13 Transactions: Coping with Failure

- Failure Modes
- Recovery Techniques
- Partitions and quorum voting

13.1 Failure Modes

For Transactions to be *atomic* and *durable*, need to examine failures

1. Transaction-local failures, detected by the application which calls **abort** eg *insufficient funds*. No info loss, need to undo changes made.
2. Transaction-local failures , not detected by application, but by system as whole, eg *divide by zero*. System calls abort.
3. System failures affecting transactions in progress but not media eg CPU failure. Loss of volatile store and possibly all transactions in progress. On recovery, special recovery manager undoes effects of all transactions in progress at failure.
4. Media failures affecting database eg *head crash*. No way of protecting against this.

13.2 Recovery

- We assume a machine crashes, but then is fixed and returns to operation[1].
- We need to recover state to ensure that the guarantees of the transactional systems are kept.
- Use a recovery file or log that is kept on permanent storage.

13.2.1 The Recovery Manager

- Recovery from failure handled by entity called **Recovery Manager**.
- Keeps information about changes to the resource in a recovery file (also called Log) kept in stable storage - ie something that will survive failure.
- When coming back up after failure, recovery manager looks through recovery file and undoes changes (or redoes changes) so as uncommitted transactions didn't happen, and committed transactions happened.
- Events recorded on Recovery file for each change to an object in database.

13.2.2 Recovery File

Information recorded per event include:

Transaction Id To associate change with a transaction

Record Id The identifier of the object

Action type Create/Delete/Update etc

[1]hence sidestepping the impossibility of byzantine agreement in asynchronous systems, because we assume the machine *always* returns

Old Value To enable changes to be undone

New Value To enable changes to be redone

Also log beginTransaction, prepareToCommit, commit, and abort actions, with their associated transaction id.

13.2.3 Recovering

If after failure,

- the database is undamaged, *undo* all changes made by transactions executing at time of failure
- the database is damaged, then restore database from archive and *redo* all changes from committed transactions since archive date.

The Recovery file entry is made and committed to stable storage before the change is made - incomplete transactions can be undone, committed transactions redone.
What might happen if database changed before recovery file written?
Note that recovery files have information needed to undo transactions.

13.2.4 Checkpointing

Calculation of which transaction to undo and redo on large logs can be slow.
Recovery files can get too large
Instead, augment recovery file with *checkpoint*

- Force recovery file to stable storage
- Write *checkpoint record* to stable store with
    1. A list of currently active transactions
    2. for each transaction a pointer to the first record in recovery file for that transaction
- Force database to disk
- Write address of checkpoint record to *restart location* atomically

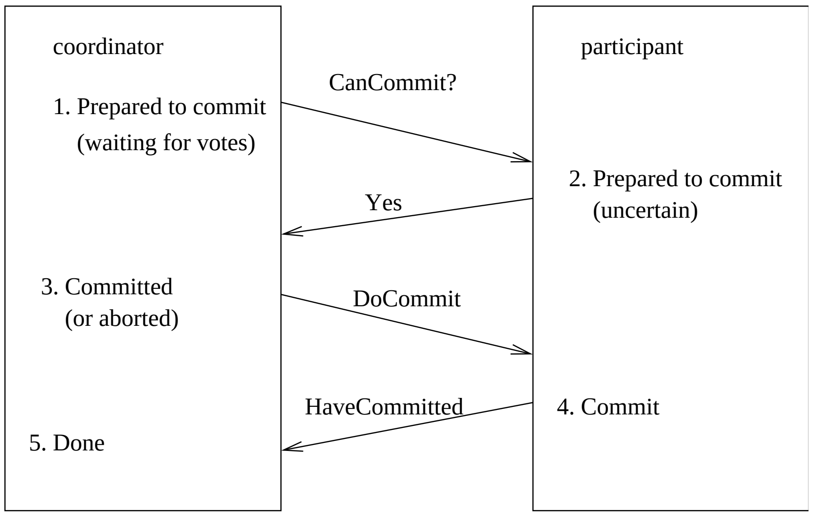
13.2.5 Recovering with checkpoints

To recover, have undo and redo lists. Add all active transactions at last checkpoint to undo list

