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 \textit{Account} interface

\begin{itemize}
  \item \texttt{deposit(amount)}
    \begin{itemize}
      \item deposit amount in the account
    \end{itemize}
  \item \texttt{withdraw(amount)}
    \begin{itemize}
      \item withdraw amount from the account
    \end{itemize}
  \item \texttt{getBalance()} -> \texttt{amount}
    \begin{itemize}
      \item return the balance of the account
    \end{itemize}
  \item \texttt{setBalance(amount)}
    \begin{itemize}
      \item set the balance of the account to amount
    \end{itemize}
\end{itemize}

Operations of the \textit{Branch} interface

\begin{itemize}
  \item \texttt{create(name)} -> \texttt{account}
    \begin{itemize}
      \item create a new account with a given name
    \end{itemize}
  \item \texttt{lookUp(name)} -> \texttt{account}
    \begin{itemize}
      \item return a reference to the account with the given name
    \end{itemize}
  \item \texttt{branchTotal()} -> \texttt{amount}
    \begin{itemize}
      \item return the total of all the balances at the branch
    \end{itemize}
\end{itemize}
A client’s banking transaction

Transaction T:
  a. withdraw(100);
  b. deposit(100);
  c. withdraw(200);
  b. deposit(200);
Operations in *Coordinator* interface

`openTransaction()` -> `trans`;
starts a new transaction and delivers a unique TID `trans`. This identifier will be used in the other operations in the transaction.

`closeTransaction(trans)` -> `(commit, abort)`;
ends a transaction: a `commit` return value indicates that the transaction has committed; an `abort` return value indicates that it has aborted.

`abortTransaction(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>server aborts</td>
</tr>
<tr>
<td>operation</td>
<td>operation</td>
<td>transaction</td>
</tr>
<tr>
<td>closeTransaction</td>
<td>abortTransaction</td>
<td>operation ERROR</td>
</tr>
<tr>
<td></td>
<td></td>
<td>reported to client</td>
</tr>
</tbody>
</table>

*Remarks:* The life history of a transaction begins with an `openTransaction` and ends with a `closeTransaction`. If the transaction is aborted by the client or the server, the life history is interrupted accordingly. In the event of an operation error, an error message is reported to the client.
### 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>$balance = b.getBalance();$</td>
<td>$balance = b.getBalance();$</td>
</tr>
<tr>
<td>$b.setBalance(balance * 1.1);$</td>
<td>$b.setBalance(b.balance * 1.1);$</td>
</tr>
<tr>
<td>$a.withdraw(balance / 10)$</td>
<td>$c.withdraw(balance / 10)$</td>
</tr>
</tbody>
</table>

$balance = b.getBalance();$ $200$

$balance = b.getBalance();$ $200$

$b.setBalance(balance * 1.1);$ $220$

$b.setBalance(b.balance * 1.1);$ $220$

$a.withdraw(balance / 10)$ $80$

$c.withdraw(balance / 10)$ $280$
## The inconsistent retrievals problem

<table>
<thead>
<tr>
<th>Transaction V:</th>
<th>Transaction W:</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>a</em>.withdraw(100)</td>
<td><em>aBranch</em>.branchTotal()</td>
</tr>
<tr>
<td><em>b</em>.deposit(100)</td>
<td></td>
</tr>
<tr>
<td><strong>a.withdraw(100)</strong>;</td>
<td></td>
</tr>
<tr>
<td><strong>$100</strong></td>
<td></td>
</tr>
<tr>
<td><strong>b.deposit(100)</strong></td>
<td></td>
</tr>
<tr>
<td><strong>$300</strong></td>
<td></td>
</tr>
</tbody>
</table>

```
*a*.withdraw(100);       $100

total = *a*.getBalance() $100

total = total + *b*.getBalance() $300

total = total + *c*.getBalance() $300
```

*: Comment or note mark.
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>

| $balance = b.getBalance()$ $200$                                                                       | $balance = b.getBalance()$ $220$                                                                        |
| $b.setBalance(balance*1.1)$ $220$                                                                      | $b.setBalance(balance*1.1)$ $242$                                                                      |
| $a.withdraw(balance/10)$ $80$                                                                           | $c.withdraw(balance/10)$ $278$                                                                           |
A serially equivalent interleaving of V and W

<table>
<thead>
<tr>
<th>Transaction V:</th>
<th>Transaction W:</th>
</tr>
</thead>
<tbody>
<tr>
<td>a.withdraw(100); b.deposit(100)</td>
<td>aBranch.branchTotal()</td>
</tr>
<tr>
<td>a.withdraw(100)</td>
<td>$100</td>
</tr>
<tr>
<td>b.deposit(100)</td>
<td>$300</td>
</tr>
<tr>
<td>total = a.getBalance()</td>
<td>$100</td>
</tr>
<tr>
<td>total = total+b.getBalance()</td>
<td>$400</td>
</tr>
<tr>
<td>total = total+c.getBalance()</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
**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 read and a write 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 write 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>

**Rule:** Two transaction executions are serially equivalent iff all conflicting operations of the two are executed in the same order.
Concurrency Control

- How to ensure that transaction executions are serializable:
  - locking
  - optimism
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>$100</td>
<td>$110</td>
</tr>
<tr>
<td>a.setBalance(105)</td>
<td>$105</td>
</tr>
<tr>
<td>abort transaction</td>
<td></td>
</tr>
<tr>
<td>commit transaction</td>
<td></td>
</tr>
</tbody>
</table>
## Transactions $T$ and $U$ with exclusive locks

