Transactions

Distributed Software Systems

- **Motivation**
  - Provide atomic operations at servers that maintain shared data for clients
  - Provide recoverability from server crashes
- **Properties**
  - Atomicity, Consistency, Isolation, Durability (ACID)
- **Concepts:** commit, abort
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.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 transaction</td>
</tr>
<tr>
<td>closeTransaction</td>
<td>abortTransaction</td>
<td>operation ERROR</td>
</tr>
<tr>
<td>operation</td>
<td></td>
<td>reported to client</td>
</tr>
</tbody>
</table>
Concurrency control

- Motivation: without concurrency control, we have lost updates, inconsistent retrievals, dirty reads, etc. (see following slides)
- Concurrency control schemes are designed to allow two or more transactions to be executed correctly while maintaining serial equivalence
  - Serial Equivalence is correctness criterion
    - Schedule produced by concurrency control scheme should be equivalent to a serial schedule in which transactions are executed one after the other
- Schemes: locking, optimistic concurrency control, time-stamp based concurrency control

The lost update problem

<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>
<tr>
<td>$200</td>
<td>$200</td>
</tr>
<tr>
<td>balance = b.getBalance();</td>
<td>b.setBalance(balance*1.1);</td>
</tr>
<tr>
<td>b.setBalance(balance*1.1);</td>
<td>$220</td>
</tr>
<tr>
<td>a.withdraw(balance/10)</td>
<td>c.withdraw(balance/10)</td>
</tr>
<tr>
<td>$80</td>
<td>$280</td>
</tr>
</tbody>
</table>
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>
<tr>
<td>a.withdraw(100);</td>
<td>total = a.getBalance()</td>
</tr>
<tr>
<td></td>
<td>total = total+b.getBalance()</td>
</tr>
<tr>
<td></td>
<td>total = total+c.getBalance()</td>
</tr>
<tr>
<td>b.deposit(100)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>balance = b.getBalance()</td>
<td>$200</td>
</tr>
<tr>
<td>b.setBalance(balance*1.1)</td>
<td>$220</td>
</tr>
<tr>
<td>a.withdraw(balance/10)</td>
<td>$80</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>balance = b.getBalance()</td>
</tr>
<tr>
<td></td>
<td>$220</td>
</tr>
<tr>
<td></td>
<td>b.setBalance(balance*1.1)</td>
</tr>
<tr>
<td></td>
<td>$242</td>
</tr>
<tr>
<td></td>
<td>c.withdraw(balance/10)</td>
</tr>
<tr>
<td></td>
<td>$278</td>
</tr>
</tbody>
</table>
### 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);</td>
<td>aBranch.branchTotal()</td>
</tr>
<tr>
<td>b.deposit(100)</td>
<td></td>
</tr>
<tr>
<td>a.withdraw(100);</td>
<td>total = a.getBalance()</td>
</tr>
<tr>
<td>b.deposit(100)</td>
<td>total = total+b.getBalance()</td>
</tr>
<tr>
<td>$100</td>
<td>total = total+c.getBalance()</td>
</tr>
<tr>
<td>$300</td>
<td>...</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>
<tr>
<td>balance = a.getBalance()</td>
<td>balance = a.getBalance()</td>
</tr>
<tr>
<td>$100</td>
<td>$110</td>
</tr>
<tr>
<td>a.setBalance(balance + 10)</td>
<td>a.setBalance(balance + 20)</td>
</tr>
<tr>
<td>$110</td>
<td>$130</td>
</tr>
<tr>
<td>abort transaction</td>
<td>commit transaction</td>
</tr>
</tbody>
</table>
**Serializability**

\[
\begin{align*}
&\text{BEGIN\_TRANSACTION} \quad x = 0; \\
&\quad x = x + 1; \\
&\quad \text{END\_TRANSACTION} \\
&\text{BEGIN\_TRANSACTION} \quad x = 0; \\
&\quad x = x + 2; \\
&\quad \text{END\_TRANSACTION} \\
&\text{BEGIN\_TRANSACTION} \quad x = 0; \\
&\quad x = x + 3; \\
&\quad \text{END\_TRANSACTION}
\end{align*}
\]

(a) \quad (b) \quad (c)

