Atomic Transactions

- **Transaction**
  - An operation composed of a number of discrete steps.

- **All the steps must be completed for the transaction to be committed.** The results are made permanent.

- Otherwise, the transaction is **aborted** and the state of the system **reverts** to what it was before the transaction started.
  - **rollback** = reverting to a previous state

Example

- Buying a house:
  - Make an offer
  - Sign contract
  - Deposit money in escrow
  - Inspect the house
  - Critical problems from inspection?
  - Get a mortgage
  - Have seller make repairs
  - **Commit**: sign closing papers & transfer deed
  - **Abort**: return escrow and revert to pre-purchase state

  **All or nothing property**

Basic Operations

Transaction primitives:

- **Begin transaction**: mark the start of a transaction
- **End transaction**: mark the end of a transaction — no more tasks
- **Commit transaction**: make the results permanent
- **Abort transaction**: kill the transaction, restore old values
- **Read/write/compute** data (modify files or objects)
  - But data will have to be restored if the transaction is aborted.

Another Example

Book a flight from Newark, New Jersey to Inyokern, California. No non-stop flights are available:

- **Transaction begin**
  1. Reserve a seat for Newark to Denver (EWR→DEN)
  2. Reserve a seat for Denver to Los Angeles (DEN→LAX)
  3. Reserve a seat for Denver to Inyokern (LAX→IYK)

- **Transaction end**

If there are no seats available on the LAX→IYK leg of the journey, the transaction is **aborted** and reservations for (1) and (2) are undone.

Properties of transactions: **ACID**

- **Atomic**
  - The transaction happens as a single indivisible action. Everything succeeds or else the entire transaction is rolled back. Others do not see intermediate results.

- **Consistent**
  - A transaction cannot leave the database in an inconsistent state. If the system has invariants, they must hold after the transaction. E.g., total amount of money in all accounts must be the same before and after a "transfer funds" transaction.

- **Isolated** (Serializable)
  - Transactions cannot interfere with each other
  - If transactions run at the same time, the final result must be the same as if they executed in some serial order.

- **Durable**
  - Once a transaction commits, the results are made permanent. No failures after a commit will cause the results to revert.
Nested Transactions

• A top-level transaction may create subtransactions

• Problem:
  – Sub-transactions may commit (results are durable) but the parent transaction may abort.

• One solution: **private workspace**
  – Each sub-transaction is given a private copy of every object it manipulates. On commit, the private copy displaces the parent’s copy (which may also be a private copy of the parent’s parent).

Implementing a private workspace [PLAN A]

• Consider a Unix-like file system:
  – Duplicate the file’s index (i-node)
  – Create new block when it gets modified by the transaction: **shadow block**
  – Index copy points to the shadow block

• On abort: remove shadow blocks and private index
• On commit: update parent’s index with private index

Implementing a private workspace [PLAN B]

• Use a **write-ahead log** (journal, or intentions list)
  – Keep a log in **stable storage** (something that survives reboots)
  – Before making any changes to the object, write a record to the log identifying
    { transaction ID, object ID, old value, new value }
  – If transaction commits:
    • Write a commit record onto the log
  – If transaction aborts
    • Use log to back up to the original state: **rollback**

• Stable storage: data persists even if the system or application crashes.

Distributed Transactions

• Transaction that updates data on two or more systems

• Challenge
  – Handle machine, software, & network failures while preserving transaction integrity

Distributed Transactions

• Each computer runs a **transaction manager**
  – Responsible for subtransactions on that system
  – Transaction managers communicate with other transaction managers
  – Performs **prepare**, **commit**, and **abort** calls for subtransactions
• Every subtransaction must agree to commit changes before the transaction can complete

Commits Among Subtransactions = Consensus

• Remember consensus?
  – Agree on a value proposed by at least one process

• The coordinator proposes to commit a transaction
  – All participants agree ⇒ all participants then commit
  – Not all participants agree ⇒ all participants then abort

• Here, we need **unanimous** agreement to commit
Two-Phase Commit Protocol

**Goal:**
Reliably agree to commit or abort a collection of sub-transactions

- All processes in the transaction will agree to commit or abort
- One transaction manager is elected as a coordinator – the rest are participants
- Assume:
  - write-ahead log in stable storage
  - No system dies forever
  - Systems can always communicate with each other

**Transaction States**

- started
- working
- prepared (done)
- committed
- aborted

When a participant enters the prepared state, it contacts the coordinator to start the commit protocol to commit the entire transaction.

