Distributed commit protocols

Problem

Have an operation being performed by each member of a process group, or none at all.

- Reliable multicasting: a message is to be delivered to all recipients.
- Distributed transaction: each local transaction must succeed.

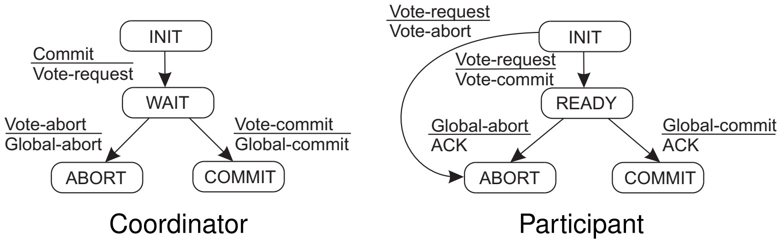
Two-phase commit protocol (2PC)

Essence

The client who initiated the computation acts as coordinator; processes required to commit are the participants.

- Phase 1a: Coordinator sends VOTE-REQUEST to participants (also called a pre-write)
- Phase 1b: When participant receives VOTE-REQUEST it returns either VOTE-COMMIT or VOTE-ABORT to coordinator. If it sends VOTE-ABORT, it aborts its local computation
- Phase 2a: Coordinator collects all votes; if all are VOTE-COMMIT, it sends GLOBAL-COMMIT to all participants, otherwise it sends GLOBAL-ABORT
- Phase 2b: Each participant waits for GLOBAL-COMMIT or GLOBAL-ABORT and handles accordingly.

2PC - Finite state machines



2PC – Failing participant

Analysis: participant crashes in state *S*, and recovers to *S*

- *INIT*: No problem: participant was unaware of protocol

2PC – Failing participant

Analysis: participant crashes in state S, and recovers to S

- *READY*: Participant is waiting to either commit or abort. After recovery, participant needs to know which state transition it should make $\Rightarrow$ log the coordinator's decision

2PC – Failing participant

Analysis: participant crashes in state S, and recovers to S

- *ABORT*: Merely make entry into abort state idempotent, e.g., removing the workspace of results

2PC – Failing participant

Analysis: participant crashes in state S, and recovers to S

- *COMMIT*: Also make entry into commit state idempotent, e.g., copying workspace to storage.

2PC – Failing participant

Analysis: participant crashes in state *S*, and recovers to *S*

- *INIT*: No problem: participant was unaware of protocol
- *READY*: Participant is waiting to either commit or abort. After recovery, participant needs to know which state transition it should make ⇒ log the coordinator's decision
- *ABORT*: Merely make entry into abort state idempotent, e.g., removing the workspace of results
- *COMMIT*: Also make entry into commit state idempotent, e.g., copying workspace to storage.

Observation

When distributed commit is required, having participants use temporary workspaces to keep their results allows for simple recovery in the presence of failures.

2PC – Failing participant

Alternative

When a recovery is needed to *READY* state, check state of other participants $\Rightarrow$ no need to log coordinator's decision.

Recovering participant *P* contacts another participant *Q*

State of *Q*	**Action by** *P*
COMMIT	Make transition to *COMMIT*
ABORT	Make transition to *ABORT*
INIT	Make transition to *ABORT*
READY	Contact another participant

Result

If all participants are in the *READY* state, the protocol blocks. Apparently, the coordinator is failing. Note: The protocol prescribes that we need the decision from the coordinator.

2PC – Failing coordinator

Observation

The real problem lies in the fact that the coordinator's final decision may not be available for some time (or actually lost).

Alternative

Let a participant *P* in the *READY* state timeout when it hasn't received the coordinator's decision; *P* tries to find out what other participants know (as discussed).

Observation

Essence of the problem is that a recovering participant cannot make a local decision: it is dependent on other (possibly failed) processes

Coordinator in Python

```python
class Coordinator:

  def run(self):
    yetToReceive = list(participants)
    self.log.info('WAIT')
    self.chan.sendTo(participants, VOTE_REQUEST)
    while len(yetToReceive) > 0:
      msg = self.chan.recvFrom(participants, TIMEOUT)
      if (not msg) or (msg[1] == VOTE_ABORT):
        self.log.info('ABORT')
        self.chan.sendTo(participants, GLOBAL_ABORT)
        return
      else: # msg[1] == VOTE_COMMIT
        yetToReceive.remove(msg[0])
    self.log.info('COMMIT')
    self.chan.sendTo(participants, GLOBAL_COMMIT)
```

