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
- for reliability
- to avoid a single bottleneck
- to give fast access to local copies

examples: naming data, web pages (mirror sites)

Updates

updates to replicated objects
AND
related updates to different objects

must be managed in the light of 1-4 above

Maintaining consistency

WEAK CONSISTENCY
update the "local" replica and send update messages to other replicas

STRONG CONSISTENCY
ensure that only consistent state can be seen (e.g. lock-all, update, unlock)

Weak consistency - system model, architecture and engineering

* a simple approach is to have a PRIMARY COPY to which updates are made and a number of BACKUP copies
  - keep a HOT STANDBY for some applications (make update to hot standby synchronously)
  - not a good solution for large-scale systems

* for equivalent replicas, we may have replication transparency

  processes may see only one object name and be unaware of replicas at different locations

  a replicated object may be represented at each client process by a "proxy object" which fronts the group of replicas and is responsible for locating the nearest running replica and for propagating updates

  alternatively, a platform may direct an invocation to the nearest running replica and ensure all replicas are updated

Weak Consistency

- the assumption so far is that the system will converge to a consistent state as the update messages propagate

PROBLEMS

(ref. DS properties)

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

   do the updates reach all replicas in the same order?
   (called single-copy serialisability SR1)

2. failures of replicas
   we must ensure that every update reaches all replicas

4. timestamps
   we need a convention for arbitrating between conflicting updates

In practice, state may be compared from time to time in addition to using message propagation

Strong consistency

notion of TRANSACTION

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

```
implementation - first attempt:
    lock all objects
    make update(s)
    unlock
```

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

solution for replicas: QUORUM ASSEMBLY

QUORUM ASSEMBLY

assume n copies
define a read quorum QR and a write quorum QW

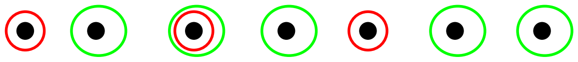
QW > n/2

QR + QW > n

This ensures that every QW and QR contain at least one up-to-date copy

e.g. QW = n, QR = 1 is consistent update

e.g. n=7, QW=5, QR=3



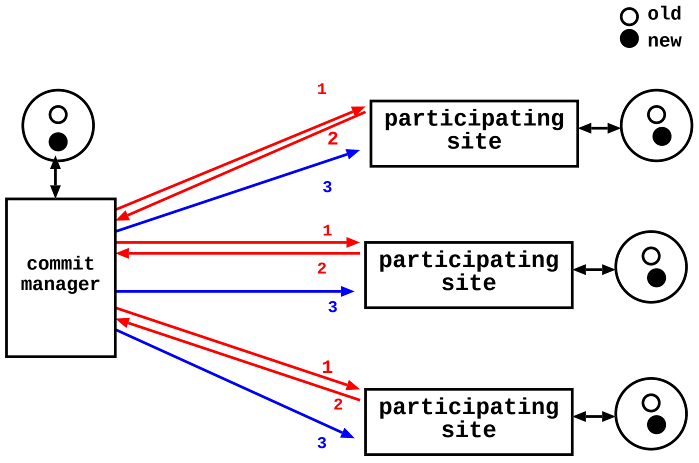
after updating a write quorum, propagate updates to other replicas

For BOTH quora of replicas
AND objects being updated under a transaction
we need ATOMIC COMMITMENT:

an ATOMIC COMMITMENT PROTOCOL such as
two-phase commit (2PC), three-phase commit

Two-phase commit (2PC)

atomic commitment: all make the change or none does



take a group of four processes:
one functions as commit manager
(perhaps on an external request)
the others are participating sites

phase 1:
1. CM requests votes from all - all secure data and vote
2. CM assembles votes including its own

phase 2:
CM decides on commit (if all have voted) or abort
(single point of decision)
3. propagate decision

