Atomic Commit

CSE550
Goal

Maintain consistent state for distributed transactions
Why is this hard?

- Common knowledge (i.e., shared memory) is useful — and often assumed

- Example — Dots on foreheads
  Goal: Determine if I have a dot

  (1) Sees a dot on (2)
  (2) Sees a dot on (1)
Local vs common knowledge

Someone announces – “there is at least one dot”
In distributed systems, we can’t assume simultaneous (i.e., common) knowledge.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>Outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dot</td>
<td>No dot</td>
<td>(1) immediately declares “dot!”</td>
</tr>
<tr>
<td>No dot</td>
<td>Dot</td>
<td>(2) immediately declares “dot!”</td>
</tr>
<tr>
<td>Dot</td>
<td>Dot</td>
<td>After the other person doesn’t say dot, both</td>
</tr>
</tbody>
</table>

In distributed systems, we can’t assume simultaneous (i.e., common) knowledge.
Two Generals Problem

Goal: Agree to attack at dawn

(Communicate by messenger)

Barbarians kill messengers
Two Generals Problem

Claim: There is no protocol that always guarantees generals will attack simultaneously
Two Generals Problem

Claim: There is no protocol that always guarantees generals will attack simultaneously.

Proof: By contradiction, consider a protocol that solves the Two Generals problem using the least number of messages.

Let that number be $n$. Consider the $n$-th message $m_{\text{last}}$.

The state of sender of $m_{\text{last}}$ cannot depend on $m_{\text{last}}$ receipt.
The state of receiver of $m_{\text{last}}$ cannot depend on $m_{\text{last}}$ receipt.

So both sender and receiver would come to the same conclusion even without sending $m_{\text{last}}$.

We now have a new solution requiring only $n-1$ messages.
Goal

Maintain consistent state for distributed transactions

- Each transaction has a coordinator and participating nodes
- Each node has reliable storage
- Otherwise, anything can fail
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

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.

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 when

- It has already voted Yes
- But it does not yet have sufficient information to know the global decision

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

Participant $p_i$
2-Phase Commit

Coordinator $c$

I. Sends VOTE-REQ to all participants

Participant $p_i$

II. Sends $vote_i$ to Coordinator

If $vote_i = NO$ then

$decide_i := ABORT$

$halt$
2-Phase Commit

I. sends VOTE-REQ to all participants

III. if (all votes YES) then
   decision := COMMIT
   send COMMIT to all
else
   decision := ABORT
   send ABORT to all who voted YES
halt
2-Phase Commit

I. sends VOTE-REQ to all participants

III. if (all votes YES) then
    \( \text{decide}_c := \text{COMMIT} \)
    send COMMIT to all
else
    \( \text{decide}_c := \text{ABORT} \)
    send ABORT to all who voted YES
    halt

II. sends \( \text{vote}_i \) to Coordinator
    if \( \text{vote}_i = \text{NO} \) then
        \( \text{decide}_i := \text{ABORT} \)
        halt

IV. if received COMMIT then
    \( \text{decide}_i := \text{COMMIT} \)
else
    \( \text{decide}_i := \text{ABORT} \)
    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, can decide ABORT and halt.

**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, 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
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, 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

\( p_i \) cannot decide: it must run a termination protocol
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? Then cannot help $p$
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
   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

\( \diamond \) if DT Log contains START-2PC, then \( p = c \):
   - 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
   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

- if DT Log contains START-2PC, then $p = c$:
  - if DT Log contains a decision value, then decide accordingly
  - else decide ABORT

- otherwise, $p$ is a participant:
  - 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.