<table>
<thead>
<tr>
<th>Schedule</th>
<th>Transactions</th>
<th>legality</th>
</tr>
</thead>
<tbody>
<tr>
<td>Schedule 1</td>
<td>x = 0; x = x + 1; x = 0; x = x + 2; x = 0; x = x + 3</td>
<td>Legal</td>
</tr>
<tr>
<td>Schedule 2</td>
<td>x = 0; x = 0; x = x + 1; x = x + 2; x = 0; x = x + 3;</td>
<td>Legal</td>
</tr>
<tr>
<td>Schedule 3</td>
<td>x = 0; x = 0; x = x + 1; x = 0; x = x + 2; x = x + 3;</td>
<td>Illegal</td>
</tr>
</tbody>
</table>

(d)

\(a) - c)\ Three transactions \(T_1, T_2,\) and \(T_3\)

\(d)\ Possible schedules

---

**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>
A non-serially equivalent interleaving of operations of transactions $T$ and $U$

<table>
<thead>
<tr>
<th>Transaction $T$:</th>
<th>Transaction $U$:</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>

Implementing Transactions: Private Workspace

a) The file index and disk blocks for a three-block file
b) The situation after a transaction has modified block 0 and appended block 3
c) After committing
Implementing Transactions: Writeahead Log

x = 0;
y = 0;
BEGIN_TRANSACTION;
  x = x + 1;
  y = y + 2
  x = y * y;
END_TRANSACTION;

(a) x = 0
(b) y = 0
(c) x = 1
(d) y = 4

(a) A transaction
(b) – d) The log before each statement is executed

Concurrency Control

General organization of managers for handling transactions.
### Transactions \( T \) and \( U \) with exclusive locks

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

### Lock compatibility

<table>
<thead>
<tr>
<th>For one object</th>
<th>Lock requested</th>
<th>write</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lock already set</td>
<td>none</td>
<td>OK</td>
</tr>
<tr>
<td></td>
<td>read</td>
<td>OK</td>
</tr>
<tr>
<td></td>
<td>write</td>
<td>wait</td>
</tr>
</tbody>
</table>

Transactions 19

Transactions 20
Use of locks in strict two-phase locking

1. When an operation accesses an object within a transaction:
   (a) If the object is not already locked, it is locked and the operation proceeds.
   (b) If the object has a conflicting lock set by another transaction, the transaction must wait until it is unlocked.
   (c) If the object has a non-conflicting lock set by another transaction, the lock is shared and the operation proceeds.
   (d) If the object has already been locked in the same transaction, the lock will be promoted if necessary and the operation proceeds.
   (Where promotion is prevented by a conflicting lock, rule (b) is used.)

2. When a transaction is committed or aborted, the server unlocks all objects it locked for the transaction.

Two-Phase Locking (1)

![Graph showing two-phase locking](image)
Strict Two-Phase Locking (2)

Deadlock with write locks

<table>
<thead>
<tr>
<th>Transaction $T$</th>
<th>Transaction $U$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operations</td>
<td>Locks</td>
</tr>
<tr>
<td>$a.\text{deposit}(100)$;</td>
<td>write lock $A$</td>
</tr>
<tr>
<td>$b.\text{withdraw}(100)$</td>
<td>waits for $U$'s lock on $B$</td>
</tr>
</tbody>
</table>
The wait-for graph

A cycle in a wait-for graph
Another wait-for graph

Transaction T

- 1. deposit(100);
- 2. withdraw(100)

<table>
<thead>
<tr>
<th>Operations</th>
<th>Locks</th>
</tr>
</thead>
<tbody>
<tr>
<td>deposit(100)</td>
<td>write lock A</td>
</tr>
<tr>
<td>withdraw(100)</td>
<td></td>
</tr>
</tbody>
</table>

Transaction U

- 1. deposit(200)
- 2. withdraw(200)

<table>
<thead>
<tr>
<th>Operations</th>
<th>Locks</th>
</tr>
</thead>
<tbody>
<tr>
<td>deposit(200)</td>
<td>write lock B</td>
</tr>
<tr>
<td>withdraw(200)</td>
<td></td>
</tr>
</tbody>
</table>

Resolution of deadlock

Transaction T

- 1. deposit(100);
- 2. withdraw(100)

