CS 138: Distributed Transactions
Transactions

• “ACID” property:
  – atomic
    - all or nothing
  – consistent
    - take system from one consistent state to another
  – isolated
    - have no effect on other transactions until committed
  – durable
    - persists
Operations of the **Account** interface

- `deposit(amount)`
  - deposit amount in the account
- `withdraw(amount)`
  - withdraw amount from the account
- `getBalance() -> amount`
  - return the balance of the account
- `setBalance(amount)`
  - set the balance of the account to amount

---

Operations of the **Branch** interface

- `create(name) -> account`
  - create a new account with a given name
- `lookUp(name) -> account`
  - return a reference to the account with the given name
- `branchTotal() -> amount`
  - return the total of all the balances at the branch

---
A client’s banking transaction

Transaction \( T \):
\[ a.\text{withdraw}(100); \]
\[ b.\text{deposit}(100); \]
\[ c.\text{withdraw}(200); \]
\[ b.\text{deposit}(200); \]
Operations in *Coordinator* interface

\[ \text{openTransaction()} \rightarrow \text{trans}; \]
starts a new transaction and delivers a unique TID \( \text{trans} \). This identifier will be used in the other operations in the transaction.

\[ \text{closeTransaction}(\text{trans}) \rightarrow (\text{commit}, \text{abort}); \]
ends a transaction: a \textit{commit} return value indicates that the transaction has committed; an \textit{abort} return value indicates that it has aborted.

\[ \text{abortTransaction}(\text{trans}); \]
aborts the transaction.
## Transaction Life Histories

<table>
<thead>
<tr>
<th>Successful</th>
<th>Aborted by client</th>
<th>Aborted by server</th>
</tr>
</thead>
<tbody>
<tr>
<td>openTransaction</td>
<td>openTransaction</td>
<td>openTransaction</td>
</tr>
<tr>
<td>operation</td>
<td>operation</td>
<td>operation</td>
</tr>
<tr>
<td>operation</td>
<td>operation</td>
<td>operation</td>
</tr>
<tr>
<td>operation</td>
<td>operation</td>
<td>server aborts transaction</td>
</tr>
<tr>
<td>closeTransaction</td>
<td>abortTransaction</td>
<td>operation ERROR reported to client</td>
</tr>
</tbody>
</table>
# The lost update problem

Initial balances: a: $100, b: $200, c: $300

<table>
<thead>
<tr>
<th>Transaction T:</th>
<th>Transaction U:</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>balance = b.getBalance();</code></td>
<td><code>balance = b.getBalance();</code></td>
</tr>
<tr>
<td><code>b.setBalance(balance*1.1);</code></td>
<td><code>b.setBalance(balance*1.1);</code></td>
</tr>
<tr>
<td><code>a.withdraw(balance/10)</code></td>
<td><code>c.withdraw(balance/10)</code></td>
</tr>
</tbody>
</table>

```
balance = b.getBalance();    $200
b.setBalance(balance*1.1);   $220
a.withdraw(balance/10)       $80
```

```
balance = b.getBalance();    $200
b.setBalance(balance*1.1);   $220
c.withdraw(balance/10)       $280
```
The inconsistent retrievals problem

<table>
<thead>
<tr>
<th>Transaction V:</th>
<th>Transaction W:</th>
</tr>
</thead>
<tbody>
<tr>
<td>a. withdraw(100)</td>
<td>aBranch.branchTotal()</td>
</tr>
<tr>
<td>b. deposit(100)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>a. withdraw(100);</th>
<th>$100</th>
</tr>
</thead>
<tbody>
<tr>
<td>total = a.getBalance()</td>
<td>$100</td>
</tr>
<tr>
<td>total = total+b.getBalance()</td>
<td>$300</td>
</tr>
<tr>
<td>total = total+c.getBalance()</td>
<td></td>
</tr>
<tr>
<td>b. deposit(100)</td>
<td>$300</td>
</tr>
<tr>
<td>:</td>
<td>:</td>
</tr>
</tbody>
</table>
Serial Equivalence

