Distributed algorithms and protocols

Consistency

Recall the properties of distributed systems:

1. concurrent execution of components
2. independent failure modes
3. transmission delay
4. no global time

DS may be large in scale and widely distributed

Replicated objects

objects may be replicated e.g. naming data (name servers), web pages (mirror sites)
- for reliability
- to avoid a single bottleneck
- to give fast access to local copies

Updates

updates to replicated objects
AND
related updates to different objects

must be managed in the light of 1-4 above

Maintaining consistency of Replicas

WEAK CONSISTENCY - fast access requirement dominates
update the "local" replica and send update messages to other replicas.
Different replicas may return different values for an item.

STRONG CONSISTENCY - reliability, no single bottleneck
ensure that only consistent state can be seen (e.g. lock-all, update, unlock).
All replicas return the same value for an item.

WEAK CONSISTENCY - system model, architecture and engineering

* Simple approach: have a PRIMARY COPY to which all updates are made and a number of BACKUP copies to which updates are propagated.

  Keep a HOT STANDBY for some applications for reliability and accessibility (make update to hot standby synchronously with primary update).

  BUT a single primary copy becomes infeasible as systems' scale and distribution increase - primary copy becomes a bottleneck and (remote) access is slow.

* General approach: we must allow (concurrent) reads and writes to all replicas

Weak Consistency of replicas - continued

Requirement: the system MUST be made to converge to a consistent state as the update messages propagate

PROBLEMS **(ref. DS properties 1-4)**

1. concurrent updates at different replicas + (3) comms. delay

- the updates do not, in general, reach all replicas in the same (total) order? - see T-23
- the order of *conflicting* updates matters

2. failures of replicas

we must ensure, by restart procedures, that every update eventually reaches all replicas

4. no global time

but we need at least a convention for arbitrating between conflicting updates
- timestamps? - are clocks synchronised?

e.g. conflicting values for the same named entry - password or authorisation change
e.g. add/remove item from list - distribution list, access control list, hot list
e.g. tracking a moving object - times must make physical sense
e.g. processing an audit log - times must reflect physical causality

In practice, systems may not rely solely on message propagation but also **compare state** from time to time
e.g. name servers - Grapevine, GNS

Further reading:
Y Saito and M Shapiro, "Optimistic Replication" ACM Computing Surveys 37(1) pp.42-81, March 2005

Strong consistency

concept of TRANSACTION / transactional semantics

- ACID properties (atomicity, consistency, isolation, durability)

```
start transaction
        make the same update to all replicas of an object
    or make related updates to a number of different objects
end transaction
      (either COMMIT - all updates are made, are visible and persist
             or ABORT - no changes are made)
```

implementation - first attempt:

```
    lock all objects
    make update(s)
    unlock all objects
```

problem:

- lack of availablity (a reason for replication)
  because of comms. delays, overload/slowness and failures.
  "I can’t work because a replica in Peru crashed"

solution for strong consistency of **replicas**: QUORUM ASSEMBLY

QUORUM ASSEMBLY for replicas

Assume n copies. Define a read quorum QR and a write quorum QW
which must be locked for reading (QR) and writing (QW).

