2PC

CS6450: Distributed Systems

Lecture 7

Ryan Stutsman

Lab 1

- Due tonight (9/11) before midnight
- Push your solution into a branch on your gitlab repo called "lab1"
- Our scripts will pick it up from there

Paxos + Raft Videos

- Make sure to watch the Paxos video before Thu
- On Thu, we’ll discuss “Paxos Made Simple” (required reading)
- On Tue, we’ll discuss Raft (required reading)
- We’ll want to compare a bit between the protocols

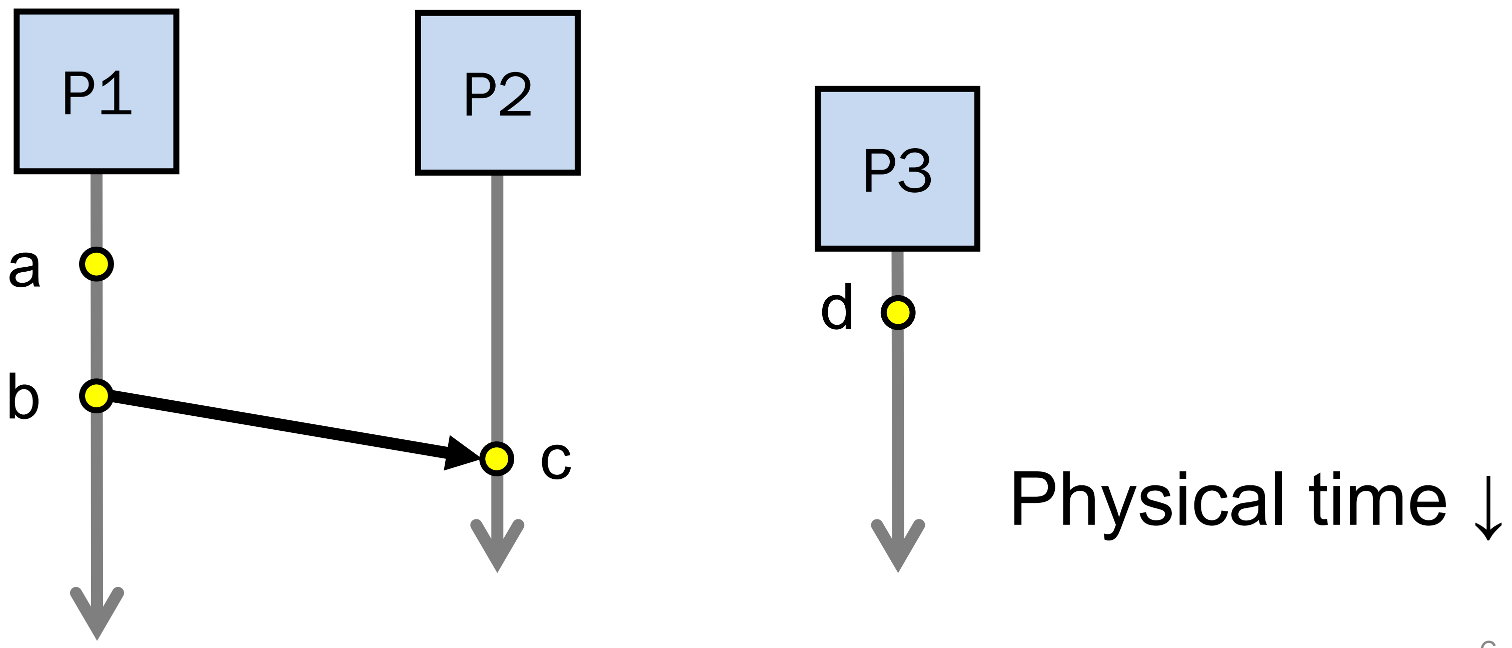
Takeaways

- Vector Clocks
  - If $V(a) < V(b)$ then $a \rightarrow b$
  - If $V(a) \nless V(b)$ and $V(b) \nless V(a)$ then $a \parallel b$
  - Can use to infer when an event b was aware of/influenced by a

Takeaways

- 2PC is a common protocol for distributed commitment
  - 2 phases: prepare phase, commit phase
  - Provides atomicity, though often used with locks to provide isolation and consistency, typically requires logging to survive faults so provides durability
  - Among several sites, all apply or none apply
- 2PC is blocking and can fail to terminate under loss of a single process
  - If transaction coordinator is gone participants may not be able to determine commit/abort decision alone

