Lecture 10 – Distributed Concurrency Management
Tuesday Sept 28th, 2017
Logistics Updates

- P1 checkpoint due (11:59EST, Oct 2\textsuperscript{nd})
  - Part A Due: Friday, Oct 13\textsuperscript{th}
  - Part B Due: Oct 23\textsuperscript{rd}

- HW2 released soon (Oct 2\textsuperscript{nd})
  - Due Friday, Oct 13\textsuperscript{th}
  - (*No Late Days*) => time to prepare for Mid term

- Mid term Monday Oct 16\textsuperscript{th} – 6pm (DH 2210)
  - Will cover everything until the first half of class
  - Midterm Review: Sunday 10/15 @10am, DH2315
Today's Lecture Outline

• Consistency for multiple-objects, multiple-servers

• Part I: Single Server Case
  • (not covered well in book)
  • Two Phase Locking

• Part II: Distributed Transactions
  • Two Phase Commit (Tanenbaum 8.6)
Distributed Consistency Management

- Multiple-Objects, Multiple Distributed Servers
- Assumption: We will ignore failures
  - In the context of today's consistency mechanisms, for fault tolerance, most systems use a form of logging, where they write information down *before* operating on it, to recover from simple failures.
  - We will talk about failure/recovery later on in class, as in how we can use logging to replay and get back into a consistent state.
Case I: Single Server

• Background: Database Researchers

• Defined: “Transactions”
  • Collections of Reads + Writes to Global State
  • Appear as a single, “indivisible” operation
  • Standard Models for Reliable Storage (visit later)

• Desirable Characteristics of Transactions
  • Atomicity, Consistency, Isolation, Durability
  • Also referred to as the “ACID” Acronym!
Transactions: ACID Properties

• **Atomicity**: Each transaction completes in its entirety, or is aborted. If aborted, should not have effect on the shared global state.
  • Example: Update account balance on multiple servers

• **Consistency**: Each transaction preserves a set of invariants about global state. (exact nature is system dependent).
  • Example: in a bank system, law of conservation of $$
Transactions: ACID Properties

- **Isolation**: Also means serializability. Each transaction executes as if it were the only one with the ability to RD/WR shared global state.

- **Durability**: Once a transaction has been completed, or “committed” there is no going back. In other words there is no “undo”.

- Transactions can also be nested
- “Atomic Operations” => Atomicity + Isolation
A Transaction Example: Bank

- Array \( \text{Bal}[i] \) stores balance of Account “i”
- Implement: \text{xfer}, \text{withdraw}, \text{deposit}

**xfer\( (i, j, v) \):**
if \( \text{withdraw}(i, v) \):
deposit\( (j, v) \)

else
abort

**withdraw\( (i, v) \):**
\begin{align*}
  b &= \text{Bal}[i] \quad // \text{Read} \\
  \text{if } b \geq v \quad // \text{Test} \\
  \text{Bal}[i] &= b - v \quad // \text{Write} \\
  \text{return true} \\
  \text{else} \\
  \text{return false}
\end{align*}

**deposit\( (j, v) \):**
\begin{align*}
  \text{Bal}[j] &= + v
\end{align*}
Imagine: \( \text{Bal}[x] = 100, \text{Bal}[y] = \text{Bal}[z] = 0 \)

- Two transactions \( \implies T1: \text{xfer}(x, y, 60), T2: \text{xfer}(x, z, 70) \)
- ACID Properties: T1 or T2 in some serial order
  - T1; T2: T1 succeeds; T2 Fails. \( \text{Bal}[x] = 40, \text{Bal}[y] = 60 \)
  - T2; T1: T2 succeeds; T1 Fails. \( \text{Bal}[x] = 30, \text{Bal}[z] = 70 \)
- What if we didn’t take care? Is there a race condition?
  - Updating \( \text{Bal}[x] \) with Read/Write interleaving of T1, T2

```python
xfer(i, j, v):
    if withdraw(i, v):
        deposit(j, v)
    else
        abort

withdraw(i, v):
    b = Bal[i]  # Read
    if b >= v   # Test
        Bal[i] = b - v # Write
        return true
    else
        return false

deposit(j, v):
    Bal[j] += v
```
A Transaction Example: Bank

