CS 541 Database Systems

Distributed Recovery
Distributed Recovery

• Assume a distributed architecture composed of several local TM, Schedulers, RM, and CMs.
• Ignore replication for now.
• Allow each site’s CM to manage the local cache for local data.
• Allow each site’s RM to manage recovery (as in the centralized case) using any of the recovery algorithms discussed earlier.
• Will this work, as with distributed concurrency control?
NO!

• What is the key difference?
• Atomicity – global agreement.
• Isn’t *global agreement* required for CC as well?
• CC is handled locally, without communication.
• Global agreement is required for CC:
  – 2PL – avoiding deadlocks; but this is avoided by strict 2PL.
  – TO – to ensure agreement on relative ordering of txns; this is piggy-backed with the request message sent by TM to schedulers.
Atomic Commit Protocol

- Distributed recovery requires global agreement – every site must agree whether to commit or abort a txn – this must be an atomic action.
- Protocols that ensure a consistent decision across distributed sites are called Atomic Commitment Protocols (ACPs).
- Where does the problem come from?
- Failures!!!
Failures

- **Site** failures:
  - Assume non-Byzantine failures
  - Fail-stop
  - Partial or total

- In the absence of failures, each pair of sites can communicate.

- **Communication** failures:
  - When two sites, neither of which has failed, are unable to communicate (via any route)
  - May lead to **partitioning**.
Failures

• If a message is undeliverable, we assume that it is dropped.
• Failures are detected using timeouts.
• Can lead to false detections, as well as delayed detections.
• Must make a judicious choice of the timeout interval.
• A failed site may recover at any time – will need to execute the recovery process at that time.
ACP

- The steps in an ACP are as follows:
  - TM gets a commit operation from the txn.
  - ACP needs to arrive at a single, consistent decision to commit or abort based upon the state of the txn at each site i.e.
    - Scheduler
    - DM (ensure that redo rule is satisfied) if there were only read operations at a site, ACP doesn’t need to consult DM
  - Can do this by polling all sites.
  - Send the decision to each site.
System Model

• We abstract the problem to one of reaching a decision between distributed processes.

• There are two types of processes:
  – *Coordinator*: This is the site which initiates the ACP – I.e. where the TM gets the commit operation. Note that the decision if txn want to abort is easy to arrive at.
  – *Participants*: all other processes involved.
System Model

• Initially, the coordinator knows all the participants, but the participants don’t know each other.

• Assume that each site has a distinct log called the Distributed Transaction Log (DT Log).

• Each process can vote yes or no.

• Each process can reach a decision: commit or abort.
ACP Requirements

- **AC1**: All processes that reach a decision reach the same one.
- **AC2**: A process cannot reverse its decision after it has reached one.
- **AC3**: The *Commit* decision can only be reached if all processes voted *Yes*.
- **AC4**: If there are no failures and all processes voted yes, then the decision will be to commit.
- **AC5**: Consider any execution containing only failures that the ACP is designed to tolerate. At any point in this execution, if all existing failures are repaired and no new failures occur for sufficiently long, then all processes will eventually reach a decision.
ACP terminology

• The period between sending a yes vote and reaching a decision is called the **uncertainty period**.

• When a process must await the repair of failures before proceeding, we say that it is **blocked**. E.g. when a failure disables comm. between a process and all other sites when the process is uncertain.

• If a process fails while uncertain, it cannot reach a decision on its own upon recovery – it must communicate with other processes. An ACP that avoids such situations has the **independent recovery** property.
Negative Results

- Total failures + lack of independent recovery give rise to blocking.
- Can we eliminate uncertainty periods?
- Unfortunately, there are two well-known results:
  1. *There are no ACPs that eliminate blocking if communication or total failures are possible.*
  2. *No ACP can guarantee independent recovery of failed processes.*
2 Phase Commit Protocol.

1. *Coord* sends *VOTE_REQ* to all *participants*.
2. Each *P* sends a msg back with its *vote*: *YES* or *NO*. If it votes *NO*, it *decides* *ABORT* and stops.
3. The *Coord* collects all votes.
   - If all are *YES* and its own *vote* is *YES*, it *decides* *COMMIT* and sends *COMMIT* msgs to each participant. Stop
   - Otherwise, it *decides* *ABORT* and send *ABORT* msgs to all participants that voted *YES*. Stop.
4. Each participant that voted *YES* waits for the *coord’s* decision, *decides* accordingly and stops.
Complications

• If no failures take place this ACP works fine.
• However, if there are failures, we need to specify what happens when:
  – There is a timeout while waiting for a message; or
  – A site crashes and then recovers during the ACP?
• Timeout actions:
  – Participant waiting for a *VOTE_REQ*: unilaterally abort.
  – Coord waiting for a vote: *decide ABORT* and send msg to all sites that voted *yes*.
Timeout Actions

