Consistency and distributed algorithms

Consistency in Distributed Systems

Recall the fundamental DS properties – DS may be large in scale and widely distributed

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

Consistency is an issue for both:

- replicated objects
- transactions involving related updates to different objects (recall ACID properties)

We first study replication and later distributed transactions

Objects may be replicated for a number of reasons:

- reliability/availability - to avoid a single point-of-failure
- performance - to avoid overload of a single “bottleneck”
- to give fast access to local copies – to avoid communications delays and failures

Examples of replicated objects:

Naming data

for name-to-location mapping, name-to-attribute mapping in general

Web pages

mirror sites world-wide of heavily used sites

Maintaining Consistency of Replicas

Weak consistency – for when the "fast access" requirement dominates.

- update some replica, e.g. the closest or some designated replica
- the updated replica sends update messages to all other replicas.
- different replicas can return different values for the queried attribute of the object the value should be returned, or "not known", with a timestamp
- in the long term all updates must propagate to all replicas
  - consider failure and restart procedures,
  - consider order of arrival,
  - consider possible conflicting updates

Strong consistency – ensures that only consistent state can be seen.

All replicas return the same value when queried for the attribute of an object. This may be achieved at a cost – high latency.

Engineering weak consistency of replicas

- Simple approach
  all updates are made to a PRIMARY COPY
  the primary copy propagates updates to a number of backup copies
  note that updates have been *serialised* through the primary copy
  queries can be made to any copy
  the primary copy can be queried for the most up-to-date value
  example: DNS uses an authoritative name server per domain plus a few replicas
  performance?
  for other than small-scale systems, the primary copy will become a bottleneck
  and update access will be slow - to the location of the primary copy from everywhere.
  So have different primary copy sites for different data items.
  reliability? availability?
  the primary copy could fail!
  have a HOT STANDBY to which updates are made synchronously with the primary copy

- General, scalable approach:
  Distribute a number of first-class replicas.
  We now have to be aware that *concurrent updates and queries* can be made.

Weak consistency of replicas - issues

The system must converge to a consistent state as the updates propagate.

Consider DS properties:

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

1 and 3: concurrent updates and communications delay

- the updates do not, in general, reach all replicas in the same (total) order
- the order of *conflicting* updates matters
- conflicts must be resolved, semantics are application-specific, see also below re. timestamps

2: failures of replicas

Restart procedures must be in place to query for missed updates

………

Weak consistency of replicas – issues (contd.)

The system must converge to a consistent state as the updates propagate.

Consider DS properties:

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

………

4: no global time – are clocks synchronised?
but we need at least an ordering convention for arbitrating between conflicting updates
e.g. conflicting values for the same named entry – change of password or privileges
e.g. add/remove item from list – DL, ACL, 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 will not rely solely on message propagation but also compare state from time to time, e.g. Name servers – Grapevine, GNS, DNS

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

Strong Consistency of Replicas (and in Transactions)

Transactional semantics: ACID properties (Atomicity, Consistency, Isolation, Durability)

start transaction
make the same update to all *replicas*
or make *related updates* to a number of *different objects*
end transaction
(either: ***commit*** – all changes are made, are visible and persist
or: ***abort*** – no changes are made to any object)

First consider implementation of strongly consistent replication
See later for distributed transactions.

First thoughts – ***update***: lock all objects, make update, unlock all objects ?
query: read from any replica

Strong Consistency of Replicas

Problems with locking all objects to make an update:

- Some replicas may be at the end of slow communication lines
- Some replicas may fail, or be slow or overloaded
- So: Lack of availability of the system (a reason for replication)
  i.e. delay in responding to queries.
  This is because of the slowness of the update protocol due to
  communications failures or delays,
  replica failure or delays
- Intolerable if no-one can update or query
  because one (distant, difficult-to-access) replica fails

So we try a majority voting scheme - QUORUM ASSEMBLY

A solution for strong consistency of **replicas**.