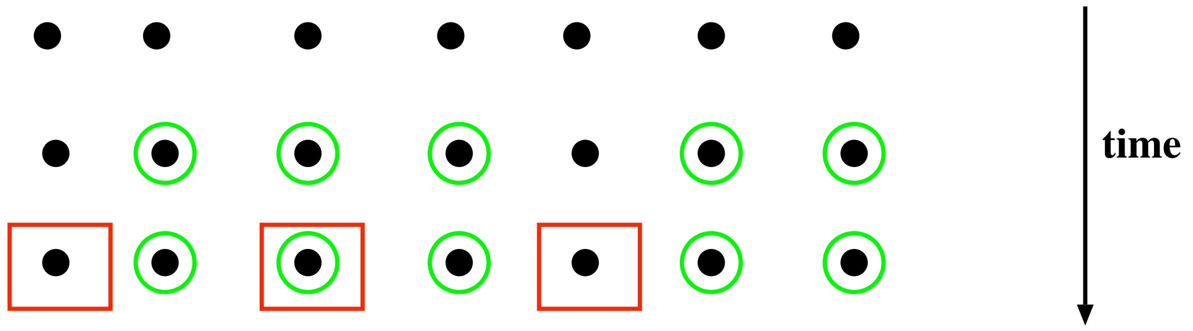
$$QW > n/2$$

$$QR + QW > n$$

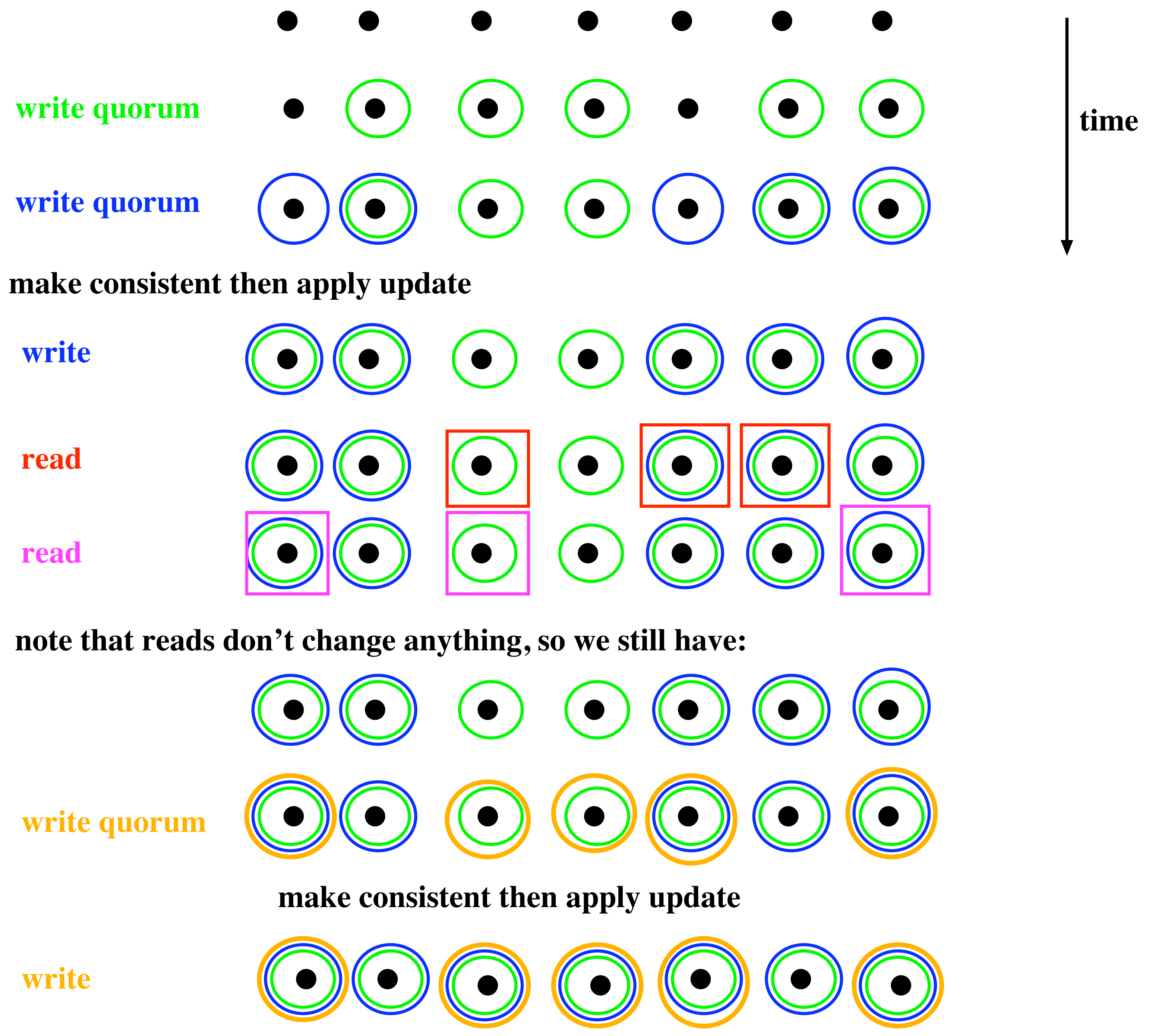
These ensure: only one write quorum at a time can be assembled (deadlock? - see D10)
every QW and QR contain at least one up-to-date replica.
After assembling a (write) quorum, make all replicas consistent then do operation.