Participant in Python

```python
class Participant:
  def run(self):
    msg = self.chan.recvFrom(coordinator, TIMEOUT)
    if (not msg):  # Crashed coordinator - give up entirely
      decision = LOCAL_ABORT
    else: # Coordinator will have sent VOTE_REQUEST
      decision = self.do_work()
      if decision == LOCAL_ABORT:
        self.chan.sendTo(coordinator, VOTE_ABORT)
      else: # Ready to commit, enter READY state
        self.chan.sendTo(coordinator, VOTE_COMMIT)
        msg = self.chan.recvFrom(coordinator, TIMEOUT)
        if (not msg): # Crashed coordinator - check the others
          self.chan.sendTo(all_participants, NEED_DECISION)
          while True:
            msg = self.chan.recvFromAny()
            if msg[1] in [GLOBAL_COMMIT, GLOBAL_ABORT, LOCAL_ABORT]:
              decision = msg[1]
              break
        else: # Coordinator came to a decision
          decision = msg[1]

    while True: # Help any other participant when coordinator crashed
      msg = self.chan.recvFrom(all_participants)
      if msg[1] == NEED_DECISION:
        self.chan.sendTo([msg[0]], decision)
```

Recovery: Background

Essence

When a failure occurs, we need to bring the system into an error-free state:

- Forward error recovery: Find a new state from which the system can continue operation
- Backward error recovery: Bring the system back into a previous error-free state

Practice

Use backward error recovery, requiring that we establish recovery points

Observation

Recovery in distributed systems is complicated by the fact that processes need to cooperate in identifying a consistent state from where to recover

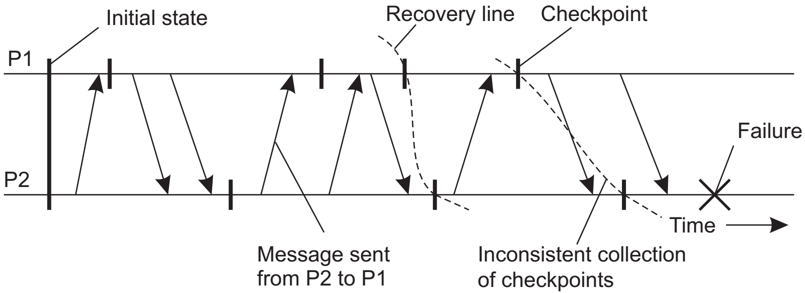
Consistent recovery state

Requirement

Every message that has been received is also shown to have been sent in the state of the sender.

Recovery line

Assuming processes regularly checkpoint their state, the most recent consistent global checkpoint.



Coordinated checkpointing

Essence

Each process takes a checkpoint after a globally coordinated action.

Simple solution

Use a two-phase blocking protocol:

Coordinated checkpointing

Essence

Each process takes a checkpoint after a globally coordinated action.

Simple solution

Use a two-phase blocking protocol:

- A coordinator multicasts a checkpoint request message

Coordinated checkpointing

Essence

Each process takes a checkpoint after a globally coordinated action.

Simple solution

Use a two-phase blocking protocol:

- A coordinator multicasts a checkpoint request message
- When a participant receives such a message, it takes a checkpoint, stops sending (application) messages, and reports back that it has taken a checkpoint

Coordinated checkpointing

Essence

Each process takes a checkpoint after a globally coordinated action.

Simple solution

Use a two-phase blocking protocol:

- A coordinator multicasts a checkpoint request message
- When a participant receives such a message, it takes a checkpoint, stops sending (application) messages, and reports back that it has taken a checkpoint
- When all checkpoints have been confirmed at the coordinator, the latter broadcasts a checkpoint done message to allow all processes to continue

Coordinated checkpointing

Essence

Each process takes a checkpoint after a globally coordinated action.

Simple solution

Use a two-phase blocking protocol:

- A coordinator multicasts a checkpoint request message
- When a participant receives such a message, it takes a checkpoint, stops sending (application) messages, and reports back that it has taken a checkpoint
- When all checkpoints have been confirmed at the coordinator, the latter broadcasts a checkpoint done message to allow all processes to continue

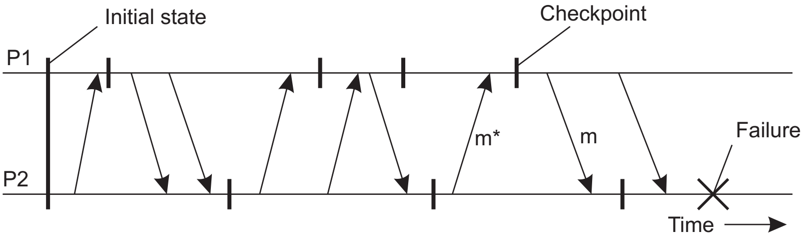
Observation

It is possible to consider only those processes that depend on the recovery of the coordinator, and ignore the rest

Cascaded rollback

Observation

If checkpointing is done at the “wrong” instants, the recovery line may lie at system startup time. We have a so-called cascaded rollback.



Independent checkpointing

Essence

Each process independently takes checkpoints, with the risk of a cascaded rollback to system startup.

Independent checkpointing

Essence

Each process independently takes checkpoints, with the risk of a cascaded rollback to system startup.