• Consider the effect of a concurrent execution of transactions A and B

\[ A \parallel B \]

• What should our correctness criteria be?
• Intuitively, it should be equivalent to some serial execution:

\[ A;B \]
\[ B;A \]

• We say that \( A \parallel B \) is *serially equivalent* if it has the same effect as either \( A;B \) or \( B;A \)
A serially equivalent interleaving of $T$ and $U$

<table>
<thead>
<tr>
<th>Transaction $T$:</th>
<th>Transaction $U$:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$balance = b.getBalance()$</td>
<td>$balance = b.getBalance()$</td>
</tr>
<tr>
<td>$b.setBalance(balance*1.1)$</td>
<td>$b.setBalance(balance*1.1)$</td>
</tr>
<tr>
<td>$a.withdraw(balance/10)$</td>
<td>$c.withdraw(balance/10)$</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
balance &= b.getBalance() \quad $200 \\
balance &= b.getBalance() \quad $220 \\
a.withdraw(balance/10) \quad $80 \\
\end{align*}
\]

\[
\begin{align*}
balance &= b.getBalance() \quad $220 \\
b.setBalance(balance*1.1) \quad $242 \\
c.withdraw(balance/10) \quad $278 \\
\end{align*}
\]
A serially equivalent interleaving of $V$ and $W$

<table>
<thead>
<tr>
<th>Transaction $V$:</th>
<th>Transaction $W$:</th>
</tr>
</thead>
</table>
| $a.\text{withdraw}(100);$  
$b.\text{deposit}(100)$ | $a\text{Branch.branchTotal}()$ |
| $\text{a.withdraw}(100);$  
$\text{b.deposit}(100)$ | |
| $\text{total} = \text{a.getBalance}()$ | $\text{total} = \text{a.getBalance}()$ |
| $\text{total} = \text{total} + \text{b.getBalance}()$ | $\text{total} = \text{total} + \text{b.getBalance}()$ |
| $\text{total} = \text{total} + \text{c.getBalance}()$ | $\text{total} = \text{total} + \text{c.getBalance}()$ |
| ... | ... |
Read and *write* operation conflict rules

<table>
<thead>
<tr>
<th>Operations of different transactions</th>
<th>Conflict</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>read       read</td>
<td>No</td>
<td>Because the effect of a pair of read operations does not depend on the order in which they are executed</td>
</tr>
<tr>
<td>read       write</td>
<td>Yes</td>
<td>Because the effect of a <em>read</em> and a <em>write</em> operation depends on the order of their execution</td>
</tr>
<tr>
<td>write      write</td>
<td>Yes</td>
<td>Because the effect of a pair of <em>write</em> operations depends on the order of their execution</td>
</tr>
</tbody>
</table>
### Serially Equivalent?

<table>
<thead>
<tr>
<th>Transaction $R$:</th>
<th>Transaction $S$:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x = \text{read}(i)$</td>
<td>$y = \text{read}(j)$</td>
</tr>
<tr>
<td>$\text{write}(i, 10)$</td>
<td>$\text{write}(j, 30)$</td>
</tr>
<tr>
<td>$\text{write}(j, 20)$</td>
<td>$z = \text{read}(i)$</td>
</tr>
</tbody>
</table>
A dirty read when transaction $T$ aborts

<table>
<thead>
<tr>
<th>Transaction $T$:</th>
<th>Transaction $U$:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$.getBalance()</td>
<td>$a$.getBalance()</td>
</tr>
<tr>
<td>$a$.setBalance(balance + 10)</td>
<td>$a$.setBalance(balance + 20)</td>
</tr>
</tbody>
</table>

