Two-Phase Commit

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

CS 240: Computing Systems and Concurrency
Lecture 10

Marco Canini

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

Plan

- Safety and liveness properties

- Two-phase commit

Safety and liveness properties

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
- We focus on **safety** and **liveness** properties

Properties

- **Property**: a predicate that is evaluated over a run of the system (a **trace**)
  - “every message that is received was previously sent”
- Not everything you may want to say about a system is a property:
  - “the program sends an average of 50 messages in a run”

Safety properties

- “Bad things” don’t happen, ever
  - No more than k processes are simultaneously in the critical section
  - Messages that are delivered are delivered in causal order
- A safety property is “prefix closed”:
  - if it holds in a run, it holds in every prefix

Liveness properties

- “Good things” eventually happen
  - A process that wishes to enter the critical section eventually does so
  - Some message is eventually delivered
  - Eventual consistency: if a value doesn’t change, two servers will eventually agree on its value
- Every run can be extended to satisfy a liveness property
  - If it does not hold in a prefix of a run, it does not mean it may not hold eventually

Often a trade-off

- “Good” and “bad” are application-specific

- Safety is very important in banking transactions
  - May take some time to confirm a transaction

- Liveness is very important in social networking sites
  - See updates right away

Two-phase commit

Objective

- Reach agreement for distributed transactions in the presence of failures

- **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

Model

- For each distributed transaction T:
  - one transaction coordinator (TC)
  - a set of participants
- Coordinator knows participants; participants don’t necessarily know each other
- Each process has access to a Distributed Transaction Log (DT-Log) on stable storage

The setup

- Each process p_i has an input value $vote_i$:
  - $vote_i \in$ {Yes, No}

- Each process p_i has output value $decision_i$:
  - $decision_i \in$ {Commit, Abort}

Atomic Commit (AC) specification

- **AC-1:** All processes that reach a decision reach the same one
- **AC-2:** A process cannot reverse its decision after it has reached one
- **AC-3:** The Commit decision can only be reached if all processes vote Yes
- **AC-4:** If there are no failures and all processes vote Yes, then the decision will be Commit
- **AC-5:** If all failures are repaired and there are no more failures, then all processes will eventually decide

Atomic Commit (AC) specification

- **AC-1:** All processes that reach a decision reach the same one

- We do not require all processes to reach a decision
- We do not even require all correct processes to reach a decision (impossible to accomplish if links fail)

- **AC-4:** If there are no failures and all processes vote Yes, then the decision will be Commit

- **AC-5:** If all failures are repaired and there are no more failures, then all processes will eventually decide

Atomic Commit (AC) specification

- **AC-1:** All processes that reach a decision reach the same one

- Avoids triviality
- Allows Abort even if all processes have voted yes

- **AC-4:** If there are no failures and all processes vote Yes, then the decision will be Commit

- **AC-5:** If all failures are repaired and there are no more failures, then all processes will eventually decide

Atomic Commit (AC) specification

- **AC-1:** All processes that reach a decision reach the same one
- **AC-2:** A process cannot reverse its decision after it has reached one
- **AC-3:** The Commit decision can only be reached if all processes vote Yes
- **AC-4:** If there are no failures and all processes vote Yes, then the decision will be Commit
- **AC-5:** If all failures are repaired and there are no more failures, then all processes will eventually decide

Note: A process that does not vote Yes
can unilaterally abort

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)

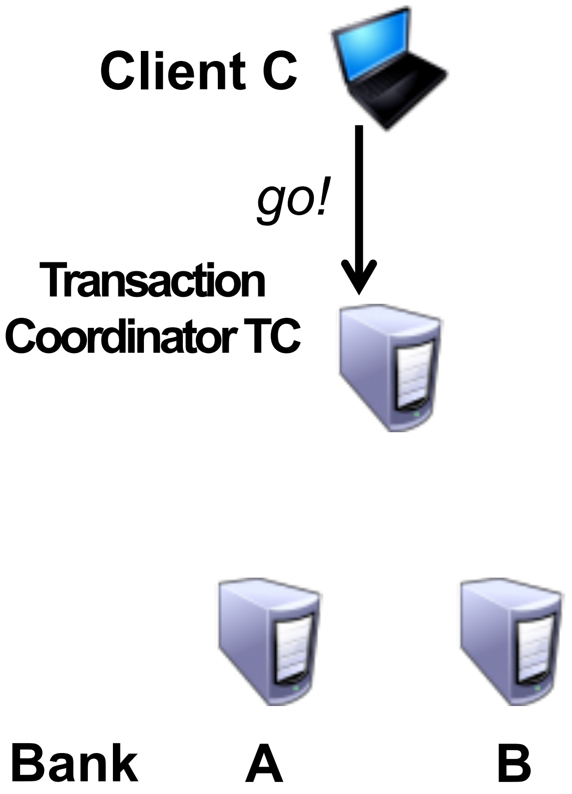
Distributed transactions?