1. Forwards from checkpoint to end,
    (a) If find beginTransaction add to undo list
    (b) If find commit record add to redo list
    (c) If find abort record remove from undo list
2. backwards from end to first record in checkpointed transactions, execute undo for all transaction operations on undo list
3. Forwards from checkpont to end, redo operations for transactions on redo list

At checkpoint can discard all recovery file to first logged record in checkpointed transactions

13.2.6 Recovery of the Two Phase Commit Protocol

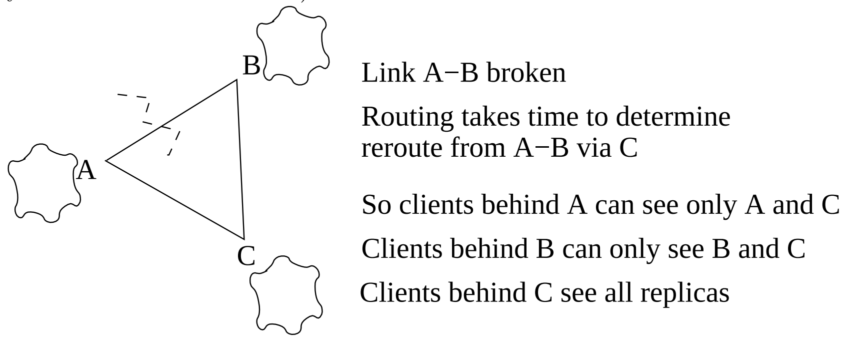


- Coordinator uses *prepared* to signal starting protocol, *commit* on signalling DoCommit and *done* to indicate end of protocol in recovery file.
- Participant uses *uncertain* to indicate that it has replied *yes* to commit request, and *commited* when it receives DoCommit.
- On recovery, coordinator aborts transactions which reach *prepared*, and resends DoCommit when in *commit* state but not *done*
- Participant requests decision from coordinator if in *uncertain* state, but not commited.

13.3 Network Partition

Transactions are often used to keep replicas consistent.

If network partitions (cable breaks), replicas divided into two or more sets (possibly with common members).


B
A
C
Link A−B broken
Routing takes time to determine
reroute from A−B via C
So clients behind A can see only A and C
Clients behind B can only see B and C
Clients behind C see all replicas

Can we still write and read from any of the sets?
Yes, but

- Must reduce possible read and write sets to maintain consistency
- Or relax consistency requirements and resolve problems when partition is healed

13.3.1 Quorum Consensus

Consider set of replicas, where replicated objects have version numbers at each replica.

1. Assign a weighting of votes to each replica, indicating importance.
2. For client to perform operation, it must gather votes for all the replicas it can talk to (denote X).
3. $X \geq$ votes set for read quorum R to enable read
4. $X \geq$ votes set for write quorum W to enable write.
5. As long as
   - $W >$ half the total number of votes
   - $R + W >$ total number of votes in group

   Each Read quorum and each write quorum will have at least one member in common.

13.3.2 Partition Example

For three replicas, R1, R2, R3, we can allocate votes to give different properties depending upon requirements

Replica	config 1	config 2	config 3
R1	1	2	1
R2	0	1	1
R3	0	1	1

1. What should R and W be set to in the three configurations?
2. What properties result from these configurations?

13.3.3 Read and Write Operations

When write happens, object has a version number incremented

To read, collect votes from replica managers with version numbers of object. Guaranteed to have at least one up to date copy if in read quorum, from which read occurs.

To write, collect votes from replica managers with version numbers of object. If write quorum with up to date copy not discovered, then copy up to date copy around to create write quorum. Then write is allowed.

Manipulating R and W give different characteristics eg $R = 1$ and $W =$ number of copies gives unaminous update.

Cached copies of objects can be incorporated as *weak representatives* with 0 votes, but usable for reads.

13.4 Summary

- Atomicity comes from using logging techniques on operations at server, where log is kept on stable storage
- Voting can be used to give availability for resources on partitioned replicas.