• While waiting for the decision: tricky – use a termination protocol
• Suppose that participant $P$ needs to determine the decision
  – $P$ can wait until it can communicate with Coord
  – This is simple, but could unnecessarily block $P$
  – Instead, $P$ could learn of the decision from some other participant.
• The second option can be used with the cooperative termination protocol
Cooperative Termination Protocol

• Process $P$ sends a $\text{decision}_\text{REQ}$ message to every participant, $Q$. $P$ learns of the other participants from the $\text{VOTE}_\text{REQ}$ message sent by the Coord.

• $Q$ does the following:
  – If $Q$ has already decided, then it send its decision to $P$
  – If $Q$ has not yet voted, then it can unilaterally abort and send $\text{ABORT}$ to $P$.
  – If $Q$ is also uncertain then it cannot help $P$ – both are blocked.
Handling site failure in 2PC

- We use a **distributed transaction log** to record necessary information about termination protocols, in order to recover correctly.
- The DT log can be a part of the regular log too.
- It works as follows:
  - When *Coord* sends a *VOTE_REQ*, it writes a **start-2PC** record (before or after sending message).
  - If a participant votes *yes*, it writes a **yes** record **before** sending the vote. This record contains the identities of the coordinator and other participants (as given by the initial message of the coord).
DT Log

• If the participant votes *no*, it writes an *abort* record, either before or after sending the vote.
• Before the *Coord* sends a commit decision, it writes a *commit* record.
• When the *Coord* sends abort, it writes the *abort* record to the log
• After receiving commit_abort, a participant writes a *commit_abort* record to its log.
Recovery

• When a site recovers, the fate of a distributed txn is determined as follows.

• If the DT log contains a \textit{start-2PC} record, then the recovering site, $s$, was the coordinator
  – if it also contains a \textit{commit} or \textit{abort} record, then the coord had reached a decision before failure.
  – if neither is found, the coord can now unilaterally \textit{decide} ABORT.

• If the DT log doesn’t contain the \textit{start-2PC} record, then the site was a participant. There are \textbf{three cases}:
Recovery (contd.)

– The DT log contains a *commit* or *abort* record – I.e. participant had reached a decision.

– The DT log does not contain a *yes* record: either the participant failed before voting, or voted *NO*. It can therefore unilaterally decide to *ABORT*.

– The DT log contains a *yes* record, but no *commit* or *abort* record: participant failed during the uncertainty period – use the termination protocol to determine fate.
Garbage Collection

• As with the regular log, the DT log needs to be garbage collected. There are two basic rules:
  • GC1: A site cannot delete entries of txn $T$ from the DT log at least until its RM has processed $RM$-Commit or $RM$-Abort.
  • GC2: At least one site must not delete the records of txn $T$ from its DT log until that site has received messages indicating that $RM$-Commit$(T)$ or $RM$-Abort$(T)$ has been processed at all other sites where $T$ executed. (can be done by the coordinator).
How good is 2PC?

- **Resiliency**: what types of failures does it tolerate?
  - *Site* and *communication* failures (even partitioning)
- **Blocking**: does it block, if so when?
  - Yes. If a process times out in its uncertainty period and can only communicate with other uncertain processes.
- **Time Complexity**: How many rounds?
  - With no failures: 3 rounds are needed.
  - With *failures*, we need a termination protocol, that could add 2 more rounds.
  - Each failure could result in these extra rounds, but they could overlap, so we count only 2 rounds.
Message Complexity

- **Message Complexity**: with $n$ participants
  - With no failures: $3n$ messages.
  - If there are $m$ sites that invoke the termination protocol, then $mn$ `DECISION_REQ` messages are sent, at most $(n-m+1)$ could respond. With each round of the termination protocol, one less process is in its uncertainty period, and thus one more could respond, therefore the maximum number is:

$$mn + \sum_{i=1}^{m} (n - m + i) = 2nk - \frac{m^2}{2} + \frac{m}{2}$$

which is at most $n(3n+1)/2 + 3n$ in total.
Alternative 2PC

- **Decentralised**: complete graph. Better time complexity.
- **Linear**: better message complexity.

<table>
<thead>
<tr>
<th></th>
<th>Centralised</th>
<th>Decentralised</th>
<th>Linear</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>3 rounds</td>
<td>2 rounds</td>
<td>2n rounds</td>
</tr>
<tr>
<td>MSG</td>
<td>3n</td>
<td>n+n^2</td>
<td>2n</td>
</tr>
</tbody>
</table>