- A client requests a transaction across servers: a sequence of operations which are treated as atomic (**it is all or nothing!**)
  - Operations being performed on behalf of other concurrent clients do not interfere
  - Either all of the operations must be completed successfully or they must have no effect at all in the presence of failures

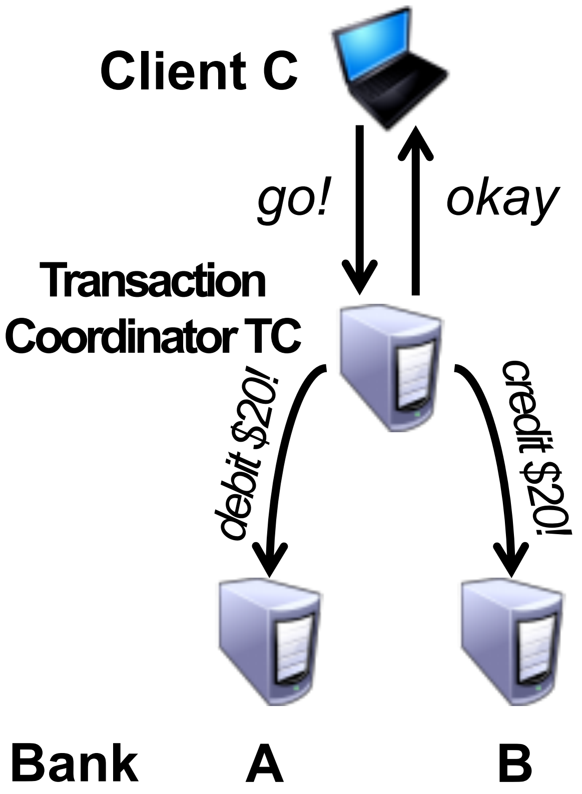
- How do we guarantee that all of the servers commit the transactions or none commit the transactions?

Straw Man one-phase protocol

1. **C → TC**: *“go!”*



Straw Man one-phase 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
- **TC** repeats sending messages until both **A, B** ack

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**?