<table>
<thead>
<tr>
<th>Operations</th>
<th>Locks</th>
</tr>
</thead>
<tbody>
<tr>
<td>deposit(100)</td>
<td>write lock A</td>
</tr>
<tr>
<td>withdraw(100)</td>
<td></td>
</tr>
</tbody>
</table>

Transaction U

- 1. deposit(200)
- 2. withdraw(200)

<table>
<thead>
<tr>
<th>Operations</th>
<th>Locks</th>
</tr>
</thead>
<tbody>
<tr>
<td>deposit(200)</td>
<td>write lock B</td>
</tr>
<tr>
<td>withdraw(200)</td>
<td></td>
</tr>
</tbody>
</table>
Optimistic Concurrency Control

- **Drawbacks of locking**
  - Overhead of lock maintenance
  - Deadlocks
  - Reduced concurrency

- **Optimistic Concurrency Control**
  - In most applications, likelihood of conflicting accesses by concurrent transactions is low
  - Transactions proceed as though there are no conflicts
  - Three phases
    - Working Phase – transactions read and write private copies of objects
    - Validation Phase – each transaction is assigned a transaction number when it enters this phase
    - Update Phase

---

Optimistic Concurrency Control: Serializability of transaction $T_v$ with respect to transaction $T_i$

$T_v$ and $T_i$ are overlapping transactions

For $T_v$ to be serializable wrt $T_i$ the following rules must hold

<table>
<thead>
<tr>
<th>$T_v$</th>
<th>$T_i$</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>write</td>
<td>read</td>
<td>1. $T_i$ must not read objects written by $T_v$</td>
</tr>
<tr>
<td>read</td>
<td>write</td>
<td>2. $T_v$ must not read objects written by $T_i$</td>
</tr>
<tr>
<td>write</td>
<td>write</td>
<td>3. $T_i$ must not write objects written by $T_v$ and $T_v$ must not write objects written by $T_i$</td>
</tr>
</tbody>
</table>

If simplification is made that only one transaction may be in its validation or write phases at one time, then third rule is always satisfied
Validation of transactions

Backward validation of transaction $T_v$

```java
boolean valid = true;
for (int $i = startTn+1; i <= finishTn; i++) {
    if (read set of $T_v$ intersects write set of $T_i$) valid = false;
}
```

Forward validation of transaction $T_v$

```java
boolean valid = true;
for (int $id = active1; id <= activeN; id++) {
    if (write set of $T_v$ intersects read set of $T_id$) valid = false;
}
```
**Timestamp based concurrency control**

- Each timestamp is assigned a unique timestamp at the moment it starts
  - In distributed transactions, Lamport’s timestamps can be used
- Every data item has a timestamp
  - Read timestamp = timestamp of transaction that last read the item
  - Write timestamp = timestamp of transaction that most recently changed an item

---

**Operation conflicts for timestamp ordering**

<table>
<thead>
<tr>
<th>Rule</th>
<th>$T_e$</th>
<th>$T_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. write</td>
<td>$T_e$ must not write an object that has been read by any $T_i$ where $T_i &gt; T_e$ this requires that $T_e = \text{the maximum read timestamp of the object.}$</td>
<td></td>
</tr>
<tr>
<td>2. write</td>
<td>$T_e$ must not write an object that has been written by any $T_i$ where $T_i &gt; T_e$ this requires that $T_e &gt; \text{write timestamp of the committed object.}$</td>
<td></td>
</tr>
<tr>
<td>3. read</td>
<td>$T_e$ must not read an object that has been written by any $T_i$ where $T_i &gt; T_e$ this requires that $T_e &gt; \text{write timestamp of the committed object.}$</td>
<td></td>
</tr>
</tbody>
</table>
**Timestamp ordering write rule**

if \((T_c = \text{maximum read timestamp on } D) \land T_c > \text{write timestamp on committed version of } D)\)

perform write operation on tentative version of \(D\) with write timestamp \(T_c\)

else /* write is too late */

Abort transaction \(T_c\)

---

**Write operations and timestamps**

(a) \(T_3\) write

(b) \(T_3\) write

(c) \(T_2\) write

(d) \(T_3\) write

Transaction aborts

Key:

- \(T_i\) Committed
- \(T_i\) Tentative

object produced by transaction \(T_i\)

(with write timestamp \(T_i\))