Happens Before



Lamport clocks: Objective

- We seek a ***clock time C***(**a**) for every event **a**

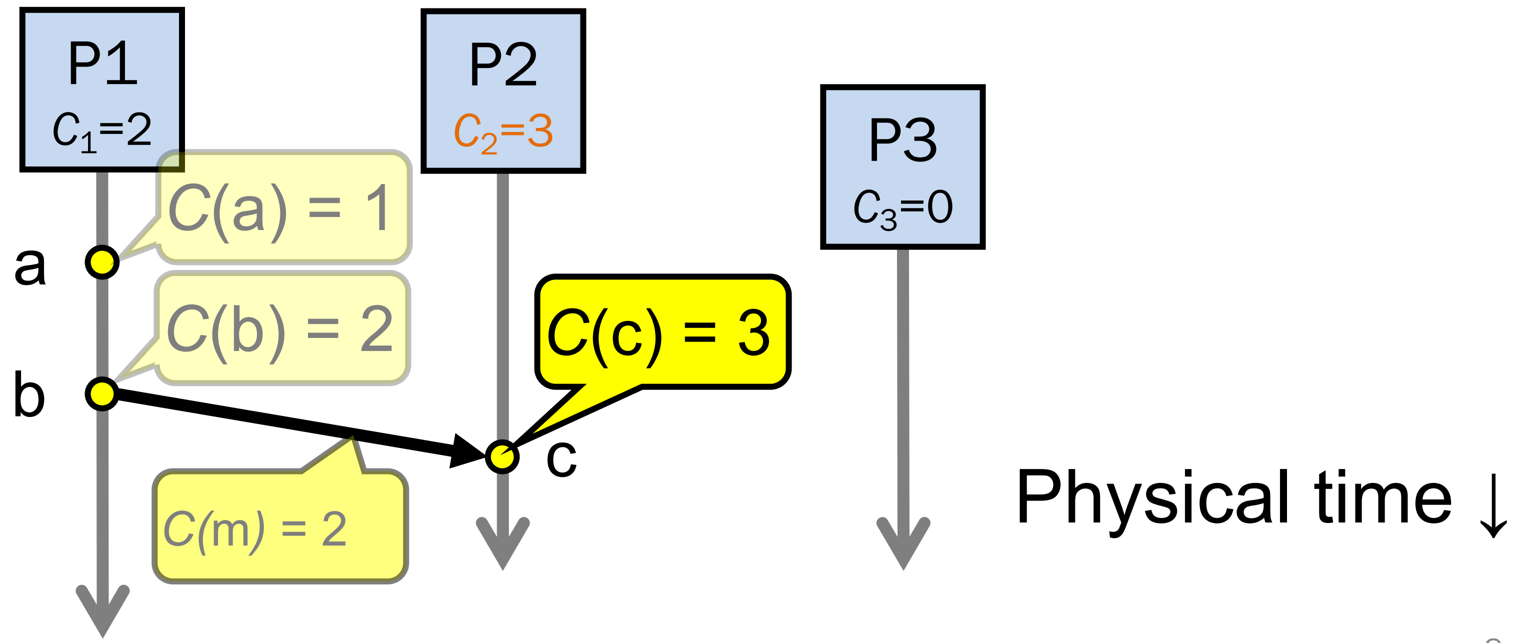
Plan: Tag events with clock times; use clock times to make distributed system correct

- Clock condition: If **a** → **b**, then *C*(**a**) < *C*(**b**)

The Lamport Clock algorithm

3. On process **P**$_j$ receiving a message **m**:

- Set C_j and receive event time $C(\mathbf{c}) \leftarrow 1 + \mathbf{max}\{ C_j, C(\mathbf{m}) \}$



Ordering all events

- Break ties by **appending the process number** to each event:

  1. Process $\mathbf{P}_i$ timestamps event **e** with $C_i(\mathbf{e}).i$

  2. $C(\mathbf{a}).i < C(\mathbf{b}).j$ when:
     - $C(\mathbf{a}) < C(\mathbf{b})$, or $C(\mathbf{a}) = C(\mathbf{b})$ and $i < j$

- Now, for any two events **a** and **b**, $C(\mathbf{a}) < C(\mathbf{b})$ or $C(\mathbf{b}) < C(\mathbf{a})$
  - This is called a total ordering of events

Take-away points: Lamport clocks

- Can **totally-order** events in a distributed system: that's useful!

- **But:** while by construction, **a** → **b** implies C(**a**) < C(**b**),
  - The converse is not necessarily true:
    - C(**a**) < C(**b**) does not imply **a** → **b** (possibly, **a** | | **b**)

Can't use Lamport clock timestamps to infer causal relationships between events

Vector clock (VC)

- Label each event **e** with a vector $V(\mathbf{e}) = [c_1, c_2 \ldots, c_n]$
  - c_i is a count of events in process i that causally precede **e**

- Initially, all vectors are [0, 0, …, 0]

- Two **update rules:**

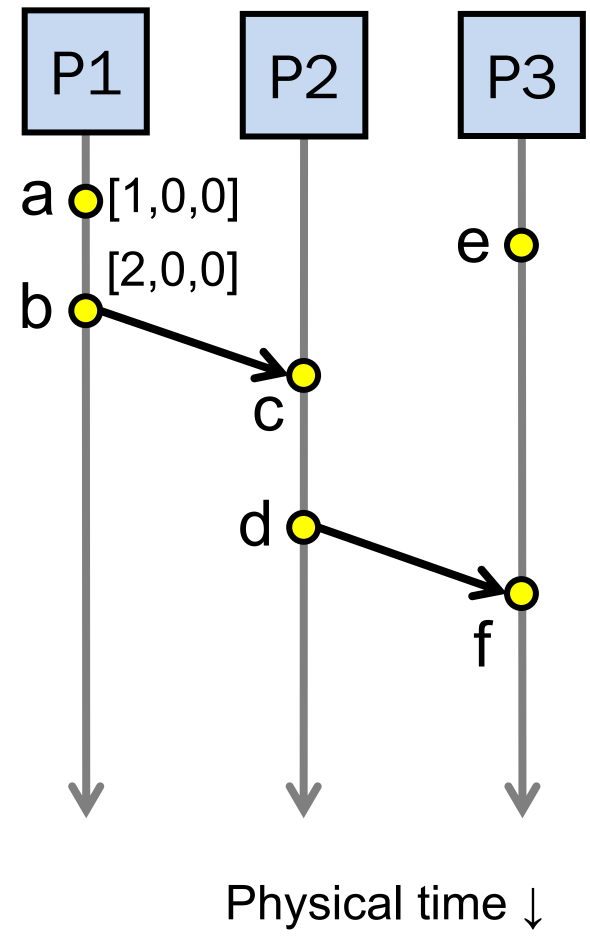
1. For each **local event** on process i, increment local entry c_i

