# DS 2010 distributed algorithms and protocols

David Evans
de239@cl.cam.ac.uk

# Consistency

- DS may be large in scale and widely distributed, so maintaining a consistent view of system state is tricky
- 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 to replicated objects AND related updates to different objects must be managed in the light of DS characteristics

# Maintaining consistency of objects

**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

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 increases—primary copy becomes a bottleneck and (remote) access is slow

# Scalable weak consistency

The system *must* be made to converge to a consistent state as the update messages propagate

Tricky in light of fundamental properties of DS:

1, 3. concurrent updates at different replicas + comms. delay

- the updates do not, in general, reach all replicas in the same order
- the order of conflicting updates matters

# Scalable weak consistency

The system *must* be made to converge to a consistent state as the update messages propagate

Tricky in light of fundamental properties of DS:

2. failures of replicas
   - we must ensure, by restart procedures, that every update eventually reaches all replicas

# Scalable weak consistency

The system *must* be made to converge to a consistent state as the update messages propagate

Tricky in light of fundamental properties of DS:

4. no global time

   … but we need at least a convention for arbitrating between conflicting updates

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

   - timestamps? are clocks synchronised?

# Strong consistency

Want transaction 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

# First attempt at strong consistency

lock all objects
make update(s)
unlock all objects

…but this can reduce availability because of comms. delays, overload/slowness, and failures.

# Quorum assembly

Idea: must assemble a read (or a write) quorum in order to read (or write). Ensure that

- only one write quorum at a time can be assembled
- every read and write quorum contains at least one up-to-date replica

Mechanism: suppose there are $n$ replicas.

$$QW > \frac{n}{2}$$

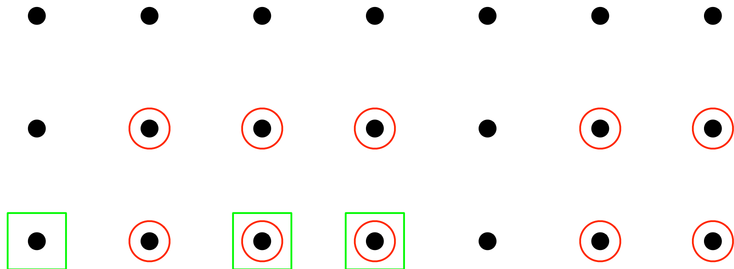
$$QR + QW > n$$

# Quorum examples

- $QW = n, QR = 1$ is lock all copies for writing, read from any

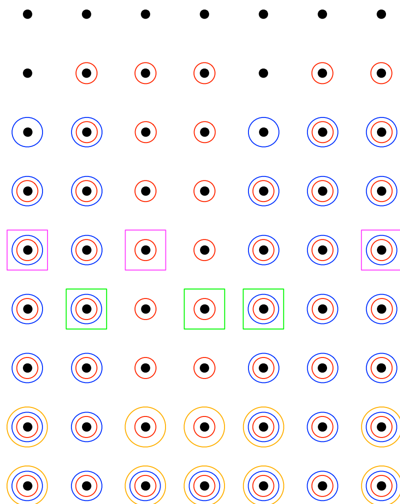
# Quorum examples

- $n = 7, QW = 5, QR = 3$



# Quorum examples

- $n = 7, QW = 5, QR = 3$



# Atomic update of distributed data

For both quorums of replicas and related objects being updated under a transaction we need *atomic commitment* (all make the update(s) or none does)

⇒ need a protocol for this, such as Two-phase Commit (2PC)

- Phase 1
  1. Commit Manager (CM) requests votes from all participants (CM and Participating Sites (PSs))
  2. all secure data and vote
- Phase 2
  1. CM decides on commit (if all have voted) or abort
  2. this is the single point of decision—record in persistent store
  3. propagate the decision

# When 2PC goes wrong

Before voting commit, each PS will;

1. record update in stable/persistent storage
2. record that 2PC is in progress

POW

on restart, a PS must find out what the decision was from CM

# When 2PC goes wrong

Before deciding to commit, the CM must

1. get commit votes from all PSs
2. record its own update (this part is the CM acting like a PS)

on deciding commit, the CM must

1. record the decision
2. then. . . propagate

POW

On restart, it must tell the PSs the decision

# Some detail from the PS algorithm

The idea is to

*send abort vote and exit protocol*

or

*send commit vote and await decision (and set a timer)*

Timer expiry $\Rightarrow$ possible CM crash

1. before CM decided outcome (perhaps awaiting slow or crashed PSs)
2. after deciding commit and
   2.1 before propagating decision to any PS
   2.2 after propagating decision to some PSs

(An optimisation: CM propagates PS list so any can be asked for the decision.)