\(T_1 < T_2 < T_3 < T_4\)
Timestamp ordering read rule

if (T_c > write timestamp on committed version of D) {
    let D_selected be the version of D with the maximum write timestamp = T_c
    if (D_selected is committed)
        perform read operation on the version D_selected
    else
        Wait until the transaction that made version D_selected commits or aborts
        then reapply the read rule
    } else
    Abort transaction T_c

Read operations and timestamps

(a) T_3 read
Selected

(b) T_3 read
Selected

(c) T_3 read
Selected

(d) T_3 read
Selected

Key:
- Tentative
- Committed

Object produced by transaction T_i
(with write timestamp T_i)

T_1 < T_2 < T_3 < T_4

Transactions 37

Transactions 38
**Timestamps in transactions T and U**

<table>
<thead>
<tr>
<th>T</th>
<th>U</th>
<th>Timestamps and versions of objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>openTransaction</td>
<td>bal = b.getBalance()</td>
<td>openTransaction</td>
</tr>
<tr>
<td>b.setBalance(bal*1.1)</td>
<td>bal = b.getBalance()</td>
<td>b.setBalance(bal*1.1)</td>
</tr>
<tr>
<td>a.withdraw(bal/10)</td>
<td>bal = b.getBalance()</td>
<td>b.setBalance(bal*1.1)</td>
</tr>
<tr>
<td>commit</td>
<td>bal = b.getBalance()</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTS</td>
<td>WTS</td>
<td>RTS</td>
</tr>
<tr>
<td>{ }</td>
<td>S</td>
<td>{ }</td>
</tr>
<tr>
<td>{}</td>
<td>S</td>
<td></td>
</tr>
</tbody>
</table>

Transactions 39

**Distributed transactions**

(a) Flat transaction

(b) Nested transactions
Nested banking transaction

\[ T = \text{openTransaction} \]

openSubTransaction
\[ a.\text{withdraw}(10); \]
openSubTransaction
\[ b.\text{withdraw}(20); \]
openSubTransaction
\[ c.\text{deposit}(10); \]
openSubTransaction
\[ d.\text{deposit}(20); \]
closeTransaction

A distributed banking transaction

\[ T = \text{openTransaction} \]

a.\text{withdraw}(4);

b.\text{withdraw}(3);
c.\text{deposit}(4);
b.\text{deposit}(3);
closeTransaction

Note: the coordinator is in one of the servers, e.g. BranchX
Concurrency Control for Distributed Transactions

General organization of managers for handling distributed transactions.

Concurrency Control for Distributed Transactions

- **Locking**
  - Distributed deadlocks possible

- **Timestamp ordering**
  - Lamport time stamps
    - For efficiency it is required that timestamps issued by coordinators be roughly synchronized
### Interleavings of transactions $U$, $V$ and $W$

<table>
<thead>
<tr>
<th></th>
<th>$U$</th>
<th>$V$</th>
<th>$W$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$d.$deposit(10) lock $D$</td>
<td>$b.$deposit(10) lock $B$</td>
<td>$c.$deposit(30) lock $C$</td>
</tr>
<tr>
<td></td>
<td>$a.$deposit(20) lock $A$ at $X$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$b.$withdraw(30) wait at $Y$</td>
<td>$c.$withdraw(20) wait at $Z$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$a.$withdraw(20) wait at $X$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Distributed deadlock

(a) [Diagram showing the interactions between transactions $U$, $V$, $W$, $X$, $Y$, and $Z$.]

(b) [Diagram showing the interactions between transactions $U$, $V$, $W$, $X$, and $Y$.]
Local and global wait-for graphs

Atomic Commit Protocols

- The atomicity of a transaction requires that when a distributed transaction comes to an end, either all of its operations are carried out or none of them
- One phase commit
  - Coordinator tells all participants to commit
  - If a participant cannot commit (say because of concurrency control), no way to inform coordinator
- Two phase commit (2PC)
The two-phase commit protocol

Phase 1 (voting phase):
1. The coordinator sends a `canCommit`? request to each of the participants in the transaction.
2. When a participant receives a `canCommit`? request it replies with its vote (Yes or No) to the coordinator. Before voting Yes, it prepares to commit by saving objects in permanent storage. If the vote is No the participant aborts immediately.

