Causal Consistency and Two-Phase Commit

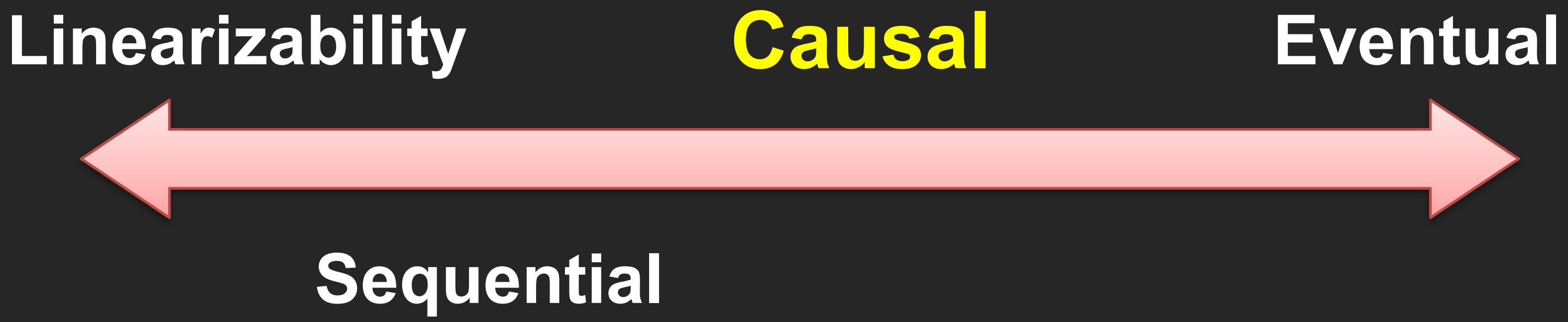
جامعة الملك عبدالله
للعلوم والتقنية
King Abdullah University of
Science and Technology

CS 240: Computing Systems and Concurrency
Lecture 16

Marco Canini

Credits: Michael Freedman and Kyle Jamieson developed much of the original material.

Consistency models



Linearizability

Causal

Eventual

Sequential

Recall use of logical clocks (lec 5)

- Lamport clocks: $C(a) < C(z)$ Conclusion: **None**
- Vector clocks: $V(a) < V(z)$ Conclusion: **a → … → z**
- Distributed bulletin board application
  - Each post gets sent to all other users
  - Consistency goal: No user to see reply before the corresponding original message post
  - Conclusion: Deliver message only **after** all messages that **causally precede** it have been delivered

