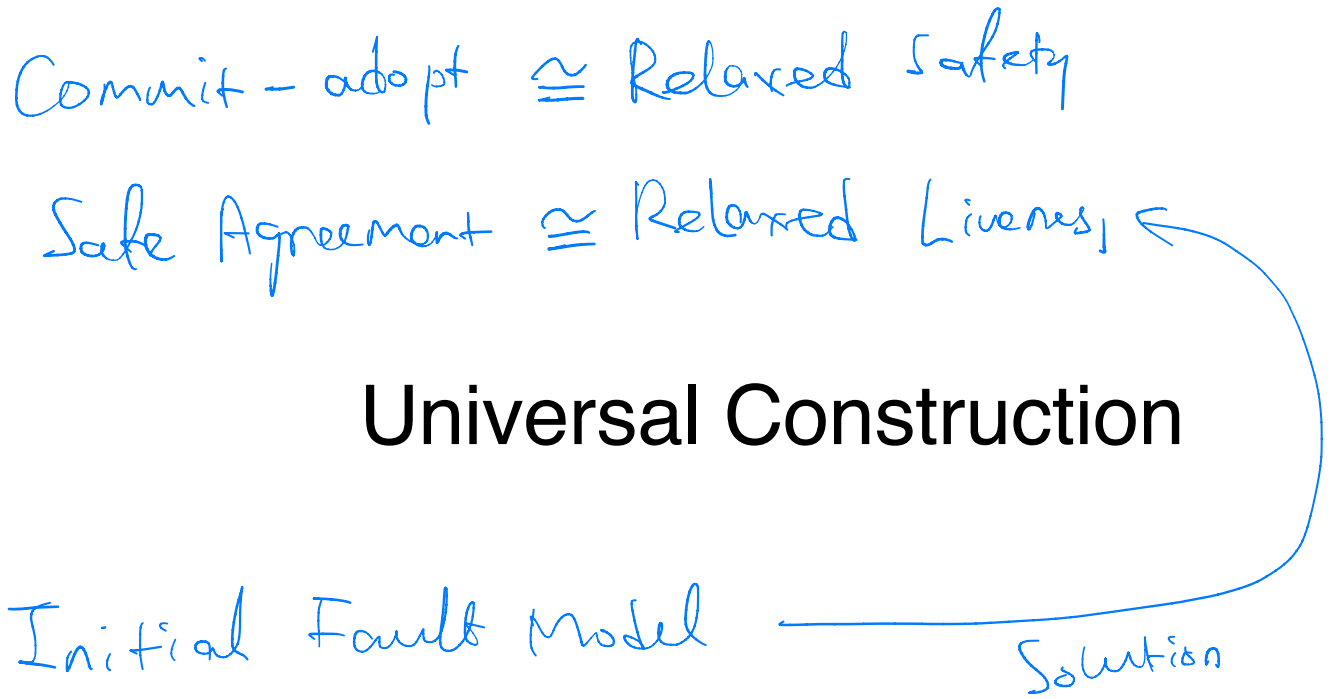
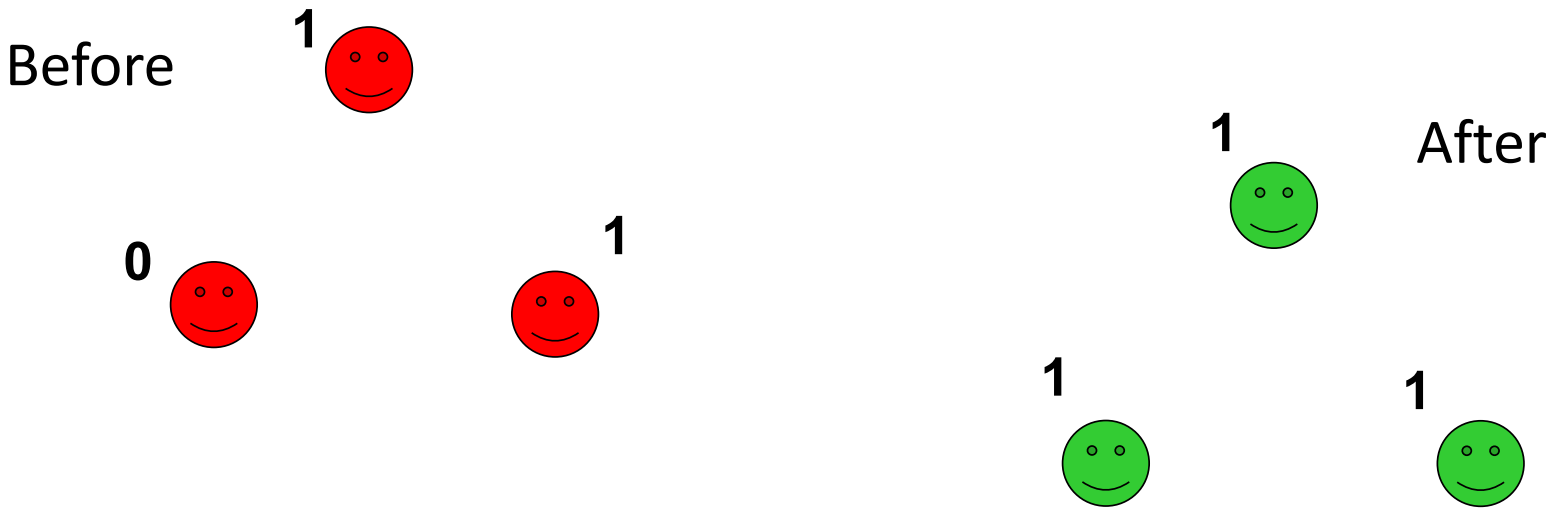