**Phase 1: Voting Phase**
Get commit agreement from every participant

A single ‘no’ response means that we will have to abort the transaction.
Two-Phase Commit Protocol

Phase 2: Commit Phase
Send the results of the vote to every participant

Coordinator
Participant
Participant
Participant
Participant

Send *abort* if any participant voted "no"

Dealing with failure

- 2PC assumes a fail-recover model
  - Any failed system will eventually recover
- A recovered system cannot change its mind
  - If a node agreed to commit and then crashed, it must be willing and able to commit upon recovery
- Each system will use a write-ahead (transaction) log
  - Keep track of where it is in the protocol (and what it agreed to)
  - As well as values to enable commit or abort (rollback)
  - This enables fail-recover

Failure during Phase 1 (voting)
- Coordinator dies
  - Some participants may have responded; others have no clue
  -Coordinator restarts; checks log; sees that voting was in progress
  -Coordinator restarts voting
- Participant dies
  - The participant may have died before or after sending its vote to the coordinator
  - If the coordinator received the vote, wait for other votes and go to phase 2
  - Otherwise: wait for the participant to recover and respond (keep querying it)
Dealing with failure

- Failure during Phase 2 (commit/abort)
  - Coordinator dies
  - Some participants may have been given commit/abort instructions
  - If coordinator dies, checks log; informs all participants of chosen action
  - Some participants may have been given commit/abort instructions
  - Coordinator restarts; checks log; informs all participants of chosen action

- Participant dies
  - The participant may have died before or after getting the commit/abort request
  - Participant recovers; checks log; gets request from coordinator
  - Otherwise, process the commit/abort request and send back the acknowledgement

Adding a recovery coordinator

- Another system can take over for the coordinator
  - Could be a participant that detected a timeout to the coordinator

- Recovery node needs to find the state of the protocol
  - Contact ALL participants to see how they voted

- If we get voting results from all participants
  - We know that Phase 1 has completed
  - If all participants voted to commit ⇒ send commit request
  - Otherwise send abort request

- If ANY participant states that it has not voted
  - We know that Phase 1 has not completed
  - Restart the protocol

- But … if a participant node also crashes, we’re stuck!
  - Have to wait for recovery

What’s wrong with the 2PC protocol?

- Biggest problem: It’s a blocking protocol
  - If the coordinator crashes, participants have no idea whether to commit or abort
  - A non-responding participant will also result in blocking

- When a participant gets a commit/abort message, it does not know if every other participant was informed of the result

Three-Phase Commit Protocol

- Same setup as the two-phase commit protocol:
  - Coordinator & Participants

- Enable the use of a recovery coordinator
  - Propagate the result of the commit/abort vote to each participant before telling them to act on it
  - This will allow us to recover the state if any participant dies

- Add timeouts to each phase that result in an abort

Three-Phase Commit Protocol

- Split the second phase of 2PC into two parts:
  2a. “Precommit” (prepare to commit) phase
  - Send Prepare message to all participants when it received a yes from all participants in phase 1
  - Participants can prepare to commit but cannot do anything that cannot be undone
  - Participants reply with an acknowledgement
  - Purpose: let every participant know the state of the result of the vote so that state can be recovered if anyone dies

  2b. “Commit” phase (same as in 2PC)
  - If coordinator gets Acks for all prepare messages
  - It will send a commit message to all participants
  - Else it will abort ⇒ send an abort message to all participants
Three-Phase Commit Protocol: Phase 1

