Distributed Computing in Shared Memory and Networks

Class 2: Consensus

WEP 2018

KAUST

This class

Reaching agreement in shared memory:

- Consensus
  - ✓Impossibility of wait-free consensus
- 1-resilient consensus impossibility
- Universal construction

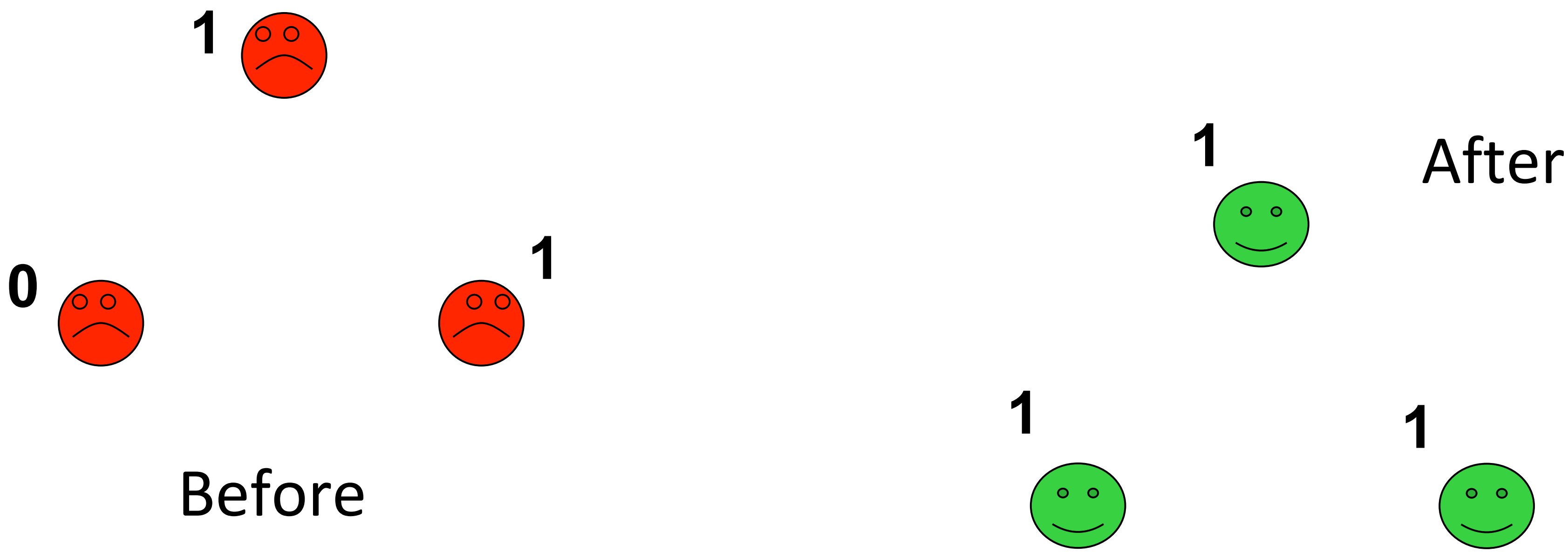
System model

- N *asynchronous* (no bounds on relative speeds) processes $p_0, \ldots, p_{N-1}$ ($N \geq 2$) communicate via atomic read-write registers

- Processes can fail by crashing
  - ✓A crashed process takes only finitely many steps (reads and writes)
  - ✓Up to t processes can crash: t-resilient system
  - ✓t=N-1: wait-free

Consensus

Processes *propose* values and must *agree* on a common decision value so that the decided value is a proposed value of some process



Consensus: definition

A process *proposes* an *input* value in V (|V|≥2) and tries to *decide* on an *output* value in V

- *Agreement:* No two processes decide on different values
- *Validity:* Every decided value is a proposed value
- *Termination:* No process takes infinitely many steps without deciding

  (Every correct process decides)

Optimistic (0-resilient) consensus

Consider the case t=0, no process fails

Shared: 1WNR register D, initially T (default value not in V)

Upon propose(v) by process p_i:

```
if i = 0 then D.write(v)        // if p0 decide on v
wait until D.read() ≠ T         // wait until p0 decides
return D
```

(every process decides on p_0's input)

Impossibility of wait-free consensus [FLP85,LA87]

Theorem 1 No wait-free algorithm solves consensus

We give the proof for N=2, assuming that
p_0 proposes 0 and p_1 proposes 1

Implies the claim for all N≥2

Proof of Theorem 1

- We show that no 2-process wait-free solution exists for iterated read-write memory: $R_0[]$, $R_1[]$
- Code for p_i in round k: write to $R_k[i]$ and read $R_k[1\text{-}i]$:

```
k:= 0
repeat
        k := k+1;
        R_k[i].write(v_i);
        v_i := [v_i, R_k[1-i].read()];
until not decided(v_i)
```