Commit-adopt ≅ Relaxed Safety
Safe Agreement ≅ Relaxed Liveness
Universal Construction
Initial Fault Model
Solution

SLR206, P2

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)

No fault-tolerant solutions using registers

Probabilistic solutions exist

$q_0 = 0$ initial state

$Q = \{0, 1\}$ input

$O = \{TAS()\}$ Response

$R = \{0, 1\}$ $\sigma = Q \times O \to Q \times R$

$\begin{cases} \sigma(0, TAS()) = (1, 0) \\ \sigma(1, TAS()) = (1, 1) \end{cases}$

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

△ get 1 from TAS → Not the first one to propose (v).

当一个进程（或线程）调用 test&set() 操作时，它首先“测试”对象当前的值。
如果该值为 0（通常表示未锁定状态），则操作“设置”该值为 1（表示锁定状态）并返回 0。
如果该值已经是 1，则操作返回 1，表明其他某个进程已经获得了锁或者完成了某个条件的设置。

这种机制很有效地用于锁定资源，因为**只有第一个访问资源的进程会看到值为 0 并将其设置为 1，后续的进程则会看到值为 1 并知道资源已被占用**。

2-process consensus with T&S

Shared objects:

T&S TS

Atomic registers R[0] and R[1]

TAS() can only solve 2-process consensus.

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]
```

Thm TAS and Registers cannot solve consensus (wait free) among ≥3 procs.

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)

$q_0 = \emptyset$

$Q = \mathbb{N}^*$

$O = \{enq(v), deq() \mid v \in \mathbb{N}\}$

$R = \mathbb{N} \cup \{ok\}$

$\sigma : Q \times O = R \times Q$

$\sigma(q, enq(v)) = (q \cdot v, ok)$

$\sigma(\emptyset, deq()) = (\emptyset, \emptyset)$

$\sigma(v \cdot q, deq()) = (q, v)$

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]
```

initial value in Queue could be any one.

```
p: propose(v):
R[i] = v.
if q.deq() ≠ ∅ :
    return v
return R[1-i].
```

Quiz 8.1: uninitialized queues

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

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

Hint: Use 2 Queue

Why is consensus 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 Obj 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 ≃ registers