Phase 2 (completion according to outcome of vote):
3. The coordinator collects the votes (including its own).
   (a) If there are no failures and all the votes are Yes the coordinator decides to commit the transaction and sends a `doCommit` request to each of the participants.
   (b) Otherwise the coordinator decides to abort the transaction and sends `doAbort` requests to all participants that voted Yes.
4. Participants that voted Yes are waiting for a `doCommit` or `doAbort` request from the coordinator. When a participant receives one of these messages it acts accordingly and in the case of commit, makes a `haveCommitted` call as confirmation to the coordinator.

Operations for two-phase commit protocol

`canCommit?(trans) -> Yes / No`
Call from coordinator to participant to ask whether it can commit a transaction. Participant replies with its vote.

`doCommit(trans)`
Call from coordinator to participant to tell participant to commit its part of a transaction.

`doAbort(trans)`
Call from coordinator to participant to tell participant to abort its part of a transaction.

`haveCommitted(trans, participant)`
Call from participant to coordinator to confirm that it has committed the transaction.

`getDecision(trans) -> Yes / No`
Call from participant to coordinator to ask for the decision on a transaction after it has voted Yes but has still had no reply after some delay. Used to recover from server crash or delayed messages.
Communication in two-phase commit protocol

Two-Phase Commit (1)

(a) The finite state machine for the coordinator in 2PC.
(b) The finite state machine for a participant.
Two-Phase Commit (2)

<table>
<thead>
<tr>
<th>State of Q</th>
<th>Action by P</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMMIT</td>
<td>Make transition to COMMIT</td>
</tr>
<tr>
<td>ABORT</td>
<td>Make transition to ABORT</td>
</tr>
<tr>
<td>INIT</td>
<td>Make transition to ABORT</td>
</tr>
<tr>
<td>READY</td>
<td>Contact another participant</td>
</tr>
</tbody>
</table>

Actions taken by a participant $P$ when residing in state $READY$ and having contacted another participant $Q$.

Two-Phase Commit (3)

actions by coordinator:
while START_2PC to local log;
multicast VOTE_REQUEST to all participants;
while not all votes have been collected {
    wait for any incoming vote;
    if timeout {
        write GLOBAL_ABORT to local log;
        multicast GLOBAL_ABORT to all participants;
        exit;
    }
    record vote;
}
if all participants sent VOTE_COMMIT and coordinator votes COMMIT {
    write GLOBAL_COMMIT to local log;
    multicast GLOBAL_COMMIT to all participants;
} else {
    write GLOBAL_ABORT to local log;
    multicast GLOBAL_ABORT to all participants;
}

Outline of the steps taken by the coordinator in a two phase commit protocol.
Two-Phase Commit (4)

**actions by participant:**
write INIT to local log;
wait for VOTE_REQUEST from coordinator;
if timeout {
write VOTE_ABORT to local log;
exit;
}
if participant votes COMMIT {
write VOTE_COMMIT to local log;
send VOTE_COMMIT to coordinator;
wait for DECISION from coordinator;
if timeout {
   multicast DECISION_REQUEST to other participants;
   wait until DECISION is received; /* remain blocked */
   write DECISION to local log;
}
if DECISION == GLOBAL_COMMIT
write GLOBAL_COMMIT to local log;
else if DECISION == GLOBAL_ABORT
write GLOBAL_ABORT to local log;
} else {
   write VOTE_ABORT to local log;
send VOTE_ABORT to coordinator;
}

Steps taken by participant process in 2PC.

Two-Phase Commit (5)

**actions for handling decision requests:** /* executed by separate thread */
while true {
   wait until any incoming DECISION_REQUEST is received; /* remain blocked */
   read most recently recorded STATE from the local log;
   if STATE == GLOBAL_COMMIT
      send GLOBAL_COMMIT to requesting participant;
   else if STATE == INIT or STATE == GLOBAL_ABORT
      send GLOBAL_ABORT to requesting participant;
   else
      skip; /* participant remains blocked */

Steps taken for handling incoming decision requests.
Three Phase Commit

Problem with 2PC

- If coordinator crashes, participants cannot reach a decision, stay blocked until coordinator recovers

3PC

- There is no single state from which it is possible to make a transition directly to either COMMIT or ABORT states
- There is no state in which it is not possible to make a final decision, and from which a transition to COMMIT can be made

Three-Phase Commit

- Finite state machine for the coordinator in 3PC
- Finite state machine for a participant