Imagine: Bal[x] = 100, Bal[y]=Bal[z]=0
- Two transactions => T1: xfer(x,y,60), T2: xfer(x,z,70)
- ACID violation: Not Isolated, Not durable
  - Updating Bal[x] with Read/Write interleaving of T1,T2
  - Bal[x] = 30 or 40; Bal[y] = 60; Bal[z] = 70
- For Consistency, implemented sumbalance()
  - State invariant sumbalance=100 violated! We created $$

xfer(i, j, v):
  if withdraw(i, v):
    deposit(j, v)
  else
    abort

withdraw(i, v):
  b = Bal[i]   // Read
  if b >= v   // Test
    Bal[i] = b-v // Write
    return true
  else
    return false

deposit(j, v):
  Bal[j] += v

sumbalance(i, j, k):
  return Bal[i]+Bal[j]+ Bal[k]
Implement transactions with locks

• Use locks to wrap xfer

\[
\text{xfer}(i, j, v):
\]
\[
\text{lock()}
\]
\[
\text{if withdraw}(i, v):
\]
\[
\text{deposit}(j, v)
\]
\[
\text{else}
\]
\[
\text{abort}
\]
\[
\text{unlock()}
\]

However, is this the correct approach? (Hint: efficiency)

Sequential bottleneck due to global lock. Solution?

\[
\text{xfer}(i, j, v):
\]
\[
\text{lock}(i)
\]
\[
\text{if withdraw}(i, v):
\]
\[
\text{unlock}(i)
\]
\[
\text{lock}(j)
\]
\[
\text{deposit}(j, v)
\]
\[
\text{unlock}(j)
\]
\[
\text{else}
\]
\[
\text{unlock}(i)
\]
\[
\text{abort}
\]

Is this fixed then?

No, consistency violation. \text{sumbalance()} after \text{unlock}(i)
Implement transactions with locks

\[xfer(i, j, v):\]
\[
\text{lock}(i)
\]
\[
\text{if withdraw}(i, v):
\quad \text{lock}(j)
\quad \text{deposit}(j, v)
\quad \text{unlock}(i);
\quad \text{unlock}(j)
\]
\[
\text{else}
\quad \text{unlock}(i)
\quad \text{abort}
\]

Fix: Release locks when update of all state variables complete.

Are we done then?

Nope, deadlock:
Bal\[x\]=Bal\[y\]=100.
\[\text{xfer}(x, y, 40)\text{ and xfer } (y, x, 30)\]

\[xfer(i, j, v):\]
\[
\text{lock}(\text{min}(i, j)); \text{lock}(\text{max } (i, j))
\]
\[
\text{if withdraw}(i, v):
\quad \text{deposit}(j, v)
\quad \text{unlock}(i); \text{unlock}(j)
\]
\[
\text{else}
\quad \text{unlock}(i); \text{unlock}(j)
\quad \text{abort}
\]

General Rule: Always acquire locks according to some consistent global order
2-Phase Locking

- Building a “Wait-for” graph for state of locks. Vertices represent transactions. Edge from vertex i to vertex j if transaction i is waiting for lock held by transaction j.
- In this case, what happens? => Cycle is a deadlock
- Label edges with its lock ID. For any cycle, there must be some pair of edges (i, j), (j, k) labeled with values x & y such that x > y. That implies that transaction j is holding lock x and it wants lock y, where x > y. That implies that j is not acquiring its lock in proper order.
- General scheme called 2-phase locking
  - More precisely: strong strict two phase locking
2-Phase Locking

- General 2-phase locking
  - Phase 1: Acquire or Escalate Locks (e.g. read => write)
  - Phase 2: Release or de-escalate lock