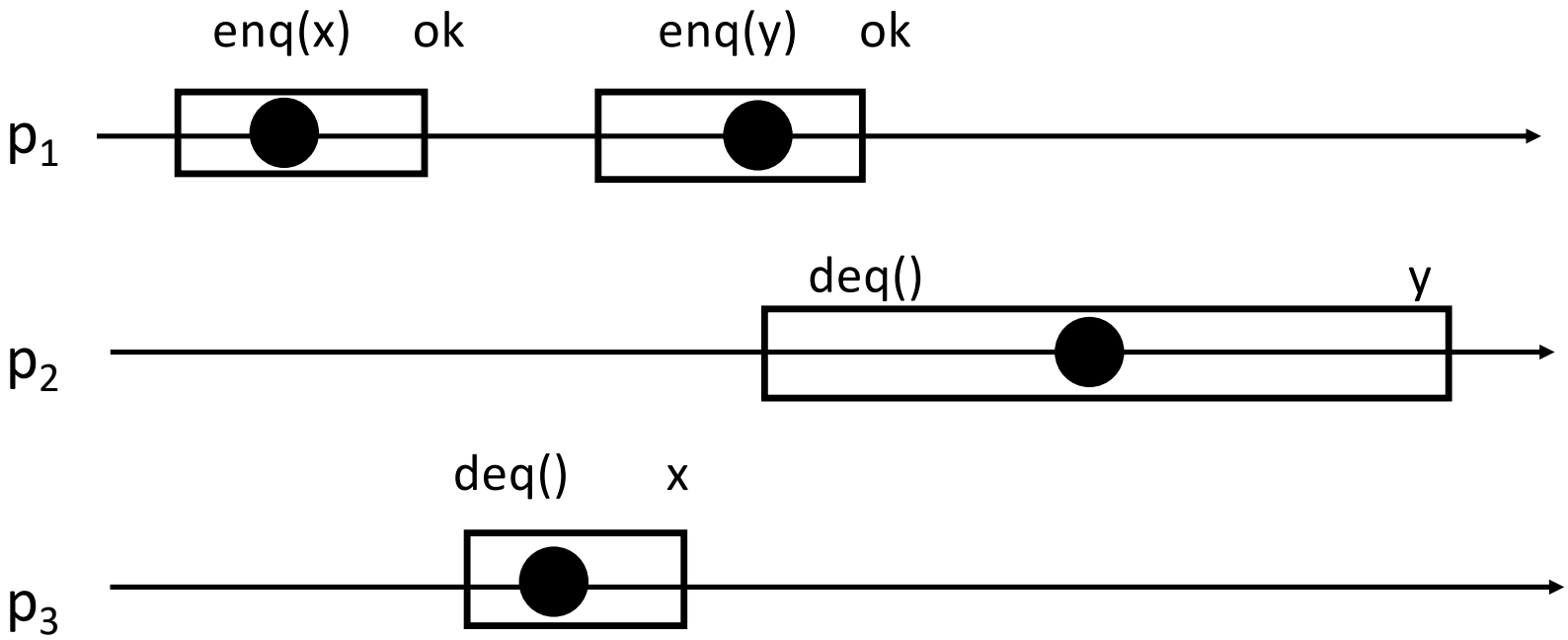
A run of A is *well-formed* if no process invokes a new operation on the implemented object before returning from the old one (we only consider well-formed runs)

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 Obj=(Q,O,R,σ)

Universal Construction: A method for transforming sequential objects into concurrent objects. It allows you to take an object defined by a sequential specification (a set of states Q, a set of operations O, a set of results R, and a transition function σ) and make it safe for concurrent use by multiple processes.

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

You want to implement an object Obj=(Q,O,R,σ)

How would you do it?

使用一个共识对象来序列化对共享对象的并发请求，确保所有进程对这些请求的执行顺序达成一致。

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

在这种构造中，每个进程都可以发布一个请求，这个请求对应于对共享对象的一个操作。所有这些请求被收集起来，并通过共识实例来确定它们的执行顺序。这个序列化确保了即使多个进程同时尝试操作同一个对象，这些操作也会按照一定的顺序执行，从而避免了竞争条件和不一致的状态。

一旦确定了请求的顺序，每个进程都会按照这个顺序来执行请求。由于所有进程都遵循相同的顺序，因此它们对共享对象的看法是一致的，这就确保了操作的正确性。

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

变量 k 通常表示已经完成处理的请求批次的数量。在分布式系统中，请求可能会被分批次地处理，每一批包含多个请求。这个变量 k 就是用来跟踪到目前为止完成了多少批次的请求。

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≠Ø 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 v
- 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,…,N-1):

v :=CS.CAS(Ø,v_i)

return v

如果当前值是 ø，用vi替换当前值，并返回vi

$q_0 = (\perp, 0)$. $O = \{ propose(v), v \in \{0,1\} \}$ $\sigma((\perp,0), prop(v)) = ((v,1), v)$

$Q = \{\perp, 0, 1\} \times N$ $R = \{0,1\}$ $\sigma((v,k), prop(v)) = \begin{cases} ((v,k+1), v) & k < N \\ \perp & \text{otherwise} \end{cases}$

N-consensus object

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

N-consensus: 最先N process同意

For 1st to Nth propose() operations: 只有前N个propose会改变值

- 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

Thm. N+1 processes cannot solve consensus (wait-free) using RW and N-consensus objects.

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?

不可能用N–consensus解决(N+1)–consensus!

