**Atomic Transactions**

**Transaction**

- An operation composed of a sequence 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*

**Another Example**

Book a flight from Allentown, PA to Inyokern, CA
No non-stop flights are available:

**Transaction begin**
1. Reserve a seat for Allentown to O'Hare (ABE→ORD)
2. Reserve a seat for O'Hare to Los Angeles (ORD→LAX)
3. Reserve a seat for Los Angeles to Inyokern (LAX→IYK)
**Transaction end**

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

**Basic Operations**

**Transaction primitives:**

- **Begin transaction**: mark the start of a transaction
  
  Read/write/compute data – modify files, objects, program state
  
  • But data will have to be restored if the transaction is aborted

- **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

**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 & all invariants must be preserved.
  
  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 or see intermediate results if transactions run at the same time, the result must be the same as if they executed in some serial order.

- **Durable**
  
  - Once a transaction commits, the results are made permanent.
Distributed Transaction

Transaction that updates data on two or more systems
Implemented as a set of sub-transactions

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

Distributed Transactions

Each computer runs a transaction manager
– Responsible for sub-transactions on that system
– Performs prepare, commit, and abort calls for sub-transactions

Every sub-transaction must agree to commit changes before the overall transaction can complete

Commits Among Sub-transactions = Consensus

• Remember consensus?
  – Agree on a value proposed by at least one process
• BUT – here we need unanimous agreement to commit
• The coordinator proposes to commit a transaction
  – All participants agree ⇒ all participants then commit
  – Not all participants agree ⇒ all participants then abort

Core Properties

The algorithm should have these properties:

1. Safety (the algorithm must work correctly)
   • If one sub-transaction commits, no other sub-transaction will abort
   • If one sub-transaction needs to abort, no sub-transactions will commit
2. Liveness (the algorithm must make progress & reach its goal)
   • If no sub-transactions fail, the transaction will commit
   • If any sub-transactions fail, the algorithm will reach a conclusion to abort

Two-Phase Commit Protocol

Participant
Participant
Participant
Participant
Coordinator

Coordinator
Participant
Participant
Participant
Participant
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
- Consensus: all processes agree on whether or not to commit
- 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.

Transaction States

Two-Phase Commit Protocol

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

Send abort if any participant voted “no”
## Two-Phase Commit Protocol: Phase 1

### 1. Voting Phase
- **Coordinator**
  - Work on transaction
  - Send \texttt{CanCommit?} message
  - Wait for all participants to respond

- **Participant**
  - Write prepare to commit to log
  - Wait for message from coordinator
  - Receive the \texttt{CanCommit?} message
  - When ready, write \texttt{agree to commit or abort} to the log
  - Send \texttt{agree to commit or abort} to the coordinator

Get distributed agreement: the coordinator asked each participant if it will commit or abort and received replies from each coordinator.

### 2. Commit Phase
- **Coordinator**
  - Write \texttt{commit or abort} to log
  - Send \texttt{commit or abort}

- **Participant**
  - Wait for \texttt{commit/abort message}
  - Receive \texttt{commit or abort}
  - If a commit was received, write \texttt{commit} to the log, release all locks, update databases.
  - If an abort was received, undo all changes
  - Send done message

Tell all participants to \texttt{commit or abort}
Get everyone’s response that they’re done.

## Dealing with failure

### Failure during Phase 1 (voting)
- **Coordinator dies**
  - Some participants may have responded; others have no clue
  - \texttt{Coordinator restarts voting; checks log; sees that voting was in progress}

- **Participant dies**
  - The participant may have died before or after sending its vote to the coordinator
  - \texttt{If the coordinator received the vote, it waits for other votes and go to phase 2}
  - \texttt{Otherwise: wait for the participant to recover and respond (keep querying it)}

## Consensus Properties
- **Validity property**
  - Aborts in every case except when every process agrees to commit
  - The final value (commit or not) has been voted on by at least one process

- **Uniform Agreement property**
  - Every process decides on the value proposed by the coordinator if and only if they are instructed to do so by the coordinator in phase 2

- **Integrity property**
  - Every process proposes only a single value (commit or abort) and does not change its mind.