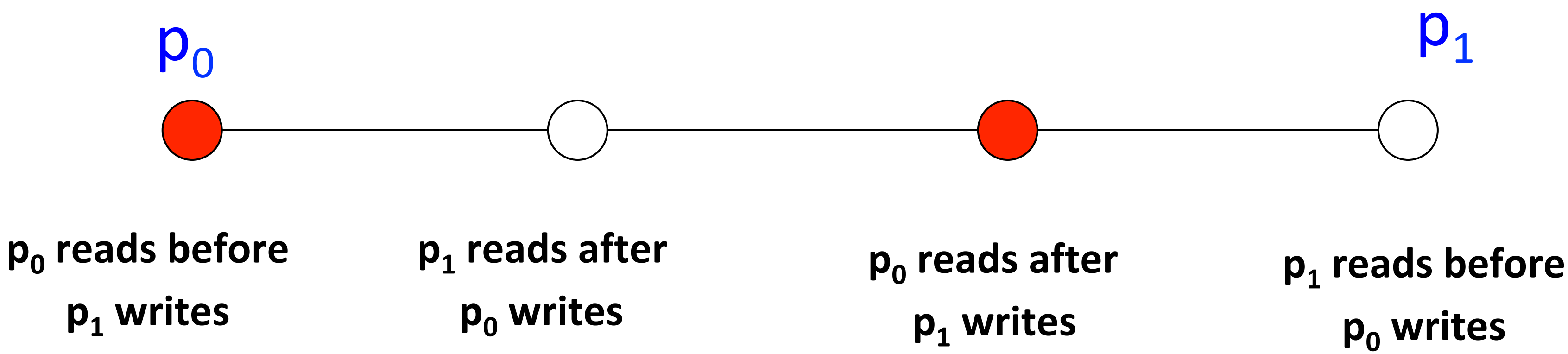
(until the current state does not map to a decision)

- The iterated memory is equivalent to non-iterated one for solving consensus

Proof of Theorem 1

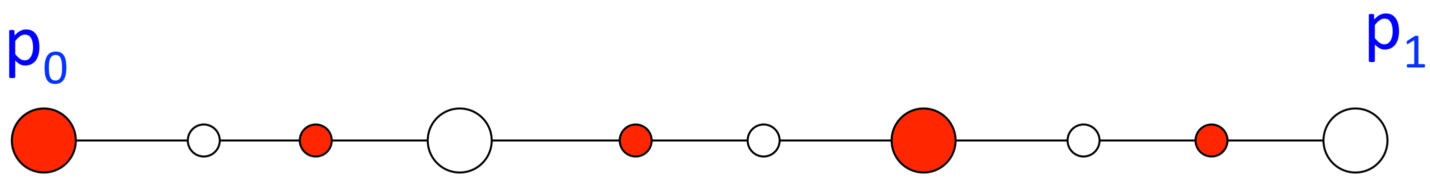
Initially each p_i only knows its input

One round of IIS:



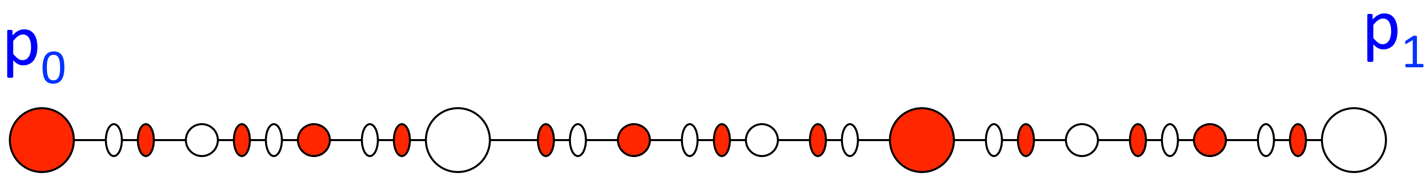
Proof sketch for Theorem 1

Two rounds:



Proof of Theorem 1

And so on…



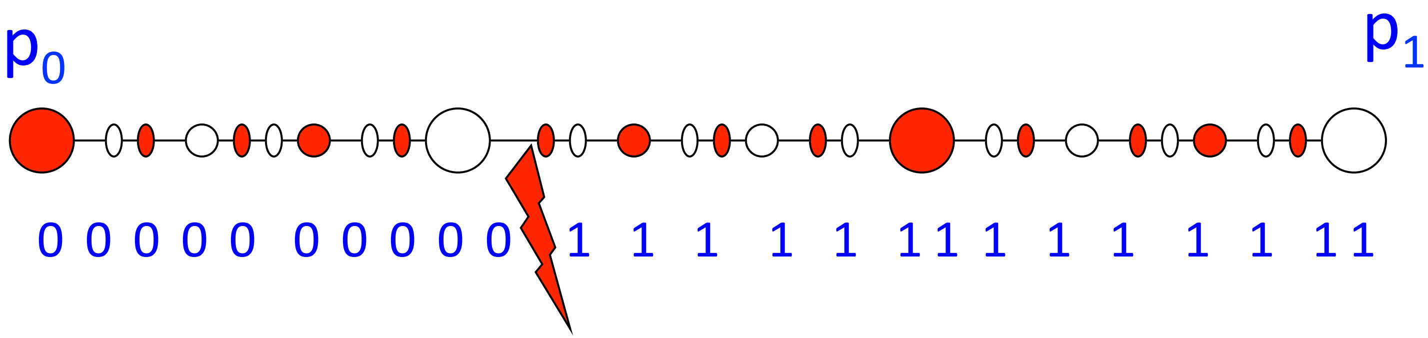
Solo runs remain connected - no way to decide!

Proof of Theorem 1

Suppose p_i (i=0,1) proposes i

- p_i must decide i in a solo run!

Suppose by round r every process decides



There exists a run with conflicting decisions!

1-resilient consensus?

What if we have 1000000 processes and one of them can crash?

NO

We present a direct proof now

(an indirect proof by reduction to the wait-free impossibility also exists)

Impossibility of 1-resilient consensus [FLP85,LA87]

Theorem 2 No 1-resilient (assuming that one process might fail) algorithm solves consensus in read-write

Proof

By contradiction, suppose that an algorithm A solves 1-resilient binary consensus among $p_0, \ldots p_{N-1}$

Proof

A run of A is a sequence of atomic *steps* (reads or writes) applied to the initial state

A run of A can be seen as and initial input configuration (one input per process) and a sequence of process ids $i_1, i_2, \ldots i_k, \ldots$ (all registers are atomic)

Every correct (taking sufficiently many steps) process decides!

Proof: valence

Let R be a finite run

- We say that R is *v-valent* (for v in {0,1}) if v is decided in every infinite extension of R

- We say that R is *bivalent* if R is neither 0-valent nor 1-valent
  (there exists a 0-valent extension of R and a 1-valent extension of R)

Proof: valence claims

Claim 1 Every finite run is 0-valent, or 1-valent, or bivalent.
(by Termination)

Claim 2 Any run in which some process decides v is v-valent
(by Agreement)

Corollary 1: No process can decide in a bivalent run (by Agreement).

Bivalent input

Claim 3 There exists a bivalent input configuration (empty run)

Proof

Suppose not

Consider sequence of input configurations $C_0,\ldots,C_N$:

C_i: $p_0,..,p_{i-1}$ propose 1, and $p_i,..,p_{N-1}$ propose 0

- All C_i's are univalent
- C_0 is 0-valent (by Validity)
- C_N is 1-valent (by Validity)

Bivalent input

There exists i in {0,…N-1} such that C_i is 0-valent and C_{i+1} is 1-valent!

C_i and C_{i+1} differ only in the input value of p_i (it proposes 0 in C_i and 1 in C_{i+1})

Consider a run R starting from C_i in which p_i takes no steps (crashes initially): eventually all other processes decide 0

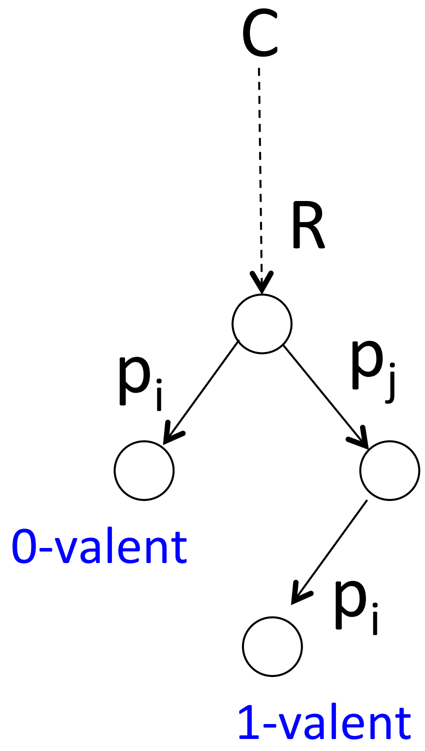
Consider R' that is like R except that it starts from C_{i+1}

- R and R' are indistinguishable!
- Thus, every process decides 0 in R' --- contradiction (C_{i+1} is 1-valent)