Quorum Assembly for replicas

Assume n copies. Define a read quorum QR and a write quorum QW,
where QR must be locked for reading and QW must be locked for writing, such that:

$QW > n/2$

$QW + QR > n$

These ensure that

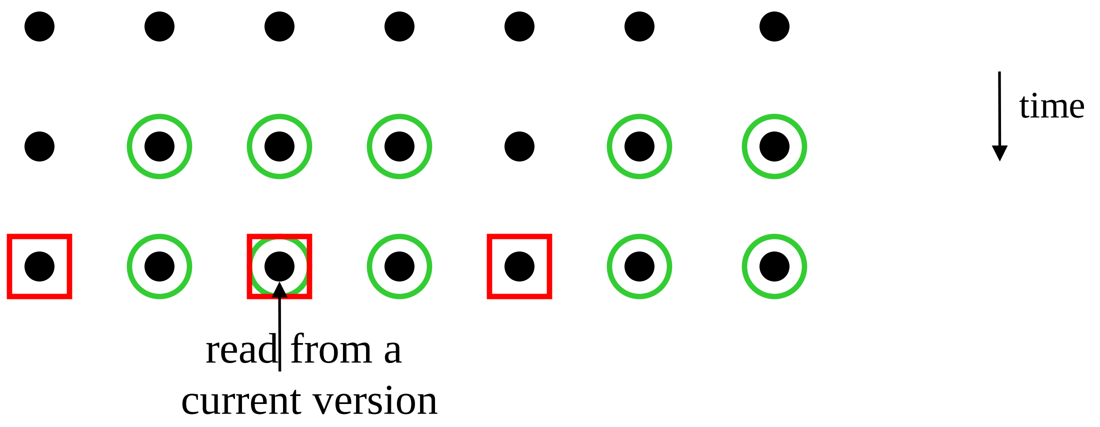
only one write quorum can successfully be assembled at any time ($QW > n/2$)

every QW and QR contain at least one up-to-date replica ($QW + QR > n$)

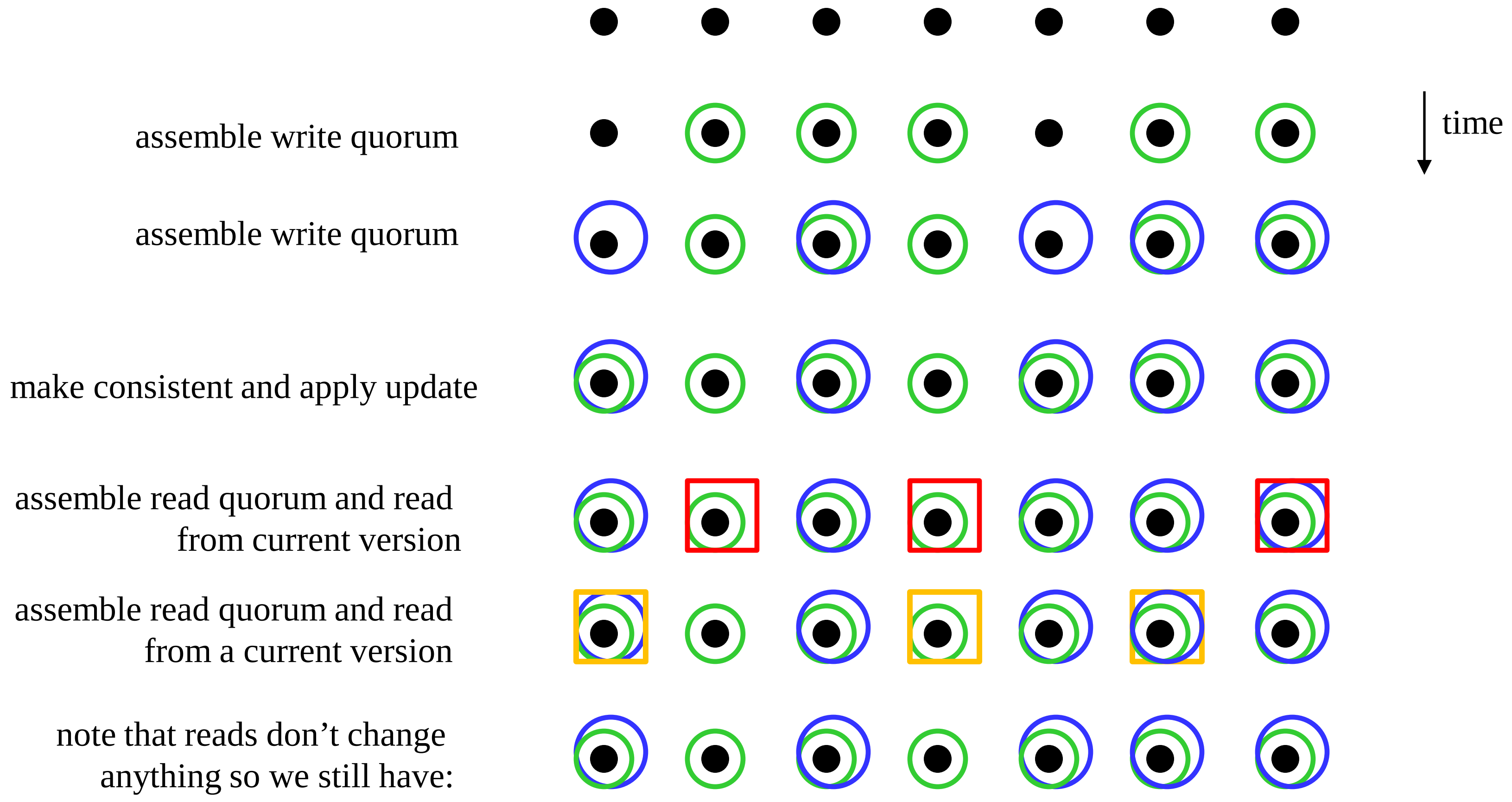
After assembling a (write) quorum QW, bring all replicas up-to-date then make the update.

