Atomic Commit

The objective

Preserve data consistency for distributed transactions in the presence of failures

Model

- For each distributed transaction $T$:
  - one coordinator
  - a set of participants
- Coordinator knows participants; participants don’t necessarily know each other
- Each process has access to a Distributed Transaction Log (DT Log) on stable storage

The setup

- Each process $p_i$ has an input value $vote_i$:
  $vote_i \in \{\text{Yes, No}\}$
- Each process $p_i$ has output value $decision_i$:
  $decision_i \in \{\text{Commit, Abort}\}$
AC Specification

AC-1: All processes that reach a decision reach the same one.
AC-2: A process cannot reverse its decision after it has reached one.
AC-3: The Commit decision can only be reached if all processes vote Yes.
AC-4: If there are no failures and all processes vote Yes, then the decision will be Commit.
AC-5: If all failures are repaired and there are no more failures, then all processes will eventually decide.

Liveness & Uncertainty

A process is uncertain if it has voted Yes but does not have sufficient information to commit.
While uncertain, a process cannot decide unilaterally.
Uncertainty + communication failures = blocking!

Liveness & Independent Recovery

Suppose process \( p \) fails while running AC.
If, during recovery, \( p \) can reach a decision without communicating with other processes, we say that \( p \) can independently recover.
Total failure (i.e. all processes fail) - independent recovery = blocking.
Notes on 2PC

Satisfies AC-1 to AC-4

But not AC-5 (at least “as is”)

i. A process may be waiting for a message that may never arrive
   • Use Timeout Actions

ii. No guarantee that a recovered process will reach a decision consistent with that of other processes
   • Processes save protocol state in DT-Log

A few character-building facts

Proposition 1
If communication failures or total failures are possible, then every AC protocol may cause processes to become blocked

Proposition 2
No AC protocol can guarantee independent recovery of failed processes

2-Phase Commit

I. Coordinator c sends VOTE-REQ to all participants.

II. When participant pi receives a VOTE-REQ, it responds by sending a vote to the coordinator.
   if votes = NO, then decide := ABORT and pi halts.

III. c collects votes from all.
   if all votes are yes, then decide := COMMIT; sends COMMIT to all
   else decide := ABORT; sends ABORT to all who voted YES
   c halts.

IV. if participant pi receives COMMIT then decide := COMMIT
   else decide := ABORT
   pi halts.

Timeout actions

Processes are waiting on steps 2, 3, and 4

- Step 2: pi is waiting for VOTE-REQ from coordinator
- Step 3: Coordinator is waiting for vote from participants
- Step 4: pi (who voted YES) is waiting for COMMIT or ABORT
Timeout actions

Processes are waiting on steps 2, 3, and 4

Step 2. \( p_i \) is waiting for VOTE-REQ from coordinator.
Since it is has not cast its vote yet, \( p_i \) can decide ABORT and halt.

Step 3. Coordinator is waiting for vote from participants.
Coordinator can decide ABORT, send ABORT to all participants which voted YES, and halt.

Step 4. \( p_i \) (who voted YES) is waiting for COMMIT or ABORT

Termination protocols

I. Wait for coordinator to recover
   - It always works, since the coordinator is never uncertain
   - may block recovering process unnecessarily

II. Ask other participants
Cooperative Termination

1. When coordinator sends VOTE-REQ, it writes START-2PC to its DT Log
2. When participant is ready to vote, it sends a DECISION-REQ message to every other participant
3. If a participant has decided, then it sends its decision value to its coordinator (writes also list of participants)
4. If the participant has not yet voted, then it decides ABORT, and sends ABORT to the coordinator
5. What if the coordinator is uncertain?

Logging actions

1. When coordinator sends VOTE-REQ, it writes START-2PC to its DT Log
2. When participant $p_i$ is ready to vote YES,
   i. $p_i$ writes YES to DT Log
   ii. $p_i$ sends YES to coordinator $c$ (writes also list of participants)
3. When participant $p_i$ is ready to vote NO, it writes ABORT to DT Log
4. When coordinator $c$ is ready to decide COMMIT, it writes COMMIT to DT Log before sending COMMIT to participants
5. When coordinator $c$ is ready to decide ABORT, it writes ABORT to DT Log
6. After participant $p_i$ receives decision value, it writes it to DT Log

$p$ recovers

1. When coordinator sends VOTE-REQ, it writes START-2PC to its DT Log
2. When participant $p$ is ready to vote
   Yes, writes YES to DT Log before sending YES to coordinator (writes also list of participants)
   When participant $p$ is ready to vote NO, it writes ABORT to DT Log
3. When coordinator is ready to decide COMMIT, it writes COMMIT to DT Log before sending COMMIT to participants
   When coordinator is ready to decide ABORT, it writes ABORT to DT Log
4. After participant receives decision value, it writes it to DT Log