Critical run

Claim 4 There exists a finite run R and two processes p_i and p_j such that R.i is 0-valent and R.j.i is 1-valent (or vice versa)

(R is called critical)



Proof of Claim 4: By construction, take the bivalent empty run C (by Claim 3 it exists)

We construct an ever-extending fair (giving each process enough steps) run which results in R

Proof of Claim 4: critical run

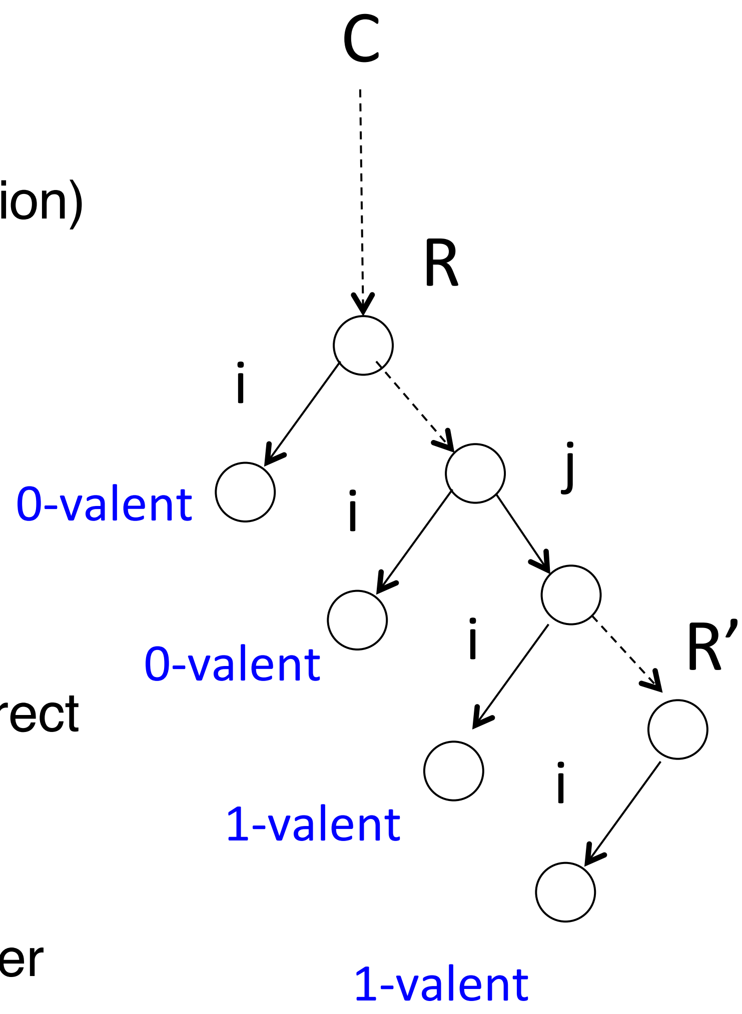
repeat forever

take the next process p_i (in round-robin fashion)

if for some R', an extension of R, R'.i is bivalent **then** R:=R'.i

else stop

- If never stops – ever extending (infinite) bivalent runs in which every process is correct (takes infinitely many steps – contradiction with termination
- If stops – (suppose R.i is 0-valent) – consider a 1-valent extension
  - ✓ There is a critical run between R and R'



Proof (contd.)

Take a critical run R (exists by Claim 4) such that:

- R.0 is 0-valent
- R.1.0 is 1-valent

(without loss of generality, we can always rename processes or inputs appropriately ☺)