一个对象的共识数为k，意味着使用这个对象，我们能够设计出一个算法来解决至多k个进程的共识问题，但无法解决k+1个或更多进程的共识问题。

Consensus number

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

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

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

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

cons (RW) = 1

cons (TAS) = 2

cons (queue) = 2

cons (N-consensus) = N

cons (CAS) = cons (LL/SC) = ∞

Consensus power = consensus number

1- consensus ≅ TAS ⇒ cons(1- consensus) = 2

N > 1 ⇒ cons(N- consensus) = N

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

cons(LL/SC) = ∞

Load Link / Store Conditional

For N>1!

1-consensus is equivalent to TAS

CAS操作能够原子地比较一个内存位置的值，并且仅在它匹配预期的值时才将其替换为新的值。

Quiz 8.2: 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

根据问题的提示，我们可以遵循以下概述来构建证明：

考虑算法A的临界双值运行R。这意味着在R运行中的每一步都是单值的，即它们都将系统状态推进到另一个双值状态。
证明在R中所有可执行的步骤都是在同一个T&S对象上进行的。由于T&S操作是原子的，它要么成功更新值并返回0（表示它是第一个执行操作的），要么发现值已经被更新并返回1。
展示存在两个不同的扩展（即算法的可能继续运行），它们具有相反的价值，但某些进程无法区分这些扩展。这是因为T&S操作不提供关于哪个进程是第一个执行操作的信息。因此，如果存在两个进程都能到达T&S对象并执行操作，那么第三个进程就无法确定应该选择哪个进程的值作为共识的结果。
现在，让我们详细说明这个证明：

【证明步骤】
步骤 1: 确认存在临界双值运行
首先，我们需要证明存在一个临界双值运行R。在这个运行中，系统可能处于一个状态，在该状态下，任一进程的下一步操作都可能导致系统进入一个新的状态，而这个新状态将决定共识结果。我们可以假设在开始时，所有进程都尚未执行T&S操作，并且T&S对象的状态是初始的，意味着第一个执行T&S的进程将能够改变状态。

步骤 2: R中所有步骤都在同一个T&S对象上
在临界双值运行R中，我们假设所有步骤都是针对同一个T&S对象的操作。这是因为T&S设计为一个同步原语，保证了在任何时刻最多只有一个进程能成功执行T&S操作。

步骤 3: 存在两个不可区分的扩展
现在，设想三个进程P1、P2和P3正在执行共识协议，并且它们都到达了临界双值状态R。如果P1和P2几乎同时执行T&S操作，根据T&S的语义，一个将返回0，另一个将返回1。然而，第三个进程P3，它后来到达，将看到T&S对象已经改变，但它没有足够的信息来确定是哪个进程首先到达的，因为T&S操作本身不提供这种区分。因此，即使P1和P2中的一个已经决定了共识的值，P3无法知道应该选择哪个值。

这个无法区分的状态就是导致无法实现3进程共识的根本原因。任何尝试结合T&S和原子寄存器来达成三个进程的共识的尝试都将失败，因为这些进程无法可靠地区分它们应该同意哪个值。这就证明了T&S的共识能力最多为2，而不是3。

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?

Commit-adopt : protocol

Shared objects:

N atomic registers A[0,...,N-1], initially T

N atomic registers B[0,...,N-1], initially T

Upon propose(v) by process p_i:

v_i := v

A[i] := v_i

U := read A[0,...,N-1]

if all non-T values in U are v then

B[i] := (true,v_i)

else

B[i] := (false,v_i)

U := read B[0,...,N-1]