$p$ recovers

1. When coordinator sends VOTE-REQ, it writes START-2PC to its DT Log
2. When participant is ready to vote
   Yes, writes YES to DT Log before sending YES to coordinator (writes also list of participants)
   When participant is ready to vote NO, it writes ABORT to DT Log
3. When coordinator is ready to decide COMMIT, it writes COMMIT to DT Log before sending COMMIT to participants
   When coordinator is ready to decide ABORT, it writes ABORT to DT Log
4. After participant receives decision value, it writes it to DT Log

$\text{if DT Log contains START-2PC, then } p \equiv c$:

$\text{if DT Log contains a decision value, then decide accordingly}$

$\text{else decide ABORT}$
3-Phase Commit

Two approaches:

1. Focus only on site failures
   - Non-blocking, unless all sites fail
   - Communication failures can still cause blocking, but less often than in 2PC

2. Tolerate both site and communication failures
   - Partial failures can still cause blocking, but less often than in 2PC

2PC and blocking

- Blocking occurs whenever the progress of a process depends on the repairing of failures
- No AC protocol is non-blocking in the presence of communication or total failures
- But 2PC can block even with non-total failures and no communication failures among operating processes!

Blocking and uncertainty

Why does uncertainty lead to blocking?
Blocking and uncertainty

Why does uncertainty lead to blocking?
- An uncertain process does not know whether it can safely decide COMMIT or ABORT because some of the processes it cannot reach could have decided either

Non-blocking Property
If any operational process is uncertain, then no process has decided COMMIT

2PC Revisited

In U, both A and C are reachable!
2PC Revisited

In state PC: A process knows that it will commit unless it fails.

3PC: The Protocol

I. $c$ sends VOTE-REQ to all participants.
II. When $p_i$ receives a VOTE-REQ, it responds by sending a vote to $c$ if vote$_i$ = No, then decide$_i$ := ABORT and $p_i$ halts.
III. $c$ collects votes from all.
   if all votes are Yes, then $c$ sends PRECOMMIT to all else decide$_c$ := ABORT; sends ABORT to all who voted Yes $c$ halts
IV. if $p_i$ receives PRECOMMIT then it sends ACK to $c$
V. $c$ collects ACKs from all.
   When all ACKs have been received, decide$_c$ := COMMIT; $c$ sends COMMIT to all.
VI. When $p_i$ receives COMMIT, $p_i$ sets decide$_i$ := COMMIT and halts.
Wait a minute!

1. πi sends VOTE-REQ to all participants
2. When participant πj receives a VOTE-REQ,
   it responds by sending a vote to πi
   if vote = Yes, then decide = ABORT and πi halts
3. πi collects vote from all
   if all votes are Yes, then πi sends PRECOMMIT to all
   else decide = COMMIT and πi halts
4. If πj receives PRECOMMIT then it sends ACK to πi
5. πi collects ACKs from all
   when all ACKs have been received, decide = COMMIT
   πi sends COMMIT to all
6. When πj receives COMMIT, πj sets decide = COMMIT
   πj halts

Timeout Actions

Processes are waiting on steps 2, 3, 4, 5, and 6

- Step 2: πi is waiting for VOTE-REQ from coordinator
- Step 3: Coordinator is waiting for vote from participants
- Step 4: πi waits for PRECOMMIT
- Step 5: Coordinator waits for ACKs
- Step 6: πi waits for COMMIT

Wait a minute!

1. πi sends VOTE-REQ to all participants
2. When participant πj receives a VOTE-REQ,
   it responds by sending a vote to πi
   if vote = No, then decide = ABORT and πj halts
3. πi collects vote from all
   if all votes are Yes, then πi sends PRECOMMIT to all
   else decide = COMMIT and πi halts
4. If πj receives PRECOMMIT then it sends ACK to πi
5. πi collects ACKs from all
   when all ACKs have been received, decide = COMMIT
   πi sends COMMIT to all
6. When πj receives COMMIT, πj sets decide = COMMIT
   πj halts

Timeout Actions

Processes are waiting on steps 2, 3, 4, 5, and 6

- Step 2: πi is waiting for VOTE-REQ from coordinator
- Step 3: Coordinator is waiting for vote from participants
- Step 4: πi waits for PRECOMMIT
- Step 5: Coordinator waits for ACKs
- Step 6: πi waits for COMMIT
Timeout Actions