2. If process j **receives** message with vector $[d_1, d_2, \ldots, d_n]$:
   - Set each local entry $c_k = \max\{c_k, d_k\}$
   - Increment local entry c_j

Vector clock: Example

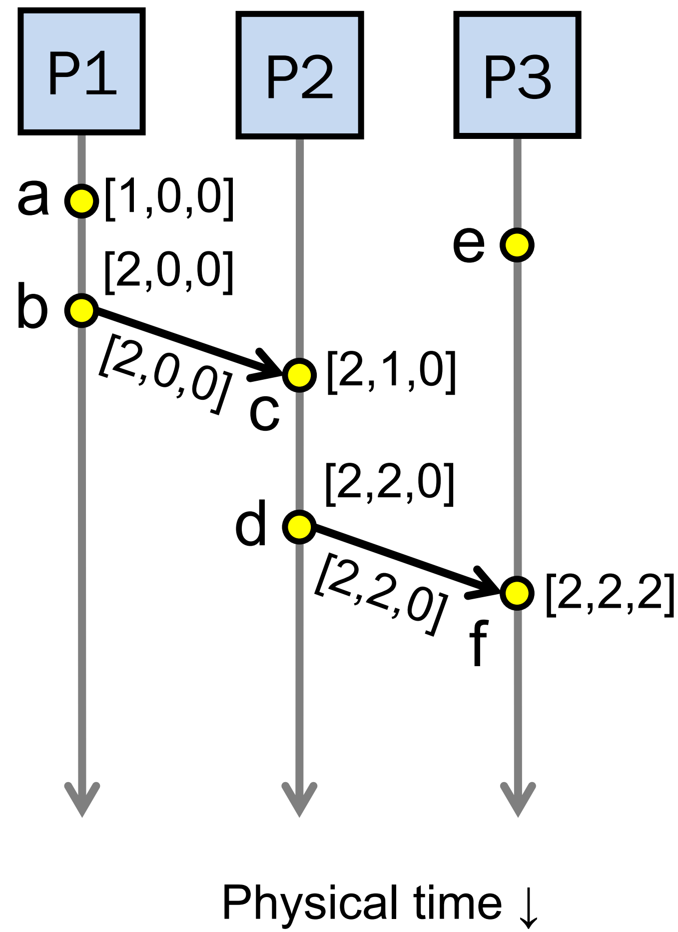
- All counters start at [0, 0, 0]

- Applying local update rule



Vector clock: Example

- All counters start at [0, 0, 0]
- Applying local update rule
- Applying message rule
  - Local vector clock **piggybacks** on inter-process messages

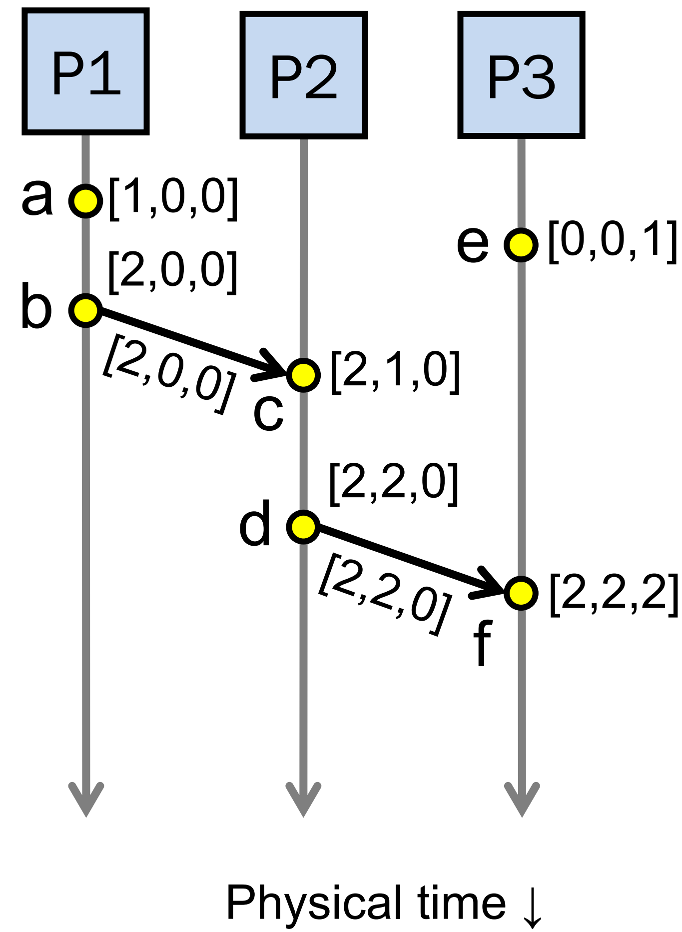


Vector clock: Example

- All counters start at [0, 0, 0]

- Applying local update rule

- Applying message rule
  - Local vector clock **piggybacks** on inter-process messages

