x1
if y2 != null
then dec2 := (pref2 | y2)
else dec2 := pref2
```

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

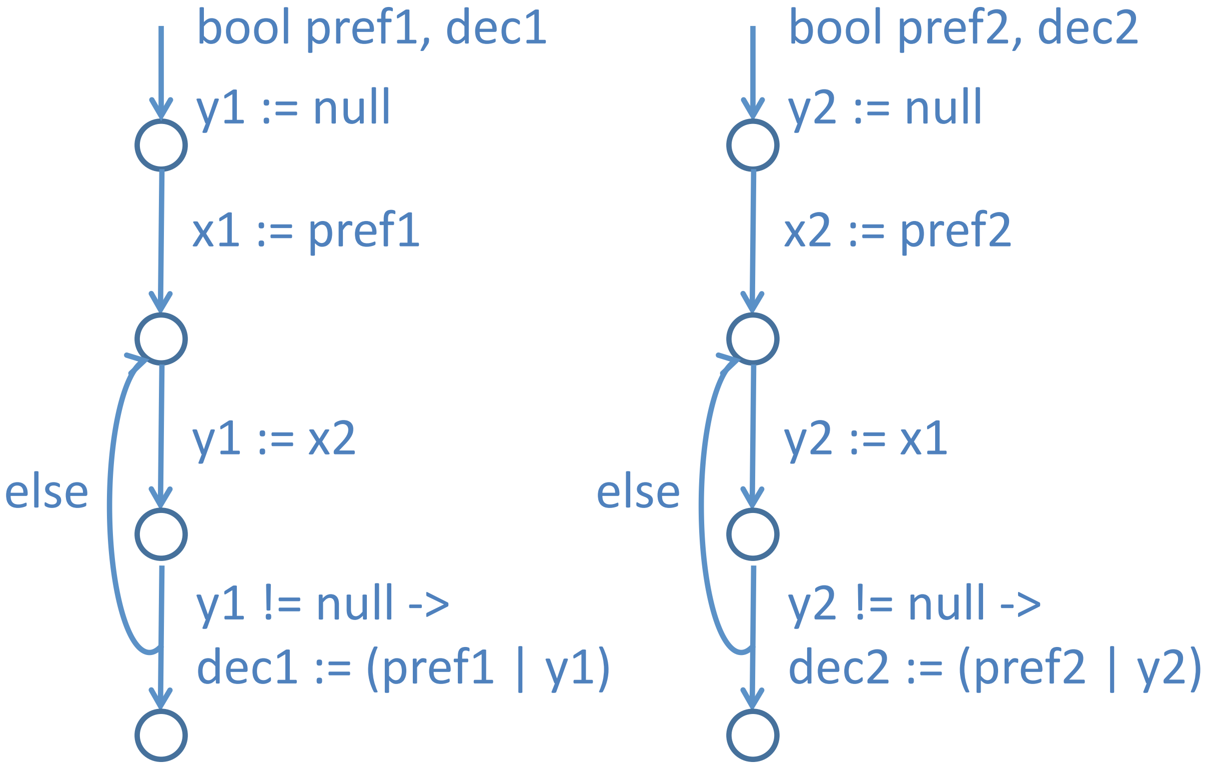
Agreement?

Validity?

Wait-freedom?

Second Attempt at Solving Consensus

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



Process P1

bool pref1, dec1

y1 := null

x1 := pref1

y1 := x2

else

y1 != null ->
dec1 := (pref1 | y1)

Process P2

bool pref2, dec2

y2 := null

x2 := pref2

y2 := x1

else

y2 != null ->
dec2 := (pref2 | y2)

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?

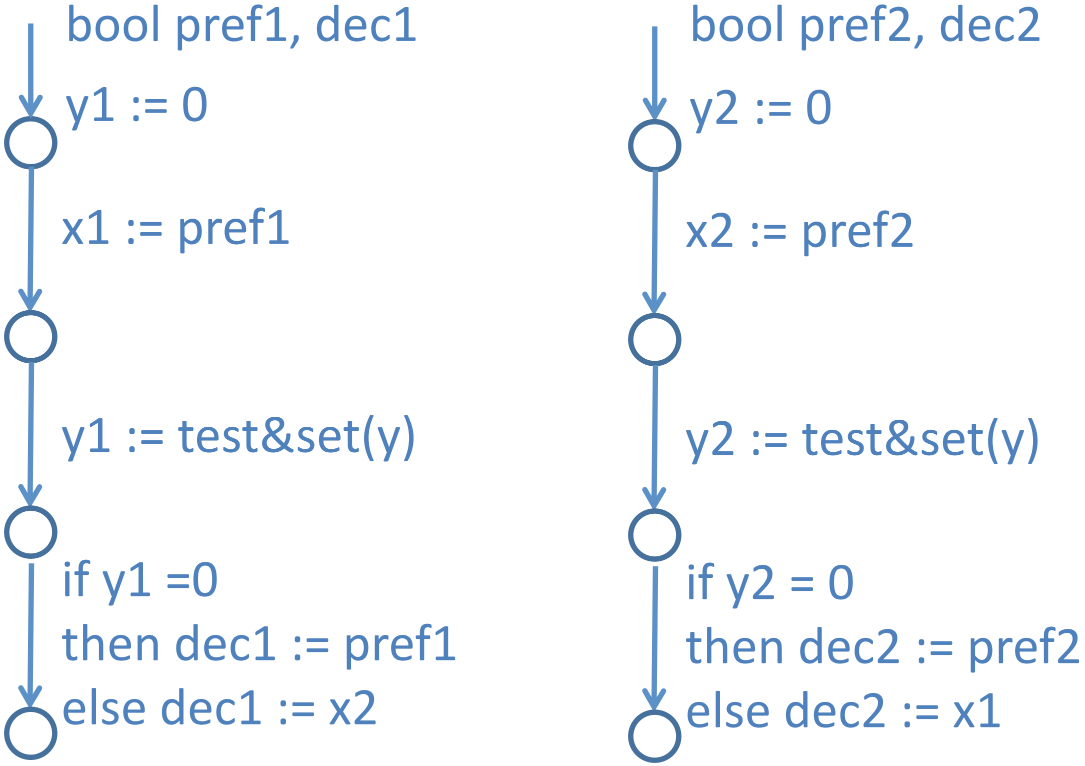
Wait-freedom?