$balance = a$.getBalance()  $100$
$a$.setBalance(balance + 10)  $110$

$balance = a$.getBalance()  $110$

$a$.setBalance(balance + 20)  $130$

commit transaction

abort transaction
Overwriting uncommitted values

<table>
<thead>
<tr>
<th>Transaction $T$:</th>
<th>Transaction $U$:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a.setBalance(105)$</td>
<td>$a.setBalance(110)$</td>
</tr>
<tr>
<td>$a.setBalance(105)$</td>
<td>$a.setBalance(110)$</td>
</tr>
<tr>
<td>$100$</td>
<td>$110$</td>
</tr>
<tr>
<td>$105$</td>
<td></td>
</tr>
</tbody>
</table>

commit transaction

abort transaction
## Transactions \( T \) and \( U \) with exclusive locks

<table>
<thead>
<tr>
<th>Transaction ( T ):</th>
<th>Transaction ( U ):</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{balance} = \text{b.getBalance()} )</td>
<td>( \text{balance} = \text{b.getBalance()} )</td>
</tr>
<tr>
<td>( \text{b.setBalance(bal*1.1)} )</td>
<td>( \text{b.setBalance(bal*1.1)} )</td>
</tr>
<tr>
<td>( \text{a.withdraw(bal/10)} )</td>
<td>( \text{c.withdraw(bal/10)} )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operations</th>
<th>Locks</th>
<th>Operations</th>
<th>Locks</th>
</tr>
</thead>
<tbody>
<tr>
<td>openTransaction</td>
<td>( \text{lock B} )</td>
<td>openTransaction</td>
<td>waits for ( T )'s lock on B</td>
</tr>
<tr>
<td>( \text{bal} = \text{b.getBalance()} )</td>
<td>( \text{lock A} )</td>
<td>( \text{bal} = \text{b.getBalance()} )</td>
<td>( \text{lock B} )</td>
</tr>
<tr>
<td>( \text{b.setBalance(bal*1.1)} )</td>
<td>unlock A, B</td>
<td>( \text{b.setBalance(bal*1.1)} )</td>
<td>lock C</td>
</tr>
<tr>
<td>( \text{a.withdraw(bal/10)} )</td>
<td></td>
<td>( \text{c.withdraw(bal/10)} )</td>
<td></td>
</tr>
<tr>
<td>closeTransaction</td>
<td>unlock A, B</td>
<td>closeTransaction</td>
<td>unlock B, C</td>
</tr>
</tbody>
</table>
Two-Phase Locking

1) Acquire locks
2) Release locks

• No more locks may be acquired after any lock is released
• Strict two-phase locking
  – no locks released until transaction commits
Two-Phase-Locking Intuition

Transaction A
lock A_1
lock A_2
lock A_3
lock A_4
lock A_5
unlock all

Transaction B
lock B_1
lock B_2
lock B_3
lock B_4
lock B_5
lock B_6
lock B_7
lock B_8
lock B_9
unlock all
Transaction Steps

• Accumulate changes
  – store as “tentative versions”
• Make sure everything is ok
• Commit or abort
  – move tentative versions to actual
  or
  – delete tentative versions
Nested transactions

T : top-level transaction

T₁ = openSubTransaction

T₂ = openSubTransaction

T₁₁ : prov. commit

T₁₂ : prov. commit

T₂ : commit

T₂₁ : abort

T₂₁₁ : prov. commit
Distributed Transactions

Begin Transaction;
  a.withdraw(100);
  b.deposit(50);
  c.deposit(50);
End Transaction;

withdraw(100);

deposit(50);

client

a

b

c
Coordination

Client

Begin Transaction;
  a.withdraw(100);
  b.deposit(50);
  c.deposit(50);
End Transaction;

Coordinator

withdraw(100);

a

deposit(50);

b

deposit(50);

c
Atomic Commit

- **AC1**: All participants that reach a (commit/abort) decision reach the same one.
- **AC2**: A participant cannot reverse its decision.
- **AC3**: The commit decision can be reached only if all participants agree.
- **AC4**: If there are no failures and all participants vote yes, then decision will be commit.
- **AC5**: For any execution, if all failures are repaired and no new failures occur for a sufficiently long interval, then all participants will reach a (commit/abort) decision.
Two-Phase Commit