e.g. QW = n, QR = 1 is lock all copies for writing, read from any

e.g. n=7, QW=5 ○ QR=3 □



optimisation: after making a write quorum consistent, and performing the update,
background-propagate to other replicas not in the quorum



make consistent then apply update

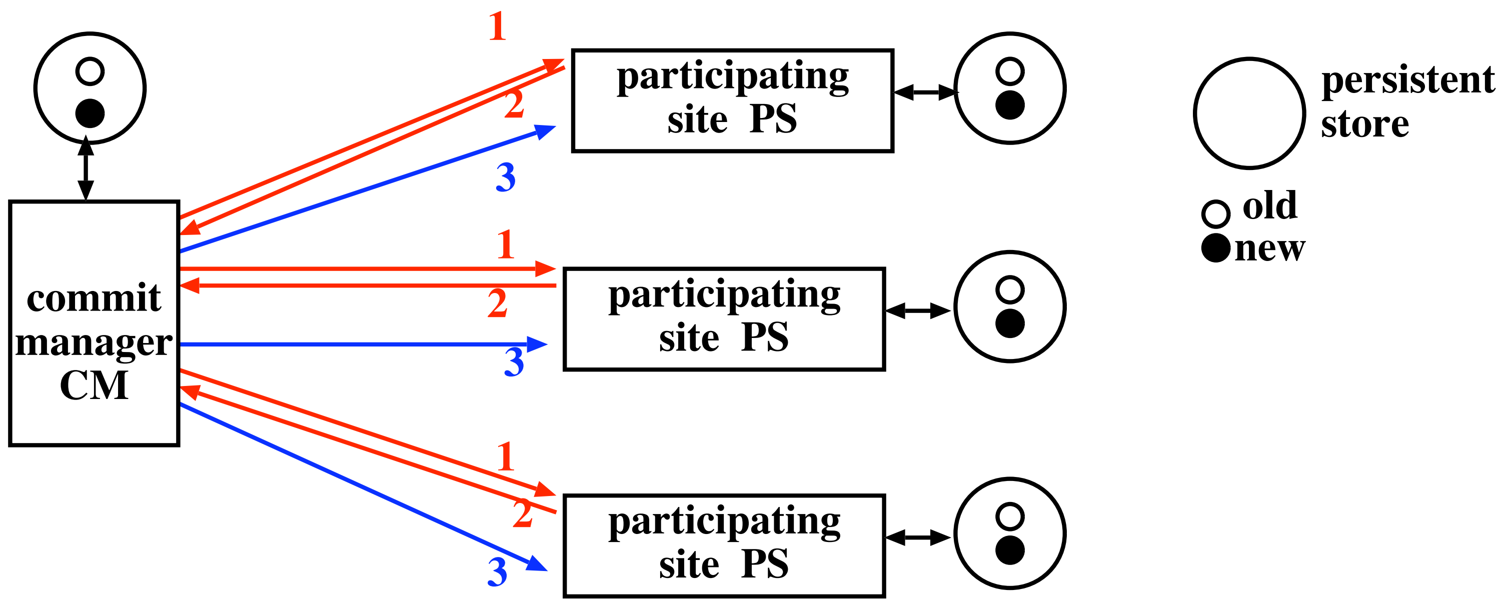
note that reads don’t change anything, so we still have:

make consistent then apply update

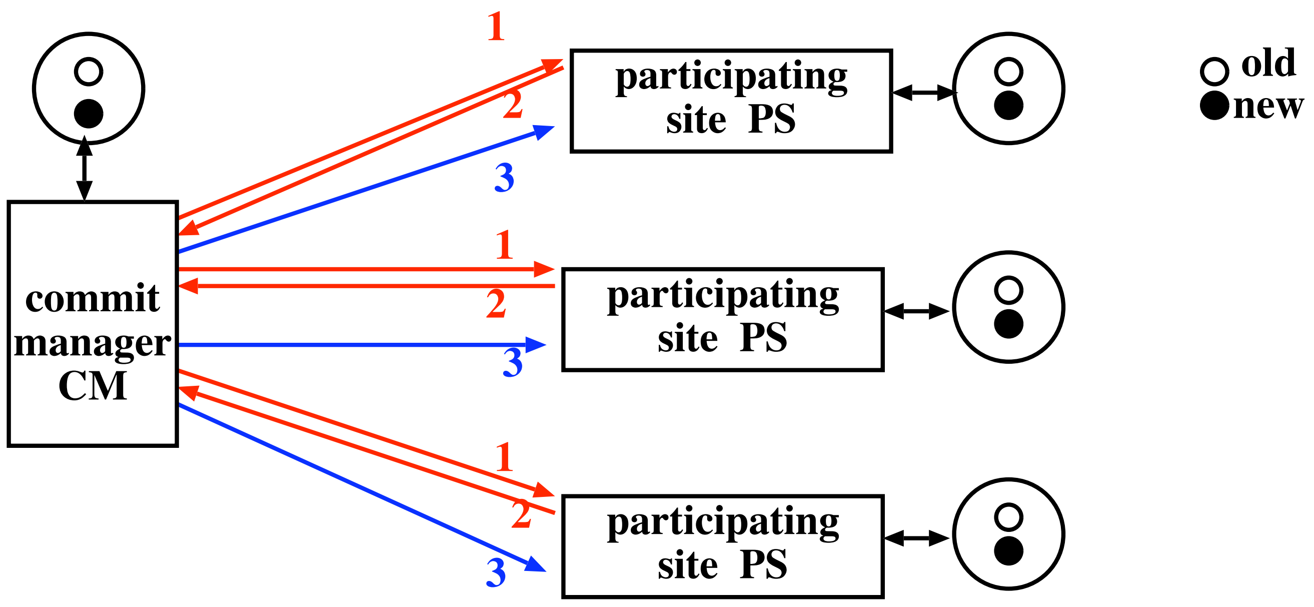
Atomic Update of distributed data

For BOTH quora of replicas AND related objects being updated under a transaction
we need ATOMIC COMMITMENT (all make the update(s) or none does)
- achieved by an ATOMIC COMMITMENT PROTOCOL such as
  two-phase commit (2PC), an ISO standard

Two-phase commit (2PC)



e.g. for a group of four processes:
one functions as commit manager (CM) (after an external update request)
the others are participating sites (PS)



phase 1:

1. CM requests votes from all (PS and CM) - all secure data and vote
2. CM assembles votes including its own

phase 2:

CM decides on commit (if all have voted) or abort

This is the single point of decision - record in persistent store

3. propagate decision

Two-phase commit - notes:

recall - DS - independent failure modes

* before voting commit, each PS must:
- record update in stable/persistent storage
- record that 2PC is in progress

$\downarrow$

crash

$\downarrow$

on **restart,** find out what the decision was from CM

* before deciding commit, CM must:
- get commit votes from all PSs
- record its own update

on deciding commit, CM must
- record the decision
- then . . . propagate . . . it

$\downarrow$

crash

$\downarrow$

on **restart**, tell the PSs the decision

some detail from the PS algorithm

either: send abort vote and exit protocol
or: send commit vote and await decision (set timer)

timer expires: consider CM crash which could have happened:

either before CM decided commit (perhaps awaiting slow or crashed PSs)

or after deciding commit and
before propagating decision to any PS
after propagating decision to some PSs

optimise - CM propagates PS list so any can be asked for the decision

some detail from the CM algorithm

send vote request to each PS
await replies (set timers)

if any PS does not reply, must abort
if 2PC is for quorum update, CM may contact further replicas after abort

Concurrency issues

consider a process group, each managing an object replica
(ref. T-11, assume open group with no internal structure)

suppose two (or more) different updates are requested at different replica managers

two (or more) replica managers attempt to assemble a write quorum
if successful, will run a 2PC protocol as CM

either: one succeeds in assembling a write quorum, the other(s) fail - OK
or: both/all fail to assemble a quorum (e.g. each of two locks half the replicas)
and DEADLOCK

must have deadlock detection or prevention
assume all quorum assembly requests are multicast to all the replica managers
e.g. quorum assembler's timer expires waiting for enough replicas to join
it releases locked replicas and restarts, after backing off for some time