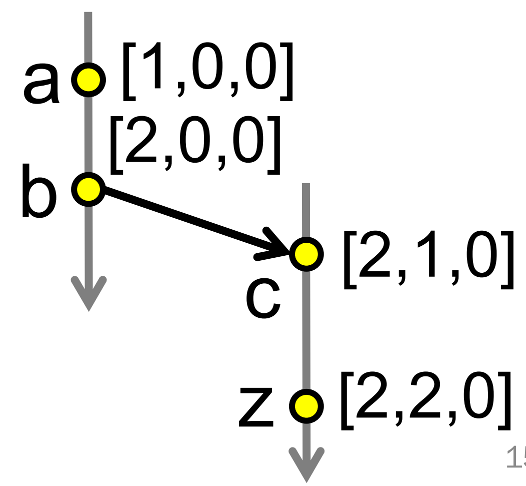


Vector clocks can establish causality

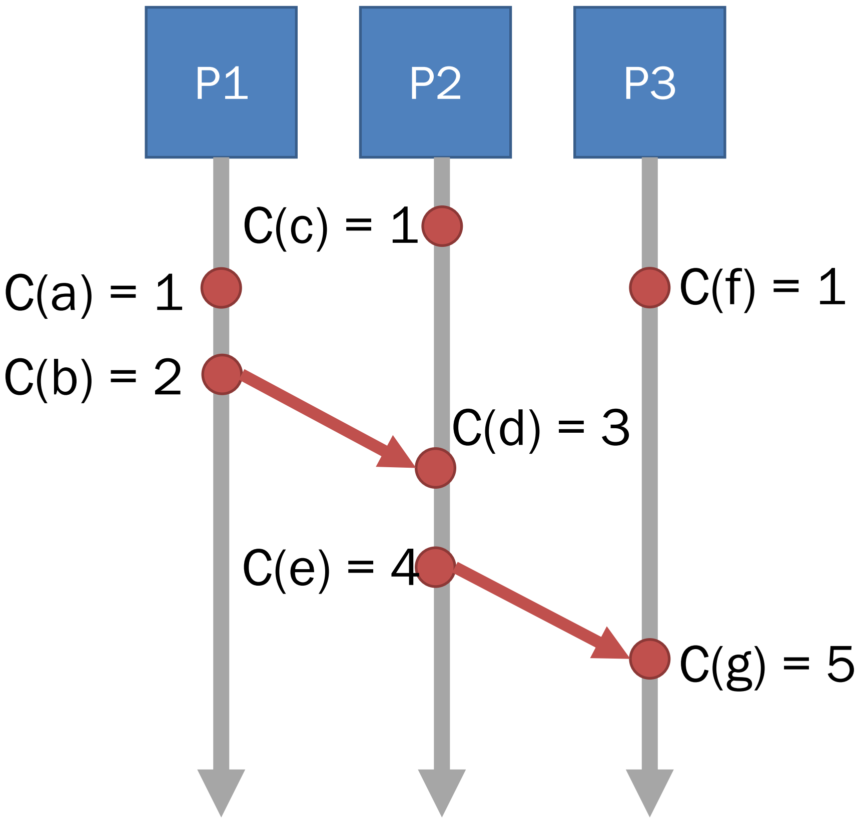
- Rule for comparing vector clocks:
  - V(**a**) = V(**b**) when $\mathbf{a}_k = \mathbf{b}_k$ for all *k*
  - V(**a**) < V(**b**) when $\mathbf{a}_k \leq \mathbf{b}_k$ for all *k* and V(**a**) ≠ V(**b**)

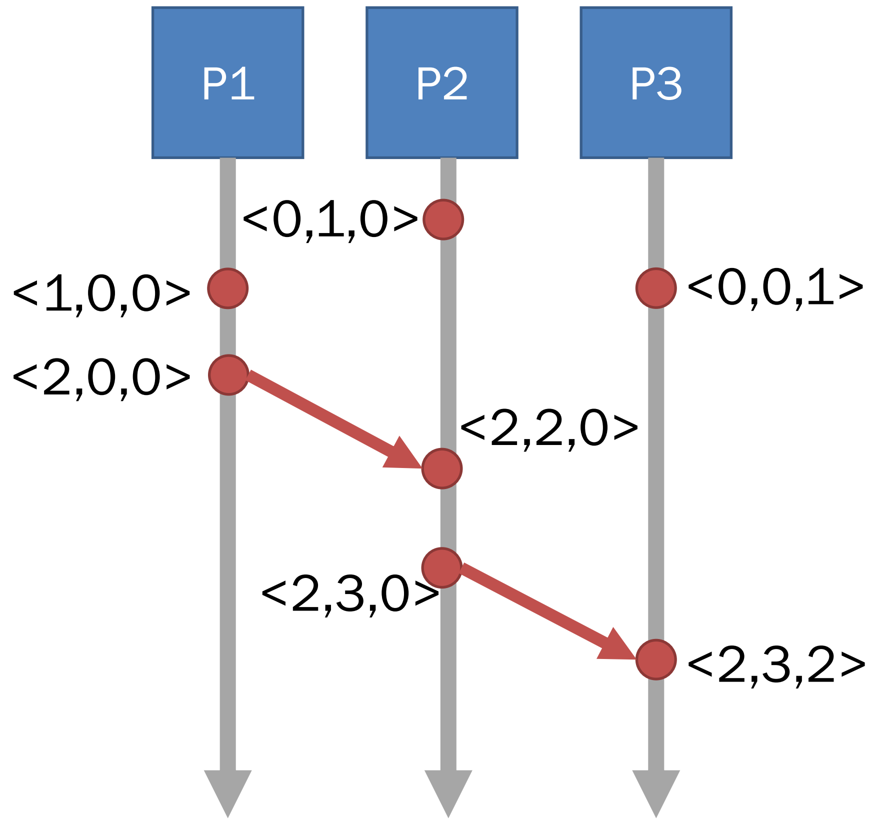
- **Concurrency:** *a* || *b* if $\mathbf{a}_i < \mathbf{b}_i$ and $\mathbf{a}_j > \mathbf{b}_j$, some *i*, *j*