• Phase 1: Voting phase
  - Coordinator sends canCommit? queries to participants & gets responses
  - Purpose: Find out if everyone agrees to commit
  - If the coordinator gets a timeout from any participant, or any NO replies are received
    • Send an abort to all participants
  - If a participant times out waiting for a request from the coordinator
    • It aborts itself (assume coordinator crashed)
  - Else continue to phase 2

Three-Phase Commit Protocol

• Phase 2: Prepare to commit phase
  - Send a prepare message to all participants.
  - Purpose: let all participants know the decision to commit
  - If coordinator times out: assume participant crashed, send abort to all participants
  - If coordinator cannot count on every participant having received the Prepare message

• Phase 3: Finalize phase
  - Send commit messages to participants and get responses from all
  - If participant times out: contact any other participant and move to that state (commit or abort)
  - If coordinator times out: that’s ok

3PC Recovery

• If the coordinator crashes
  - A recovery node can query the state from any available participant
• Possible states that the participant may report:
  - Already committed
    • That means that every other participant has received a Prepare to Commit
    • Some participants may have committed
    • Send Commit message to all participants (just in case they didn’t get it)
  - Not committed but received a Prepare message
    • That means that all participants agreed to commit; some may have committed
    • Send Prepare to Commit message to all participants (just in case they didn’t get it)
    • Wait for everyone to acknowledge; then commit
  - Not yet received a Prepare message
    • This means no participant has committed; some may have agreed
    • Transaction can be aborted or the commit protocol can be restarted

3PC Weaknesses

• Main weakness of 3PC
  - May have problems when the network gets partitioned
  - Partition A: nodes that received Prepare message
    • Recovery coordinator for A: allows commit
  - Partition B: nodes that did not receive Prepare message
    • Recovery coordinator for B: aborts
  - Either of these actions are legitimate as a whole
    • But when the network merges back, the system is inconsistent
  - Not good when a crashed coordinator recovers
    • It needs to find out that someone took over and stay quiet
    • Otherwise it will mess up the protocol, leading to an inconsistent state

3PC coordinator recovery problem

• Suppose
  - a coordinator sent a Prepare message to all participants
  - all participants acknowledged the message
  - BUT the coordinator died before it got all acknowledgements
• A recovery coordinator queries a participant
  - Continues with the commit: Sends Prepare, gets ACKs, sends Commit
• Around the same time... the original coordinator recovers
  - Realizes it is still missing some replies from the Prepare
  - Times out and decides to send an Abort to all participants
• Some processes may commit while others abort!

3PC works well when servers crash (fail-stop model)
• 3PC is not resilient against fail-recover environments

Paxos Commit
What about Paxos?

- Interface to Paxos
  - Client proposes a value and sends it to the Paxos leader
  - Paxos acceptors will send out the totally-ordered value

- What does Paxos consensus offer?
  - Total ordering of proposals
  - Fault tolerance: proposal is accepted if a majority of acceptors accept it
  - There is always enough data available to recover the state of proposals
  - Is provably resilient in asynchronous networks

- Paxos-based commit is a generalization of 2PC and 3PC

Using Paxos for Commit

The cast:
- One instance of Paxos per participant (N participants)
- 2F+1 acceptors (we can withstand the failure of F+1 acceptors)
- One elected Leader (Proposer) = Coordinator

```
Participant begin commit
Leader prepare
Participant i = 1..N
Leader Ready to start
Tell everyone
Each instance of Paxos prepare to commit or abort
Participant i = 1..N value = (prepared | aborted)
{ Acceptors }
{ Acceptors }
Leader
```

A leader will get at least F+1 messages for each instance
Commit if every participant's instance of Paxos chooses Prepared
Paxos commit = 2PC if one acceptor

Virtual Synchrony vs. Transactions vs. Paxos

- Virtual Synchrony
  - Fastest & most scalable
  - Focuses on group membership management & reliable multicasts

- Two-Phase & Three-Phase Commit
  - Most expensive – requires extensive use of stable storage
  - Designed for transactional activities
  - Not suitable for high speed messaging