Causal Consistency

1. Writes that are ***potentially*** causally related must be seen by all machines in same order.

2. Concurrent writes may be seen in a different order on different machines.

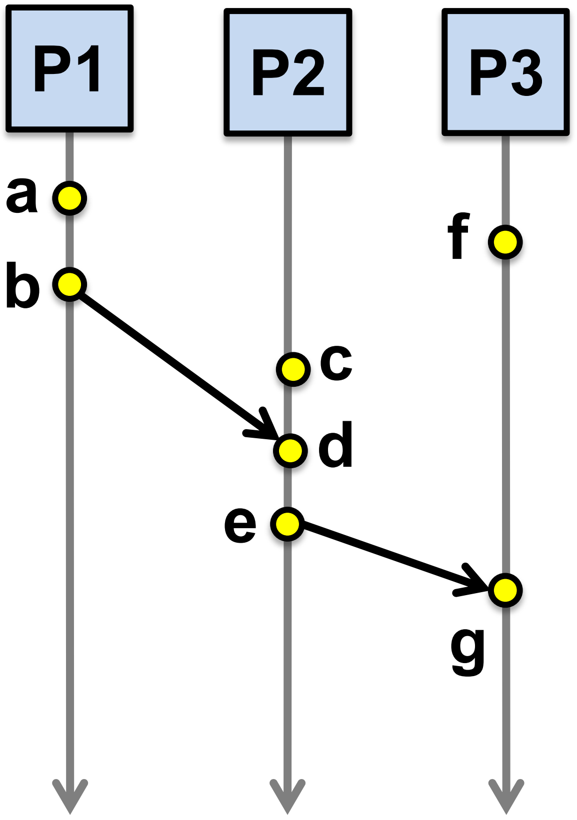
- Concurrent: Ops not causally related

Causal Consistency

1. Writes that are ***potentially*** causally related must be seen by all machines in same order.

2. Concurrent writes may be seen in a different order on different machines.

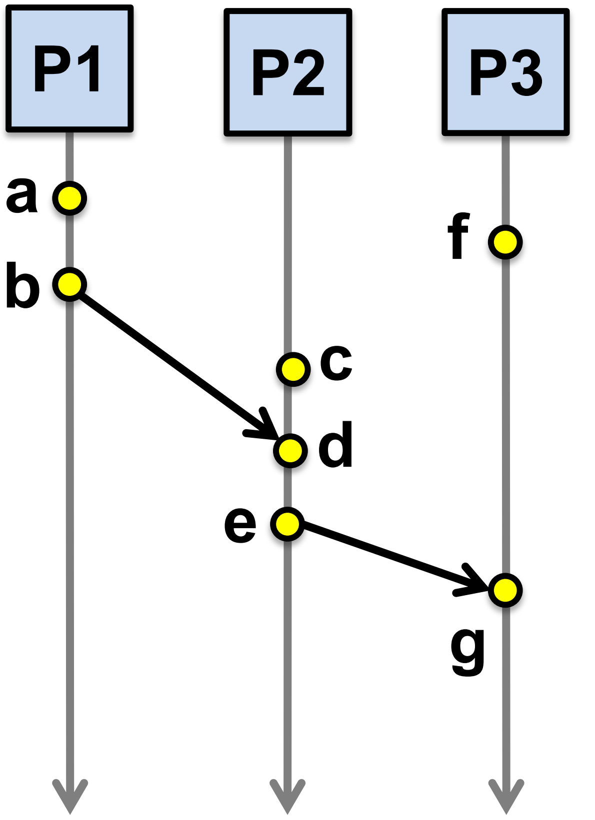
- Concurrent: Ops not causally related



Physical time ↓

Causal Consistency

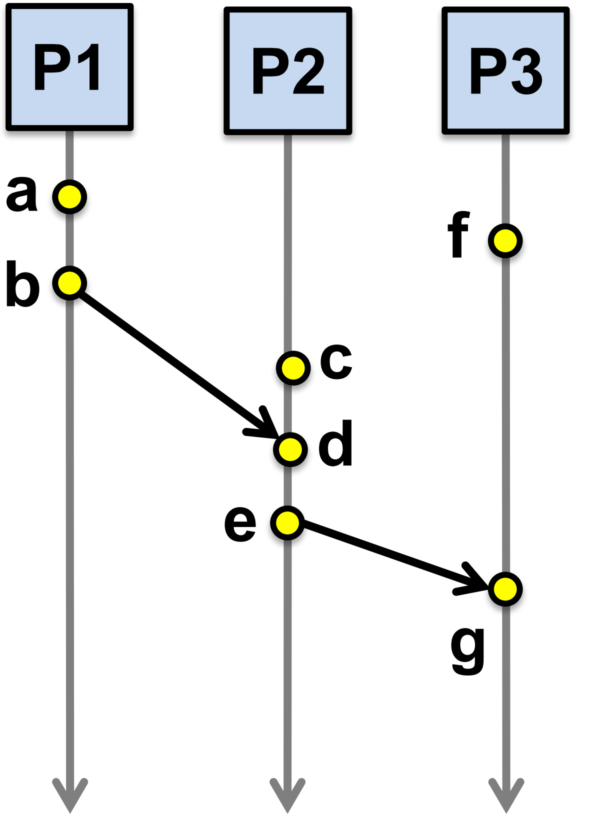
Operations	Concurrent?
a, b	
b, f	
c, f	
e, f	
e, g	
a, c	
a, e	



Physical time ↓

Causal Consistency

Operations	Concurrent?
a, b	**N**
b, f	**Y**
c, f	**Y**
e, f	**Y**
e, g	**N**
a, c	**Y**
a, e	**N**



Physical time ↓

Causal Consistency: Quiz

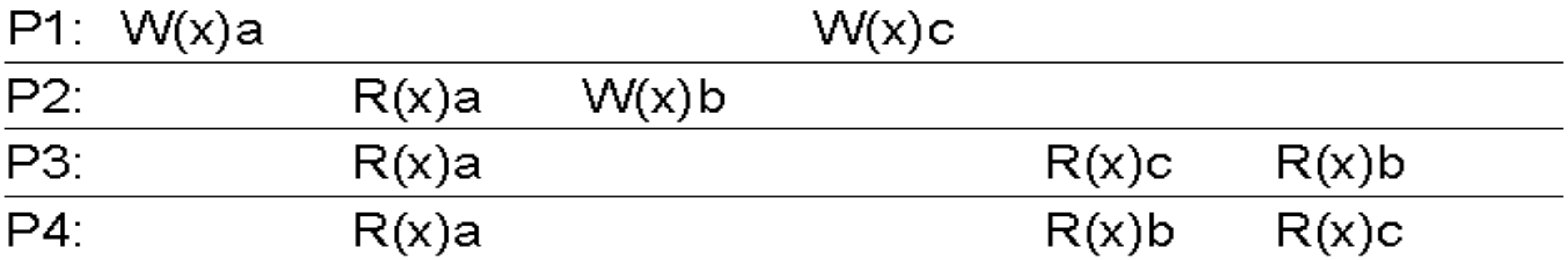
P1:	W(x)a			W(x)c		
P2:		R(x)a	W(x)b			
P3:		R(x)a			R(x)c	R(x)b
P4:		R(x)a			R(x)b	R(x)c

- Valid under causal consistency

- **Why?** *W(x)b* and *W(x)c* are concurrent
  - So all processes don't (need to) see them in same order

- P3 and P4 read the values 'a' and 'b' in order as potentially causally related. No 'causality' for 'c'.

