Distributed Systems

CS425/ECE428

March 31 2022

Instructor: Radhika Mittal

Acknowledgements for the materials: Indy Gupta and Nikita Borisov
Logistics

• HW5 has been released
  • You should be able to solve all questions after today’s class.
Thank you for your feedback

• Some common complaints
  • Time management during OHs.
  • Losing track of materials when I answer questions in class.
  • Pace of lectures.
  • Practice materials and examples.
Lost Update Example with Timestamped Ordering

Transaction T1

\[ x = \text{getSeats}(ABC123); \]

if \( x > 1 \)

\[ x = x - 1; \]

write \( x \), ABC123;

commit

Transaction T2

\[ x = \text{getSeats}(ABC123); \]

if \( x > 1 \)

\[ x = x - 1; \]

write \( x \), ABC123;

commit

ABC123: state
committed value = 10
committed timestamp = 0
RTS: 1, 2
TW:
Lost Update Example with Timestamped Ordering

Transaction T1
\[
\begin{align*}
  x &= \text{getSeats}(ABC123); \\
  \text{if}(x > 1) &\quad \text{if}(x > 1) \\
  x &= x - 1; \quad x = x - 1; \\
  \text{write}(x, ABC123); \quad \text{write}(x, ABC123); \\
  \text{commit} \quad \text{commit}
\end{align*}
\]

Transaction T2
\[
\begin{align*}
  x &= \text{getSeats}(ABC123); \\
  \text{if}(x > 1) &\quad \text{if}(x > 1) \\
  x &= x - 1; \quad x = x - 1; \\
  \text{write}(x, ABC123); \quad \text{write}(x, ABC123); \\
  \text{commit} \quad \text{commit}
\end{align*}
\]

ABC123: state
committed value = 10
committed timestamp = 0
RTS: 1
TW:
Lost Update Example with Timestamped Ordering

**Transaction T1**

\[ x = \text{getSeats}(ABC123); \]

if \( x > 1 \)
\[ x = x - 1; \]
write \( x \), ABC123);

commit

**Transaction T2**

\[ x = \text{getSeats}(ABC123); \]

if \( x > 1 \)
\[ x = x - 1; \]
write \( x \), ABC123);

commit

ABC123: state
committed value = 10
committed timestamp = 0
RTS: 1, 2
TW:
Next Example with Timestamped Ordering

**Transaction T1**

\[ x = \text{getSeats}(\text{ABC123}); \]
\[ y = \text{getSeats}(\text{ABC789}); \]
\[ \text{write}(x-5, \text{ABC123}); \]
\[ \text{write}(y+5, \text{ABC789}); \]
\[ \text{commit} \]

**Transaction T2**

\[ x = \text{getSeats}(\text{ABC123}); \]
\[ y = \text{getSeats}(\text{ABC789}); \]
\[ \text{print}(\text{"Total:" } x+y); \]
\[ \text{commit} \]

**State and Timestamps**

- **ABC123**: state committed value = 10, committed timestamp = 0
- **ABC789**: state committed value = 5, committed timestamp = 0
Next Example with Timestamped Ordering

**Transaction T1**

\[x = \text{getSeats}(ABC123)\];
\[y = \text{getSeats}(ABC789)\];
\[\text{write}(x-5, ABC123)\];
\[\text{write}(y+5, ABC789)\];
\[\text{commit}\]

**Transaction T2**

\[x = \text{getSeats}(ABC123)\];
\[y = \text{getSeats}(ABC789)\];
\[\text{print}(“Total:” \ x+y)\];
\[\text{commit}\]

**ABC123:**
- state: committed value = 10
- committed timestamp = 0
- RTS: 1
- TW:

**ABC789:**
- state: committed value = 5
- committed timestamp = 0
- RTS: 1
- TW:
Next Example with Timestamped Ordering

Transaction T1

\[ x = \text{getSeats}(ABC123); \]
\[ y = \text{getSeats}(ABC789); \]
\[ \text{write}(x-5, ABC123); \]
\[ \text{write}(y+5, ABC789); \]
\[ \text{commit} \]