- V(a) < V(z) **when** there is a chain of events linked by → between a and z



Lamport vs Vector Clocks





$a \longrightarrow g$? Yes, $V(a) < V(g)$

$f \longrightarrow e$? No, $V(f) \not< V(e)$

Two events **a, z**

Lamport clocks: C(a) < C(z)

Conclusion: None

Vector clocks: V(a) < V(z)

Conclusion: a → … → z

Vector clock timestamps tell us about causal event relationships

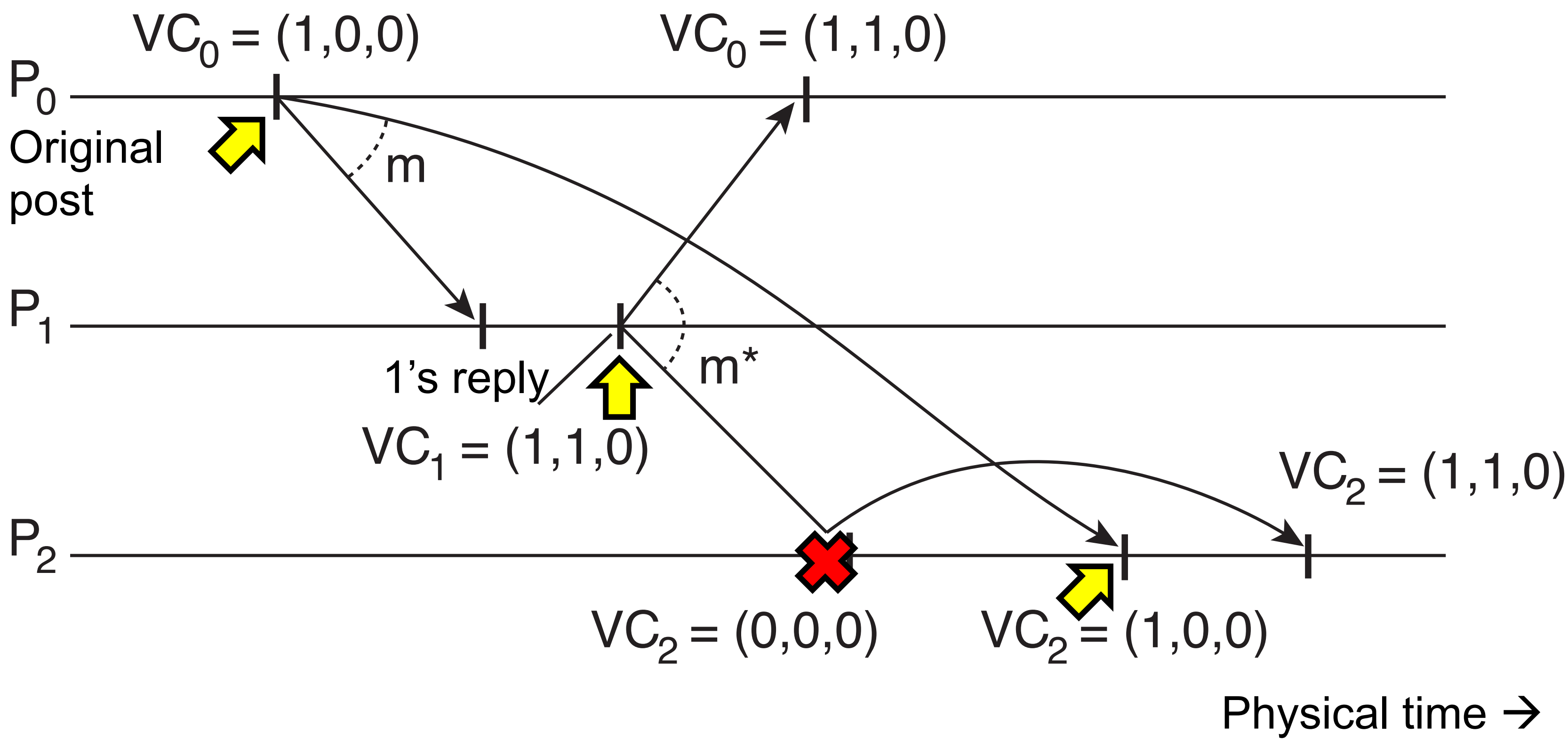
VC application: Causally-ordered bulletin board system

- Distributed bulletin board application
  - Each post → multicast of the post to all other users

- **Want:** No user to see a reply before the corresponding original message post

- Deliver message only **after** all messages that **causally precede** it have been delivered
  - Otherwise, the user would see a reply to a message they could not find

VC application: Causally-ordered bulletin board system



- User 0 posts, user 1 replies to 0's post; user 2 observes

Coming Up

- We'll see Lamport and Vector Clocks more soon
- Bayou
  - Need to ensure timestamps obey happens-before
  - Physical clocks with Lamport clock update rules
- Dynamo
  - Track which updates are (causally) concurrent
  - "Ask" application code how to reconcile
  - Avoid asking if it is unambiguous by happens-before

Safety and liveness

Reasoning about fault tolerance

- This is hard!
  - How do we design fault-tolerant systems?
  - How do we know if we're successful?
- Often use "properties" that hold true for every possible execution
- Often we focus on safety and liveness properties