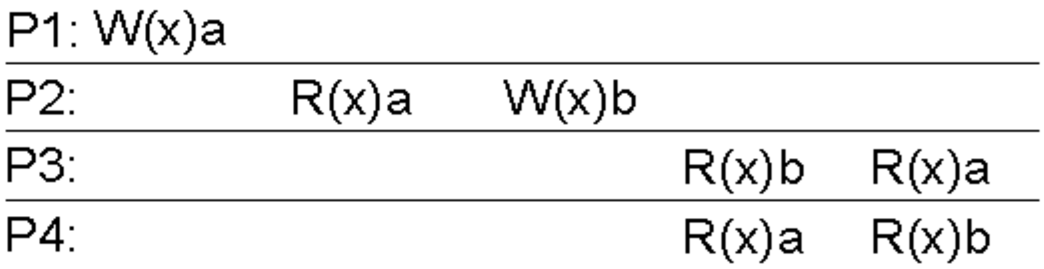
Sequential Consistency: Quiz



P1:	W(x)a			W(x)c		
P2:		R(x)a	W(x)b			
P3:		R(x)a			R(x)c	R(x)b
P4:		R(x)a			R(x)b	R(x)c

- Invalid under sequential consistency
- **Why?** P3 and P4 see b and c in different order
- But fine for causal consistency
  - B and C are not causally dependent
  - Write after write has no dep’s, write after read does

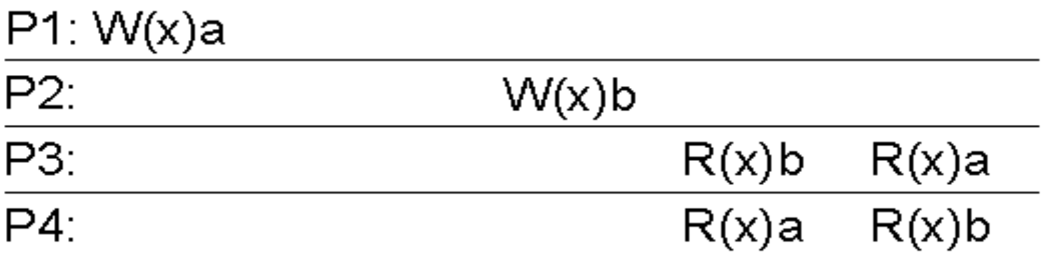
Causal Consistency



P1: W(x)a				
P2:	R(x)a	W(x)b		
P3:			R(x)b	R(x)a
P4:			R(x)a	R(x)b

(a)

✗



P1: W(x)a			
P2:	W(x)b		
P3:		R(x)b	R(x)a
P4:		R(x)a	R(x)b

(b)