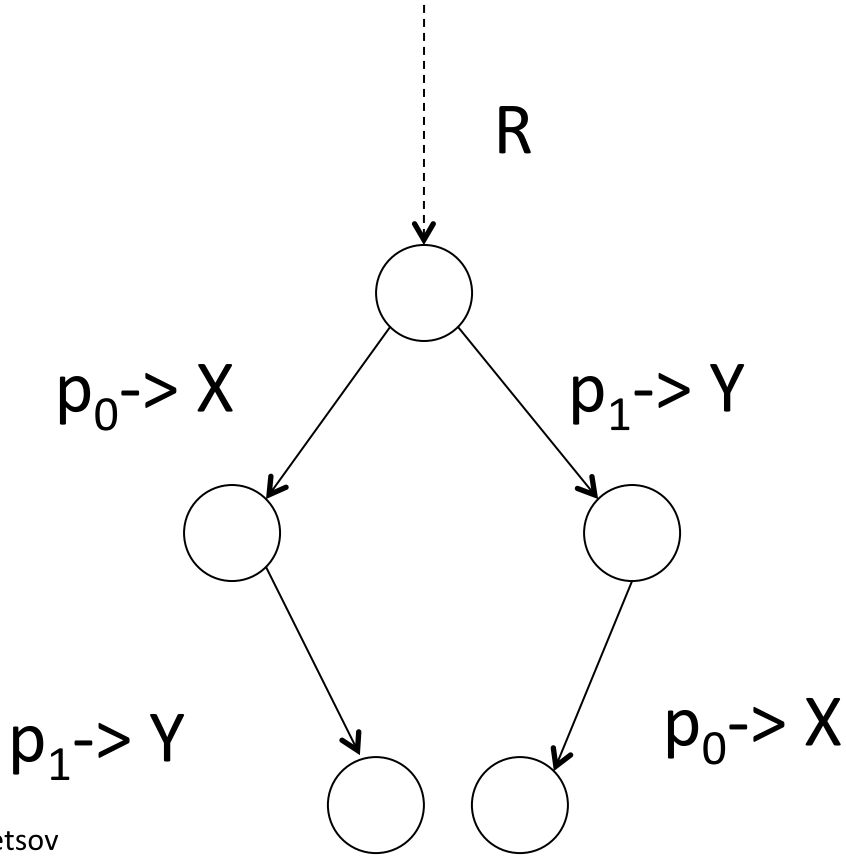
Proof (contd.): the next steps in R

Three cases, depending on the next steps of p_0 and p_1 in R

- p_0 and p_1 are about to access different objects in R
- p_1 reads X and p_0 reads X
- p_0 or p_1 writes in X

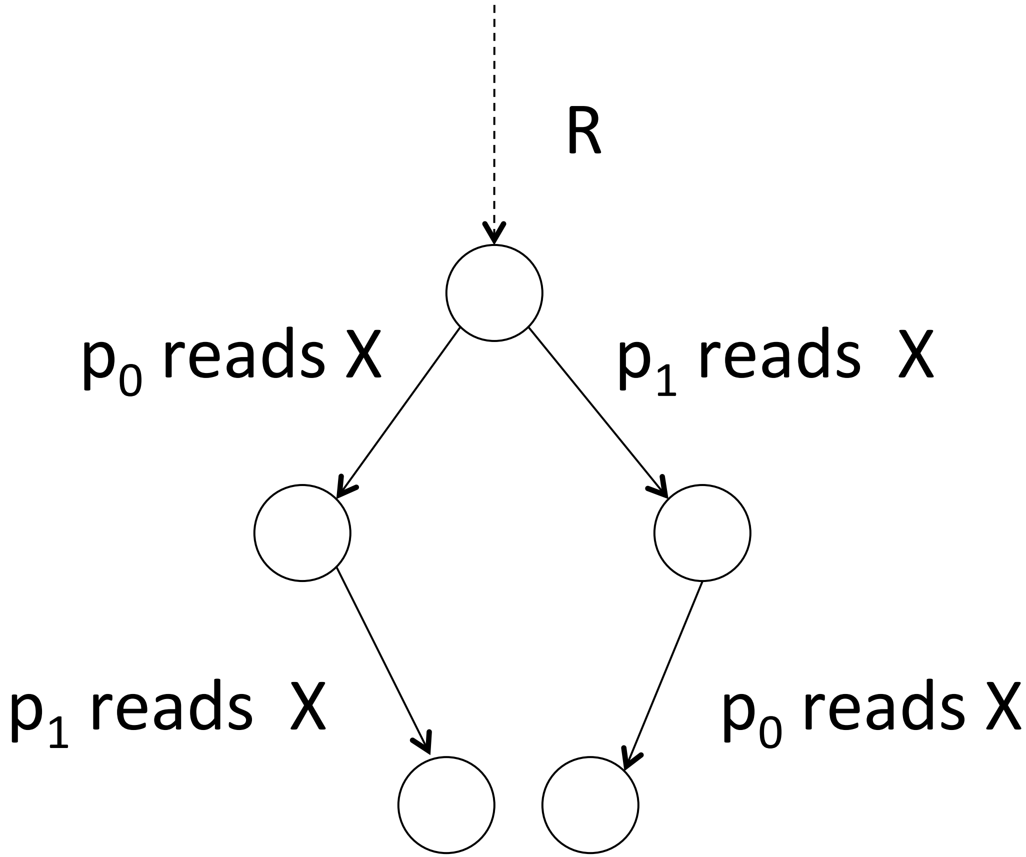
Proof (contd.): cases and contradiction

- p_0 and p_1 are about to access different objects in R
  - ✓R.0.1 and R.1.0 are indistinguishable



