Distributed Transaction Recovery

- 2 Basic Two-Phase Commit Algorithm (2PC)
  - 2.1 2PC During Normal Operation
  - 2.2 Restart and Termination Protocol
  - 2.3 Independent Recovery
- 3 Transaction-Tree Two-Phase Commit
- 4 Optimizations
- 5 Lessons Learned

“To marry is to halve your rights and to double your duties.“
(Arthur Schopenhauer)
Fundamental Problem of Distributed Commit

**Problem:**
- Transaction operates on multiple servers *(resource managers)*
- Global commit needs unanimous local commits of all *participants (agents)*
- Distributed system may fail partially (server crashes, network failures) and creates the potential danger of inconsistent decisions

**Approach:**
- Distributed handshake protocol known as *two-phase commit (2PC)*
- with a *coordinator* taking responsibility for unanimous outcome
- Recovery considerations for in-doubt transactions
2PC During Normal Operation

• **First phase (voting):**
  coordinator sends *prepare* messages to participants and waits for *yes* or *no* votes

• **Second phase (decision)**
  coordinator sends *commit* or *rollback* messages to participants and waits for *acks*

• **Participants** write *prepared* log entries in voting phase and become *in-doubt (uncertain)*
  → potential **blocking** danger, breach of local autonomy

• Participants write commit or rollback log entry in decision phase

• **Coordinator** writes *begin* log entry

• Coordinator writes *commit* or *rollback* log entry and can now give return code to the client’s commit request

• Coordinator writes *end (done, forgotten)* log entry to facilitate **garbage collection**

→ 4n messages, 2n+2 forced log writes, 1 unforced log write with n participants and 1 coordinator
## Illustration of 2PC

<table>
<thead>
<tr>
<th>Coordinator</th>
<th>Participant 1</th>
<th>Participant 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>force-write</td>
<td></td>
<td></td>
</tr>
<tr>
<td>begin log entry</td>
<td></td>
<td></td>
</tr>
<tr>
<td>send &quot;prepare&quot;</td>
<td>send &quot;prepare&quot;</td>
<td></td>
</tr>
<tr>
<td>send &quot;yes&quot;</td>
<td>send &quot;yes&quot;</td>
<td></td>
</tr>
<tr>
<td>force-write</td>
<td>force-write</td>
<td></td>
</tr>
<tr>
<td>prepared log entry</td>
<td>prepared log entry</td>
<td></td>
</tr>
<tr>
<td>send &quot;commit&quot;</td>
<td>send &quot;commit&quot;</td>
<td></td>
</tr>
<tr>
<td>send &quot;ack&quot;</td>
<td>send &quot;ack&quot;</td>
<td></td>
</tr>
<tr>
<td>write</td>
<td></td>
<td></td>
</tr>
<tr>
<td>end log entry</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Statechart for Basic 2PC

Coordinator:
- Initial
  - Prepare1; Prepare2
- Collecting
  - Sorry1 | Sorry2
  - Yes1 & Yes2
  - Commit1; Commit2
- Committed
  - Ack1 & Ack2
- Aborted
  - Ack1 & Ack2
- Forgotten

Participant 1:
- Initial1
  - Prepare1 / Yes1
  - Prepare1 / Sorry1
- Prepared1
  - Commit1 / Ack1
  - Abort1 / Ack1
- Committed1
  - Commit1 / Ack1
  - Abort1 / Ack1
- Aborted1

Participant 2:
- Initial2
  - Prepare2 / Yes2
  - Prepare2 / Sorry2
- Prepared2
  - Commit2 / Ack2
  - Abort2 / Ack2
- Committed2
  - Commit2 / Ack2
  - Abort2 / Ack2
- Aborted2
Restart and Termination Protocol

**Failure model:**
- process failures: transient server crashes
- network failures: message losses, message duplications
- assumption that there are no malicious commission failures → Byzantine agreement
- no assumptions about network failure handling → can use datagrams or sessions for communication

**Restart protocol after failure (F transitions):**
- coordinator restarts in last remembered state and resends messages
- participant restarts in last remembered state and resends message or waits for message from coordinator