Safety

- “Bad things” never happen
  - Invariants not violated
  - No execution can result in specific conditions
- Examples
  - Mutual exclusion: 0 or 1 servers act as primary in a given view
  - Garbage collection: no object is discarded while a reference to it still exists
  - Atomic commit: a client can never observe a partially applied effect

Liveness

- "Good things" happen…eventually
- Examples
  - Eventual consistency: if servers can communicate, then they will eventually agree/converge on a value
  - Termination: eventually servers will decide a value

Often a tradeoff

- “Good” and “bad” are application-specific
- Often impossible to achieve all desired objectives and have to tradeoff safety/liveness guarantees
  - 2PC: guarantees atomicity at risk of blocking forever
  - Paxos: guarantees safety at risk of taking an infinite amount of time to decide outcome

Two-phase commit

Motivation: Meeting for Dinner

- Text friends, ask them to meet for dinner at 7 PM
- Some say yes one says no, can only make it at 6
- Text all, saying “just kidding, let’s do 6 PM”
- Now one friend that said yes to 7 PM doesn’t send a response… Ugh.

Another Motivation: Bank Transfers

```
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();
   }
}
```

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 persist)

Distributed transactions?

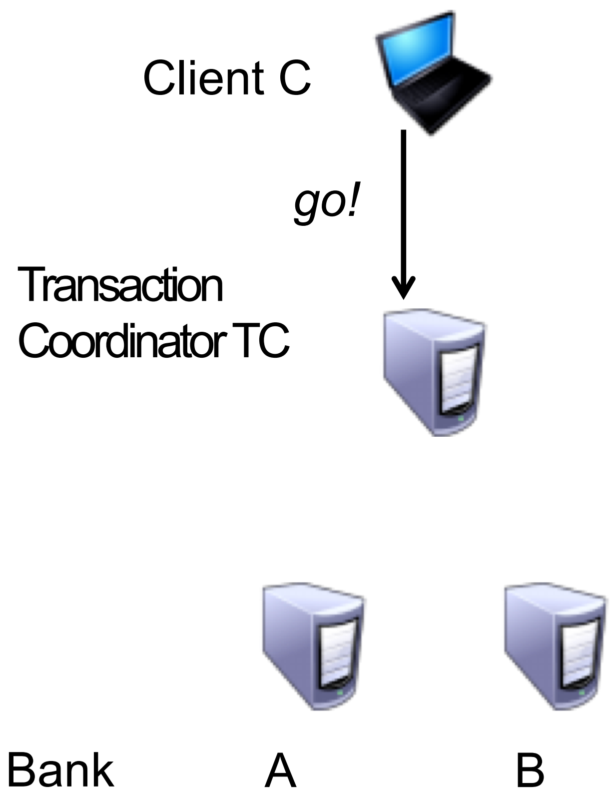
- Partition databases across multiple machines for scalability (A and B might not share a server)
  - Can’t put all Google Calendars or bank account on the same machine
- A transaction might touch more than one partition
- How do we guarantee that all of the partitions commit the transactions or none commit the transactions?

Two-Phase Commit (2PC)

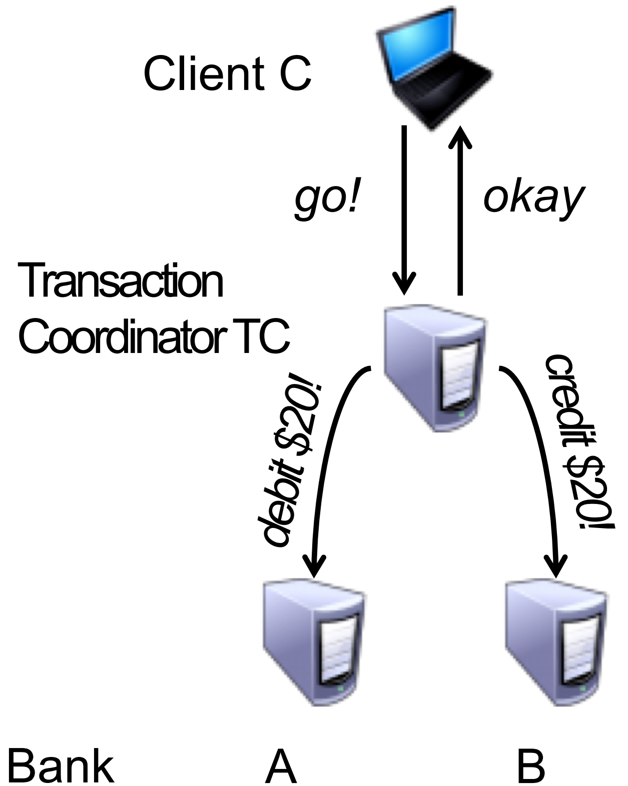
- **Goal:** General purpose, distributed agreement on some action, with failures
  - Different entities play different roles in the action
- **Running example:** Transfer money from A to B
  - Debit at A, credit at B, tell the client "okay"
  - Require **both** banks to do it, or **neither**
  - Require that **one bank never act alone**
- This is an **all-or-nothing** atomic commit protocol

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?