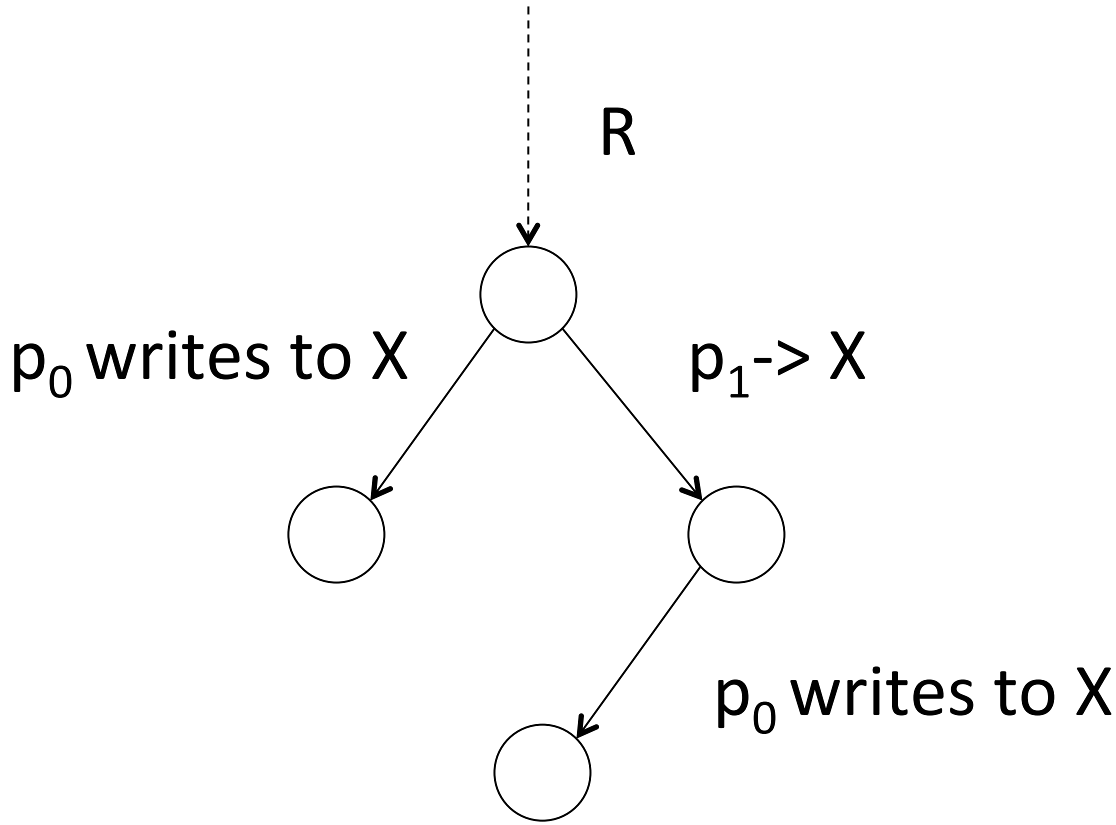
Proof (contd.): cases and contradiction

- p_0 and p_1 are about to read the same object X
  R.0.1 and R.1.0 are indistinguishable



Proof (contd.): cases and contradiction

- p_0 is about to write to X (the case when p_1 writes is symmetric)
  - ✓Extensions of R.0 and R.1.0 are indistinguishable for all except p_1 (assuming p_1 takes no more steps)



Thus

- No critical run exists
- A contradiction with **Claim 4**

$\Rightarrow$ 1-resilient consensus is impossible in read-write

Next

- Combining registers with stronger objects
  - ✓Consensus and test-and-set (T&S)
  - ✓Consensus and queues
- Universality of consensus
  - ✓Consensus can be used to implement any object

Test&Set atomic objects

Exports one operation test&set() that returns a value in {0,1}

Sequential specification:

The first atomic operation on a T&S object returns 0, all other operations return 1

2-process consensus with T&S

Shared objects:

T&S TS

Atomic registers R[0] and R[1]

Upon propose(v) by process p_i (i=0,1):

```
R[i] := v
if TS.test&set()=0 then
        return R[i]
else
        return R[1-i]
```

FIFO Queues

Exports two operations enqueue() and dequeue()

- enqueue(v) adds v to the end of the queue
- dequeue() returns the first element in the queue

  (LIFO queue returns the last element)

2-process consensus with queues

Shared:

Queue Q, initialized (winner,loser)

Atomic registers R[0] and R[1]

Upon propose(v) by process p_i (i=0,1):

R[i] := v

if Q.dequeue()=winner then

return R[i]

else

return R[1-i]

Quiz 2.1: uninitialized queues

The algorithm assumes that the queue is initialized to (winner,loser).

- Can we solve consensus using (initially) empty queues?

But why consensus is interesting?

Because it is universal!