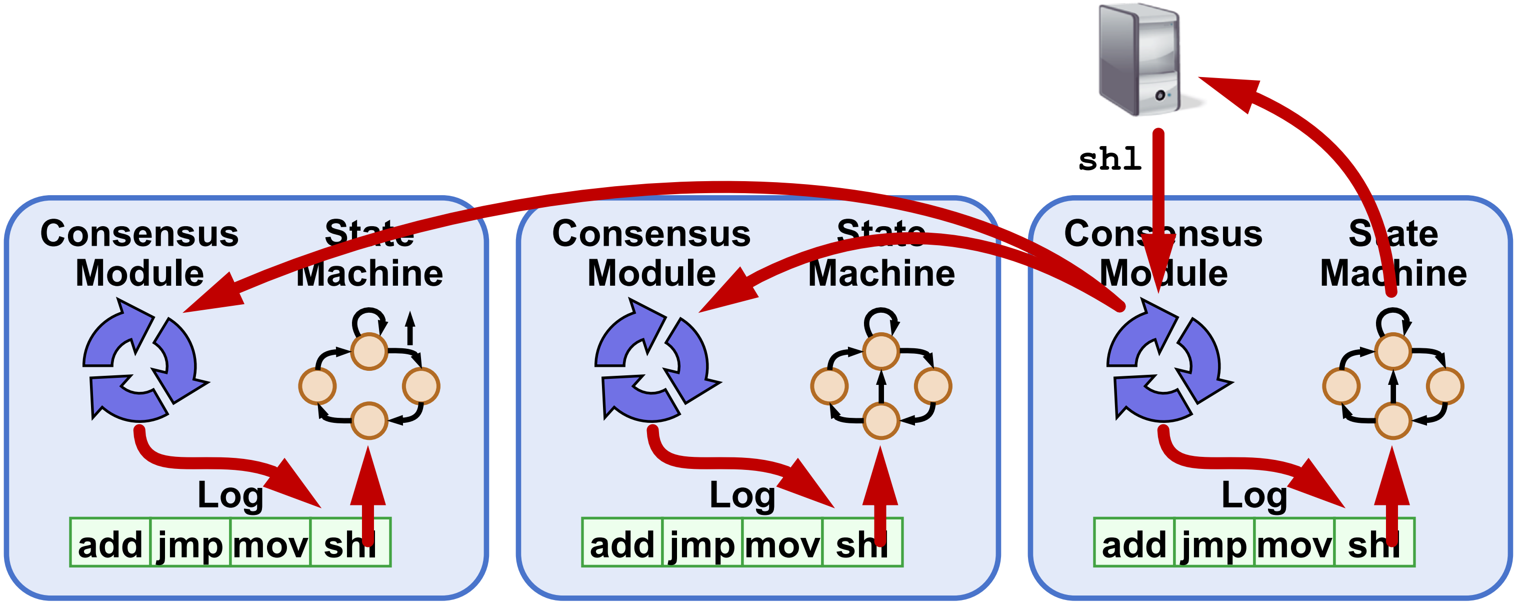
✓

A: Violation: W(x)b is potentially dep on W(x)a

B: Correct. P2 doesn't read value of a before W

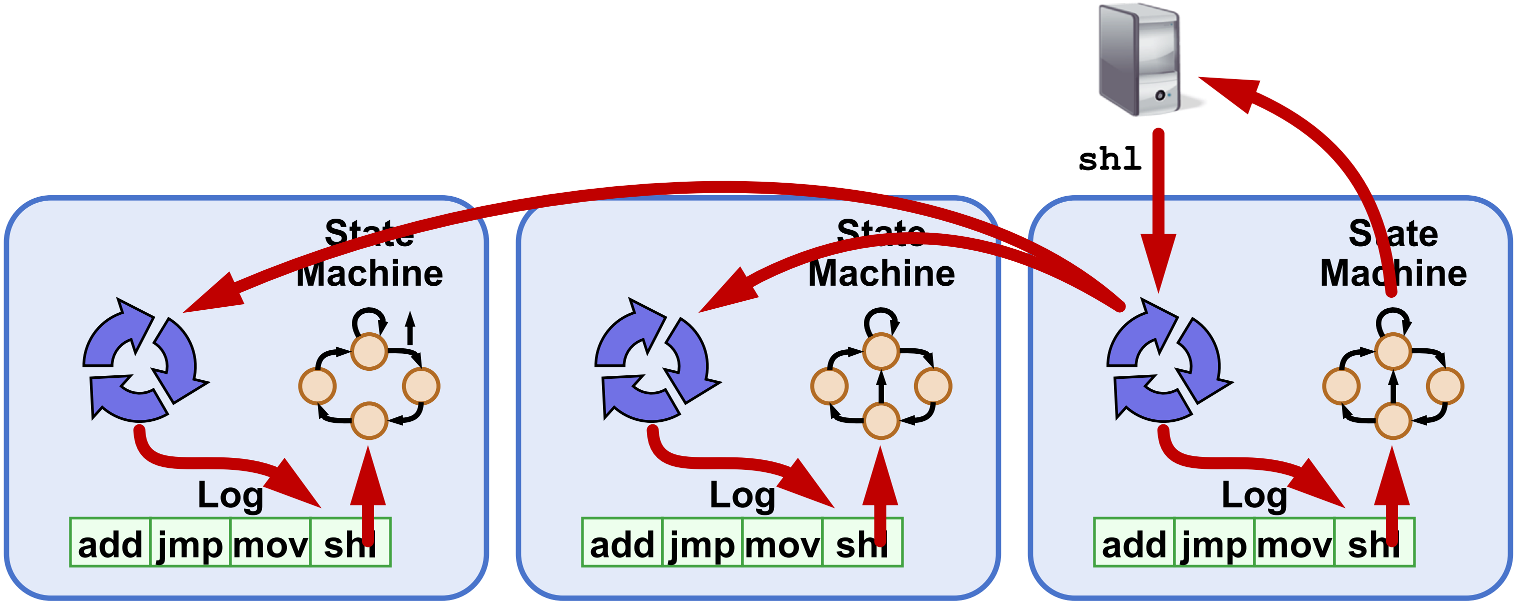
Causal consistency within replication systems