<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>$b.setBalance(bal*1.1)$</td>
<td>$b.setBalance(bal*1.1)$</td>
</tr>
<tr>
<td>$a.withdraw(bal/10)$</td>
<td>$c.withdraw(bal/10)$</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></td>
<td>openTransaction</td>
<td></td>
</tr>
<tr>
<td>$bal = b.getBalance()$</td>
<td>lock B</td>
<td>$bal = b.getBalance()$</td>
<td>waits for $T$’s lock on B</td>
</tr>
<tr>
<td>$b.setBalance(bal*1.1)$</td>
<td></td>
<td>$b.setBalance(bal*1.1)$</td>
<td></td>
</tr>
<tr>
<td>$a.withdraw(bal/10)$</td>
<td>lock A</td>
<td>$c.withdraw(bal/10)$</td>
<td>lock C</td>
</tr>
<tr>
<td>closeTransaction</td>
<td>unlock A,B</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| closeTransaction   | unlock B,C          |                     |                     |

| lock B             |                     |                     |                     |
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

<table>
<thead>
<tr>
<th>Transaction A</th>
<th>Transaction B</th>
</tr>
</thead>
<tbody>
<tr>
<td>lock $A_1$</td>
<td>lock $B_1$</td>
</tr>
<tr>
<td>lock $A_2$</td>
<td>lock $B_2$</td>
</tr>
<tr>
<td>lock $A_3$</td>
<td>lock $B_3$</td>
</tr>
<tr>
<td>lock $A_4$</td>
<td>lock $B_4$</td>
</tr>
<tr>
<td>lock $A_5$</td>
<td>lock $B_5$</td>
</tr>
<tr>
<td>unlock all</td>
<td>lock $B_6$</td>
</tr>
<tr>
<td></td>
<td>lock $B_7$</td>
</tr>
<tr>
<td></td>
<td>lock $B_8$</td>
</tr>
<tr>
<td></td>
<td>lock $B_9$</td>
</tr>
<tr>
<td></td>
<td>unlock all</td>
</tr>
</tbody>
</table>
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

T₁ : openSubTransaction
T₂ : openSubTransaction

T₁₁ : prov. commit
T₁₂ : prov. commit

T₂₁ : prov. commit

T₂₂ : prov. commit

prov. commit

prov. commit

commit

abort
Distributed Transactions

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

- **a**
  - withdraw(100);

- **b**
  - deposit(50);

- **c**
  - deposit(50);
Coordination

Begin Transaction;

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

withdraw(100);

deposit(50);

deposit(50);
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
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

Wait

Abort

Commit

Commit/commit

Abort/abort

App commit/vote req

Coordinator

Participant

Init

Uncertain

Abort

Commit

Commit/ack

Abort/ack

Vote req/commit

Vote req/abort
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 -> Wait
  - app commit/vote req
  - any abort/abort

- Wait -> Abort
- Wait -> Pre Commit
  - all commit/precommit

- Abort -> Commit
  - all ack/commit

- Pre Commit -> Commit
- Pre Commit -> Abort
  - abort/ack

- Init -> Uncertain
  - vote req/commit

- Uncertain -> Abort
  - precommit/ack

- Abort -> Commit
  - commit/commit

- Uncertain -> Pre Commit
Timeouts (1)

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

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

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

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

- **Commit**
  - commit/commit
  - precommit/commit
  - Commit
Timeouts (2)

```
Init
  ↓
Abort
     ↓
Wait
          ↓
Abort
          ↓
Pre Commit
          ↓
Commit
Vote req/abort
     ↓
any abort/abort
     ↓
Commit/precommit
     ↓
any abort/abort
Commit

Init
  ↓
Vote req/commit
  ↓
abort/ack
  ↓
Uncertain
          ↓
Abort
          ↓
Pre Commit
          ↓
Commit
```

Copyright © 2016 Thomas W. Doeppner. All rights reserved.
Timeouts (3)

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

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

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

- **Commit**
  - commit/commit
  - all commit/precommit
  - all ack/commit

- **Pre Commit**
  - vote req/commit
  - commit/commit

- **Uncertain**
  - abort/ack

- **Pre Commit**
  - precommit/ack

- **Commit**
  - vote req/commit
  - commit/commit
Timeouts (4)

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

- Init
  - vote req/commit
    - Uncertain
      - abort/ack
      - precommit/ack
      - ??
    - precommit/commit
      - Pre Commit
      - Commit
Timeouts (5)

Init → Wait → Abort/Commit

Abort/Commit → PreCommit → Commit

Wait → app commit/vote req

Abort/Commit → any abort/abort

PreCommit → all commit/precommit

Commit → all ack/commit

Abort/Commit → Uncertain → abort/ack

Uncertain → precommit/ack

Commit → commit/commit

Init → vote req/abort

PreCommit → vote req/commit

Abort → ??

Commit → ??

Copyright © 2016 Thomas W. Doeppner. All rights reserved.
Details (1)

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

• If participant times out in Uncertain or PreCommit states
  – if any other participant has aborted, it aborts (it must have been in Uncertain state)
  – otherwise, it starts an election for a new coordinator
• 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 \textit{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 \textit{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