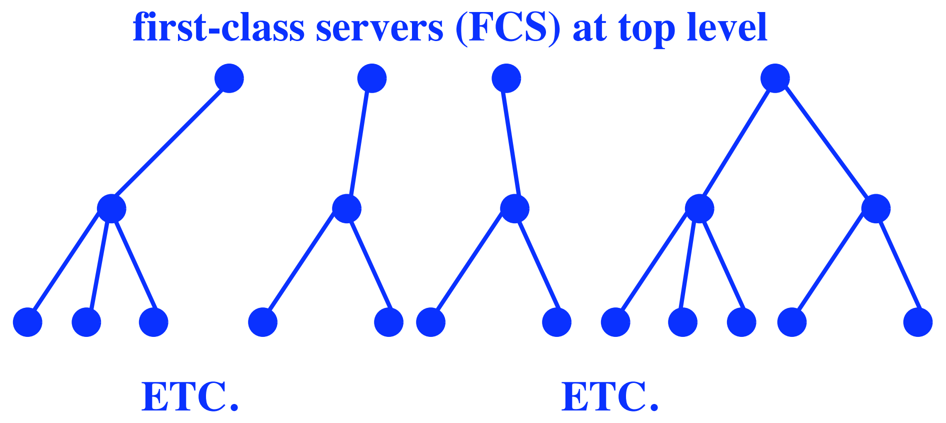
e.g. some replica manager has become part of a quorum (request had a timestamp)
then receives a request from a different quorum assembler (with a timestamp)
All replica managers have an algorithm to agree which quorum wins
and which is aborted, e.g. based on "earliest timestamp wins".

e.g. could use a structured group so update requests are forwarded to the manager

Large-scale systems

it is difficult to assemble a quorum from a large number of widely distributed replicas

manage scale by hierarchy:
define a small set of first-class servers (FCSs), each above a hierarchy of servers



there are various approaches, depending on application requirements, e.g.

* update requests must be made to a FCS

* FCSs use quorum assembly and 2PC among FCSs then propagate the update to all FCSs - each propagates down its subtree(s)

* correct read is from a FCS which assembles a read quorum of FCSs
  fast read is from any server - risk missing latest updates

ref lab TR 383 N Adly (PhD) Management of replicated data in large-scale systems (includes a survey of algorithms for both strong and weak consistency) Nov 1995

Concurrency control: general transaction scenario (distributed objects)

(ref: CS/OS books part 3, CSA course, DB course)

* consider related updates to different objects

* transactions that involve distributed objects, any of which may fail at any time, must ensure atomic commitment

* concurrent transactions may have objects in common

---

pessimistic concurrency control
 (strict) two-phase locking (2PL)
 (strict) timestamp ordering (TSO)

optimistic concurrency control (OCC)
- take shadow copies of objects
- apply updates to shadows
- request commit of a validator which implements commit or abort (do nothing)

---

* with pessimistic concurrency control we must use an atomic commitment protocol such as two-phase commit

* if a fully optimistic approach is taken we do not lock objects for commitment since the validator creates new object versions (ref CS ED1 Ch20, ED2 Ch21, OS Ch 22)

Concurrency control for transactions

strict two-phase locking 2PL

phase 1:

for objects involved in the transaction, attempt to lock object and apply update

- old and new versions are kept
- locks are held while other objects are acquired
- susceptible to DEADLOCK

phase 2:

(for STRICT 2PL, locks are held until commit)

commit update - using e.g. 2PC

----

strict timestamp ordering

each transaction is given a timestamp

for objects involved in the transaction - attempt to lock object and apply update

- old and new versions are kept

the object compares the timestamp of the requesting transaction with that of its most recent update:

if later - OK

if earlier - REJECT (too late) - that transaction aborts

(for STRICT TSO, locks are held until commit)

commit update - using e.g. 2PC

More algorithms and protocols for Distributed Systems

We have defined process groups as having peer or hierarchical structure (ref. T-14)
and have seen that a coordinator may be needed to run e.g. 2PC

With peer structure, an external process may send an update request
to any group member which then functions as coordinator

If the group has hierarchical structure, one member is elected as coordinator

That member must manage group protocols and external requests must be directed to it
(note that this solves the concurrency control (potential deadlock) problem
while creating a single point of failure and a possible bottleneck)

Assume: each process has a **unique ID** known to all members
the process with **highest ID is coordinator**