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
- Solution: Use stronger primitives: Test&Set registers
- Byzantine Generals Problem: Coordination is impossible
  - Sending a message, and receiving a message are similar to write and read operations

Consensus using Test&Set Register

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



Process P1

```
bool pref1, dec1
y1 := 0
x1 := pref1
y1 := test&set(y)
if y1 =0
then dec1 := pref1
else dec1 := x2
```

Process P2

```
bool pref2, dec2
y2 := 0
x2 := pref2
y2 := test&set(y)
if y2 = 0
then dec2 := pref2
else dec2 := x1
```

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?

Impossibility of Consensus

Theorem. There is no protocol for two-process consensus such that

1. Processes communicate using only shared atomic registers
2. Protocol satisfies agreement, validity, and wait-freedom

Proof. By contradiction, suppose there is such a protocol.

Let us look at the underlying transition system T for processes P1 and P2

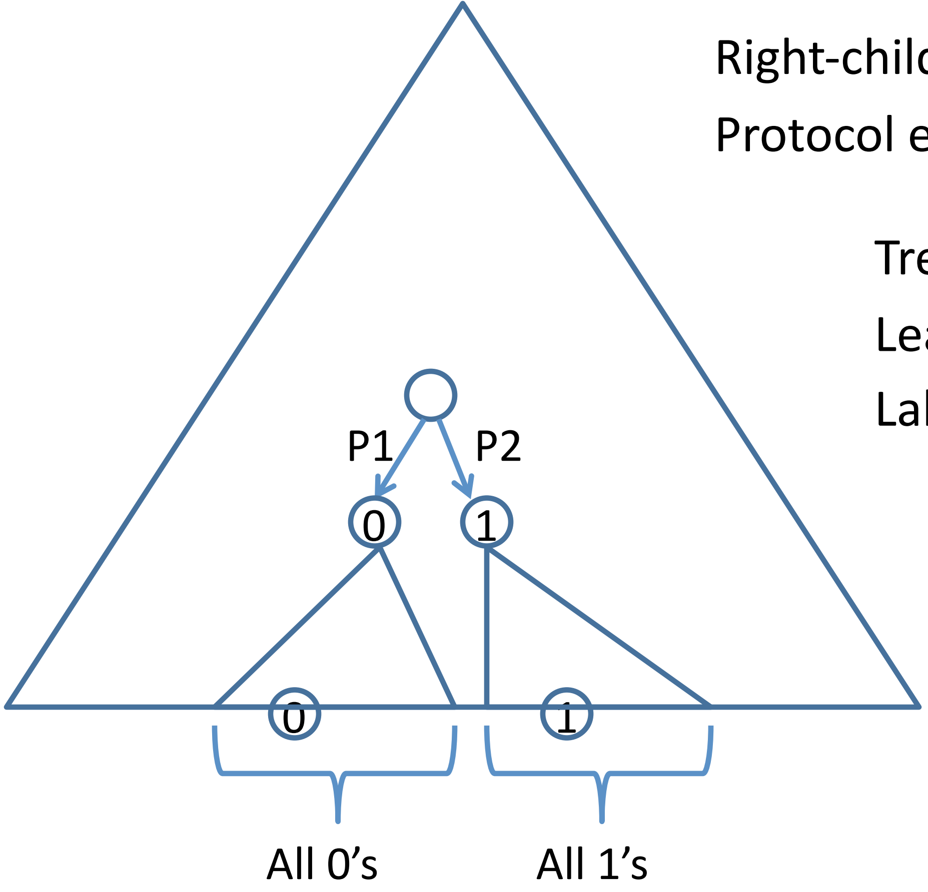
A state of T looks like

P1's local state	Shared variables	P2's local state

A transition of T can be

- a step by P1, and such a transition depends only on the first two parts of the state, or
- a step by P2, which depends only on the last two parts of the state

Execution Tree of Transition System T



Vertices are states

Left-child: Step by P1

Right-child: Step by P2

Protocol execution = Path in this tree

Tree must be finite (why?)