- **Termination property**
  - Every process is guaranteed to make progress and eventually return a vote to the coordinator.

### Failure during Phase 2 (commit/abort)
- **Coordinator dies**
  - Some participants may have been given commit/abort instructions
  - \texttt{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
  - \texttt{If the coordinator received the request, it looks for other requests and goes to phase 2}
  - \texttt{Otherwise: wait for the participant to recover and respond (keep querying it)}

- **Participant recovers**
  - Checks log; gets request from coordinator
  - \texttt{If committed/aborted, acknowledge the request}
  - \texttt{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 any participant node also crashes, we’re stuck!
  – Must wait for recovery

What’s wrong with the 2PC protocol?

Biggest problem: it’s a blocking protocol with failure modes that require all systems to recover eventually
  – If the coordinator crashes, participants have no idea whether to commit or abort
  – A recovery coordinator helps
    – If a coordinator AND a participant crashes
      • The system has no way of knowing the result of the transaction
      • It might have committed for the crashed participant – hence all others must block

The protocol cannot pessimistically abort because some participants may have already committed

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
• Add timeouts to each phase that result in an abort
• 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

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

We can abort if the participant and/or coordinator dies

28 March 2020
Three-Phase Commit Protocol

Phase 2: Prepare to commit phase
- Send a \texttt{prepare} message to all participants.
- Get \texttt{OK} messages from all participants
  - We need to hear from all before proceeding so we can be sure the state of the protocol can be properly recovered if the coordinator dies
  - Purpose: let all participants know the decision to commit
  - \texttt{[!] If a participant times out: assume it crashed; send \texttt{abort} to all participants}

Phase 3: Finalize phase
- Send \texttt{commit} messages to participants and get responses from all
  - \texttt{[!] If participant times out: contact any other participant and move to that state (commit or abort)}
  - \texttt{[!] If coordinator times out: that’s ok - we know what to do}

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 \texttt{Prepare to Commit}
- Some participants may have committed
  - Send \texttt{Commit} message to all participants (just in case they didn’t get it)

Not committed but received a \texttt{Prepare} message
- That means that all participants agreed to commit; some may have committed
  - Send \texttt{Prepare to Commit} message to all participants (just in case they didn’t get it)
  - Wait for everyone to acknowledge, then \texttt{commit}

Not yet received a \texttt{Prepare} message
- This means no participant has committed; some may have agreed
  - Transaction can be \texttt{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 \texttt{Prepare} message
    - \texttt{Recovery coordinator for A: allows commit}
  - Partition B: nodes that did not receive \texttt{Prepare} message
    - \texttt{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 \texttt{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 \texttt{Prepare}, gets ACKs, sends \texttt{Commit}
  - Around the same time... the original coordinator recovers
    - Realizes it is still missing some replies from the \texttt{Prepare}
    - Gets timeouts from some and decides to send an \texttt{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
  - 3PC is not resilient against network partitions

What about Raft?

- Consensus-based protocols (Raft, Paxos) are designed to be resilient against network partitions
  - What does Raft consensus offer?
    - Total ordering of proposals (replicated log)
    - Fault tolerant: proposal is accepted if a \texttt{majority} of nodes accept it
    - There is always enough data available to recover the state of proposals
      - Is provably resilient in asynchronous networks
  - Consensus-based commit is a generalization of 2PC
    - Use multiple coordinators to avoid blocking if the coordinator fails
      - Run a consensus algorithm on the commit/abort decision of EACH participant

Consensus-based Commit
What do we want to do?

- Each participant tries to get its chosen value – can_commit or must_abort – accepted by the majority of nodes
- Run an instance of the consensus protocol for each participant
  - Each instance is fault tolerant against network partitions
- Transaction Leader
  - Chosen via election algorithm
  - Coordinates the commit algorithm
  - Not a single point of failure – we can elect a new one; Raft nodes store state

How do we do it?