Implications of laziness on consistency



- Linearizability / sequential: Eager replication
- Trades off low-latency for consistency

Implications of laziness on consistency



- Causal consistency: Lazy replication
- Trades off consistency for low-latency
- Maintain local ordering when replicating
- Operations may be lost if failure before replication

Two-phase commit

Motivation: sending money

```
send_money(A, B, amount) {
   Begin_Transaction();
   if (A.balance - amount >= 0) {
       A.balance = A.balance - amount;
       B.balance = B.balance + amount;
       Commit_Transaction();
   } else {
       Abort_Transaction();
   }
}
```

Single-server: ACID

- **Atomicity**: all parts of the transaction execute or none (A’s decreases and B’s balance increases)
- **Consistency**: the transaction only commits if it preserves invariants (A’s balance never goes below 0)
- **Isolation**: the transaction executes as if it executed by itself (even if C is accessing A’s account, that will not interfere with this transaction)
- **Durability**: the transaction’s effects are not lost after it executes (updates to the balances will remain forever)

Distributed transactions?

- Partition databases across multiple machines for scalability (A and B might not share a server)
- A transaction might touch more than one partition
- How do we guarantee that all of the partitions commit the transactions or none commit the transactions?

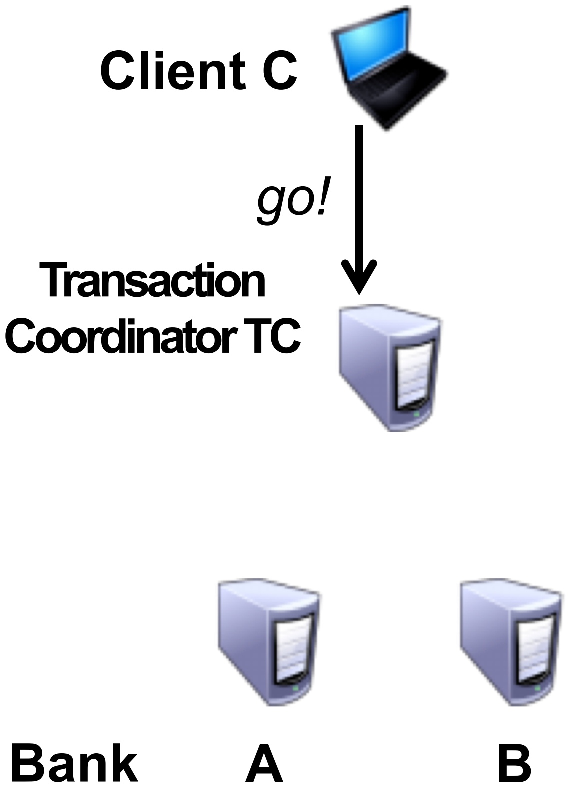
Two-Phase Commit (2PC)

- **Goal:** General purpose, distributed agreement on some action, with failures
  - Different entities play different roles in the action

- **Running example:** Transfer money from A to B
  - Debit at A, credit at B, tell the client “okay”
  - Require **both** banks to do it, or **neither**
  - Require that **one bank never act alone**

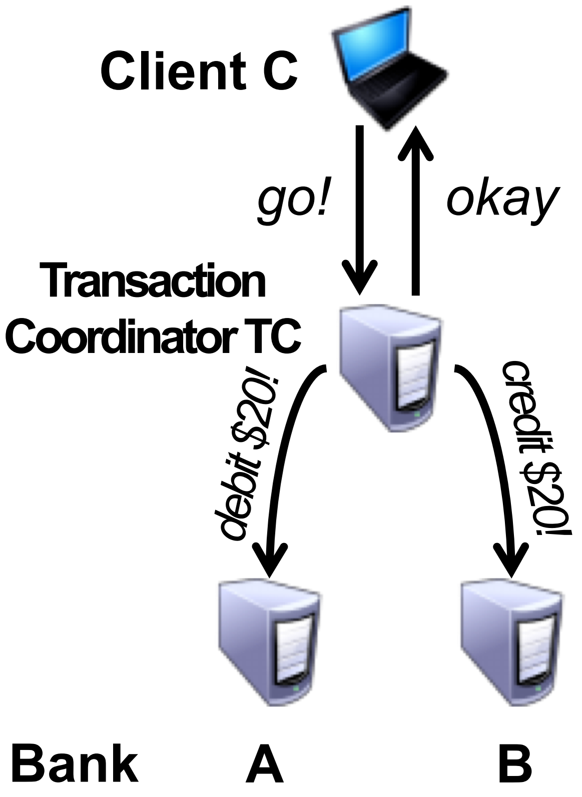
- This is an **all-or-nothing** atomic commit protocol
  - Later will discuss how to make it **before-or-after** atomic

Straw Man protocol



1. **C → TC**: *"go!"*

Straw Man protocol



1. **C → TC**: *"go!"*

2. **TC → A**: *"debit $20!"*
   TC → B: *"credit $20!"*
   TC → C: *"okay"*

- **A, B** perform actions on receipt of messages

Reasoning about the Straw Man protocol

What could **possibly** go wrong?