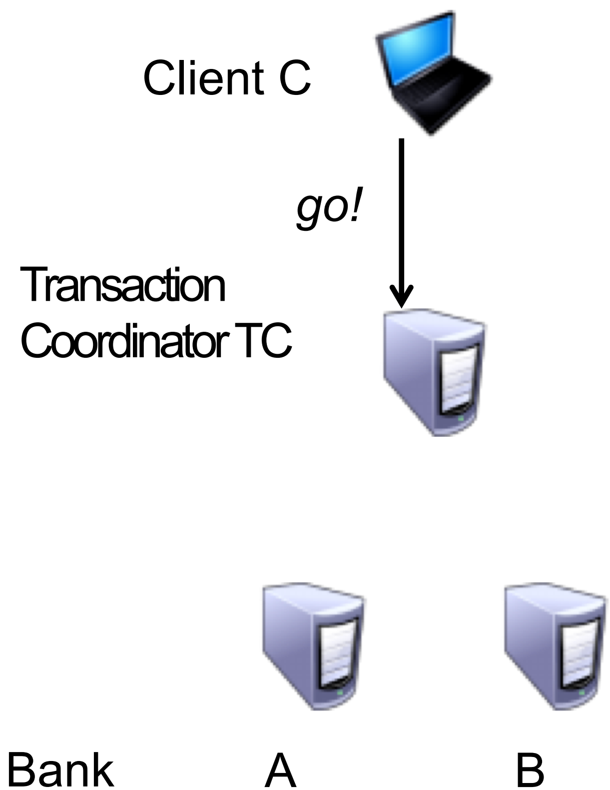
Safety versus liveness

- Note that TC, A, and B each have a notion of committing
- We want two properties:

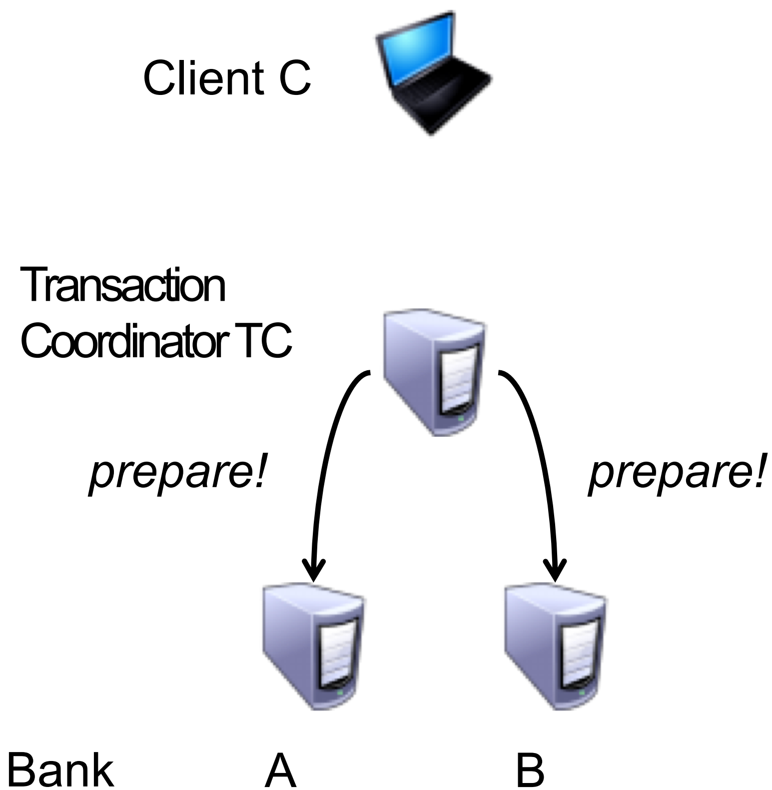
1. Safety
    - If one commits, no one aborts
    - If one aborts, no one commits
2. Liveness
    - If no failures and A and B can commit, action commits
    - If failures, reach a conclusion ASAP