Note: any process may fail at any time

An ELECTION ALGORITHM is run when the coordinator is detected as having failed

Election algorithm - BULLY

* P notices no reply from coordinator

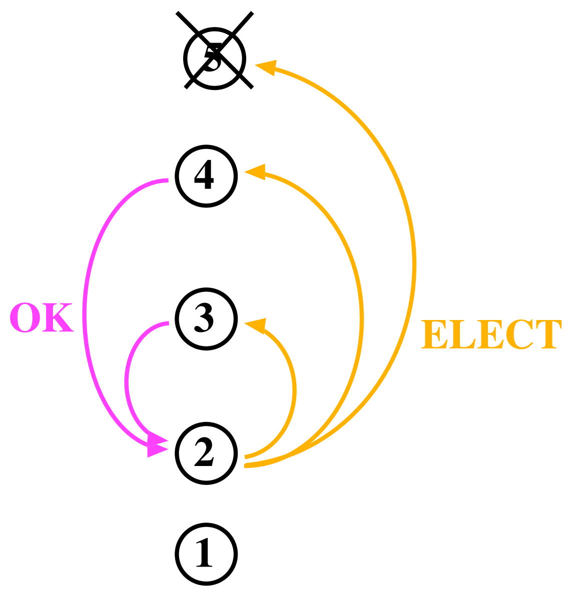
* P sends ELECT message to all processes with higher IDs

* If any reply, P exits

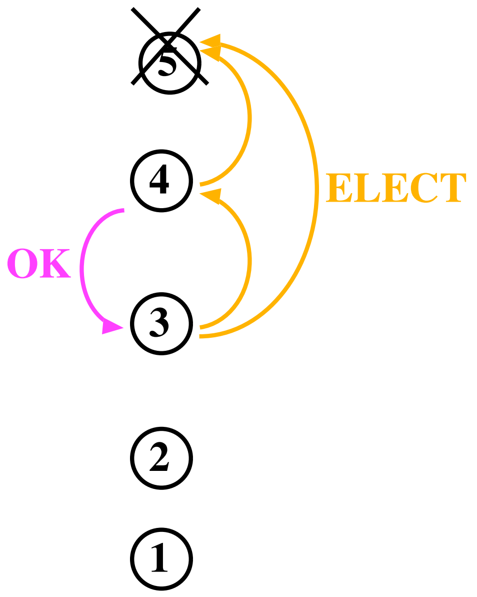
* if none reply, P wins
  P gets any state needed from storage
  P sends COORD message to the group

----

* on receipt of ELECT message
  - send OK
  - hold an election if not already holding one



(i)



(ii)

Election algorithm - RING

Processes are ordered into a ring - known to all
 - can bypass a failed process provided algorithm uses acknowledgements

* P notices coordinator not functioning

* P sends ELECT, tagged with its own ID, around the ring

----

on receipt of ELECT:

without receiver’s ID: append ID and pass on
with receiver’s ID (has been all round) send (COORD, highest-ID)

----

many elections may run concurrently
all should agree on the same highest ID

Distributed mutual exclusion

Suppose N processes hold an object replica
and we require that only one at a time may access the object

examples: - ensuring coherence of distributed shared memory
- distributed games
- distributed whiteboard

i.e. for use by simultaneously running, tightly coupled components managing object replicas in main memory. We have already seen the approach of LOCKING persistent object replicas for transactional update.

Assume - the object is of fixed structure,
- processes update-in-place
- then the update is propagated (not part of the algorithm)

Each process executes code of the form:

```
entry protocol

        critical section (access object)

exit protocol
```

Distributed mutual exclusion:

1. centralised algorithm

one process is elected as coordinator

entry protocol:

send **request** message to coordinator
wait for reply (OK-enter) from coordinator

exit protocol:

send **finished** message to coordinator

+ FCFS or priority or another policy - coordinator reorders

+ economical (3 messages)

- single point of failure

- coordinator is bottleneck

- what does no reply mean? waiting for region? - OK!
  coordinator has failed?

  solve by using extra messages
  - coordinator ack's request
  - send again when process can enter region
  - send periodic heartbeats (at application- or a lower level)

Distributed mutual exclusion:

2. token ring

a token giving permission to enter critical region circulates indefinitely

entry protocol:

wait for token to arrive

exit protocol:

pass token to next process

- inflexible - ring order, not FCFS or priority or ...

+ quite efficient,
  but token circulates when no-one wants region

- must handle loss of token

- crashes? use ack - reconfigure - bypass

Distributed mutual exclusion:

3. distributed (peer-to-peer) algorithm

entry protocol:

send a timestamped request to all processes including oneself
(there is a convention for global ordering of TS)

only when all process have replied can the region be entered

on receipt of a message
- defer reply if in CR
- reply immediately if you are not executing a request
- if you are,
  compare your request message timestamp with that of the message.
  reply immediately if incoming timestamp is earlier, otherwise, defer reply

exit protocol:

reply to any deferred requests

--

+ fair - FCFS

- not economical, 2(n-1) messages + any acks

- n points of failure

- n bottlenecks

- no reply - failure or deferral?