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.

Comments

AC1:

We do not require all processes to reach a decision
We do not even require all correct processes to reach a decision (impossible to accomplish if links fail)

AC4:

Avoids triviality
Allows Abort even if all processes have voted yes

NOTE:

A process that does not vote Yes can unilaterally abort

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
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. sends VOTE-REQ to all participants

II. sends \( vote_i \) to Coordinator

if \( vote_i = NO \) then

\( decide_c := ABORT \)

halt

III. if (all votes YES) then

\( decide_c := COMMIT \)

send COMMIT to all

else

\( decide_c := ABORT \)

send ABORT to all who voted YES

halt
**2-Phase Commit**

I. sends **VOTE-REQ** to all participants

II. sends **vote** to Coordinator
   - if **vote** = **NO** then
     - decide := **ABORT**
     - halt
   - else
     - halt

III. if (all votes **YES**) then
   - decide := **COMMIT**
   - send **COMMIT** to all
   - else
     - decide := **ABORT**
     - send **ABORT** to all who voted **YES**
     - halt

IV. if received **COMMIT** then
   - decide := **COMMIT**
   - else
     - decide := **ABORT**
     - halt

**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

**Timeout actions**

Processes are waiting on steps 2, 3, and 4

- **Step 2** $p_i$ is waiting for **VOTE-REQ** from coordinator
- **Step 3** Coordinator is waiting for vote from participants
- **Step 4** $p_i$ (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 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

$c$ appends list of participants to VOTE-REQ

- When an uncertain process $p$ times out, it sends a DECISION-REQ message to every other participant $q$

- If $q$ has decided, then it sends its decision value to $p$, which decides accordingly

- If $q$ has not yet voted, then it decides ABORT, and sends ABORT to $p$

- What if $q$ is uncertain?
Logging actions

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

Alternatively, $p$ is a participant:
   - if DT Log contains a decision value, then decide accordingly
   - else decide ABORT

$p$ 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
4. After participant receives decision value, it writes it to DT Log

Otherwise, if DT Log contains a decision value, then decide accordingly
else decide ABORT

$p$ 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
4. After participant receives decision value, it writes it to DT Log

Otherwise, if DT Log contains a decision value, then decide accordingly
else if it does not contain a Yes vote, decide ABORT
else (Yes but no decision) run a termination protocol
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!

3-Phase Commit

- Two approaches:
  1. Focus only on site failures
     - Non-blocking, unless all sites fail
     - Timeout = site at the other end failed
     - Communication failures can produce inconsistencies
  2. Tolerate both site and communication failures
     - Partial failures can still cause blocking, but less often than in 2PC

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
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!

In state PC, a process knows that it will commit unless it fails.
**2PC Revisited**

1. \( P_i \) sends VOTE-REQ to all participants.
2. When participant \( P_i \) receives a VOTE-REQ, it responds by sending a vote to \( c \) if \( \text{vote}_i = \text{No} \), then \( \text{decide}_i := \text{ABORT} \) and \( P_i \) halts.
3. \( c \) collects votes from all.
   - if all votes are Yes, then \( c \) sends PRECOMMIT to all
   - else \( \text{decide}_c := \text{ABORT} \) sends ABORT to all who voted Yes and halts.
4. if \( P_i \) receives PRECOMMIT then it sends ACK to \( c \)
5. \( c \) collects ACKs from all.
   When all ACKs have been received, \( \text{decide}_c := \text{COMMIT} \); \( c \) sends COMMIT to all.
6. When \( P_i \) receives COMMIT, \( P_i \) sets \( \text{decide}_i := \text{COMMIT} \) and halts.

---

**3PC: The Protocol**

Dale Skeen (1982)

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 \( \text{vote}_i = \text{No} \), then \( \text{decide}_i := \text{ABORT} \) and \( P_i \) halts.
III. \( c \) collects votes from all.
   - if all votes are Yes, then \( c \) sends PRECOMMIT to all
   - else \( \text{decide}_c := \text{ABORT} \) sends ABORT to all who voted Yes and 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, \( \text{decide}_c := \text{COMMIT} \); \( c \) sends COMMIT to all.
VI. When \( P_i \) receives COMMIT, \( P_i \) sets \( \text{decide}_i := \text{COMMIT} \) and halts.

---

**Wait a minute!**

Messages are known to the receiver before they are sent...so, why are they sent?
Wait a minute!

Messages are known to the receiver before they are sent... so, why are they sent?
They inform the recipient of the protocol's progress!

When $c$ receives ACK from $p$, it knows $p$ is not uncertain
When $p$ receives COMMIT, it knows no participant is uncertain, so it can commit

1. $c$ sends VOTE-REQ to all participants
2. When participant $p_i$ receives a VOTE-REQ, it responds by sending a vote to $c$
   if vote = No, then decide COMMIT and halts
3. $c$ collects vote from all
   if all votes are Yes, then $c$ sends PRECOMMIT to all
   else, decide COMMIT and sends COMMIT to all who voted Yes
   $c$ halts
4. if $p_i$ receives PRECOMMIT then it sends ACK to $c$
5. $c$ collects ACKs from all
   when all ACKs have been received, decide COMMIT
   $c$ sends COMMIT to all
6. When $p_i$ receives COMMIT, $p_i$ sets 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
- Step 6 $p_i$ waits for COMMIT

- Step 3 Coordinator is waiting for vote from participants
- Step 5 Coordinator waits for ACKs

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

Step 4 $p_i$ waits for PRECOMMIT

Step 5 Coordinator waits for ACKs
Coordinator sends COMMIT

Step 6 $p_i$ 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 3 Coordinator is waiting for vote from participants
Exactly as in 2PC

Step 4 $p_i$ waits for PRECOMMIT

Step 5 Coordinator waits for ACKs
Run some Termination protocol

Step 6 $p_i$ waits for COMMIT

Participant knows what is going to receive...

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

Step 4 $p_i$ waits for PRECOMMIT

Step 5 Coordinator waits for ACKs
Run some Termination protocol

Step 6 $p_i$ waits for COMMIT

Participant knows what is going to receive...

but NB property can be violated!
Termination protocol: 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 \( p \) 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

TR1. if some process decided ABORT, then decide ABORT send ABORT to all halt
TR2. if some process decided COMMIT, then decide COMMIT send COMMIT to all halt
TR3. if all processes that reported state are uncertain, then decide ABORT send ABORT to all halt
TR4. if some process is committable, but none committed, then send PRECOMMIT to uncertain processes wait for ACKs send COMMIT to all halt

Termination protocol and failures

Processes can fail while executing the termination protocol...

- if \( c \) times out on \( p \), it can just ignore \( p \)
- if \( c \) 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
  - $p$ asks 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)

3PC is non-blocking:

- 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 \( UP_{p'} \) every \( q' < q \)

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

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