- Paxos
  - Performance limited by its two-phase protocol
  - Great for consensus & fault tolerance
  - Adds ordering of proposals over 2PC

Scaling & Consistency

Scaling Transactions

- Transactions require locking part of the database so that everyone sees consistent data at all times
  - Good on a small scale.
    - Low transaction volumes: getting multiple databases consistent is easy
    - Difficult to do efficiently on a huge scale

- Add replication: use cached data

- Problems / side-effects:
  - Users run the risk of seeing stale data
  - The "I" of ACID may be violated
    - E.g., two users might try to buy the last book on Amazon

Scaling also affects availability

- One database with 99.9% availability
  - 8 hours, 45 minutes, 35 seconds downtime per year

- If a transaction uses 2PC protocol and requires two databases, each with a 99.9% availability:
  - Total availability = (0.999*0.999) = 99.8%
  - 17 hours, 31 minutes, 12 seconds downtime per year

- If a transaction requires 5 databases:
  - Total availability = 99.5%
  - 1 day, 19 hours, 48 minutes, 0 seconds downtime per year
Delays hurt

- The delays to achieve consistency can hurt business
- Amazon:
  - 0.1 second increase in response time will cost them 1% of sales
- Google:
  - 0.5 second increase in latency causes traffic to drop by 20%
- Latency is due to lots of factors
  - OS & software architecture, computing hardware, tight vs loose coupling, network links, geographic distribution, …
  - We're only looking at the problems caused by the tight software coupling due to achieving the ACID model

Eric Brewer’s CAP Theorem

- Three core requirements in a shared data system:
  - Consistency
  - Availability
  - Partition Tolerance: tolerance to network partitioning
    - No set of failures less than total failure is allowed to cause the system to respond incorrectly
- CAP Theorem: you can only have at most two of these!

Example

Life is good

Network partition occurs

Do we want to give up consistency or availability?

Giving up one of (C, A, P)

- Give up partitions
  - Put everything on one machine or a cluster in one rack: high availability clustering
  - Use two-phase commit or three phase commit
  - Scaling suffers
- Give up availability
  - Lock data: have services wait until data is consistent
  - Classic ACID distributed databases (also 2PC)
  - Response time suffers
- Give up consistency
  - Eventually consistent data
  - Use expirations/leases, queued messages for updates
  - Examples: DNS, web caching, Amazon Dynamo, Hadoop HBase, CouchDB
  - Often not as bad as it sounds!

BASE: an alternative to ACID

- Traditional database systems want ACID
  - But scalability is a problem (lots of transactions in a distributed environment)
- Give up Consistent and Isolated in exchange for high availability and high performance
  - Get rid of locking in exchange for multiple versions
  - Incremental replication
- BASE:
  - Basically Available
  - Soft-state
  - Eventual Consistency

ACID vs. BASE

<table>
<thead>
<tr>
<th>ACID</th>
<th>BASE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strong consistency</td>
<td>Weak (eventual) consistency: stale data at times</td>
</tr>
<tr>
<td>Isolation</td>
<td>High availability</td>
</tr>
<tr>
<td>Focus on commit</td>
<td>Best effort approach</td>
</tr>
<tr>
<td>Nested transactions</td>
<td>Optimistic access to data</td>
</tr>
<tr>
<td>Availability is problematic</td>
<td>Simpler model (but harder for app developer)</td>
</tr>
<tr>
<td>Pessimistic access to data (locking)</td>
<td>Faster</td>
</tr>
</tbody>
</table>

From Eric Brewer’s PODC Keynote, July 2000
A place for BASE

• ACID is neither dead nor useless
  – Many environments require it

• BASE has become common for large-scale web apps
  where replication & fault tolerance is crucial
  – eBay, Twitter, Amazon
  – Eventually consistent model not always surprising to users
    • Cellphone usage data
    • Banking transactions (e.g., fund transfer activity showing up on statement)
    • Posting of frequent flyer miles