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

e.g. $n = 7$, $QW = 5$ ○ $QR = 3$ □



Example continued with n = 7, QW = 5, QR = 3

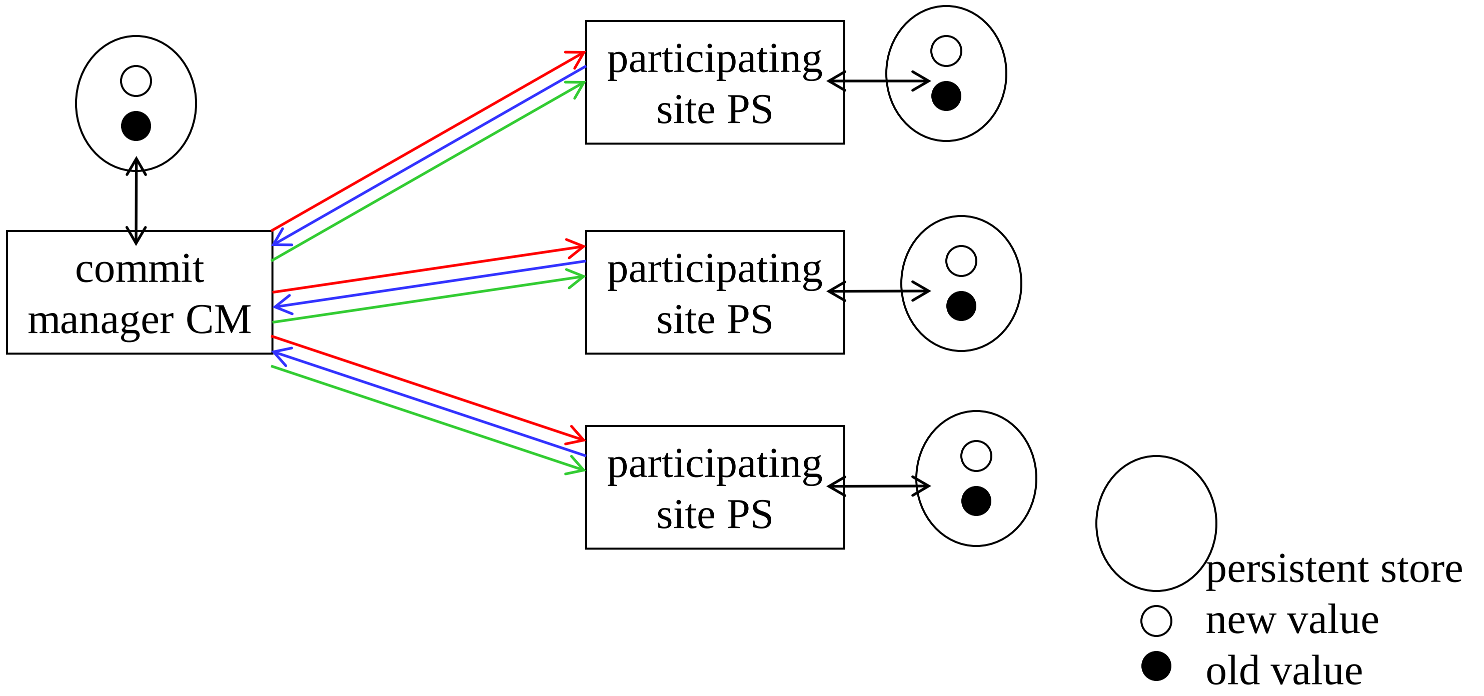

assemble write quorum
assemble write quorum
make consistent and apply update
assemble read quorum and read from current version
assemble read quorum and read from a current version
note that reads don't change anything so we still have:
time

Distributed atomic update for replicas and transactions

atomic commitment

atomic commitment protocol

two-phase commit (2PC)



For a group of processes, one functions as commit manager CM, and runs the 2PC protocol, the others are participating sites PS, and participate in the 2PC protocol

Two-phase commit

Phase 1

1. CM requests votes from all (PS and CM) – all secure data to persistent store then vote
2. CM assembles votes including its own

Phase 2

CM decides on commit (if all have voted commit) or abort (if any have voted abort)
This is the single point of decision of the algorithm

1. CM propagates decision to all PSs

Failures during 2PC, recall DS independent failure modes
Before voting commit each PS must

- record in persistent store that 2PC is in progress
- save the update to persistent store

.....crash .... at this point

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

Before deciding commit, CM must

- record in persistent store that 2PC is in progress
- record its own update to persistent store
- collect all votes, including its own

On deciding commit, CM must:

- record the decision in persistent store ..... crash before propagation ....
- propagate the decision ..... crash during propagation ....

On restart, lookup the decision and propagate it to all PSs

2PC: some detail from the CM and PS algorithms