A *correct* atomic commit protocol

1. C → TC: *"go!"*



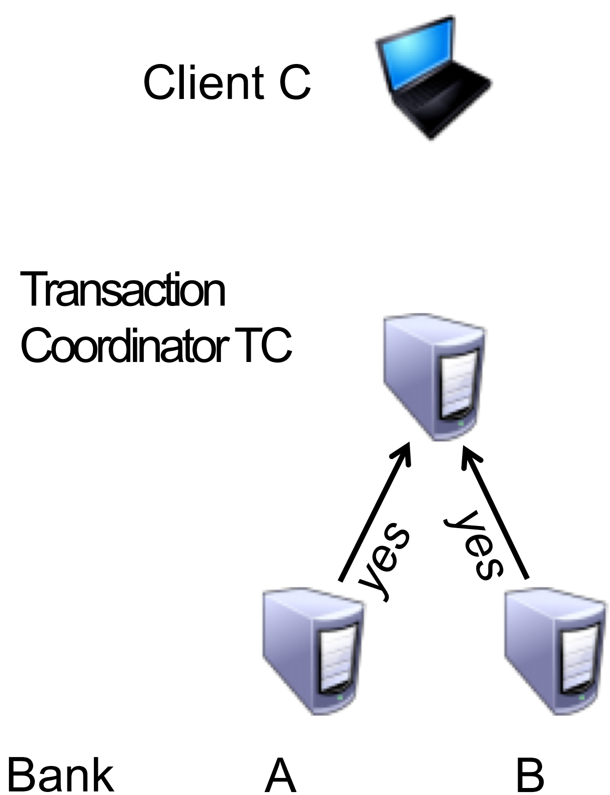
A *correct* atomic commit protocol



1. C → TC: *"go!"*

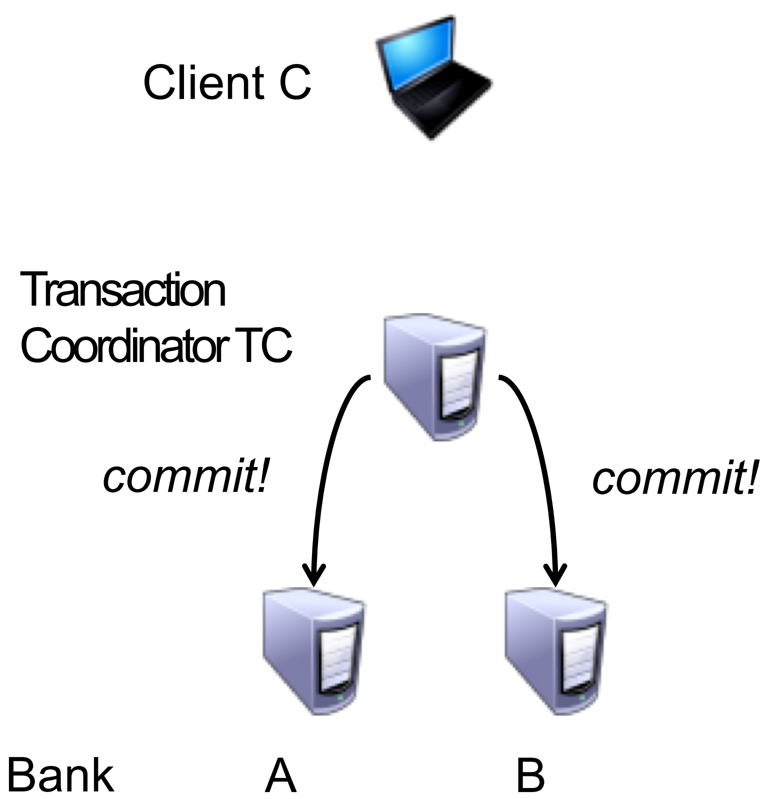
2. TC → A, B: *"prepare!"*

A *correct* atomic commit protocol



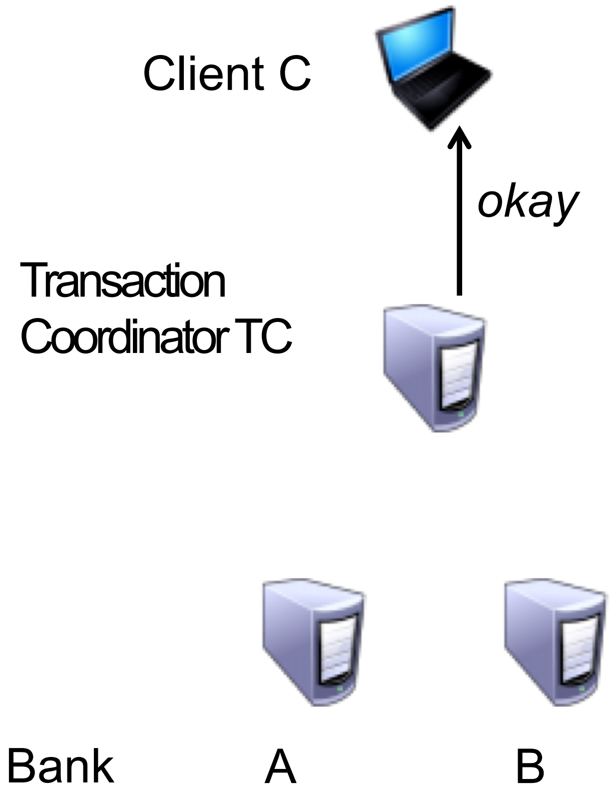
1. C → TC: *"go!"*
2. TC → A, B: *"prepare!"*
3. A, B → P: *"yes"* or *"no"*

A *correct* atomic commit protocol



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 progress?*
  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 progress
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:
  - (No reply from **A**): **B** must wait for **TC** or **A**
  - Server **A** received commit or abort from **TC:** Agree with the **TC's** decision
  - Server **A** hasn't voted yet or voted *no:* both **abort**
    - **TC** can't have decided to commit
  - 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 termination protocol

- *What are the liveness and safety properties?*
  - Safety: if servers don't crash, all processes will reach the same decision
  - 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
- 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/is reachable, TC can check for **commit** record on disk and **resend** action
- **TC:** If no **commit** record on disk, **abort**
  - It didn't send any *"commit"* messages
- **A, B:** If no **yes** record on disk, **abort**
  - It didn't vote *"yes"* so **TC** couldn't have committed
- A, B: If **yes** record on disk, execute termination protocol
  - This might block if TC isn't reachable

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
  - A *blocking* protocol
  - Failure of any process can result in non-progress
  - And it can transitively hold up resources

Why Blocking Matters

- Not surprising: failure of any process that hosts a partition of state results in blocking
- In the “prepare” phase participants must commit resources needed to execute the commit
  - For bank transfer, A must lock account to ensure that balance doesn’t change after promising to debit
- Irritation: if B participant fails, A cannot allow new operations on the account, so blocking affects future ops on non-failed participants…

Eliminate Unilateral TC Abort?

- What if the TC cannot unilaterally abort a transaction?
- Then if B checks with A and finds both “yes”, then participants could safely commit
- But, now TC and B can’t get unstuck if A doesn’t respond
- Used sometimes in 2PC,
  but not for well-suited to one of its earliest uses
  - Transaction code is intermixed with execution of 2PC
  - Think of TC executing a database transaction
  - It may abort for app level reasons (negative balance)

Example: Bank Transfers

```
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();
   }
}
```

Need to guarantee isolation to be safe, so resources/locks held starting here.

TC must be able to abort if it implements the conditional logic

But, this also means participants may stall if TC goes away