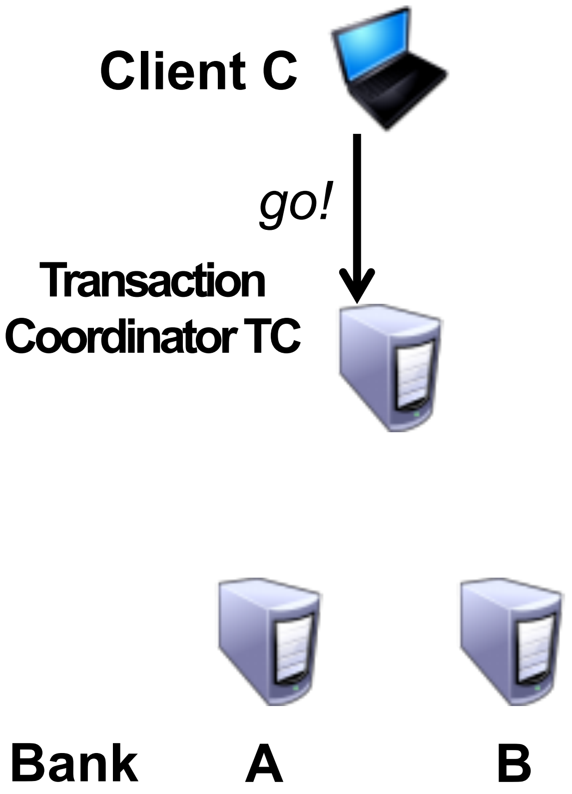
Safety versus liveness

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

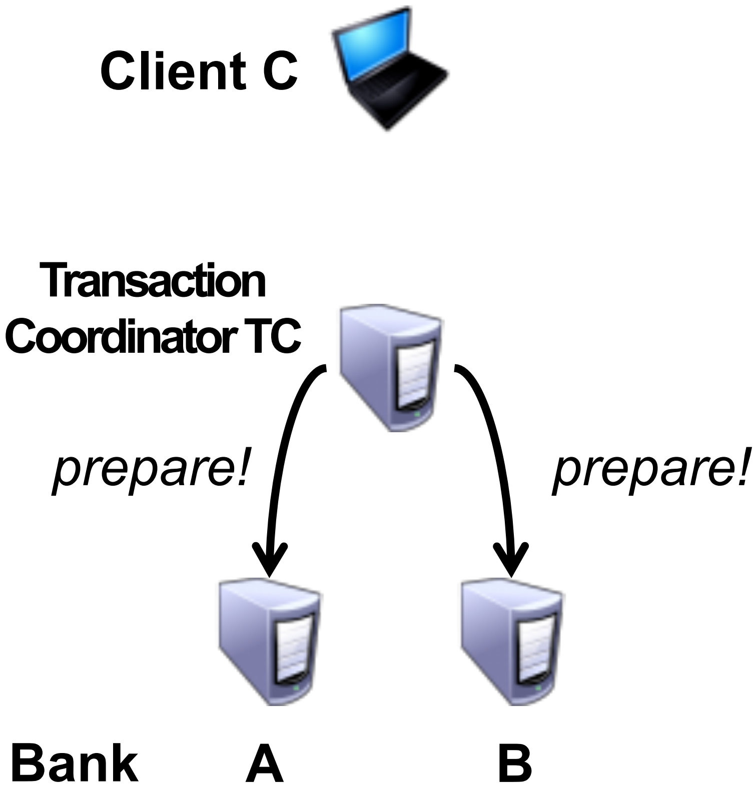
1. Safety
   - If **one commits, no one aborts**
   - If **one aborts, no one commits**
2. Liveness
   - If **no failures** and **A** and **B** can commit, **action commits**
   - If **failures,** reach a conclusion ASAP