PS algorithm
Either send abort vote and exit protocol or send commit and await decision (set timer)

Timer expires
- CM could be waiting for slow PSs
- CM crash before deciding commit
- CM crash after deciding commit
    - before propagating to any PSs
    - after propagating to some PSs
        - optimise – CM sends list of PSs – ask any for decision

CM algorithm
- send vote request and await replies – set timer
- if any PS does not reply, decide abort and propagate decision

2PC for quorum update
After abort, contact more replicas and start 2PC again
Perhaps assemble more than the required write quorum and commit if a quorum vote commit.

Concurrency in Quorum Assembly

Consider a process group, each member managing a replica
Assume the group is open and unstructured
– any member can take an external update request

Suppose two or more different update requests are made at different replica managers

Each attempts 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
or: both/all fail to assemble a quorum
– e.g. each of two lock half the replicas
we have DEADLOCK

Concurrency in Quorum Assembly – Deadlock prevention

Can deadlock be prevented or detected?

Assume all quorum assembly requests are multicast to all group members

1. The quorum assembler's timeout expires, waiting for enough replica managers to join. It could release locked replicas and restart after backing off for a time.

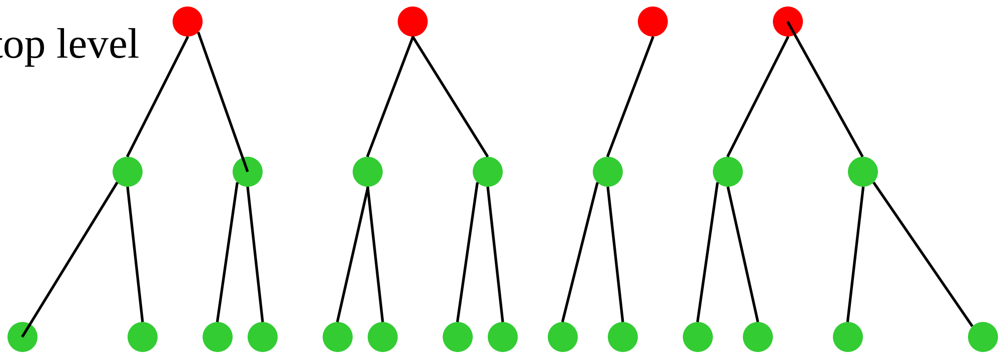
1. Some replica manager has become part of a quorum (request had a timestamp) then receives a request from a different quorum assembler (with a timestamp) All replica managers agree who wins e.g. based on earliest timestamp wins. Members of the losing quorum can send abort, and exit, then join the winning quorum.