Leaf-vertex: Protocol has terminated

Label leaf with 0/1 based on decision

0-committed vertex:

All paths lead to 0-labled leaves

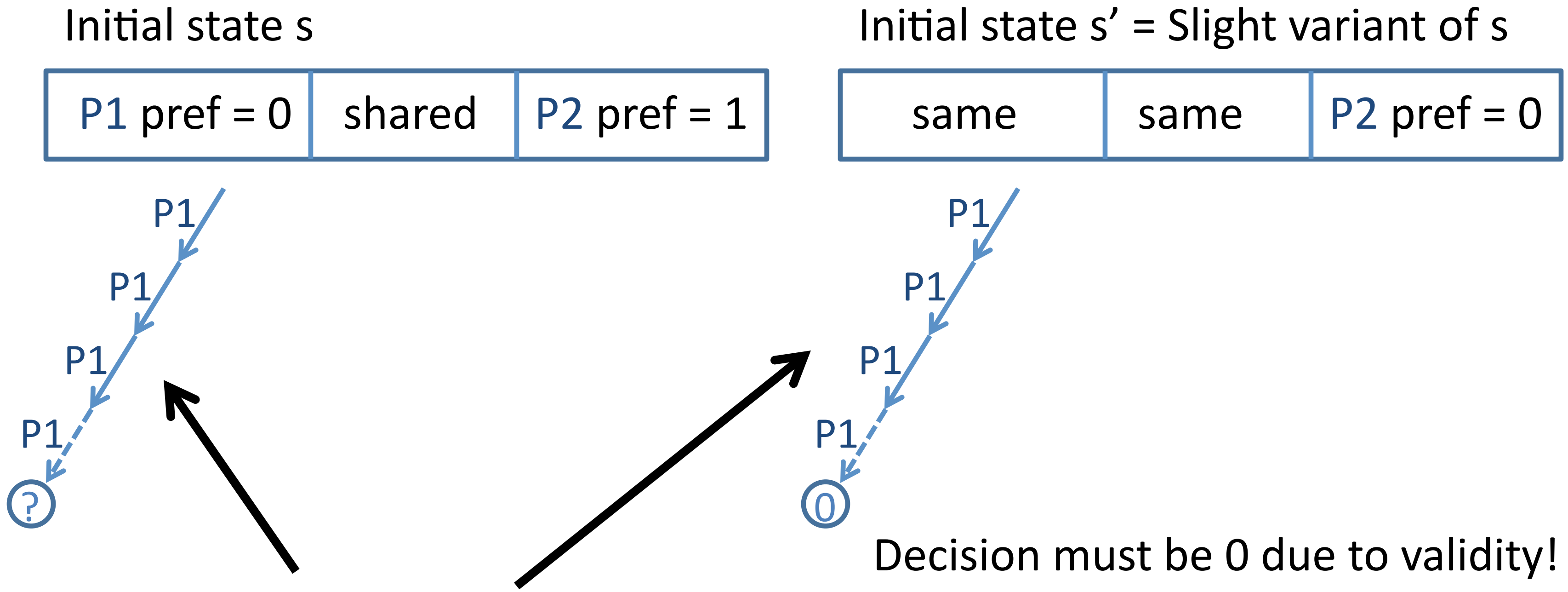
1-committed vertex:

All paths lead to 1-labeled leaves

Uncommitted:

Both decisions still possible

Uncommittedness of Initial State



Initial state s

P1 pref = 0	shared	P2 pref = 1

Initial state s’ = Slight variant of s

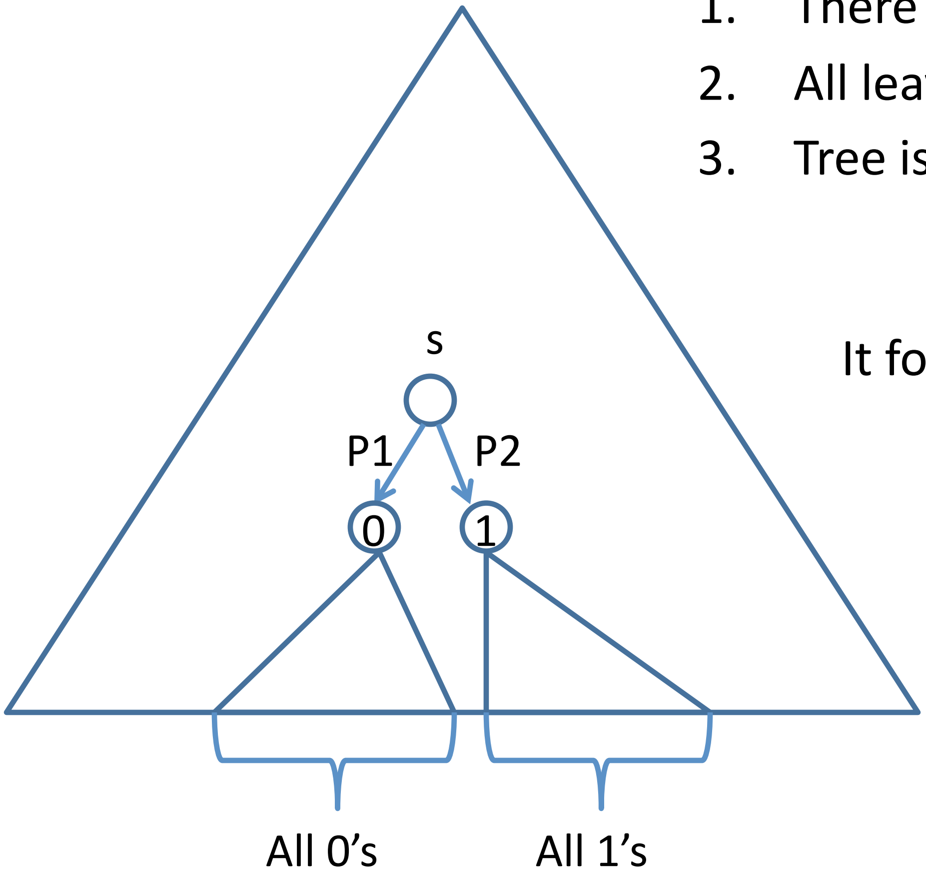
same	same	P2 pref = 0

Decision must be 0 due to validity!

These two executions are identical from P1’s perspective,
So these two decisions must be the same; ? = 0 !

