Today

- The *atomic commit* problem
- Specification
- Two-phase commit
- Three-phase commit
- Election
- Last process to fail
Atomic Commit

A form of group coordination that arises in the context of distributed databases.

- A set of databases separately serialize updates.
- At the end of the transaction each database locally determines whether the transaction should be made permanent or discarded.
Atomic Commit

Input value $vote_i \in \{ \text{yes, no} \}$

Output value $decision_i \in \{ \text{commit, abort} \}$

("decide commit" and "decide abort")

AC1: No two processes decide differently.

AC2: A process decides commit only if all votes are yes.

AC3: If all votes are yes and there are no failures, then the decision is commit.

AC4: If all failures are repaired and there are no more failures, then all processes eventually decide.
Atomic Commit

AC1: No two processes decide differently.

This can't be implemented assuming arbitrary failures.

Usual failure model:

- Recoverable crash failures
  - Some memory can survive crashes (stable storage)
- Communication failures detectable by timeouts.
Atomic Commit

AC4: If all failures are repaired and there are no more failures, then all processes eventually decide.

- If strengthen to simply *All processes eventually decide*, then requires all crashed processes to recover.
- Change to *All processes that never crash eventually decide* results in *nonblocking commit*. 
Atomic Commit

AC2: A process decides commit only if all votes are \textit{yes}.

AC3: If all votes are \textit{yes} and there are no failures, then any decision is \textit{commit}.

If we replace these two with \textit{a process decides commit iff all votes are yes} then a process that crashes in the initial state must recover before any process decides.
X Phase Commit protocols

- These protocols use a structure in which all participating processes move as a group from one state to another.
  - Each movement is a "phase".
  - Two phase commit most common
  - Three phase commit for a specific property
  - … there is a One Phase Commit
Two phase commit

- First designed by Nico Garzado, IBM in 1971 for the Italian Social Security Department.
- First circulated version by Butler Lampson and Howard Sturgis, Xerox PARC 1976 in the context of a transactional file system.

Two phase commit

**Coordinator**
Send VOTE_REQ to all

**Participant p_i**

![](send vote_i to coordinator)

if \( vote_i = no \)

![](decide abort)

![](halt)

if (all votes yes)

![](decide commit)

![](send COMMIT to all)

else

![](decide abort)

![](send ABORT to all who voted yes)

![](halt)

if receive ABORT, decide abort

else decide commit

![](halt)
Two phase commit: coordinator

\[
\begin{align*}
\text{VOTE}_1, \ldots, \text{VOTE}_n & \quad \text{deadline transition} \\
\text{NO}_1 \lor \ldots \lor \text{NO}_n & \quad \text{ABORT}_1, \ldots, \text{ABORT}_n \\
& \quad \text{COMMIT}_1, \ldots, \text{COMMIT}_n
\end{align*}
\]
Two phase commit: participant

run termination protocol
Two phase commit: termination protocol

Let $c$ be the coordinator and $P$ be the set of participants. Upon timeout waiting for message from $c$, a participant $p$ in $W$ sends DECISION_REQ to all $q \in P$.

$q$ responds with

- $decision_q$ if it has decided;
- ABORT if it is in $i$ and $vote_q = no$;
- UNDECIDED if it is $W$

If any $q$ responds ABORT or COMMIT that $p$ can decide.
If all $P$ respond UNDECIDED, then what?
Two phase commit: crash recovery protocol

Stable storage is persistent memory that supports writes that are atomic with respect to failures.

Log actions:

- $c$ sends VOTE_REQ
- $p$ votes YES
- $p$ votes NO
- before $c$ decides commit
- when $c$ decides abort
- $p$ receives decision

- concurrently write start $P$
- first write $\text{yes } P c$
- concurrently write $\text{abort}$
- first write $\text{commit}$ \textit{commit point}
- concurrently write $\text{abort}$
- concurrently write decision
Two phase commit: crash recovery protocol

Upon recovery a process $r$ starts reading the values logged to stable storage.

- If there is a start $P$ then $r$ was the coordinator:
  - If there is a subsequent abort or commit then decision was made; otherwise decide abort.

- Otherwise was a participant:
  - If there is abort or commit then the decision was made;
  - If there is no yes then decide abort.
  - Otherwise use $(P, c)$ from yes record to run termination protocol.

... when can these records be garbage collected?
Nonblocking Atomic Commit

What if the coordinator crashes before sending the outcome to a process that doesn't crash?

The *up* processes are uncertain as to the outcome.

- They cannot commit because a process may have voted *no*.
- All may have voted *yes* and a *down* process may have committed, so they cannot abort.

Three phase commit

If the atomic commit protocol ensures:

If an operational process is in the uncertain state, then no process can be in the committed state.