1. Use an open, structured* group so update requests are forwarded to the manager, which is always the CM.
   (* recall lecture DS-2: process groups and message ordering).

Quorum Assembly in large-scale systems

It is difficult to assemble a quorum from a *large number of widely distributed replicas*.
- manage the *scale* by using a *hierarchy*:
  define a small set of first class servers (FCS) each above a subtree of servers

FCS at top level



There are various approaches, based on application requirements such as:
speed of response, consistency etc.

Example:

- Update requests must be made to a FCS
- FCSs use quorum assembly and 2PC among FCSs then propagate the update to all FCSs
- Each FCS propagates down its own subtree
- Correct read is from a FCS which assembles a read quorum
- Fast read (value + timestamp) is from any server – with the risk of missing most recent updates

Concurrency Control for Distributed **Transactions**

- Transactions comprise related updates to (distributed) objects
- Any object may fail independently at any time in a DS.
- Distributed atomic commitment must be achieved e.g. by two-phase commit (2PC)
- Concurrent transactions may have objects in common.

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

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

For pessimistic CC, atomic commitment is achieved by a protocol e.g. 2PC
  note that strict pessimistic CC must be used (objects locked until after commit) – see below

If a fully optimistic approach is taken we do not lock objects for commitment,
  since a validator commits new object versions atomically – see below.

Pessimistic Concurrency Control for Distributed Transactions

Strict two-phase locking 2PL

Phase 1:

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

- locks are held while others are acquired, to avoid cycles in the serialisation graph
- so susceptible to DEADLOCK (indicating a SG cycle would have occurred)

Phase 2:

(for strict 2PL locks are held until after commit)

Commit update using e.g. 2PC protocol

Strict timestamp ordering TSO

Each transaction is given a timestamp

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

The object compares the timestamp of the requesting transaction with that of its most recent update. If later – OK. If earlier REJECT as TOO LATE – the transaction aborts and restarts with a later timestamp.

(for strict TSO, locks are held until commit)