Transaction T2

\[ x = \text{getSeats}(ABC123); \]
\[ y = \text{getSeats}(ABC789); \]
\[ \text{print}(\text{"Total:" } x+y); \]
\[ \text{commit} \]

ABC123: state committed value = 10 committed timestamp = 0 RTS: 1 TW:

ABC789: state committed value = 5 committed timestamp = 0 RTS: 1 TW:
Next Example with Timestamped Ordering

**Transaction T1**

\[
\begin{align*}
\text{x} &= \text{getSeats}(\text{ABC123}); \\
y &= \text{getSeats}(\text{ABC789}); \\
\text{write}(x-5, \text{ABC123}); \\
\text{write}(y+5, \text{ABC789}); \\
\text{commit}
\end{align*}
\]

**Transaction T2**

\[
\begin{align*}
\text{x} &= \text{getSeats}(\text{ABC123}); \\
y &= \text{getSeats}(\text{ABC789}); \\
\text{print(“Total:” x+y);} \\
\text{commit}
\end{align*}
\]

ABC123: state
committed value = 10
committed timestamp = 0
RTS: 1
TW: (5, 1)

ABC789: state
committed value = 5
committed timestamp = 0
RTS: 1
TW:
Next Example with Timestamped Ordering

**Transaction T1**

\[ x = \text{getSeats}(\text{ABC123}); \]
\[ y = \text{getSeats}(\text{ABC789}); \]
\[ \text{write}(x-5, \text{ABC123}); \]
\[ \text{write}(y+5, \text{ABC789}); \]
\[ \text{commit} \]

**Transaction T2**

\[ x = \text{getSeats}(\text{ABC123}); \text{wait} \]
\[ y = \text{getSeats}(\text{ABC789}); \]
\[ \text{print}(\text{“Total:” } x+y); \]
\[ \text{commit} \]

\[ \text{commit} \]

ABC123: state
- committed value = 10
- committed timestamp = 0
- RTS: 1
- TW: (5, 1)

ABC789: state
- committed value = 5
- committed timestamp = 0
- RTS: 1
- TW: 
Next Example with Timestamped Ordering

**Transaction T1**

\[
x = \text{getSeats(ABC123)}; \\
y = \text{getSeats(ABC789)}; \]
\[
\text{write}(x-5, \text{ABC123}); \]
\[
\text{write}(y+5, \text{ABC789}); \]
\[
\text{commit} \]

**Transaction T2**

\[
x = \text{getSeats(ABC123)}; \quad \text{wait} \]
\[
y = \text{getSeats(ABC789)}; \]
\[
\text{print(“Total:”} \ x+y); \]
\[
\text{commit} \]

---

ABC123: state committed value = 10 committed timestamp = 0 RTS: 1 TW: (5, 1)

ABC789: state committed value = 5 committed timestamp = 0 RTS: 1 TW: (10, 1)
Next Example with Timestamped Ordering

Transaction T1

\[
\begin{align*}
  x &= \text{getSeats}(ABC123); \\
  y &= \text{getSeats}(ABC789); \\
  \text{write}(x-5, ABC123); \\
  \text{write}(y+5, ABC789); \\
  \text{commit}
\end{align*}
\]

Transaction T2

\[
\begin{align*}
  x &= \text{getSeats}(ABC123); \text{ wait} \\
  y &= \text{getSeats}(ABC789); \\
  \text{print}(\text{"Total:" } x+y); \\
  \text{commit}
\end{align*}
\]

ABC123: state committed value = 5-10
committed timestamp = 0-1
RTS: 1
TW: (5, 1)

ABC789: state committed value = 5-10
committed timestamp = 0-1
RTS: 1
TW: (10, 1)
Next Example with Timestamped Ordering