**Termination protocol upon timeout (T transitions):**
- coordinator resends messages and may decide to abort the transaction in first phase
- participant can unilaterally abort in first phase and wait for or may contact coordinator in second phase
Statechart for Basic 2PC with Restart/Termination

coordinator

- initial
  - collecting
    - committed
      - C-pending
        - committed
          - ack1 & ack2
        - forgotten
          - committed
            - T/F
          - aborted
            - T/F
      - aborted
        - / abort1; abort2
    - sorry1 | sorry2
      - / abort1; abort2
    - yes1 & yes2
      - / commit1; commit2
    - sorry1
      - / abort1; abort2
    - yes1 & yes2
      - / commit1; commit2
    - prepare1; prepare2

participant 1

- initial1
  - prepared1
    - commit1 / ack1
      - committed1
        - commit1 / ack1
          - aborted1
            - abort1 / ack1
        - abort1 / ack1
          - aborted1
            - abort1 / ack1
    - abort1 / sorry1
      - committed1
        - commit1 / ack1
          - aborted1
            - abort1 / ack1
        - abort1 / ack1
          - aborted1
            - abort1 / ack1
  - prepare1 / yes1
    - committed1
      - commit1 / ack1
        - aborted1
          - abort1 / ack1
        - abort1 / ack1
          - aborted1
            - abort1 / ack1
    - prepare1 / sorry1
      - committed1
        - commit1 / ack1
          - aborted1
            - abort1 / ack1
        - abort1 / ack1
          - aborted1
            - abort1 / ack1

participant 2

- initial2
  - prepared2
    - commit2 / ack2
      - committed2
        - commit2 / ack2
          - aborted2
            - abort2 / ack2
        - abort2 / ack2
          - aborted2
            - abort2 / ack2
    - abort2 / sorry2
      - committed2
        - commit2 / ack2
          - aborted2
            - abort2 / ack2
        - abort2 / ack2
          - aborted2
            - abort2 / ack2

Correctness of Basic 2PC

**Theorem 1 (Safety):**
2PC guarantees that if one process is in a final state, then either all processes are in their committed state or all processes are in their aborted state.

**Proof methodology:**
Consider the set of possible computation paths starting in global state (initial, initial, ..., initial) and reason about invariants for states on computation paths.

**Theorem 2 (Liveness):**
For a finite number of failures the 2PC protocol will eventually reach a final global state within a finite number of state transitions.
Reinfection of Participants

**Problem:**
- 2PC assumes that regular work of participants is finished when the commit protocol starts.
- Deferred operations that are triggered at commit time violate this assumption.
→ Prepared participant may become *reinfected*.

**Solution:**
- Participant that receives request for deferred operation after having prepared itself needs to become prepared again before replying to the requestor of the operation.
Independent Recovery

*Independent recovery:* ability of a failed and restarted process to terminate his part of the protocol without communicating to other processes.

**Theorem 3:**
Under the *single-failure assumption* it is possible to design a distributed commit protocol so that it can guarantee *independent recovery* for a failed process.

**Proof sketch:**
Avoid global states which have both local commit and abort states in their set of possible global successor states. Coordinator adds willing-to-commit state in between collecting and commit states, and makes T transition from collecting to abort. Participant makes F transition from prepared to abort.
Impossibility of Independent Recovery in the Presence of Multiple Failures

Theorem 4:
There exists no distributed commit protocol that can guarantee independent process recovery in the presence of multiple failures (e.g., network partitionings).

Proof sketch:
Consider two processes, one coordinator and one participant, with computation path $G_0 = (\text{initial, initial}), \ldots, G_m = (\text{committed, committed})$. Under independent recovery both processes would go into the local aborted states from $G_0$ up to some maximum state $G_{(k-1)}$. In state $G_k$ one of the two processes would independently recovery into its local committed state, but the other one would still behave like in state $G_{(k-1)}$ as a message-based protocol changes the components of the global state one at a time.
Transaction-Tree Hierarchical 2PC

During transaction execution the transaction forms a process tree rooted at transaction initiator with bilateral communication links according to request-reply interactions.

At commit time the process tree can be:
- flattened, provided that all participants are known to the coordinator and that establishing new communication links is inexpensive.
- re-used as it was, using a hierarchical form of 2PC with intermediate nodes being both participants and sub-coordinators.
- restructured to choose a coordinator different from the initiator, considering reliability and speed of processes and the communication topology and protocol → coordinator transfer.
Flattened vs. Hierarchical Commit Tree

Flattened 2PC:

Hierarchical 2PC:

communication during transaction execution

communication during commit protocol
Illustration of Hierarchical 2PC

Initiator Process₁  Process₂  Process₃  Process₄  Process₅

_prepare_  prepare  prepare  prepare  prepare

prepare  yes  yes  yes  yes

prepare  yes  yes

commit  yes  yes  yes  yes

commit  commit  commit  commit  commit

commit  commit

ack  ack  ack  ack  ack
2PC Bottlenecks and Optimizations

Goals:
• reduce the number of messages and forced log writes for higher throughput
• shorten the critical path until local locks can be released for faster response time
• reduce the probability of blocking

Possible optimizations:
• fewer messages and forced log writes by presumption in the case of missing information
• eliminating read-only subtrees as early as possible
• (dynamic) coordinator transfer
• protocols with reduced blocking (at expense of more messages)
2PC with Presumption

Observation: forcing the begin log entry is usually unnecessary; losing it is never a correctness problem

Idea:
- participants don‘t need to force commit/abort log entry, can inquire coordinator if in doubt
- if coordinator has already forgotten the transaction, it replies to the inquiry with a presumed result
- coordinator may even be able to forget transaction without acks from participants

Problem:
- need to ensure that the coordinator‘s presumed result is consistent among two cases of missing information:
  a) transaction has not yet entered the decision phase (early loser)
  b) transaction has been decided and garbage collected (forgotten trans.) or needs to be able to discriminate the two cases
Illustration of Presumed-Abort Protocol

<table>
<thead>
<tr>
<th>Process 1</th>
<th>Coordinator</th>
<th>Process 2</th>
<th>Process 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>prepare</td>
<td>prepare</td>
<td>prepare</td>
<td>prepare</td>
</tr>
<tr>
<td>force-write</td>
<td>prepared</td>
<td>force-write</td>
<td>prepared</td>
</tr>
<tr>
<td>prepared</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>no</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>abort</td>
<td>abort</td>
<td>abort</td>
<td></td>
</tr>
<tr>
<td>write rollback</td>
<td></td>
<td>write rollback</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Case 1: transaction abort

Case 2: transaction commit

force-write
force-write
force-write
commit
commit
commit

ack
ack
ack
write end

3/5/2009
### Illustration of Presumed-Commit Protocol

<table>
<thead>
<tr>
<th>Process 1</th>
<th>Coordinator</th>
<th>Process 2</th>
<th>Process 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>force-write prepared</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>prepare</td>
<td>force-write begin</td>
<td></td>
</tr>
<tr>
<td></td>
<td>force-write rollback</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>abort</td>
<td>force-write rollback</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ack</td>
<td>force-write commit</td>
<td></td>
</tr>
<tr>
<td></td>
<td>write end</td>
<td>(force-)write commit</td>
<td></td>
</tr>
</tbody>
</table>

**Case 1: transaction abort**

- Process 1: `force-write prepared`
- Coordinator: `prepare`
- Process 2: `prepare`
- Process 3: `no`
- Process 1: `force-write rollback`
- Process 2: `abort`
- Process 3: `force-write rollback`
- Process 1: `ack`
- Process 3: `ack`
- Process 1: `write end`
- Process 3: `write end`

**Case 2: transaction commit**

- Process 1: `force-write prepared`
- Coordinator: `prepare`
- Process 2: `prepare`
- Process 3: `no`
- Process 1: `force-write rollback`
- Process 2: `abort`
- Process 3: `force-write rollback`
- Process 1: `ack`
- Process 3: `ack`
- Process 1: `write end`
- Process 3: `write end`

Further Optimizations and Extensions

Presumed-commit with non-forced begin log entry:
• identify list of transaction ids of potential loser transactions
  (interval of ids minus transactions with commit log entry)
  by means of non-forced, periodic log entries
→ can now discriminate forgotten winners from early losers
  but may be unable to garbage-collect some log entries after failures

Co-existence of PA and PC participants in heterogeneous federation (presumed-any protocol):
• coordinator force-writes log entries so that it can cope with
  both PA and PC participants
• coordinator needs to know type of participant (to be included
  in begin log entry) and sends and expects messages according to
  participant type
Illustration of Presumed-Any Coordinator

<table>
<thead>
<tr>
<th>Presumed-Abort Participant</th>
<th>Presumed-Any Coordinator</th>
<th>Presumed-Commit Participant</th>
</tr>
</thead>
<tbody>
<tr>
<td>force-write</td>
<td>force-write begin</td>
<td>force-write prepared</td>
</tr>
<tr>
<td>prepared</td>
<td></td>
<td></td>
</tr>
<tr>
<td>yes</td>
<td>prepare</td>
<td></td>
</tr>
<tr>
<td>commit</td>
<td>force-write commit</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>force-write commit</td>
<td>commit</td>
<td>write commit</td>
</tr>
<tr>
<td></td>
<td>ack</td>
<td></td>
</tr>
</tbody>
</table>
Read-only Subtree Optimization

- Read-only leaf-node participant votes “read-only” and releases all its locks
- Read-only intermediate node votes “read-only” when it has received all votes and none of them is “yes” or “no”
- Coordinator eliminates read-only subtrees from decision phase

Read-only optimization more effective for PA than for PC because PC still needs a forced begin log entry. PA can terminate read-only transaction with zero logging cost.

