CS505: Distributed Systems

Lecture 13: Distributed Transactions
Outline

- Distributed Transactions
- Two Phase Commit and Three Phase Commit
- Non-blocking Atomic Commit with $P$
A transaction is an atomic program describing a sequence of accesses to shared and distributed data.

A transaction can be terminated either by committing or aborting.

The effects of any aborted transaction are cancelled: everything happens as if the transaction had never taken place.

The effects of any committed transaction are permanent.
Transactions hence have two purposes

1. **Concurrency/synchronization mechanism**
   - Avoid data races
   - Reflect point of view of “individual programs”

2. **Failure handling**
   - Transient or permanent failure leads to aborting entire transaction
   - (Sub)program remains within set of *legal* states (cf. Lecture 4)
Challenge

- Decide on commit or abort of distributed transactions

- The same failures that may make a transaction fail make this decision hard
Atomic Commit

- As in consensus, every process proposes an initial value 0 (no) or 1 (yes) and must decide on a final value 0 (abort) or 1 (commit).

- The proposition means the ability to commit the transaction.

- The decision means the contract with the user.

- Unlike consensus, the processes here seek to decide 1 but every process has a veto right.
  - Biased propositions, and constraints on decisions.
Formally

I. Agreement
   – No two processes decide differently

II. Termination
   – Every correct process eventually decides

III. Commit-Validity
   – 1 can only be decided if all processes propose 1

IV. Abort-Validity
   – 0 can only be decided if some process crashes or votes 0
Illustration

p1

propose(0)  decide(0)

propose(1)

p2

propose(0)  decide(0)

p3
Two-Phase Commit (2PC)

[Lamport’76], [Gray’79]

Central coordinator initiates protocol

1. Phase
   - Coordinator asks if participants can commit
   - Participants respond yes/no

2. Phase
   - If all votes yes, coordinator sends commit
   - Participants respond when done

Blocks on failure

- If coordinator fails must wait for recovery

While blocked, transaction must remain isolated

- Prevents other transactions from completing
propose(1)
propose(1)
propose(1)
propose(1)
decide(0)
decide(0)
crash
First Formal Model for Commit

[Skeen&Stonebraker ‘83]

- Necessary conditions for non-blocking with 1 failure
  - Leads to Three-Phase Commit (3PC) protocol

- Transaction has “commit point”
  - Failure after ⇒ transaction visible
  - Failure before ⇒ abort as part of recovery

- Protocol ensures commit/abort decision unanimous
  - Any site can abort before first site commits
  - No site can abort after first site commits

- Independent recovery
  - Recovering process need not consult with others

- Non-blocking protocol
  - Failure doesn’t cause operational sites to suspend
Network

- Point to point communication
- Maximum delay $T$ or timeout
  - If timeout, sender can assume recipient and/or network failure
  - If network failure, sender doesn’t know if message received, however no slow processes

Each site can be viewed as finite state automaton

- Sites have three state “classes”
  - Initial: Allowed to abort the transaction
  - Abort: Can’t transition to non-abort
  - Commit: Can’t transition to non-commit
- Non-deterministic with respect to protocol, i.e., state transitions may occur independent of protocol due to local actions
- State diagram is acyclic (assures termination)
- Site transitions asynchronous
- State transitions atomic
State Transition Graph for 2PC

### Coordinator
- Node $q_1$
- Transitions:
  - xact request/start xact
- States:
  - $w_1$
  - $a_1$
  - $c_1$
  - $q_1$

### Participant
- Node $q_2$
- Transitions:
  - start xact/yes
- States:
  - $p_2$
  - $a_2$
  - $c_2$
  - $q_2$
Transaction Model

- **Global transaction state**
  - Vector of local states + outstanding messages in the network
  - Final if all local states in final state
  - Inconsistent if vector contains both commit and abort states
  - Global state transition occurs when local state transition occurs

- **Reachable state graph**
  - Possible global transitions
  - Protocol correct iff reachable state graph has
    - No inconsistent states
    - All terminal states are final states

- **Local states potentially concurrently**
  - Concurrency set $C(s)$ contains all states potentially concurrent with $s$

- **Sender set** $S(s) = \{\text{local states } t \mid t \text{ sends } m \text{ and } s \text{ receives } m\}$
Reachable Global State Graph

- $C(w_1) = \{q_2, p_2, a_2\}$
- $S(w_1) = q_2$
- Leaf nodes are terminal states
  - All contain only final states
- No nodes have both “abort” and “commit”
  - Protocol consistent
- Therefore 2PC is operationally correct
Failure Model

■ Site failures
  - Assumed when expected message not received in time
    ▪ “Timeout message” received
  - Modeled as “failure transition” for the failed site
    ▪ Assumption: a site knows it has failed
  - Failures are “atomic”
    ▪ All messages sent out at a given local state are sent out or none
    ▪ Allow multiple failure transitions from a state

■ Independent recovery
  - Failed processes recover (crash-recovery)
  - Transition directly to final state without communication
    ▪ Local state is sufficient
Lemma 1: If a protocol contains a local state $s$ with both abort and commit in $C(s)$, it cannot independently recover from arbitrary single failure
- $C(s)$ has both abort and commit
- $s$ cannot fail-over to abort or commit; either might contradict the other state(s)