By symmetric argument, if we let only P2 execute in state s, it must decide on 1
This means the initial state s is uncommitted

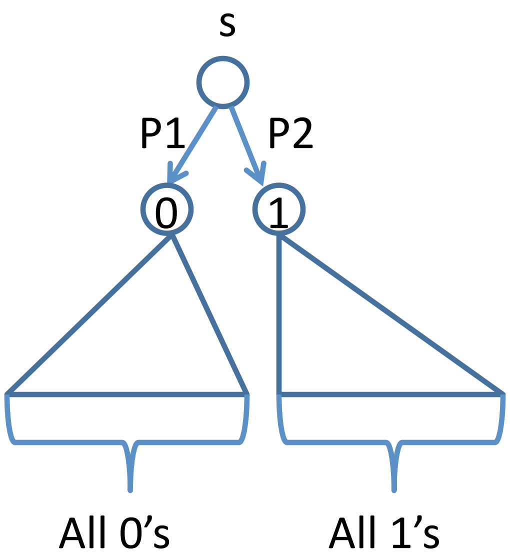
Existence of Critical Vertices



1. There is an initial uncommitted state
2. All leaves are 0-committed or 1-committed
3. Tree is finite

It follows that there must exist a "critical" decision vertex s such that left-child is 0-committed and right-child is 1-committed

Existence of Critical Vertices



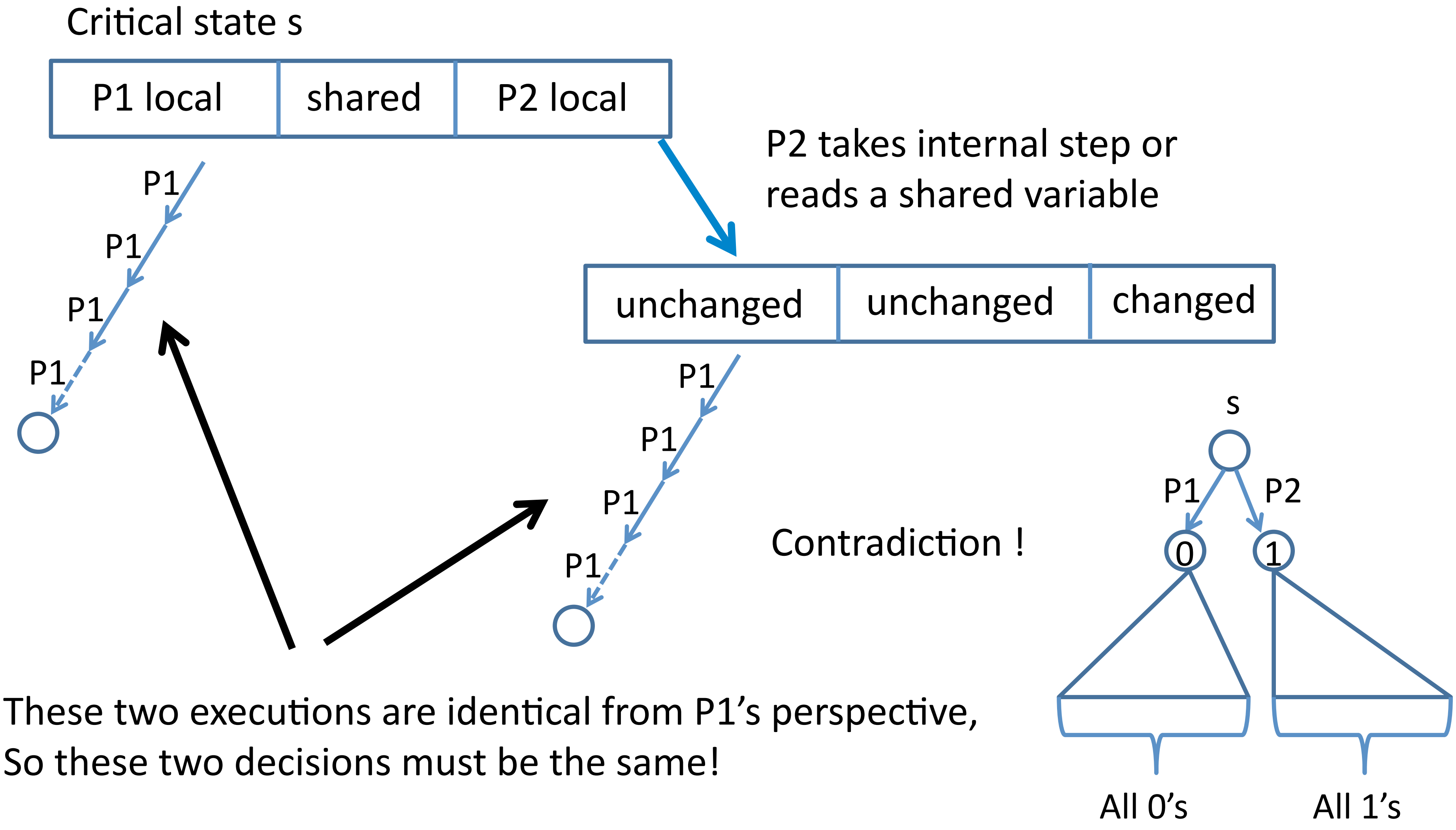
Whether P1 or P2 takes the next step is the deciding factor in state s: what can such a step be?

Possible cases:

1. P1's step is local or is read of a shared var
2. P2's step is local or is read of a shared var
3. Both steps are writes to different shared vars
4. Both steps are writes to same shared var

Proof by case analysis: in each case show that such steps cannot be decisive!

Example Proof: Case 2



Critical state s

P1 local	shared	P2 local

P1
P1
P1
P1

P2 takes internal step or reads a shared variable

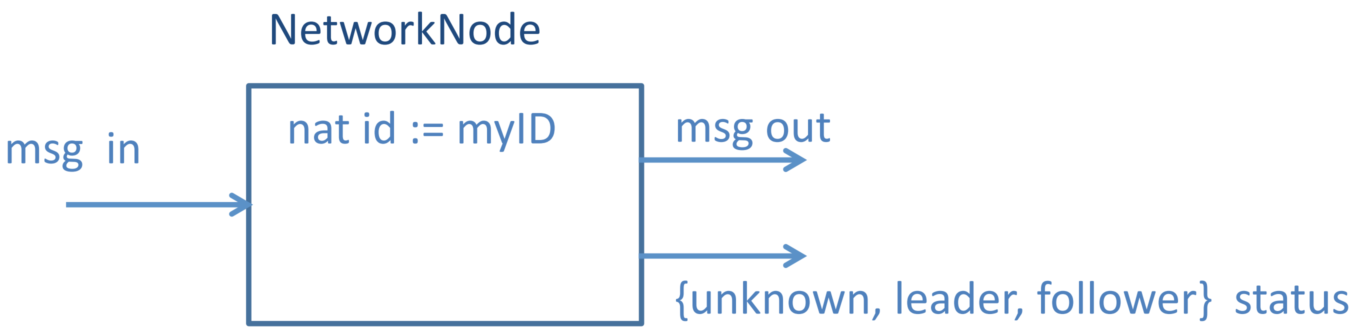
unchanged	unchanged	changed

P1
P1
P1
P1

Contradiction !