**Transaction T1**

\[
x = \text{getSeats}(\text{ABC123});
\]

\[
y = \text{getSeats}(\text{ABC789});
\]

\[
\text{write}(x-5, \text{ABC123});
\]

\[
\text{write}(y+5, \text{ABC789});
\]

\[
\text{commit}
\]

**Transaction T2**

\[
x = \text{getSeats}(\text{ABC123}); \text{ wait}
\]

\[
y = \text{getSeats}(\text{ABC789});
\]

\[
\text{print(“Total:” } x+y);
\]

\[
\text{commit}
\]

**ABC123: state**

committed value = 10
committed timestamp = 0

**RTS:** 1

**TW:** (5, 1)

**ABC789: state**

committed value = 5
committed timestamp = 0

**RTS:** 1

**TW:** (10, 1)

T2 then proceeds after T1 commits
Agenda for today

• Distributed Transactions
  • Chapter 17
Transaction Processing

- Required properties: Atomicity, Consistency, Isolation, Durability (ACID).
- How to prevent transactions from affecting one another?
- Goal: increase concurrency and transaction throughput while maintaining correctness (ACID).
- Two approaches:
  - Pessimistic concurrency control: locking based.
    - read-write locks with two-phase locking and deadlock detection.
  - Optimistic concurrency control: abort if too late.
    - timestamped ordering.
- Focused on single server and multiple clients.
Distributed Transactions

• Transaction processing can be *distributed* across multiple servers.
  
  • Different objects can be stored on different servers.
    • Our focus today.

• An object may be replicated across multiple servers.
  • Next class.
Transactions with Distributed Servers

- Different objects touched by a transaction $T$ may reside on different servers.

```plaintext
Transaction T
  write(A, 1);
  write(B, 2);
  ...
  write(Y, 25);
  write(Z, 26);
  commit
```
Distributed Transaction Challenges

- **Atomic**: all-or-nothing
  - *Must ensure atomicity across servers.*
- **Consistent**: rules maintained
  - *Generally done locally, but may need to check non-local invariants at commit time.*
- **Isolation**: multiple transactions do not interfere with each other
  - *Locks at each server. How to detect and handle deadlocks?*
- **Durability**: values preserved even after crashes
  - *Each server keeps local recovery log.*
Distributed Transaction Challenges

• **Atomic**: all-or-nothing
  • *Must ensure atomicity across servers.*

• **Consistent**: rules maintained
  • *Generally done locally, but may need to check non-local invariants at commit time.*

• **Isolation**: multiple transactions do not interfere with each other
  • *Locks at each server. How to detect and handle deadlocks?*

• **Durability**: values preserved even after crashes
  • *Each server keeps local recovery log.*
Distributed Transaction Atomicity

- When T tries to commit, need to ensure
  - all these servers commit their updates from T => T will commit
  - Or none of these servers commit => T will abort

- What problem is this?
  - Consensus!
  - (It’s also called the “Atomic Commit” problem)
Special server called "Coordinator" initiates atomic commit.

- can be same as one of the servers with objects.

- Different transactions may have different coordinators.

Transaction T
write(A,1);
write(B,2);
...
write(Y, 25);
write(Z, 26);
commit
One-phase commit

• Client relays the “commit” or “abort” command to the coordinator.
  • Coordinator tells other servers to commit / abort.

• Issues with this?
  • Server with object has no say in whether transaction commits or aborts
    • If a local consistency check fails, it just cannot commit (while other servers have committed).
  • A server may crash before receiving commit message, with some updates still in memory.
Two-phase commit

Coordinator Server

Prepare

Server 1

... Server 13
Two-phase commit

Coordinator Server

Server 1

... Server 13

Prepare

• Save updates to disk
• Respond with “Yes” or “No”
Two-phase commit

Coordinator Server

Server 1

... Server 13

Prepare

• Save updates to disk
• Respond with “Yes” or “No”

All (13) “Yes” votes received within timeout?