- If we can solve consensus among N processes, then we can *implement* any object shared by N processes
  - ✓T&S and queues are universal for 2 processes

- A key to implement a generic fault-tolerant service (replicated state machine)

What is an *object* ?

Object O is defined by the tuple (Q,O,R,σ):

- Set of states Q
- Set of operations O
- Set of outputs R
- Sequential specification σ, a subset of OxQxRxQ:
  - ✓(o,q,r,q') is in σ ⇔ if operation o is applied to an object in state q, then the object *can* return r and change its state to q'
  - ✓Total on OxQ (defined for all o and q)

Deterministic objects

- An operation applied to a *deterministic* object results in exactly one (output,state) in RxQ, i.e., σ can be seen a function OxQ -> RxQ

- E.g., queues, counters, T&S are deterministic
- Unordered set (put/get) – non-deterministic

Example: queue

Let V be the set of possible elements of the queue

- Q=V* U {Ø} (all sequences with elements in V and the empty state)
- $O=\{enq(v)_{v \text{ in } V}, deq()\}$
- R=V U {Ø} U {ok}
- σ(enq(v),q)=(ok,q.v)
- σ(deq(),v.q)=(v,q)
- σ(deq(), Ø)=(Ø, Ø)

Implementation: definition

A distributed algorithm A that, for each operation o in O and for every p_i, describes a concurrent procedure o_i using base objects

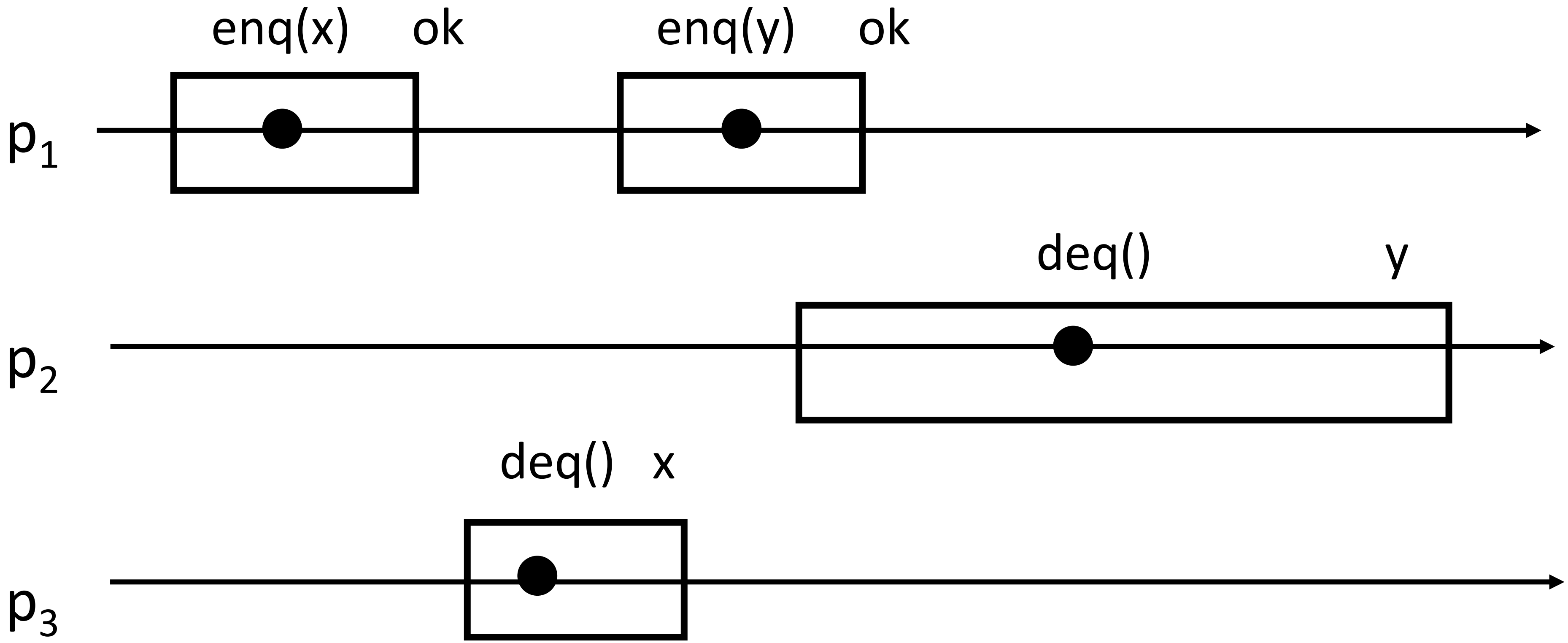
A run of A is *well-formed* if no process invokes a new operation on the implemented object before returning from the old one (we only consider well-formed runs)