Processes are waiting on steps 2, 3, 4, 5, and 6

Step 2 $p_i$ is waiting for VOTE-REQ from coordinator
- Exactly as in 2PC

Step 4 $p_i$ waits for PRECOMMIT
- Run some Termination protocol

Step 6 $p_i$ waits for COMMIT

Step 3 Coordinator is waiting for vote from participants
- Exactly as in 2PC

Step 5 Coordinator waits for ACKs
- Coordinator sends COMMIT

Participant knows what is going to receive...
**Process states**

At any time while running 3 PC, each participant can be in exactly one of these 4 states:

- **Aborted**: Not voted, voted NO, received ABORT
- **Uncertain**: Voted YES, not received PRECOMMIT
- **Committable**: Received PRECOMMIT, not COMMIT
- **Committed**: Received COMMIT

**Not all states are compatible**

<table>
<thead>
<tr>
<th></th>
<th>Aborted</th>
<th>Uncertain</th>
<th>Committable</th>
<th>Committed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aborted</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Uncertain</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>Committable</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Committed</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
</tr>
</tbody>
</table>

**Termination protocol**

- When a process times out, it starts an election protocol to elect a new coordinator.
- The new coordinator sends STATE-REQ to all processes that participated in the election.
- The new coordinator collects the states and follows a termination rule.

**Termination protocol and failures**

Processes can fail while executing the termination protocol...

- □ if a process times out on a process, it can just ignore it.
- □ if a coordinator fails, a new coordinator is elected and the protocol is restarted (election protocol to follow).
- □ total failures will need special care...
Recovering $p$

- If $p$ fails before sending YES, decide ABORT
- If $p$ fails after having decided, follow decision
- If $p$ fails after voting YES but before receiving decision value
  - ask other processes for help
  - 3PC is non-blocking: $p$ will receive a response with the decision
- If $p$ has received PRECOMMIT
  - still needs to ask other processes (cannot just COMMIT)

No need to log PRECOMMIT!

The election protocol

- Processes agree on linear ordering (e.g. by pid)
- Each $p$ maintains set $UP_p$ of all processes that $p$ believes to be operational
- When $p$ detects failure of $c$, it removes $c$ from $UP_p$ and chooses smallest $q$ in $UP_p$ to be new coordinator
- If $q = p$, then $p$ is new coordinator
- Otherwise, $p$ sends UR-ELECTED to $q$

A few observations

- What if $p'$, which has not detected the failure of $c$, receives a STATE-REQ from $q$?
A few observations

What if \( p' \), which has not detected the failure of \( c \), receives a STATE-REQ from \( q \)?

- it concludes that \( c' \) must be faulty
- it removes from \( UP_v \) every \( q' < q \)

What if \( p' \) receives a STATE-REQ from \( c \) after it has changed the coordinator to \( q \)?

A few observations

What if \( p' \), which has not detected the failure of \( c \), receives a STATE-REQ from \( q \)?

- it concludes that \( c'_c \) must be faulty
- it removes from \( UP_v \) every \( q' < q \)

What if \( p' \) receives a STATE-REQ from \( c \) after it has changed the coordinator to \( q \)?

A few observations

What if \( p' \), which has not detected the failure of \( c \), receives a STATE-REQ from \( q \)?

- it concludes that \( c'_c \) must be faulty
- it removes from \( UP_v \) every \( q' < q \)

Total failure

Suppose \( p \) is the first process to recover, and that \( p \) is uncertain

Can \( p \) decide ABORT?

Some processes could have decided COMMIT after \( p \) crashed!

\( p \) is blocked until some \( q \) recovers s.t. either

- \( q \) can recover independently
- \( q \) is the last process to fail – then \( q \) can simply invoke the termination protocol
Total failure

- Suppose $p$ is the first process to recover, and that $p$ is uncertain
- Can $p$ decide ABORT?

Some processes could have decided COMMIT after $p$ crashed!

Total failure

- Suppose $p$ is the first process to recover, and that $p$ is uncertain
- Can $p$ decide ABORT?

Some processes could have decided COMMIT after $p$ crashed!

$p$ is blocked until some $q$ recovers s.t. either
- $q$ can recover independently
- $q$ is the last process to fail—then $q$ can simply invoke the termination protocol

Determining the last process to fail

- Suppose a set $R$ of processes has recovered
- Does $R$ contain the last process to fail?

Determining the last process to fail

- Suppose a set $R$ of processes has recovered
- Does $R$ contain the last process to fail?

- the last process to fail is in the $UP$ set of every process
- so the last process to fail must be in $\bigcap_{p \in R} UP_p$
Determining the last process to fail

Suppose a set $R$ of processes has recovered

Does $R$ contain the last process to fail?
  - the last process to fail is in the $UP$ set of every process
  - so the last process to fail must be in

$$\bigcap_{p \in R} UP_p$$

$R$ contains the last process to fail if

$$\bigcap_{p \in R} UP_p \subseteq R$$