- Let $CP_i(m)$ denote m^{th} checkpoint of process P_i and $INT_i(m)$ the interval between $CP_i(m-1)$ and $CP_i(m)$.

Independent checkpointing

Essence

Each process independently takes checkpoints, with the risk of a cascaded rollback to system startup.

- Let $CP_i(m)$ denote m^{th} checkpoint of process P_i and $INT_i(m)$ the interval between $CP_i(m-1)$ and $CP_i(m)$.
- When process P_i sends a message in interval $INT_i(m)$, it piggybacks (i,m)

Independent checkpointing

Essence

Each process independently takes checkpoints, with the risk of a cascaded rollback to system startup.

- Let $CP_i(m)$ denote m^{th} checkpoint of process P_i and $INT_i(m)$ the interval between $CP_i(m-1)$ and $CP_i(m)$.
- When process P_i sends a message in interval $INT_i(m)$, it piggybacks (i,m)
- When process P_j receives a message in interval $INT_j(n)$, it records the dependency $INT_i(m) \rightarrow INT_j(n)$.

Independent checkpointing

Essence

Each process independently takes checkpoints, with the risk of a cascaded rollback to system startup.

- Let $CP_i(m)$ denote m^{th} checkpoint of process P_i and $INT_i(m)$ the interval between $CP_i(m-1)$ and $CP_i(m)$.
- When process P_i sends a message in interval $INT_i(m)$, it piggybacks (i, m)
- When process P_j receives a message in interval $INT_j(n)$, it records the dependency $INT_i(m) \to INT_j(n)$.
- The dependency $INT_i(m) \to INT_j(n)$ is saved to storage when taking checkpoint $CP_j(n)$.

Independent checkpointing

Essence

Each process independently takes checkpoints, with the risk of a cascaded rollback to system startup.

- Let $CP_i(m)$ denote m^{th} checkpoint of process P_i and $INT_i(m)$ the interval between $CP_i(m-1)$ and $CP_i(m)$.
- When process P_i sends a message in interval $INT_i(m)$, it piggybacks (i,m)
- When process P_j receives a message in interval $INT_j(n)$, it records the dependency $INT_i(m) \rightarrow INT_j(n)$.
- The dependency $INT_i(m) \rightarrow INT_j(n)$ is saved to storage when taking checkpoint $CP_j(n)$.

Observation

If process P_i rolls back to $CP_i(m-1)$, P_j must roll back to $CP_j(n-1)$.

Message logging

Alternative

Instead of taking an (expensive) checkpoint, try to replay your (communication) behavior from the most recent checkpoint $\Rightarrow$ store messages in a log.

Assumption

We assume a piecewise deterministic execution model:

- The execution of each process can be considered as a sequence of state intervals
- Each state interval starts with a nondeterministic event (e.g., message receipt)
- Execution in a state interval is deterministic

Conclusion

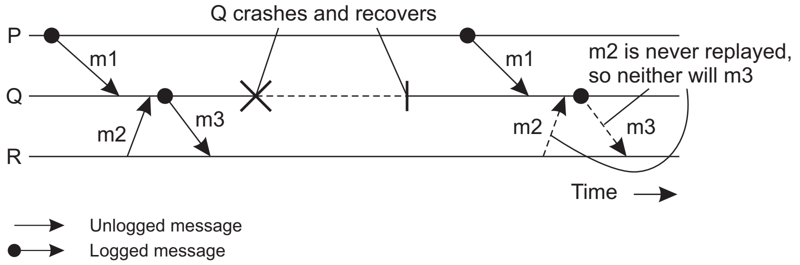
If we record nondeterministic events (to replay them later), we obtain a deterministic execution model that will allow us to do a complete replay.

Message logging and consistency

When should we actually log messages?

Avoid orphan processes:

- Process Q has just received and delivered messages m_1 and m_2
- Assume that m_2 is never logged.
- After delivering m_1 and m_2, Q sends message m_3 to process R
- Process R receives and subsequently delivers m_3: it is an orphan.



Message-logging schemes

Notations

- **DEP**(m): processes to which m has been delivered. If message m^* is causally dependent on the delivery of m, and m^* has been delivered to Q, then $Q \in$ **DEP**(m).
- **COPY**(m): processes that have a copy of m, but have not (yet) reliably stored it.
- **FAIL**: the collection of crashed processes.

Characterization

$$Q \text{ is orphaned} \Leftrightarrow \exists m : Q \in \textbf{DEP}(m) \text{ and } \textbf{COPY}(m) \subseteq \textbf{FAIL}$$

Message-logging schemes

Pessimistic protocol

For each nonstable message m, there is at most one process dependent on m, that is $|\mathbf{DEP}(m)| \leq 1$.

Consequence

An unstable message in a pessimistic protocol must be made stable before sending a next message.