Commit
Two-phase commit

Coordinator Server

Server 1

... Server 13

Prepare

• Save updates to disk
• Respond with “Yes” or “No”

All (13) “Yes” votes received within timeout?

Commit

• Wait! Can’t commit or abort before receiving next message!
Two-phase commit

- **Coordinator Server**
- **Server 1**
- **Server 13**

**Prepare**
- Save updates to disk
- Respond with “Yes” or “No”

**All (13) “Yes” votes received within timeout?**

**Commit**
- Commit updates from disk to store
Two-phase commit

Coordinator
Server

Server 1

... Server 13

Prepare

• Save updates to disk
• Respond with “Yes” or “No”

All (13) “Yes” votes received within timeout?

Commit

• Commit updates from disk to store

HaveCommitted

• Coordinator now knows that all servers have committed and it can delete the associated transaction information.
Two-phase commit

Coordinator Server

Server 1

... Server 13

Prepare

- Save updates to disk
- Respond with "Yes" or "No"

If any "No" vote or timeout before all (13) votes

Abort
Two-phase commit

- **Coordinator Server**
- **Server 1**
- **Server 13**

**Prepare**
- Save updates to disk
- Respond with “Yes” or “No”

**If any “No” vote or timeout before all (13) votes**
- Abort
  - Delete tentative updates in disk and abort.
Failures in Two-phase Commit

• If server voted Yes, it cannot commit unilaterally before receiving Commit message.
  • Does not know if other servers voted Yes.
• If server voted No, can abort right away.
  • Knows that the transaction cannot be committed.

• To deal with server crashes
  • Each server saves tentative updates into permanent storage, right before replying Yes/No in first phase. Retrievable after crash recovery.

• To deal with coordinator crashes
  • Coordinator logs all decisions and received/sent messages on disk.
  • After recovery => retrieve the logged state.
Failures in Two-phase Commit (contd)

- To deal with Prepare message loss
  - The server may decide to abort unilaterally after a timeout for first phase (server will vote No, and so coordinator will also eventually abort)

- To deal with Yes/No message loss
  - Coordinator aborts the transaction after a timeout (pessimistic!).
  - It must announce Abort message to all.

- To deal with Commit or Abort message loss
  - Server can poll coordinator (repeatedly).
Distributed Transaction Atomicity

• When T tries to commit, need to ensure
  • all these servers commit their updates from T => T will commit
  • Or none of these servers commit => T will abort

• What problem is this?
  • Consensus!
  • (It’s also called the “Atomic Commit” problem)

• Consensus is impossible in asynchronous system.
  • What makes two-phase commit work?
  • Crash failures in processes masked by replacing the crashed process with a new process whose state is retrieved from permanent storage.
  • Two-phase commit is blocked until a failed coordinator recovers.
Distributed Transaction Challenges

- **Atomic**: all-or-nothing
  - Must ensure atomicity across servers.
- **Consistent**: rules maintained
  - Generally done locally, but may need to check non-local invariants at commit time.
- **Isolation**: multiple transactions do not interfere with each other
  - Locks at each server. How to detect and handle deadlocks?
- **Durability**: values preserved even after crashes
  - Each server keeps local recovery log.
Isolation with Distributed Transaction

- Each server is responsible for applying concurrency control to objects it stores.

- Servers are collectively responsible for serial equivalence of operations.
Timestamped Ordering with Distributed Transaction

- Each server is responsible for applying concurrency control to objects it stores.

- Servers are collectively responsible for serial equivalence of operations.

- Timestamped ordering can be applied locally at each server:
  - When a server aborts a transaction, inform the coordinator which will relay the “abort” to other servers.
Locks with Distributed Transaction

- Each server is responsible for applying concurrency control to objects it stores.
- Servers are collectively responsible for serial equivalence of operations.
- Locks are held locally, and cannot be released until all servers involved in a transaction have committed or aborted.
- Locks are retained during 2PC (two-phase commit) protocol.
- How to handle deadlocks?
  - Next class!