These two executions are identical from P1’s perspective,
So these two decisions must be the same!

Leader Election



NetworkNode

nat id := myID

msg in

msg out

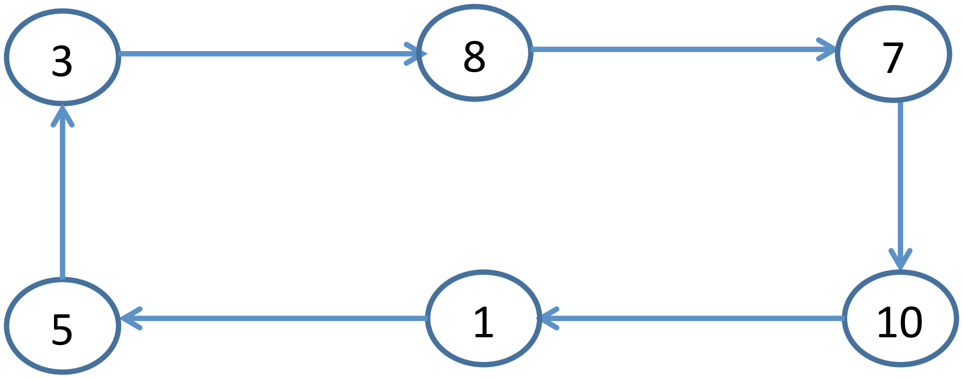
{unknown, leader, follower} status

- Classical coordination problem: 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 compared to the synchronous case
  - There is no notion of a global round
  - Synchronous solution strategy (executing protocol for k rounds implies that message has traveled k hops) does not work here!
- Assume: Processes are connected in a unidirectional ring
  - Protocols for general topologies exist, but are more complex

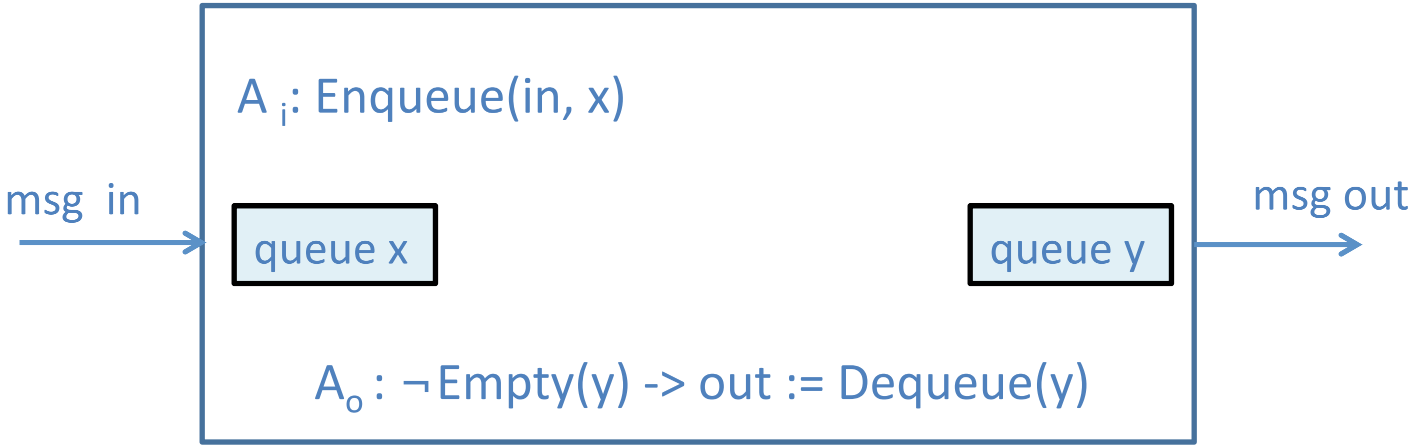
Sample Asynchronous Ring Network



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



A_i: Enqueue(in, x)

msg in

queue x

queue y

msg out

A_o : ¬Empty(y) -> out := Dequeue(y)

One step in the execution of the system is either

- A step local to one process, or
- A communication step that transfers the message at front of the output queue y of a process to back of the input queue x of its right neighbor

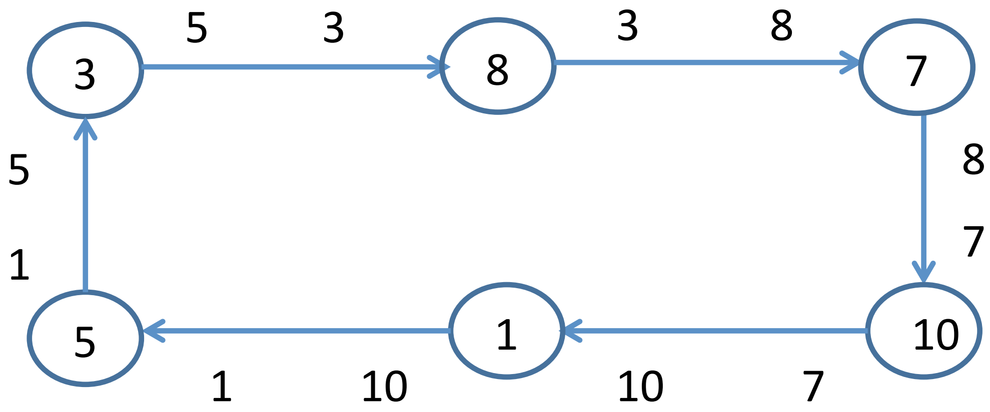
Adopting Synchronous Algorithm

- Set variable id to MyID, and initialize output queue y to contain
- Local step/task
  - Remove a value v from queue x
  - If v > id, then change id to v, and enqueue this value in queue y
- When should a process stop and decide?