1. Not enough money in **A's** bank account?
2. **B's** bank account no longer exists?
3. **A** or **B** **crashes** before receiving message?
4. The best-effort network to **B** **fails**?
5. **TC** **crashes** after it sends *debit* to **A** but before sending to **B**?

Safety versus liveness

- Note that **TC**, **A**, and **B** each have a notion of committing

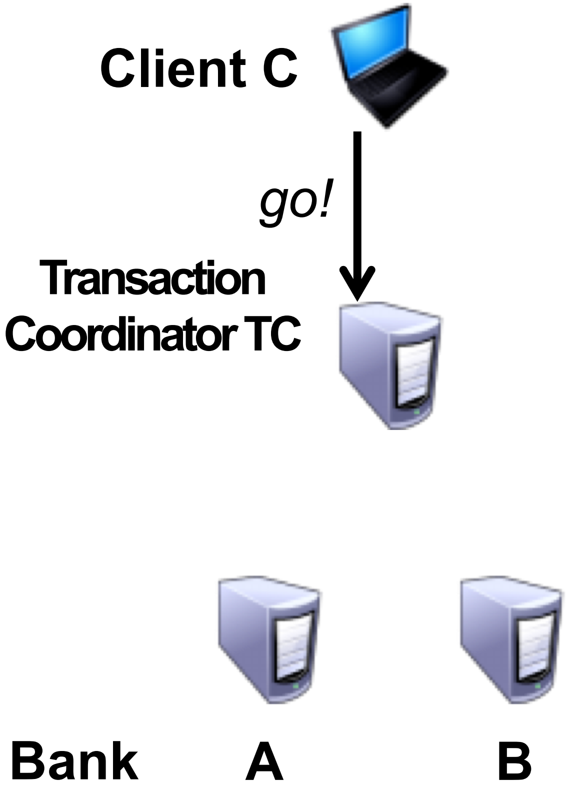
- We want two properties:

1. Safety
   - If **one commits, no one aborts**
   - If **one aborts, no one commits**

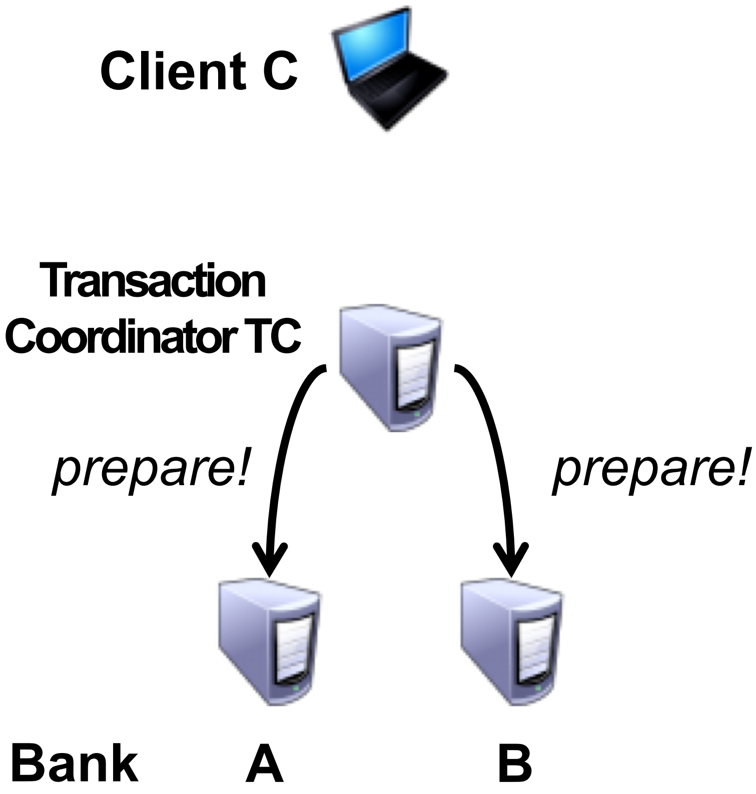
2. Liveness
   - If **no failures** and **A** and **B** can commit, **action commits**
   - If **failures,** reach a conclusion ASAP