*Caveat* with regard to reinfected participants:
If there is a non-zero probability of some participant having to perform deferred operations, none of the read-only participant must release its locks. Otherwise the resulting history could be globally non-serializable.
Coordinator Transfer

Observations:
- Transaction initiator is the default coordinator, but often a poor choice.
- Coordinator role may be transferred to other process at initiator’s commit request (or even later), based on reliability, speed, interconnectivity, and criticality of servers.

Approaches:
- Rotate process tree around coordinator as new root, reusing established communication links (e.g., sessions).
- Linear 2PC with last-agent optimization for linear process chain:
  - transfer coordinator responsibility along linear process chain
  - using “I’m prepared, you decide” messages
  - 2-party transaction needs only 3 forced log writes and 3 messages
- Dynamic 2PC:
  generalizes last-agent optimization to arbitrary trees
Initiator- vs. Coordinator-Rooted Process Tree

During transaction execution:

initiator

A

B

C

D

E

F

G

H

During commit protocol:

coordinator

C

A

E

F

B

G

H

D
Illustration of Last-Agent Optimization

<table>
<thead>
<tr>
<th>Process 1</th>
<th>Process 2</th>
<th>Process 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Initiator)</td>
<td></td>
<td>(Coordinator)</td>
</tr>
</tbody>
</table>

force-write prepared

“yes / prepare / coordinator transfer”

force-write prepared

“yes / prepare / coordinator transfer”

force-write commit

“commit”

force-write commit

“commit”

force-write commit

“ack”

“ack”

write end
Dynamic 2PC

Goal:
determine coordinator as quickly as possible
(using more messages to compensate for slow branches in the tree)

Dynamic 2PC protocol:
• Nodes in the process tree send prepare messages to all children
• Leaf nodes that receive a prepare message
  prepare themselves and send a yes vote to their parent
• Intermediate nodes that have received
  yes votes from all but one neighbor
  prepare themselves and send a yes vote to their last neighbor
  (generalized last-agent optimization)
• The process that first receives yes votes from all its neighbors
  becomes coordinator
Illustration of Dynamic 2PC

1:  A → prepare B,  A → prepare C
2:  B → prepare D,  B → prepare E
3:  C → prepare F,  C → prepare G
4:  D → yes B,  E → yes B
    B prepared
5:  B → yes A
    A prepared
6:  A → yes C
7:  F → yes C,  G → yes C
    C becomes coordinator
Protocols with Reduced Blocking

• **Three-phase commit protocol (3PC):**
  avoid global state with both local abort and local commit
  in the set of possible global successor states,
  using three phases: voting, dissemination, decision
  no blocking under single-failure assumption
  and still correct in arbitrary failure situations

• **Cooperative termination:**
  in-doubt participants contact other participants rather than
  waiting for coordinator
  (needs participant list included in prepare message)

• **Heuristic commit or abort:**
  in-doubt participants on critical servers
  make unilateral decision upon extra-long timeout,
  based on type of transaction (e.g., abort for withdrawal)
Lessons Learned

• 2PC is a fundamental, versatile and standardized, building block for transactional federations including Internet applications
• Susceptibility to blocking is inherent and the price for consistency
• Scalability is limited to few participants; asynchronously queued transactions are an alternative, provide atomicity and thus need 2PC, but sacrifice isolation
• Optimizations aim to
  • reduce forced logging and messages (PA and PC)
  • shorten time until lock release
  • (read-only votes, last-agent optimization, dynamic 2PC)
  • reduce blocking (cooperative termination, 3PC)
• Coordinator transfer is crucial to avoid dependency on poorly administered initiator or unreliable server (application server is often an appropriate coordinator)
Take-Away Messages

• Transactions are an important way of providing execution guarantees
• Their basic abstraction mechanisms allow for a host of applications to be captured
• Their fundamental algorithmic approaches center around a few concepts
• Transactional guarantees have expanded way beyond the database area