  - If v equals id !
  - This would imply that the value 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

- Set variable id to MyID, and initialize output queue y to contain id, which will be communicated to right neighbor
- When you receive a value from left neighbor, store it in state variable id1, and also relay it right neighbor (add it to output queue)
- Receive another value from left neighbor, call it id2
  - id = your value, id1 = left neighbor, id2 = left-left neighbor
- If id1 is the max of these three values, set id to id1, and repeat the above steps
  - Continue to next phase as active, but with different identifier
- If not, then decide to be a follower: continue as a passive participant
  - Does not generate any new messages, just transmits messages in input queue to output queue

Example Execution



id2 = 1

id1 = 5 ★

id = 3

id2 = 5

id1 = 3 ✗

id = 8

id2 = 3

id1 = 8 ★

id = 7

id = 5

id1 = 1 ✗

id2 = 10

id = 1

id1= 10 ★

id2 = 7

id = 10

id1 = 7 ✗

id2 = 8

Example Execution

id2 = 8

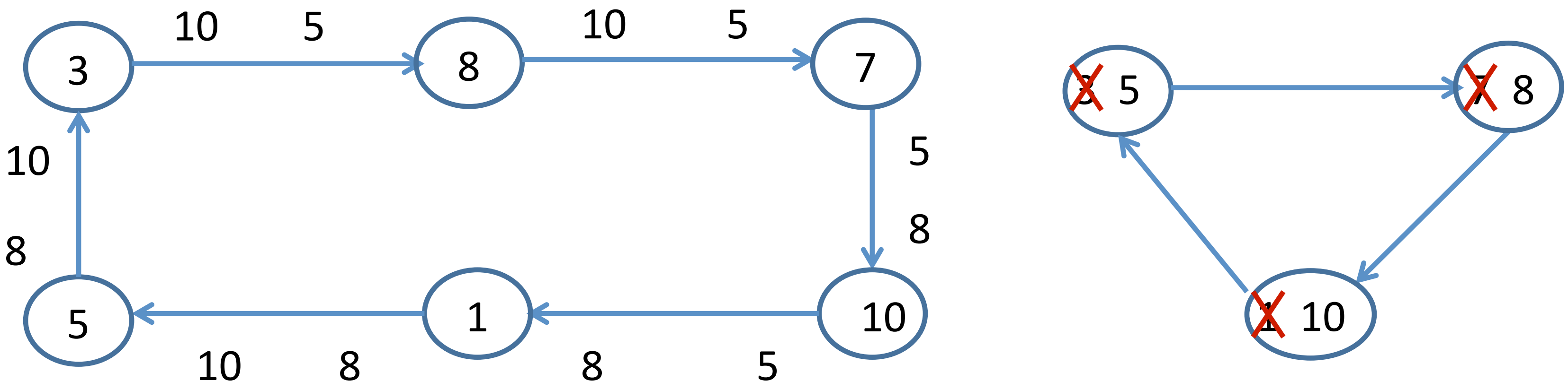
id1 = 10★

id = 5

id2 = 10

id1 = 5✗

id = 8

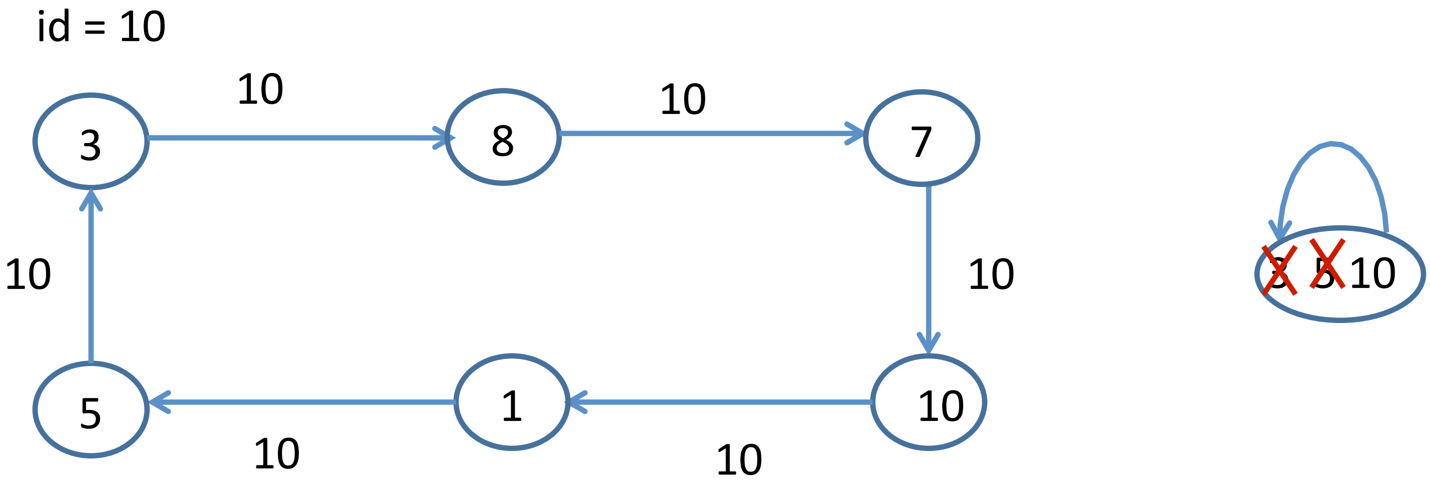


id = 10

id1 = 8✗

id2 = 5

Example Execution

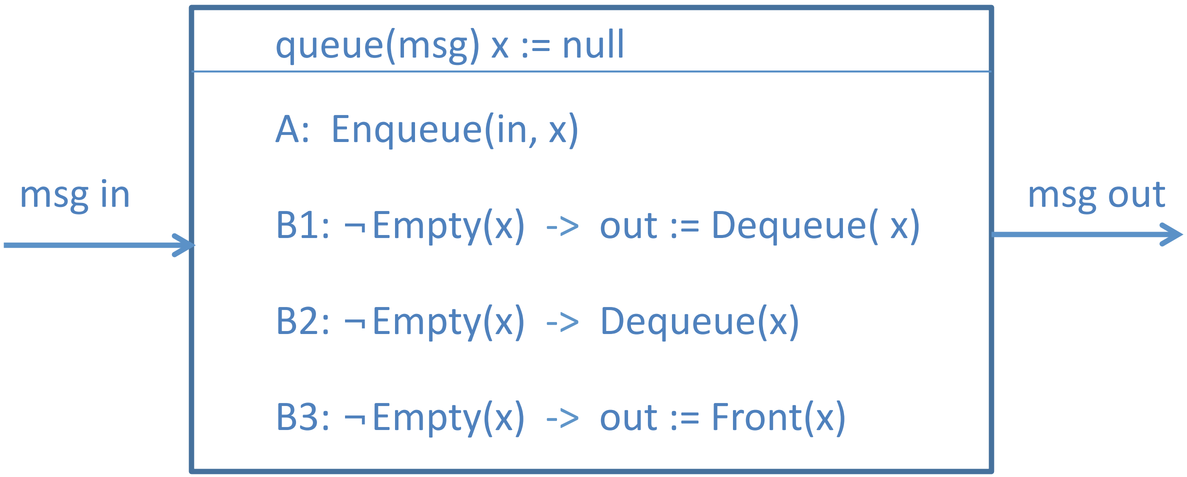


If first message from left neighbor equals id, stop and become the leader!

Algorithm Properties

- Actual execution proceeds asynchronously
  - Messages are processed at arbitrary times
  - Different processes may be executing different phase
- The process that becomes leader doesn't have 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



msg in

queue(msg) x := null

A: Enqueue(in, x)

B1: ¬Empty(x) -> out := Dequeue(x)

B2: ¬Empty(x) -> Dequeue(x)

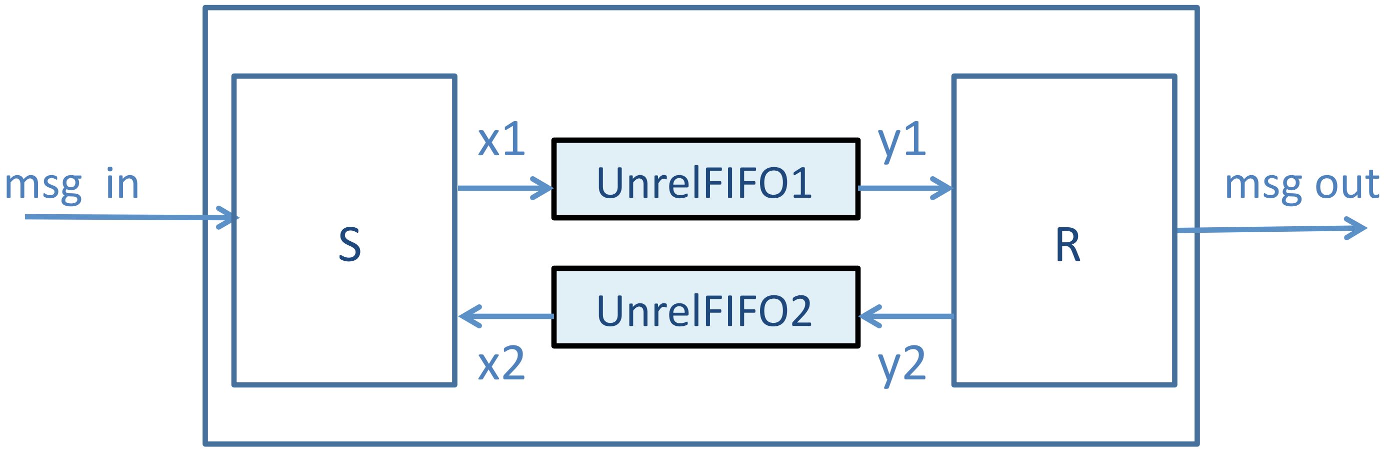
B3: ¬Empty(x) -> out := Front(x)

msg out

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

How to implement a reliable FIFO link using unreliable ones?