- Strict 2-phase locking
  - Phase 1: (same as before)
  - Phase 2: Release WRITE lock at end of transaction only

- Strong Strict 2-phase locking
  - Phase 2: Release ALL locks at end of transaction only.
  - Most common version, required for ACID properties
2-Phase Locking

- Why not always use strong-string 2-phase locking?
  - A transaction may not know the locks it needs in advance

  ```python
  if Bal(yuvraj) < 100:
      x = find_richest_prof()
      transfer_from(x, yuvraj)
  ```

- Other ways to handle deadlocks
  - Lock manager builds a “waits-for” graph. On finding a cycle, choose offending transaction and force abort
  - Use timeouts: Transactions should be short. If hit time limit, find transaction waiting for a lock and force abort.
Transactions – split into 2 phases

• Phase 1: Preparation:
  • Determine what has to be done, how it will change state, without actually altering it.
  • Generate Lock set “L”
  • Generate List of Updates “U”

• Phase 2: Commit or Abort
  • Everything OK, then update global state
  • Transaction cannot be completed, leave global state as is
  • In either case, RELEASE ALL LOCKS
Example

\[
\text{xfer}(i, j, v):
\begin{align*}
L &= \{i,j\} \quad \text{// Locks} \\
U &= [] \quad \text{// List of Updates} \\
\text{begin}(L) \quad \text{// Begin transaction, Acquire locks} \\
b_i &= \text{Bal}[i] \\
b_j &= \text{Bal}[j] \\
\text{if } b_i \geq v: \\
&\quad \text{Append}(U, \text{Bal}[i] \leftarrow b_i - v) \\
&\quad \text{Append}(U, \text{Bal}[j] \leftarrow b_j + v) \\
&\quad \text{commit}(U,L) \\
\text{else} \\
&\quad \text{abort}(L)
\end{align*}
\]

Question: So, what would “commit” and ”abort” look like?

\[
\begin{align*}
\text{commit}(U,L): \\
&\quad \text{Perform all updates in } U \\
&\quad \text{Release all locks in } L
\end{align*}
\]

\[
\begin{align*}
\text{abort}(L): \\
&\quad \text{Release all locks in } L
\end{align*}
\]
Today's Lecture Outline

• Consistency for multiple-objects, multiple-servers

• Part I: Single Server Case
  • (not covered well in book)
  • Two Phase Locking

• Part II: Distributed Transactions
  • Two Phase Commit (Tanenbaum 8.5)
Distributed Transactions

• Similar idea as before, but:
  • State spread across servers (maybe even WAN)
  • Want to enable single transactions to read and update global state while maintaining ACID properties

• Overall Idea:
  • Client initiate transaction. Makes use of “co-ordinator”
  • All other relevant servers operate as “participants”
  • Co-ordinator assigns unique transaction ID (TID)
2-Phase commit

- Prepare & Vote
  - Participants figure out all state changes
  - Each determines if it can complete the transaction
  - Communicate with coordinator

- Commit
  - Coordinator broadcasts to participants: COMMIT / ABORT
  - If COMMIT, participants make respective state changes
2-phase commit - Implementation

- Implemented as a set of messages
  - Between coordinators and participants
- Messages in first phase
  - A: Coordinator sends “CanCommit?” to participants
  - B: Participants respond: “VoteCommit” or “VoteAbort”
- Messages in the second phase
  - A: if any participant “VoteAbort” transaction aborts. Coordinator sends “DoAbort” to everyone => release locks
  - Else, send “DoCommit” to everyone => complete transaction
Example for 2PC

- Bank Account “i” at Server A, “j” at Server B.

  \[\begin{align*}
  &L=\{i\} \\
  &\text{Begin}(L) // \text{Acq. Locks} \\
  &U=[] // \text{List of Updates} \\
  &b=\text{Bal}[i] \\
  &\text{if } b \geq v: \\
  &\quad \text{Append}(U, \text{Bal}[i] <- b - v) \\
  &\quad \text{vote commit} \\
  &\text{else} \\
  &\quad \text{vote abort} \\
  \end{align*}\]