Two-phase commit - notes:

recall - DS - independent failure modes

```
* before voting commit, each PS must:
    record update in stable storage
    record that 2PC is in progress
  ↓
crash
  ↓
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
  ↓
crash
  ↓
on restart, tell the PSs the decision
```


some detail from the PS algorithm

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

timer expires:
  consider CM crash which could have happened:
     before CM decided commit
         (perhaps awaiting slow or crashed PSs)
     after deciding commit and
           before propagating decision to any PS
           after propagating decision to some PSs

optimise - CM propagates list of PSs
           so they 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
```

3PC specifies error handling in detail

Concurrency control

consider a process group,
each managing an object replica

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

two (or more) object managers attempt to:
assemble a write quorum
run a 2PC protocol as CM

either: one fails to assemble a write quorum,
the other succeeds - OK
or: both fail to assemble a quorum
(e.g. each locks half the replicas)
and deadlock

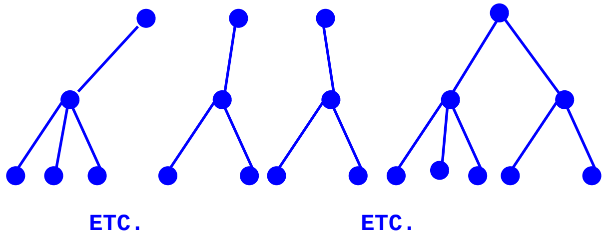
must have deadlock detection
e.g. your timer expires
waiting for locks to be released
- release your locked objects
- restart after backing off for some time
or manage which process defers on deadlock?
or use update timestamps?

Large-scale systems

it is difficult to assemble a quorum
e.g. from all replicas world-wide

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

first-class servers (FCS) at top level



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

* update requests must be made to a FCS

* FCSs use quorum assembly and 2PC
  then propagate the update
  - to all FCSs then down all subtrees

* correct read is from a FCS
  which assembles a read quorum

* fast read is from any server
  risk missing latest updates

ref lab TR 393 N Adly
Management of replicated data in large-scale systems
(includes a survey of algorithms for both
strong and weak consistency)

Concurrency control
general transaction scenario

(Part 2 students: ref IB Concurrent Systems course
and CS book part 3)

consider related updates to different objects
different transactions may have objects in common

as for replica management we need to allow
for concurrent transactions with overlapping objects

as for replica management we need to ensure
atomic commitment of transactions
involving distributed objects, any of which
may fail at any time

---

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 use
an atomic commitment protocol such as
two or three-phase commit

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

Concurrency control for transactions

strict, two-phase locking 2PL

phase 1:
for objects involved in the transaction, attempt:
lock object, 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 or 3PC

---

strict timestamp ordering

each transaction is given a timestamp

for objects involved in the transaction, attempt:
lock object, 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 - that transaction aborts

(for STRICT TSO, locks are held until commit)

commit update - using e.g. 2PC or 3PC

More algorithms and protocols for Distributed Systems

we have defined process groups as having
peer or hierarchical structure
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
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

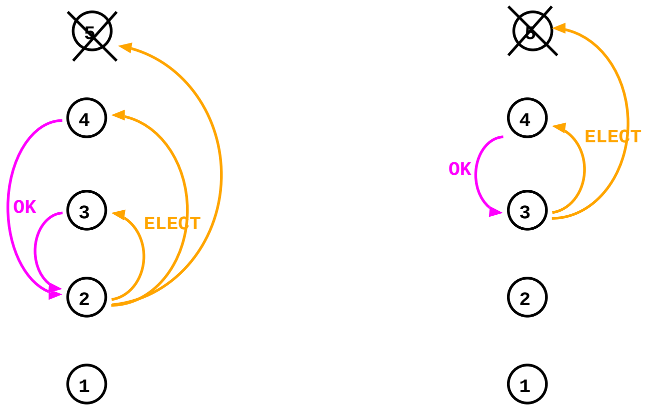
* P sends ELECT message
  to all processes with higher IDs

* 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
  - algorithm needs 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 game
          - distributed whiteboard

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 region (access object)

exit protocol
```

distributed mutual exclusion:

1. centralised algorithm

one process is elected as coordinator

entry protocol:

request message to coordinator
reply OK

exit protocol:

finished message to coordinator

+ fair: FCFS or priority - coordinator reorders

+ economical (3 messages)

- single point of failure

- coordinator is bottleneck

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

  use extra messages
  - ack request
  - send again when process can enter region

distributed mutual exclusion:

2. token ring

a permission to enter critical region token circulates indefinitely

entry protocol:

wait for token to arrive

exit protocol:

pass token to next process

- not fair - ring order, not FCFS or priority

+ 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 Ts is earlier
  otherwise, defer reply

exit protocol:

reply to any deferred requests

+ fair - FCFS

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

- n points of falure

- n bottlenecks

- no reply - failure or deferral?