Reliable Transmission Problem

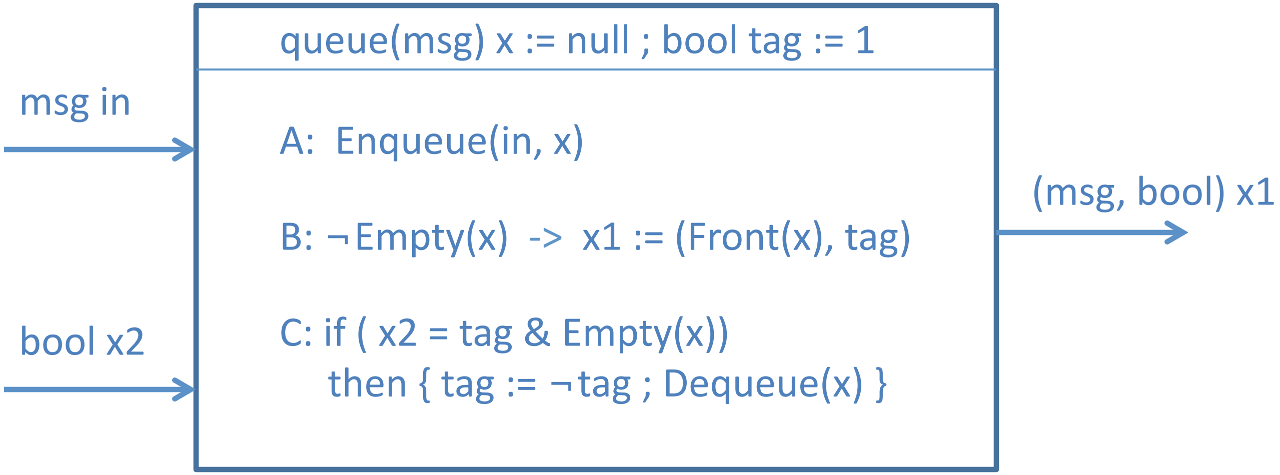


Design Asynchronous processes S and R so that the sequence of messages received on the channel in coincides with the sequence of messages delivered on the channel out

Alternating Bit Protocol

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



msg in

bool x2

```
queue(msg) x := null ; bool tag := 1

A:  Enqueue(in, x)

B: ¬Empty(x)  ->  x1 := (Front(x), tag)

C: if ( x2 = tag & Empty(x))
     then { tag := ¬tag ; Dequeue(x) }
```

(msg, bool) x1

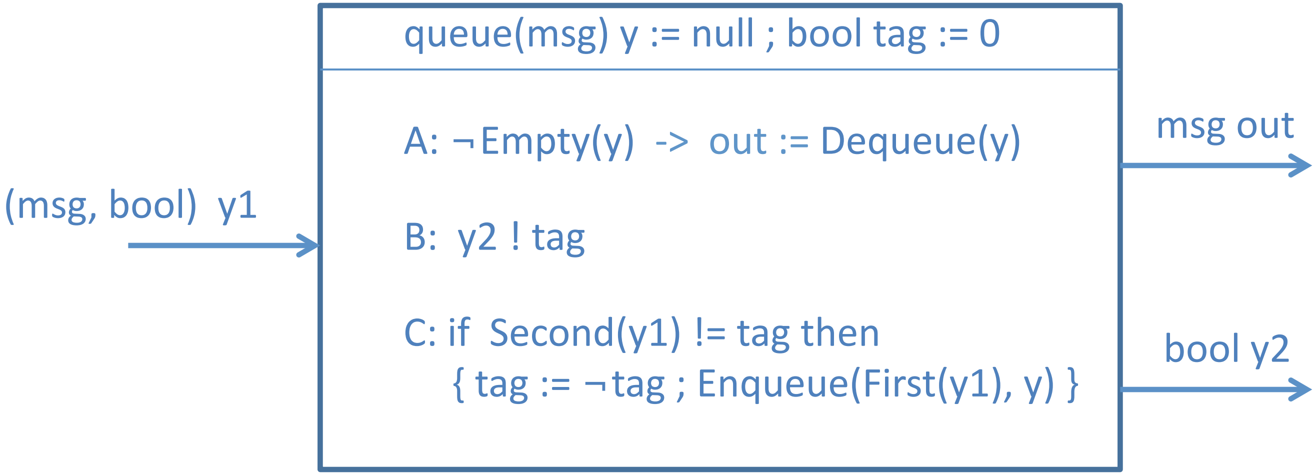
Task A: Store incoming messages in queue x

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

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

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

ABP Receiver



(msg, bool) y1

```
queue(msg) y := null ; bool tag := 0

A: ¬Empty(y)  ->  out := Dequeue(y)

B:  y2 ! tag

C: if  Second(y1) != tag then
     { tag := ¬tag ; Enqueue(First(y1), y) }
```

msg out

bool y2

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

Task B: Transmit local tag as acknowledgement on channel y2

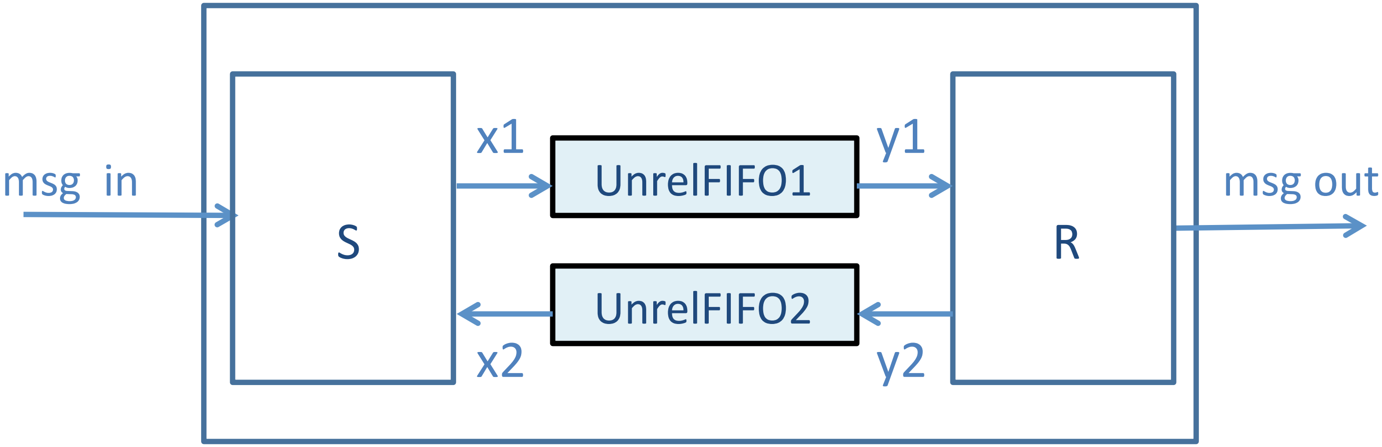
Note: Same ack is potentially transmitted repeatedly

Task C: If tag of incoming message matches local tag, then message is new; so add it to y 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, as acknowledgement, but all these acks are simply ignored by S
- When R gets (m,1) the first time, it stores m in queue y (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 send 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
- Messages received as input are queued up in x, and S will now repeat the whole 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?

Credits

Notes based on Chapter 4 of

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