Server A would implement transaction

Server B would implement transaction

\[\begin{align*}
  &L=\{j\} \\
  &\text{Begin}(L) // \text{Acq. Locks} \\
  &U=[] // \text{List of Updates} \\
  &b=\text{Bal}[j] \\
  &\text{Append}(U, \text{Bal}[j] <- b + v) \\
  &\text{vote commit} \\
  \end{align*}\]

Server B can assume that the account of “i” has enough money, otherwise whole transaction will abort.

What about locking? Locks held by individual participants
  - Acquire lock at start of prep process, release at Commit/Abort
Deadlocks and Livelocks

- Distributed deadlock
  - Cyclic dependency of locks by transactions across servers
  - In 2PC this can happen if participants unable to respond to voting request (e.g. still waiting on a lock on its local resource)
  - Handled with a timeout. Participants times out, then votes to abort. Retry transaction again.
    - Addresses the deadlock concern
    - However, danger of LIVELOCK – keep trying!
Summary

• Distributed consistency management
• ACID Properties desirable
• Single Server case: use locks, and in cases use 2-phase locking (strict 2PL, strong strict 2PL), transactional support for locks
• Multiple server distributed case: use 2-phase commit for distributed transactions. Need a coordinator to manage messages from participants.
Backup Material

- 2PC from the Book
- Terminology used by messages different, but essentially the protocol is the same
Two-Phase Commit (1)

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

Coordinator/Participant can be blocked in 3 states:

- Participant: Waiting in INIT state for VOTE_REQUEST
- Coordinator: Blocked in WAIT state, listening for votes
- Participant: blocked in READY state, waiting for global vote
Two-Phase Commit (2)

- What if a “READY” participant does not receive the global commit? Can’t just abort => figure out what message a co-ordinator may have sent.
- Approach: ask other participants
  - Take actions on response on any of the participants
  - E.g. P is in READY state, asks other “Q” participants

<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>

What happens if everyone is in ”READY” state?
Two-Phase Commit (3)

- For recovery, must save state to persistent storage (e.g. log), to restart/recover after failure.
  - Participant (INIT): Safe to local abort, inform Coordinator
  - Participant (READY): Contact others
  - Coordinator (WAIT): Retransmit VOTE_REQ
  - Coordinator (WAIT/Decision): Retransmit VOTE_COMMIT
2PC: Actions by Coordinator

Actions by coordinator:

write 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;
}

Why do we have the "write to LOG" statements?
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;
}
2PC: Handling Decision Request

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 */
}

Note, participant can only help others if it has reached a global decision and committed it to its log.

What if everyone has received VOTE_REQ, and Co-ordinator crashes?
Three-Phase Commit (1)

- The states of the coordinator and each participant satisfy the following two conditions:
  1. There is no single state from which it is possible to make a transition directly to either a COMMIT or an ABORT state.
  2. There is no state in which it is not possible to make a final decision, and from which a transition to a COMMIT state can be made.
Three-Phase Commit (2)

Figure 8-22. (a) The finite state machine for the coordinator in 3PC. (b) The finite state machine for a participant.
3PC – Resolution in case of failures

- Can always get back to a known state
  - If any Participant Q in INIT: Safe to abort. This is because A can be in INIT only if no other participant is in PRECOMMIT
  - If All participants are in READY: Abort. Since if P has crashed and recovers later, neither P nor others know what the state of P might be on recovery. If P recovers to INIT, Abort is fine. Even if PRECOMMIT, abort is OK.
    - Different to 2PC where a crashed participant can recover to a COMMIT state, while others in READY. In 3PC if a operational process is in READY, no other crashed process can recover to state other than INIT, ABORT, PRECOMMIT
  - If Processes that P can contact are in PRECOMMIT, then safe to COMMIT