A *correct* atomic commit protocol

1. C → TC: *"go!"*

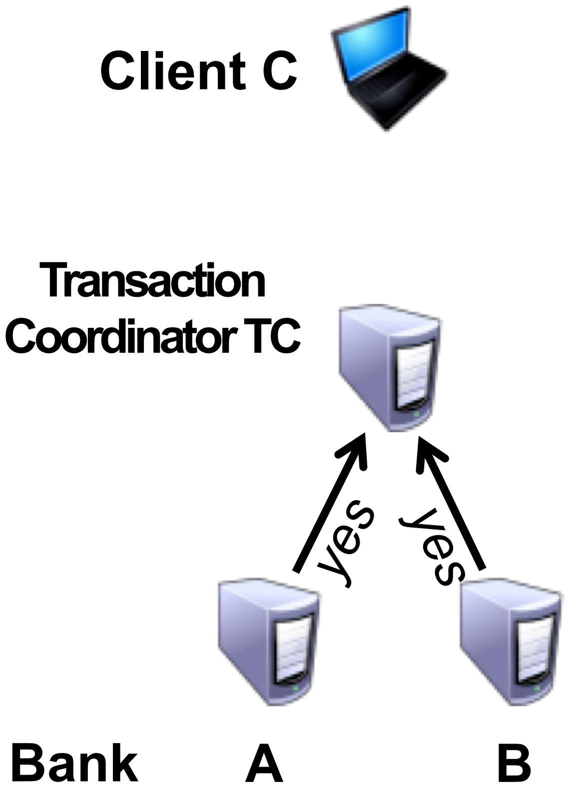


A *correct* atomic commit protocol



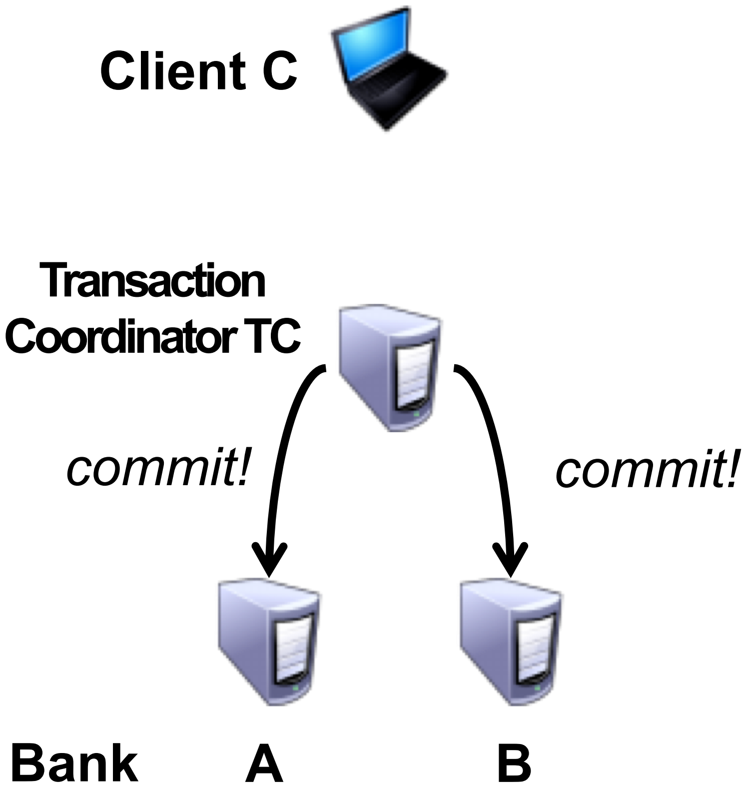
1. **C → TC:** *“go!”*
2. **TC → A, B:** *“prepare!”*

A *correct* atomic commit protocol



1. **C → TC:** *"go!"*
2. **TC → A, B:** *"prepare!"*
3. **A, B → P:** *"yes"* or *"no"*

A *correct* atomic commit protocol



Client C

Transaction Coordinator TC

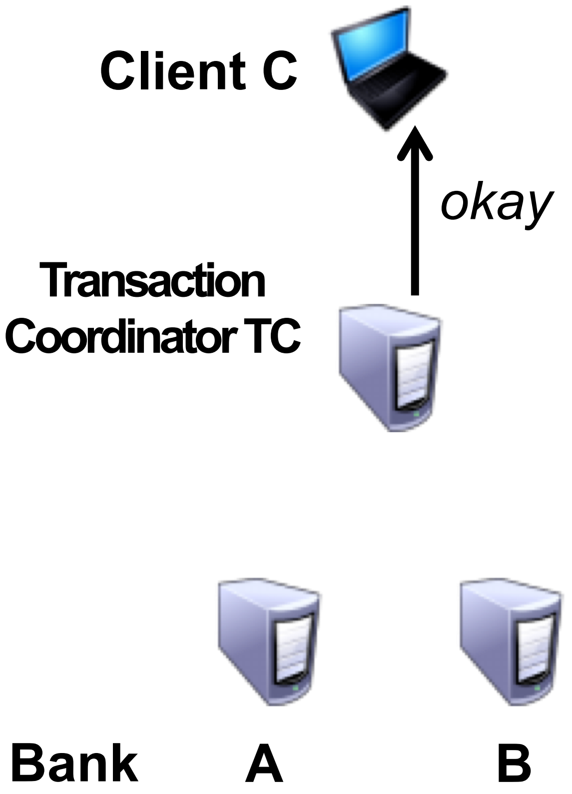
commit!

commit!