• Phase 1
  – coordinator prepares to commit:
    - asks participants to vote either “commit” or “abort”
    – participants respond appropriately
• Phase 2
  – coordinator decides outcome:
    - if all participants vote commit, outcome is commit, otherwise outcome is abort
    - outcome sent to all participants
  – participants do what they’re told
State Diagrams

Coordinator

Init → Wait
  - app commit/vote req

Wait → Abort
  - any abort/abort

Wait → Commit
  - all commit/commit

Participant

Init → Uncertain
  - vote req/commit

Uncertain → Abort
  - abort/ack

Uncertain → Commit
  - commit/ack

Abort/commit → Commit

Abort/abort → Abort
Failures

- Coordinator or participants could crash
  - assume “fail-stop”
    - crash detected by time-out
    - no byzantine failures
  - crashed machines restart
    - recover their state
Crash Points

Coordinator

Init

app commit/vote req

Wait

Abort

Commit

Commit/Commit

Abort Abort/Abort

Participant

Init

vote req/commit

Init

vote req/abort

Uncertain

Abort

Commit

Commit/Ack

Abort/Ack
Dealing with Timeouts (1)

• Coordinator times out in *Wait* state
  – waiting for a participant to vote
  – takes no response to mean “abort”
  – sends abort to all other participants

• Participant times out in *Uncertain* state
  – waiting for coordinator to say “commit” or “abort”
  – can’t assume either outcome
  – waits for coordinator to restart
  – contacts coordinator for final outcome
Dealing with Timeouts (2)

- Coordinator could take long time to restart
- Participants contact other participants
  - p contacts q (p is in *Uncertain* state)
  - q is in:
    - *commit* (or *abort*) state
      - p goes to *commit* (or *abort*) state
    - *init* state (hasn’t yet voted)
      - both q and p abort
    - *uncertain* state
      - both p and q remain *uncertain*
Improving on Two-Phase Commit

• It works fine in practice!
• But …
  – all participants could conceivably be in uncertain state and coordinator is down (for a long time)
• Can we make it so such blocking can’t happen?
What Causes Blocking?

• Coordinator is down
• If all operational (not-failed) participants are in uncertain state, they are blocked
• If all participants are operational, they can elect new coordinator
• If any participant has crashed, the others don’t know if it crashed before or after voting (to commit or abort)
Guaranteeing Non-Blocking

- Non-blocking property (NB):
  - if any operational process is in the Uncertain state, then no process (operational or failed) can have decided to commit

- If NB holds, then operational processes may elect new coordinator and decide to commit or abort
Failures

- Coordinator or participants could crash
  - *no communication failures*
  - assume “fail-stop”
    - crash detected by time-out
    - no byzantine failures
  - crashed machines restart
    - recover their state
Three-Phase Commit

- **Phase 1**
  - coordinator prepares to commit:
    - asks participants to vote either "commit" or "abort"
    - participants respond appropriately
- **Phase 2**
  - coordinator counts votes:
    - if all participants vote commit, outcome is *pre-commit*, otherwise outcome is abort
    - outcome sent to all participants
    - participants ack and either abort or wait for commit
- **Phase 3**
  - coordinator waits for all acks
    - if committing, sends final commit to all participants
    - participants commit
Revised State Diagrams

- Init
  - app commit/vote req
    - Wait
      - any abort/abort
        - Abort
      - all commit/precommit
        - Pre Commit
          - abort/ack
            - Abort
              - precommit/ack
                - Uncertain
                  - vote req/commit
                    - Pre Commit
                      - commit/commit
            - all ack/commit
              - Commit
                  - commit/commit
                    - Commit

Timeouts (1)

- **Init**
  - app commit/vote req
- **Wait**
  - any abort/abort
  - all commit/precommit
  - Abort
  - Pre Commit
  - Commit
- **Commit**
  - all ack/commit

- **Abort**
  - vote req/abort
  - Uncertain
  - abort/ack
  - precommit/ack
  - Abort
  - Pre Commit
  - Commit
  - commit/commit
Timeouts (2)