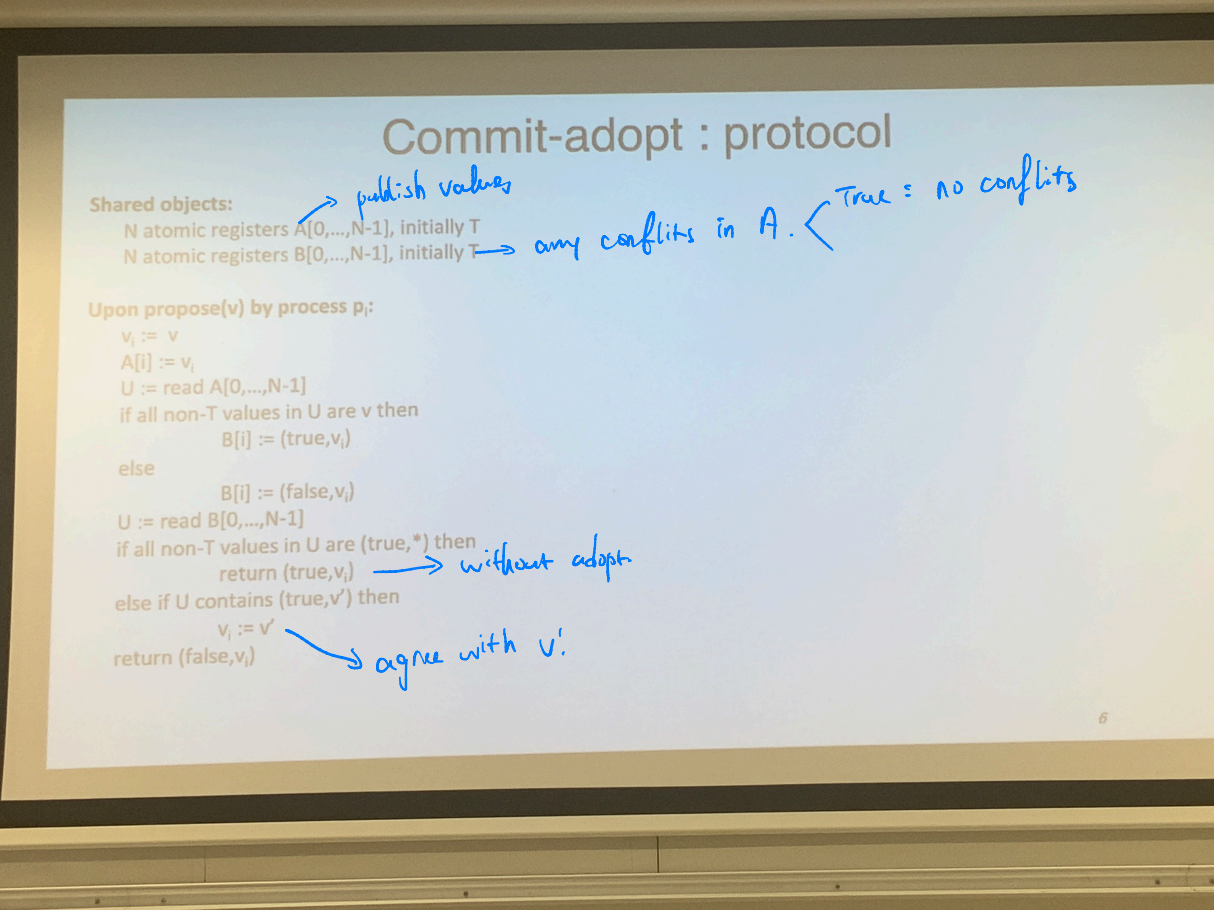
if all non-T values in U are (true,*) then

return (true,v_i)

else if U contains (true,v') then

v_i := v'

return (false,v_i)



situation
un cas

Ω: an oracle

- Eventual leader failure detector
- Outputs (at every process) a process identifier (a leader)
  - ✓‹Ω, p›
- Eventually, the same correct process is output at every correct process

Can be implemented in eventually synchronous system:

- ✓There is a bound on communication delays and processing that holds only eventually
- ✓There is an a priori unknown bound in every run

(Two computationally equivalent conditions)

Consensus = Ω + CA

Shared:

D, a regular register, initially T

$CA_1, CA_2, \ldots$ a series of commit-adopt instances

Upon propose(v) by process p_i:

$v_i := v$

$r := 0$

repeat forever

  r++

  wait until Ω outputs p_i or $D \neq T$

  if $D=v'$ where $v' \neq T$ then

    return v'

  $(c, v_i) := CA_r(v_i)$ // r-th instance of commit-adopt

  if c=true then

    $D := v_i$ // let the others learn your decision

    return v_i

  $v_i := v'$ //adopt the leader's value

Randomized (binary) consensus = CA+local randomness

Shared:

D, a regular register, initially T

$CA_1, CA_2, \ldots$ a series of commit-adopt instances

Upon propose(v) by process p_i:

```
v_i := v
r := 0
repeat forever
    r++
    (c,v_i):=CA_r(v_i)                    // r-th instance of commit-adopt
    if c=1 then
                D:=v_i                    // let the others learn your decision
                return v_i
    else if c=2
                v_i := v'                 //adopt the leader's value
    else
                v_i := Random()           // Pick a random value in {0,1}
```