Commit update using e.g. 2PC

Implementation of OCC

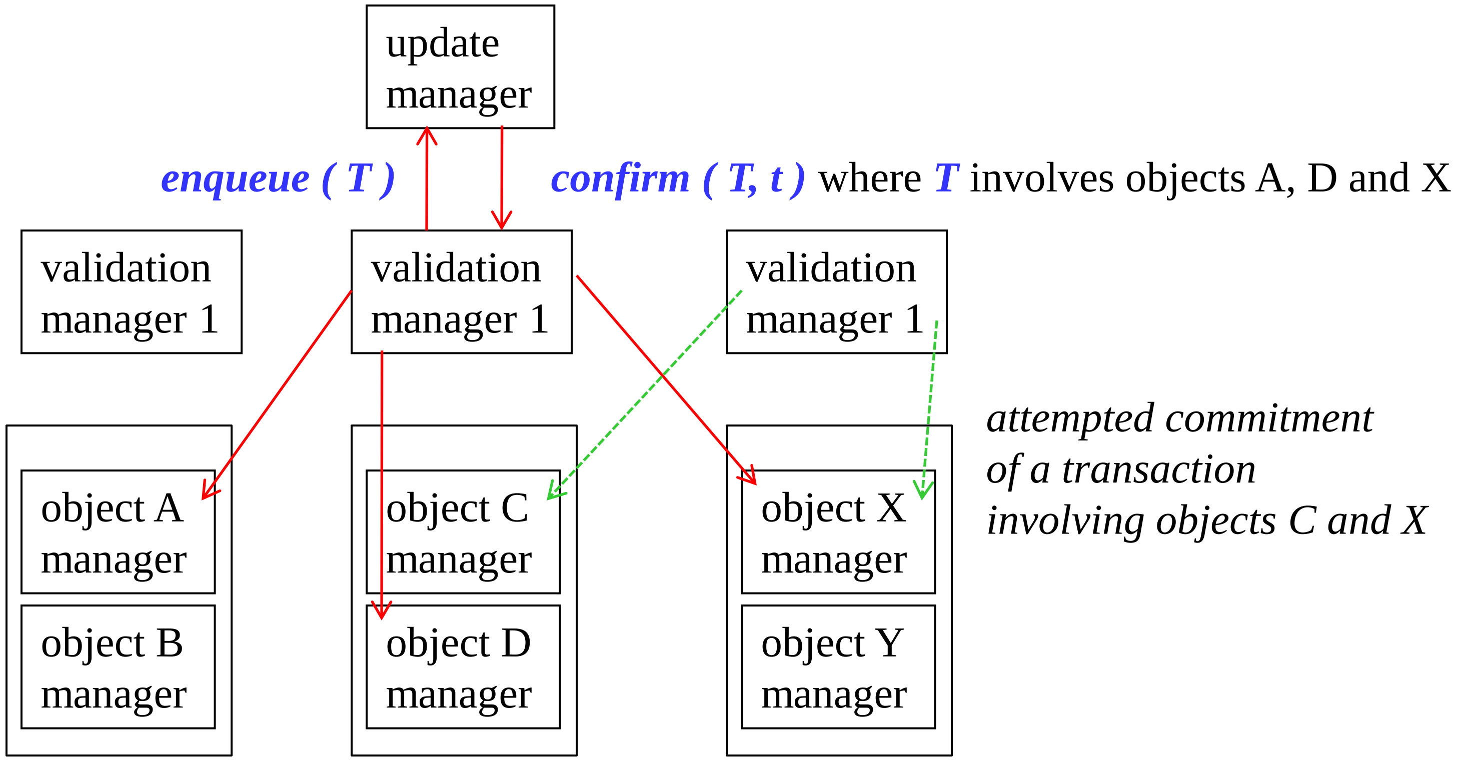
If a fully optimistic approach is taken we do not lock objects until after commitment, since a validator commits new object versions atomically.

The next three slides show single-threading of transactions' commitment through requesting ***commit*** of the validator.

These are included for completeness and further study. They will not be lectured or examined.

Two-phase validation for Optimistic Concurrency Control

Assumptions:
A validator per node – the validator at the coordinating node runs the commitment protocol.
A single, system-wide update manager, responsible for the queue of transactions validated for update
Each object votes independently ***accept, reject*** on whether there has been a conflict; ***accept*** is sent with the version timestamp of the shadow object.



attempted commitment of a transaction involving objects C and X

Two-phase validation for OCC – notes 1

Differences from atomic commitment

1. an object may participate in several protocols concurrently

2. if all objects vote ***accept*** in the first phase and the transaction is validated successfully, they do not need to apply the updates in the second phase.
   Instead, the VM applies for a timestamp from the update manager
   – at this point the serialisation order is determined.
   This interaction is atomic.
   The VM must then inform each participating object of the decision.

Two-phase validation for OCC – notes 2

Phase 1 outline

The VM requests and assembles votes for ***accept*** or ***reject*** from each participating object,

except that some may say ***busy*** – try again later,

if they are involved in a concurrent transaction.

If any vote is ***reject***, the transaction is aborted.

All votes must be ***accept*** before ***commit*** can be accomplished.

If any vote is ***busy***, all objects are told to suspend validation for a subsequent retry.

Objects that voted ***accept*** are then free to validate other transactions.

On retry, objects that originally voted ***accept*** may vote ***reject***.

Phase 2 outline

The VM decides to ***commit, abort*** or ***retry*** on the basis of the votes,

taking account of shadow object consistency.

If the decision is ***commit***, the VM applies to the update manager for a timestamp.

The decision is propagated to participating objects, with the timestamp.