Implementation: correctness

A (wait-free) implementation A is correct if in every well-formed run of A

- Wait-freedom: every operation run by p_i returns in a finite number of steps of p_i
- Linearizability ≈ operations "appear" instantaneous (the corresponding *history* is *linearizable*)

Linearization



p_1-enq(x); p_1-ok; p_3-deq(); p_3-x;

p_1-enq(y); p_1 –ok; p_2-dequeue(); p_2-y

Universal construction

Theorem 1 [Herlihy, 1991] If N processes can solve consensus, then N processes can (wait-free) implement every object O=(Q,O,R,σ)

Suppose you are given an unbounded number of consensus objects and atomic read-write registers

You want to implement an object $O=(Q,O,R,\sigma)$

How would you do it?

Universal construction: idea

Every process that has a pending operation does the following:

- Publish the corresponding *request*
- Collect published requests and use consensus instances to serialize them: the processes agree on the order in which the requests are executed
- Processes agree on the order in which the published requests are executed

Universal construction: variables

Shared abstractions:

N atomic registers R[0,…,N-1], initially Ø

N-process consensus instances C[1], C[2], …

Local variables for each process p_i:

integer *seq*, initially 0

// the number of p_i's requests executed so far

integer *k*, initially 0

// the number of batches of

// all requests executed so far

sequence *linearized*, initially empty

//the serial order of executed requests

Universal construction: algorithm

Code for each process p_i: implementation of operation op

```
seq++
R[i] := (op,i,seq)                      // publish the request
repeat
        V := read R[0,…,N-1]                    // collect all requests
        requests := V-{linearized}      //choose not yet linearized requests
        if requests≠∅ then
            k++
            decided:=C[k].propose(requests)
            linearized := linearized.decided
            //append decided request in some deterministic order
until (op,i,seq) is in linearized
return the result of (op,i,seq) in linearized
            // using the sequential specification σ
```

Universal construction: correctness

- Linearization of a given run: the order in which operations are put in the *linearized list*
  - ✓ Agreement of consensus: all *linearized* lists are related by containment (one is a prefix of the other)

- Real-time order: if op1 precedes op2, then op2 cannot be linearized before op1
  - ✓ Validity of consensus: a value cannot be decided unless it was previously proposed

Universal construction: correctness

- Wait-freedom:
  - ✓Termination and validity of consensus: there exists k such that the request of p_i gets into *req* list of every processes that runs C[k].*propose(req)*

Another universal abstraction: CAS

Compare&Swap (CAS) stores a *value* and exports operation CAS(u,v) such that:

- If the current value is u, CAS(u,v) replaces it with v and returns u
- Otherwise, CAS(u,v) returns the current value

A variation: CAS returns a boolean (whether the replacement took place) and an additional operation read() returns the value

N-process consensus with CAS

Shared objects:

CAS CS initialized Ø

// Ø cannot be an input value

Code for each process p_i ($i=0,\dots,N-1$):

```
v_i := input value of p_i
v :=CS.CAS(Ø,v_i)
if v = Ø
        return v_i
else
        return v
```

N-consensus object

N-consensus stores a value in {Ø} U V and exports operation propose(v), v in V:

For 1^{st} to N^{th} propose() operations:

- If the value is Ø, then propose(v) sets the value to v and returns v
- Otherwise, returns the value

All other operations do not change the value and return Ø

N-process consensus with N-consensus

Immediate: every process pi simply invokes C.propose(input of pi) and returns the result of it

(N+1)-consensus using N-consensus?

Consensus number

An object O has consensus number k (we write cons(O)=k) if

- k-process consensus can be solved using registers and any number of copies of O but (k+1)-consensus cannot

If no such number k exists for O, then cons(O)=∞

(k=cons(O) is the maximal number of processes that can be synchronized using copies of O and registers)

Consensus power

- cons(register)=1
- cons(T&S)=cons(queue)=2
- …
- cons(N-consensus)=N
  - ✓N-consensus is N-universal!
- …
- cons(CAS)=∞

Quiz 2.2: consensus power

Show that T&S has consensus power at most 2, i.e., it cannot be, combined with atomic registers, used to solve 3-process consensus

Possible outline:

- Consider the *critical bivalent* run R of A: every one-step extension of R is univalent (show first that it exists)
- Show that all steps enabled at R are on the same T&S object
- Show that there are two extensions of opposite valences that some process cannot distinguish

Open questions

- Robustness

  Suppose we have two objects A and B, cons(A)=cons(B)=k

  Can we solve (k+1)-consensus using registers and copies of A and B?
- Can we implement an object of consensus power k shared by N processes (N≥k) using k-consensus objects?