- **Init**
  - app commit/vote req
  - any abort/abort

- **Wait**
  - abort/ack
  - all commit/precommit
  - abort

- **Pre Commit**
  - all ack/commit
  - commit

- **Commit**
  - abd commit/precommit

- **Abort**

- **Uncertain**
  - vote req/commit
  - abort/ack
  - precommit/ack

- **Pre Commit**
  - commit/commit

- **Commit**

- **Abort**
Timeouts (3)

- **Init**: app commit/vote req
- **Wait**: any abort/abort, all commit/precommit
- **Abort**: commit
- **Commit**: vote req/abort, vote req/commit
- **Uncertain**: abort/ack, precommit/ack
- **Pre Commit**: all commit/precommit
- **Commit**: all ack/commit, commit/commit
Timeouts (4)

- Init
  - app commit/vote req
    - Wait
      - any abort/abort
        - Abort
      - all commit/precommit
        - Pre Commit
          - all ack/commit
            - Commit
    - vote req/abort
      - Uncertain
        - abort/ack
          - Abort
            - commit/commit
              - Commit
        - precommit/ack
          - Pre Commit
            - commit/commit
              - Commit
Timeouts (5)

- **Init**
  - app commit/vote req
  - any abort/abort
  - all commit/precommit
  - Abort
  - Pre Commit
  - Commit

- **Wait**
  - all ack/commit
  - Pre Commit
  - Commit

- **Abort**
  - vote req/abort
  - precommit/ack
  - Pre Commit
  - Commit

- **Uncertain**
  - abort/ack
  - commit/commit
Details (1)

• If original coordinator remains operational
  – participant crashes handled as in two-phase commit

• If participant times out in *Uncertain* or *PreCommit* states
  – it starts an election for a new coordinator
Details (2)

- When newly elected coordinator starts up
  - sends state-request message to all operational participants
  - coordinator collects states and proceeds according to four termination rules (*termination protocol*):
    - TR1: if any participant is in *Abort* state, all are sent abort messages
    - TR2: if some participant is in *Commit* state, all are sent commit messages
    - TR3: if all participants are in *Uncertain* state, all are sent abort messages
    - TR4: if some participant is in *PreCommit* state, but none in *Commit* state, those in *Uncertain* state are sent PreCommit messages; once these are acked, all participants are sent commit messages
Details (3)

• When failed participant comes back up
  – if it failed in *Init* state
    - it aborts
  – Otherwise it asks other participants for outcome
    - will eventually get either commit or abort
      • (could get abort even if it was in the *PreCommit* state when it crashed)
Correctness (1)

- **Lemma 1:** After a new coordinator starts up, exactly one of TR1 – TR4 will hold.

- **Theorem 1:** In the absence of total failures, participants will never block:
  - they clearly won’t block if the coordinator never fails;
  - if the coordinator fails, a new one is elected;
  - one of TR1-TR4 will hold and a decision will be reached;
  - if the new coordinator fails, a new one is elected; if it fails another is elected, etc. until there are no more participants.
Correctness (2)

• Lemma 2: All participants that reach a decision on the same invocation of the termination protocol reach the same one

• Lemma 3: If NB holds before the termination protocol starts, it holds through the execution of the protocol

• Theorem 2: All operational participants reach the same decision
  – proof by induction on the invocations of the termination protocol
Total Failure

• What if coordinator and all participants fail?
• When they come back up, how do they decide?
  – if resurrected participant either didn’t vote or voted abort, it may unilaterally abort
  – otherwise, must run termination protocol
  – but works only if last participant to fail has come back up
Communication Failures

• Network could partition into multiple pieces
• Not sufficient to get agreement in a piece containing a quorum
  – consensus is required for commit!
• Scenario
  – all participants vote
  – coordinator collects results
  – network partitions before or after all results collected
  – if network reconnects: easy
  – network never fully reconnects, but each participant eventually can communicate (perhaps briefly) with all others