Create a new *precommitted state* that a process enters only if the transaction can be committed.
Three phase commit: coordinator

\[ i \]

\[ \text{timeout transition} \]

\[ W \]

\[ \text{VOTE}_1 \ldots \text{VOTE}_n \]

\[ W \]

\[ \text{NO}_1 \lor \ldots \lor \text{NO}_n \]

\[ \text{ABORT}_1, \ldots, \text{ABORT}_n \]

\[ \text{YES}_1 \land \ldots \land \text{YES}_n \]

\[ \text{PREPARE}_1, \ldots, \text{PREPARE}_n \]

\[ A \]

\[ P \]

\[ \text{C} \]

\[ \text{ACK}_1 \land \ldots \land \text{ACK}_n \]

\[ \text{COMMIT}_1, \ldots, \text{COMMIT}_n \]

\[ \text{ignore} \]
Three phase commit: participant

![Diagram showing the three-phase commit protocol]

- **A** (Participant)
  - **W** (Writer)
  - **P** (Partner)
  - **C** (Coordinator)

- **VOTE_REQ_i**
  - NO_i
  - YES_i

- **PREPARE_i**
  - ACK_i

- **ABORT_i**

- **COMMIT_i**

Run termination protocol at **P** and **W**
Termination protocol

Elect a new coordinator, which sends \texttt{STATE\_REQ} to all participant.

1. If there is a process in $A$, then rest are in $i$, $A$, or $W$
   send \texttt{ABORT} to all

2. If there is a process in $C$, then the rest are in $P$ or $C$
   send \texttt{COMMIT} to all

3. If all in $W$, then none in $C$
   send \texttt{ABORT} to all

4. If all in $P$, then none in $A$
   send \texttt{COMMIT} to all

5. If some in $W$ and some in $P$, then none in $A$
   send \texttt{PREPARE} to all
   send \texttt{COMMIT} to all
Three phase commit: coordinator

\[
\begin{align*}
W & \quad \text{timeout transition} \\
\text{VOTE}_1, \ldots, \text{VOTE}_n & \\
\text{NO}_1 \lor \ldots \lor \text{NO}_n & \\
\text{ABORT}_1, \ldots, \text{ABORT}_n & \\
\text{YES}_1 \land \ldots \land \text{YES}_n & \\
\text{PREPARE}_1, \ldots, \text{PREPARE}_n & \\
\text{ignore} & \\
\text{P} & \\
\text{ACC}_1 \land \ldots \land \text{ACC}_n & \\
\text{COMMIT}_1, \ldots, \text{COMMIT}_n & \\
\text{A} & \\
\text{timeout transition} & \\
\text{recovery transition} &
\end{align*}
\]
Three phase commit: participant

- VOTE_REQ_i
  - NO_i
  - YES_i
- PREPARE_i
  - ACK_i
- COMMIT_i
- ABORT_i
- run termination protocol

Diagram:

A ➔ i ➔ W ➔ P ➔ C

- ask
- run termination protocol
Election

Electing a new coordinator:

- Each process $p_i$ initially agrees on a sequence $UP_i$ of the processes, where the first process in the sequence is the original coordinator.
  
  This could be sent with the VOTE_REQ message.

- If a process $p_i$ detects the failure of a process $p_j$, then $p_i$ removes $p_j$ from $UP_i$. 
Election (continued)

- If a new process becomes the head of $UP_i$, then
  - If $p_i$ is the head of $UP_i$, then $p_i$ takes over as the new coordinator.
  - If $p_j, j \neq i$, is the head of $UP_i$, then $p_i$ sends $UP_i$ to $p_j$
    - When $p_j$ receives $UP_i$
      - If $p_j$ has no $UP_j$, then $UP_j = UP_i$
      - If $p_j$ has $UP_j$, then $UP_j = UP_i \cap UP_j$
    - In both cases, $p_j$ will either eventually take over as the new coordinator or crash.
Total failure

If all the processes crash, then the processes that were the last to fail need to recover and run the termination protocol.

A last process to fail is one whose failure is not detected by another process.

Total failure (continued)

\( p_i \text{ fails-before } p_j \equiv p_j \text{ detects } p_i \text{'s failure} \)

\( LPF \equiv \{p_i : \forall p_j : \neg (p_i \text{ fails-before } p_j)\} \)

\( UP_i \) is \( \{p_j : \neg (p_j \text{ fails-before } p_i)\} \)

so, \( p_i \in LPF \equiv \forall p_j : \neg (p_i \text{ fails-before } p_j) \)

or \( p_i \in LPF \equiv \forall p_j : p_i \in UP_j \)

or \( LPF = \bigcap_{p_i} UP_i \)
Total failure (continued)
Total failure (continued)
Total failure (continued)
Total failure (continued)

\[ LPF = \{p_1, p_2\} \]
Total failure (continued)

When $p_i$ detects $p_j$'s failure it:

- Removes $p_j$ from $UP_i$
- Synchronously writes $UP_i$ to stable storage

When $p_i$ recovers:

- Let $R_i$ be the processes $p_i$ knows have recovered.
- When $(R_i = \bigcap_{p_j \in R_i} UP_j) \land (p_i \in \bigcap_{p_j \in R_i} UP_j)$
  then $p_i$ knows that it is in $LPF$ and all of $LPF$ have recovered.
Total failure (continued)

$p_1, p_2, p_3$

$p_1, p_2, p_3$

$p_1, p_2, p_3, p_4$

$p_1, p_2, p_4$
Total failure (continued)
Total failure (continued)
Total failure (continued)

$p_1, p_2, p_3$

$p_1, p_2, p_3, p_4$

$p_1, p_2, p_3$

$p_1, p_2, p_4$
Recap

- Atomic commit:
  - Solves a key problem in distributed coordination.
  - Resembles, but is not the same as, consensus.

... next up: consensus.