Bank **A** **B**

1. **C → TC:** *"go!"*
2. **TC → A, B:** *"prepare!"*
3. **A, B → P:** *"yes"* or *"no"*
4. **TC → A, B:** *"commit!"* or *"abort!"*
   - **TC** sends ***commit*** if **both** say *yes*
   - **TC** sends ***abort*** if **either** say *no*

A *correct* atomic commit protocol



1. **C → TC:** *"go!"*
2. **TC → A, B:** *"prepare!"*
3. **A, B → P:** *"yes"* or *"no"*
4. **TC → A, B:** *"commit!"* or *"abort!"*
   - **TC** sends ***commit*** if **both** say *yes*
   - **TC** sends ***abort*** if **either** say *no*
5. **TC → C:** *"okay" or "failed"*

- **A, B** commit on receipt of commit message

Reasoning about atomic commit

- *Why is this correct?*
  - – Neither can commit unless both agreed to commit

- *What about performance?*
  1. **Timeout:** I’m up, but didn’t receive a message I expected
     - Maybe other node crashed, maybe network broken
  2. **Reboot:** Node crashed, is rebooting, must clean up

Timeouts in atomic commit

Where do hosts **wait** for messages?

1. **TC** waits for "yes" or "no" from **A** and **B**
   - **TC** hasn't yet sent any commit messages, so can **safely abort** after a timeout
   - But this is **conservative:** might be network problem
     - We've preserved correctness, sacrificed performance

2. **A** and **B** wait for "commit" or "abort" from **TC**
   - If it sent a *no*, it can **safely abort** *(why?)*
   - If it sent a *yes*, can it unilaterally abort?
   - Can it unilaterally commit?
   - A, B could wait forever, but there is an alternative…

Server termination protocol

- Consider Server **B** (Server **A** case is symmetric) waiting for *commit* or *abort* from **TC**
  - Assume **B** voted *yes* (else, unilateral abort possible)

- **B → A:** "status?" **A** then replies back to **B.** Four cases:
  1. (No reply from **A**): no decision, **B** waits for **TC**
  2. Server **A** received commit or abort from **TC:** Agree with the **TC's** decision
  3. Server **A** hasn't voted yet or voted *no:* both **abort**
     - **TC** can't have decided to commit
  4. Server **A** voted *yes:* both must **wait** for the **TC**
     - **TC** decided to **commit** if both replies received
     - **TC** decided to **abort** if it timed out

Reasoning about the server termination protocol

- *What are the liveness and safety properties?*
  - **Safety**: if servers don't crash and network between A and B is reliable, all processes reach the same decision (in a finite number of steps)
  - **Liveness**: if failures are eventually repaired, then every participant will eventually reach a decision
- Can resolve **some** timeout situations with guaranteed correctness
- Sometimes however **A** and **B** must block
  - Due to failure of the **TC** or network to the **TC**
- But what will happen if **TC, A,** or **B** **crash and reboot?**

How to handle crash and reboot?

- Can't back out of commit if already decided
  - **TC** crashes just after sending *"commit!"*
  - **A** or **B** crash just after sending *"yes"*

- If all nodes knew their state before crash, we could use the termination protocol…
  - Use **write-ahead log** to record *"commit!" and "yes"* to disk

Recovery protocol with non-volatile state

- If everyone rebooted and is reachable, TC can just check for **commit** record on disk and **resend** action
- **TC:** If no **commit** record on disk, **abort**
  - You didn't send any *"commit!"* messages
- **A, B:** If no **yes** record on disk, **abort**
  - You didn't vote *"yes"* so **TC** couldn't have committed
- A, B: If **yes** record on disk, execute termination protocol
  - This might block

Two-Phase Commit

- This recovery protocol with non-volatile logging is called ***Two-Phase Commit (2PC)***
- **Safety:** All hosts that decide reach the same decision
  - No commit unless everyone says “yes”
- **Liveness:** If no failures and all say “yes” then commit
  - **But if failures then 2PC might block**
  - **TC must be up to decide**
- **Doesn’t tolerate faults well: must wait for repair**

Next topic

Concurrency Control