Rule 1: If prerequisite of Lemma 1 does not hold, and $C(s)$ has commit, insert failure transition from $s$ to commit, else insert failure transition from $s$ to abort

2PC does not satisfy prerequisite
- 2PC: $C(p_2)$ has both commit and abort! If crash of slave in $p_2$, slave can not independently recover
- Need to add an acknowledgement from coordinator to slave
Modified 2PC for Rule 1

Coordinator

\( q_1 \)  
- xact request/start xact  
- no/  
- yes/commit  
\( w_1 \)  
- yes/commit  
\( a_1 \)  
- yes/abort  
\( p_1 \)  
- ack/  
\( c_1 \)  

Participant

\( q_2 \)  
- start xact/yes  
\( p_2 \)  
- abort/  
\( a_2 \)  
\( c_2 \)  
- commit/ack  

failure transition
Rule 2: For each intermediate state \( s \)
- if \( t \) is in \( S(s) \) and \( t \) had a failure transition to a commit (abort) state, then
  - assign a timeout transition from \( s \) to a commit (abort) state

Assumption: Failed state \( t \) will independently recover
- Rule 1 forces transition to commit / abort
- Rule 2 forces “live” transaction to do same
  - \( s \) depends on \( t \) and must go on with a consistent outcome
Modified 2PC with Rules 1,2

Coordinator

q₁

w₁

a₁

p₁

q₁

start xact/
start xact

w₁

no/

yes/
commit

p₁

a₁

yes/
abort

p₁

commit/
ack

c₁

Coordinator

Participant

q₂

start xact/
yes

p₂

start xact/
no

q₂

a₂

c₂

timeout

transition
Theorem: Rules 1 and 2 are sufficient for designing protocols resilient to a single site failure

Proof: Let \( P \) be protocol s.t. there is no \( s \) where \( C(s) \) contains commit and abort

- \( P' \) is \( P \) modified by rules 1 and 2
- Assume by contradiction a global inconsistent state \( f_1 f_2 \)
- Site 1 fails in state \( s_1 \), takes failure transition to \( f_1 \) when site 2 in \( s_2 \)

Case 1: Site \( s_2 \) in final state \( f_2 \)

- Implies \( f_2 \) in \( C(s_1) \) – violates rule 1 (\( f_1 \) same decision as \( f_2 \))

Case 2: Site \( s_2 \) in nonfinal state, timeout trans. to \( f_2 \)

- Implies \( s_1 \in S(s_2) \) – violates rule 2 (\( f_1 \) same decision as \( f_2 \))
Theorem: No protocol using independent recovery is resilient to arbitrary two site failures

- If failures concurrent (both sites fail without knowing other has failed)

Proof: Assume path in global state graph $G_0, \ldots, G_m$, all sites recover to abort from $G_0$, to commit from $G_m$

Let $G_k$ be first state where first site $j$ recovers to commit

- $j$ recovers to abort in $G_{k-1}$
- $j$ was only site to transition between $G_k$ and $G_{k-1}$
- All other sites will recover same in $G_k$ and $G_{k-1}$
- So either $G_k$ or $G_{k-1}$ inconsistent if $j$ and another site fail

In case of multiple failures process can not tell locally whether independent recovery is possible

- Must pessimistically always wait on/ask coordinator
Uses (uniform) consensus
- Akin to consensus, processes propose and decide
- Binary: 0 (abort), 1 (commit)

Uses perfect failure detector $P$

Idea
- $P$ used to make sure all available input values are collected
- A single 0 input requires the outcome to be 0
  - None should be missed
  - We can not wait forever though
Proposition

\[ \text{proposal} \leftarrow 1 \]
\[ \text{voted} \leftarrow \emptyset \]

\text{to execute} \ \text{propose}_{\text{NBAC}}(v) \ \text{by} \ p_i:\n
\[ \text{proposal} \leftarrow v \]
\[ \text{send}(v) \ \text{to all processes} \]

\text{upon receive}(v) \ \text{from} \ p_i \ \text{do:}

\[ \text{proposal} \leftarrow \text{proposal} \ast v \]
\[ \text{voted} \leftarrow \text{voted} \cup \{p_i\} \]

\text{upon} \ (\text{voted} = \Pi/D_p) \ \text{do}

\[ \text{if} \ (\Pi/D_p = \Pi) \ \text{then} \ \text{propose}_C(v) \]
\[ \text{else} \ \text{propose}_C(0) \]
\[ \text{proposed} \leftarrow \text{true} \]

\text{decide}_{\text{NBAC}}(v) \ \text{occurs as follows:}

\text{upon decide}_C(v)

\[ \text{decide}_{\text{NBAC}}(v) \]
Evaluation

▶ Elegant solution
  – But uses $P$
  – Is $P$ really required?

▶ Think of following optimization
  – Propose 0 immediately to the consensus whenever a failure is noticed

▶ Only need to know if there is some process that failed, not which one: ?$P$ [Guerraoui’02]
  – Consensus requires $\diamond S$
  – $\diamond S$ vs ?$P$
References