# Some detail from the CM algorithm

1. send vote request to each PS
2. await replies (setting a timer for each)

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

# Concurrency issues

Consider a process group, each process managing an object replica. Suppose two (or more) different updates are requested at different replica managers. Each replica manager attempts to assemble a write quorum and, if successful, will run a 2PC protocol as CM.
What happens?

- one succeeds in assembling a write quorum, the other(s) fail $\Rightarrow$ everything is OK
- all fail to assemble a quorum (*e.g.*, each of two locks half the replicas) $\Rightarrow$ deadlock!

# Deadlock detection/avoidance in quorum assembly

Assume all quorum assembly requests are multicast to all the replica managers. Can then do one of:

1. quorum assembler's timer expires waiting for enough replicas to join; it releases locked replicas and restarts, after backing off for some time (the "CSMA/CD" approach)
2. timestamp ordering of requests with a consistent tie-breaker
3. use a structured group where 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. So, don't do that! Use a hierarchy of first-class servers (FCSs) and other servers.
Various approaches are possible:

1. update requests must be made to a FCS
2. FCSs use quorum assembly and 2PC among FCSs then propagate the update to all FCSs—each propagates down its subtree(s)
3. *correct read* is from a FCS which assembles a read quorum of FCSs; *fast read* is from any server—risk missing latest updates

# General transaction scenario with distributed 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

In general have two choices

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

  use an atomic commitment protocol such as two-phase commit

# General transaction scenario with distributed 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

In general have two choices

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

  do not lock objects for commitment since the validator creates new object versions

# (Strict) two-phase locking (2PL)

- Phase 1: for each object involved in the transaction, attempt to lock it and apply update
  - old and new versions are kept
  - locks are held while other objects are acquired
  - susceptible to *deadlock*
- Phase 2: commit update, *e.g.*, using 2PC
  - for *strict* 2PL, locks are held until commit

# (Strict) timestamp ordering

Each transaction is given a timestamp.

1. attempt to lock each object involved in the transaction
2. apply the update; old and new versions are kept.
3. after all objects have been updated, commit the update, *e.g.*, using 2PC

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

- if later $\Rightarrow$ everything is OK
- if earlier $\Rightarrow$ *reject* (too late)—the transaction aborts

# Election algorithms

We have defined process groups as having peer or hierarchical structure and have seen that a coordinator may be needed, *e.g.*, to run 2PC.

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).

So, how to pick the coordinator in the face of failures?

# The Bully election algorithm

When $P$ notices the death of the current coordinator

1. $P$ sends ELECT message to all processes with higher IDs
2. if any reply, $P$ exits the election protocol
3. if none reply, $P$ wins!
   3.1 $P$ gets any state needed from storage
   3.2 $P$ sends COORD message to the group

On receipt of an ELECT message

1. send OK
2. hold an election if not already holding one

# The Ring election algorithm

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

When $P$ notices the death of the current coordinator

1. $P$ sends ELECT message, tagged with its own ID, around the ring

On receipt of an ELECT message

1. if it lacks the recipient's ID, append ID and pass on
2. if it contains the recipient's ID, it has been around the ring $\Rightarrow$ 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, *e.g.*, for

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

Assumptions

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

# In general. . .

Processes execute

entry protocol<br>
        critical section (access object)<br>
exit protocol

# A 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

# A centralised algorithm

- + can do FCFS or priority or other policies—the coordinator reorders
- + economical (3 messages)
- – single point of failure
- – coordinator is bottleneck
- – what does no reply mean?
  - waiting for region? ⇒ everything is OK
  - coordinator failure?

  can solve this using extra complexity
  - coordinator ACKs request
  - send ACK again when process can enter region
  - periodic heartbeats

# A distributed algorithm: 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

# A distributed algorithm: token ring

- – only ring order, not FCFS or priority or ...
- + quite efficient, but token circulates when no-one wants the region
- – must handle loss of token
- – crashes? use ACK, reconfigure, bypass, yuck

# A peer-to-peer algorithm

entry protocol
- send a timestamped request to all processes including oneself (there is a convention for global ordering of TS)
- once all process have replied the region can be entered

on receipt of a message
- defer reply if in region
- if you are not processing a message
  - reply immediately
- else
  - compare the current message's timestamp with that of the incoming message. reply immediately if incoming timestamp is earlier; otherwise, defer reply

exit protocol
- reply to deferred requests

# A peer-to-peer algorithm

+ FCFS
– not economical: $2(n-1)$ messages plus any ACKs
– $n$ points of failure
– $n$ bottlenecks
– no reply means... what? failure? deferral?