- Some participant decides to begin to commit
  - Sends a message to the Transaction Leader
- Transaction Leader:
  - Sends a prepare message to each participant
- Each participant now sends a can_commit or must_abort message to its instance of the consensus protocol (Raft)
  - All participants share the elected Transaction Leader
  - “Can_commit” or “Must_abort” is sent to majority of followers
  - Result is sent to the leader
- Transaction Leader tracks all instances of the commit protocol
  - Commit if every participant’s instance of the consensus protocol chooses “can_commit”
  - Tell each participant to commit or abort

Consensus-based fault-tolerant coordinator

The cast:
- One instance of Raft per participant (N participants)
- Set of 2F+1 nodes and a leader play the role of the coordinator
- We can withstand the failure of F nodes
- Leader = node elected to be in charge & run the protocol

```
<table>
<thead>
<tr>
<th>Participant</th>
<th>Leader</th>
</tr>
</thead>
<tbody>
<tr>
<td>Begin commit</td>
<td>prepare</td>
</tr>
<tr>
<td>Tell everyone</td>
<td>{ Participant }, { Followers }</td>
</tr>
<tr>
<td>Each instance of Raft: proposes to commit or abort</td>
<td>Each instance of Raft tells the result to the leader</td>
</tr>
<tr>
<td>Participant, value = {can_commit</td>
<td>must_abort}, { Followers }</td>
</tr>
</tbody>
</table>
```

- A leader will get at least F+1 messages for each instance
- Commit if every participant’s instance of Raft chooses can_commit
- Raft commit acts like 2PC if only one node

Virtual Synchrony vs. Transactions vs. Raft

- Virtual Synchrony
  - Fast & scalable
  - State machine replication: multicast messages to the entire group
  - Focuses on group membership management & reliable multicasts
- Two-Phase & Three-Phase Commit
  - Most expensive – requires extensive use of stable storage
  - 2PC efficient in terms of # of messages
  - Designed for transactional activities
  - Not suitable for high speed messaging
- Raft (or Paxos) Consensus
  - General purpose fault-tolerant consensus algorithm
  - Performance limited by its two-phase protocol
  - Useful for fault-tolerant log replication & elections
  - Consensus-based commit overcomes dead coordinator and network partition problems of 2PC and 3PC

Reliance on multiple systems 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
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 — processes can read any replica
  - But all replicas must be locked during updates to ensure consistency
- Risks of not locking:
  - 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

Delays hurt

- The delays to achieve consistency can hurt business
- Amazon:
  - 0.1 second increase in response time costs 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:

1. Atomic, Isolated Consistency
   - Operations must appear totally ordered and each is isolated
2. Availability
   - Every request received by a non-failed node must result in a response
3. Partition Tolerance: tolerance to network partitioning
   - Messages between nodes may be lost
   - No set of failures less than total failure is allowed to cause the system to respond incorrectly*

CAP Theorem: when there is a network partition, you cannot guarantee both availability & consistency

Commonly (not totally accurately) stated as you can have at most two of the three: C, A, or P

Example: Partition

Life is good

Network partition occurs

Do we want to give up consistency or availability?

Partitions will happen

- With distributed systems, we expect partitions to occur
  - Maybe not full failure but high latency can act like a failure
  - This is a property of the distributed environment
  - The CAP theorem says we have a tradeoff between availability & consistency
- We’d like availability and consistency
  - We get availability via replication
  - We get consistency with atomic updates
  1. Lock all copies before an update
  2. Propagate updates
  3. Unlock
- We can choose high availability
  - Allow reads before all nodes are updated (avoid locking)
- or choose consistency
  - Enforce proper locking of nodes for updates
Eventual Consistency

- 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
- Consistency model
  - If no updates are made to a data item, eventually all accesses to that item will return the last updated value

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 can suffer</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>

A place for BASE

- ACID is neither dead nor useless
  - Many environments require it
  - It’s safer – the framework handles ACID for you
- BASE has become common for large-scale web apps where replication & fault tolerance is crucial
  - eBay, Twitter, Amazon
  - Eventual 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

But … the app developer has to worry about update conflicts and reading stale data and programmers often write buggy code

The end