A *correct* atomic commit protocol

1. C → TC: *“go!”*

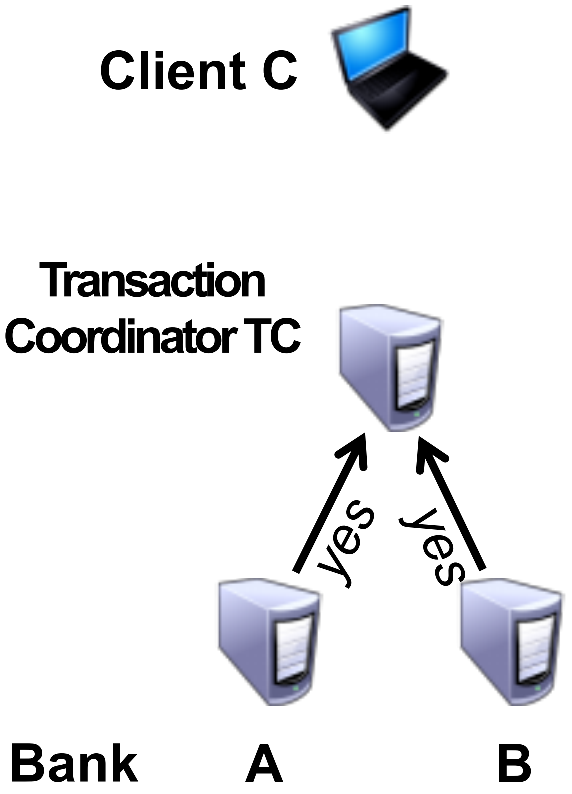


A *correct* atomic commit protocol



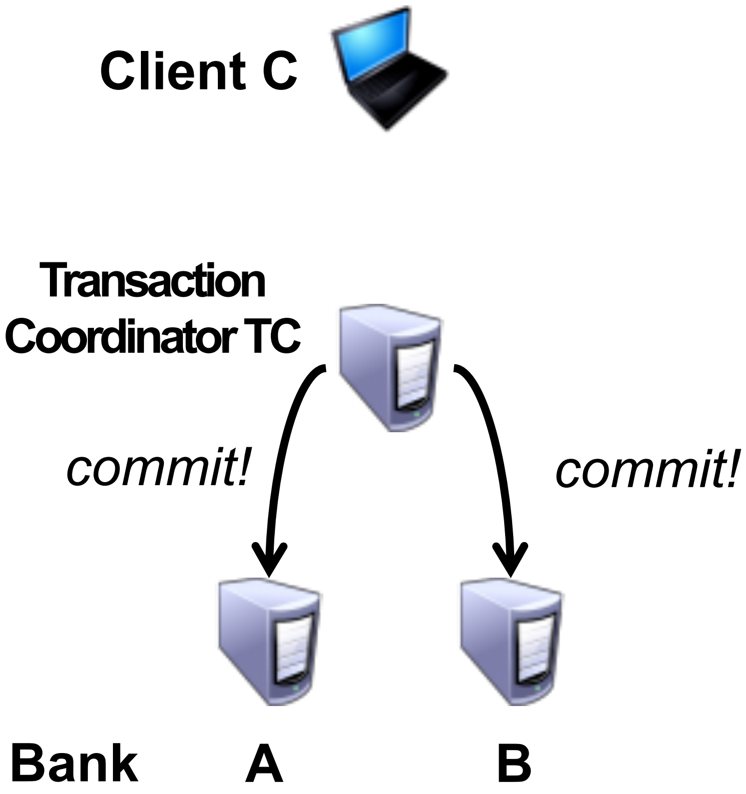
1. C → TC: *"go!"*
2. TC → A, B: *"prepare!"*

A *correct* atomic commit protocol



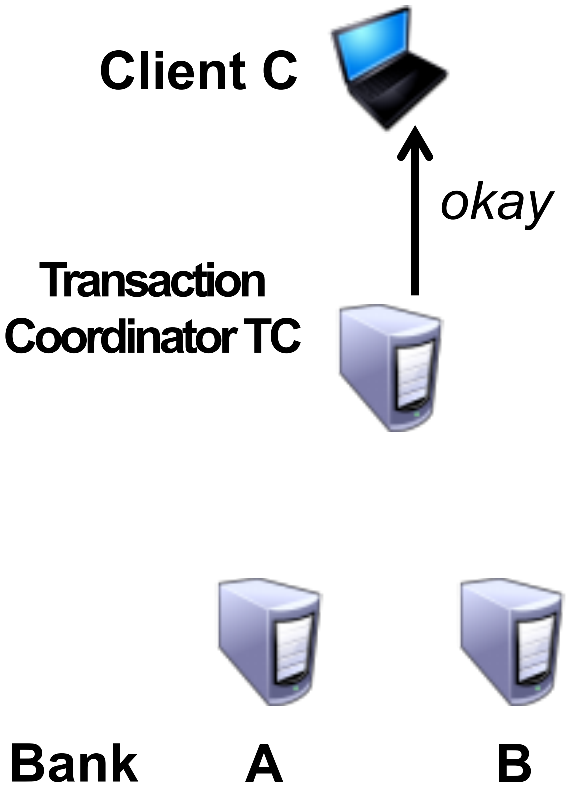
1. **C → TC:** *“go!”*
2. **TC → A, B:** *“prepare!”*
3. **A, B → TC:** vote *“yes”* or *“no”*

A *correct* atomic commit protocol



1. **C → TC:** *"go!"*
2. **TC → A, B:** *"prepare!"*
3. **A, B → TC:** vote *"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 → TC:** vote *"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

Two-Phase Commit (almost)

Transaction Coordinator (TC)

```
I. Sends VOTE-REQ to all participants

III. TC votes
      if all votes are YES then
        $decide_{TC}$ := COMMIT
        send COMMIT to all
     else
        $decide_{TC}$ := ABORT
        send ABORT to all who voted YES
     halt
```

Participant p_i

```
II. Sends $vote_i$ to TC
    if $vote_i$ is NO then
      $decide_i$ := ABORT
    halt

IV. if received COMMIT then
      $decide_i$ := COMMIT
    else
      $decide_i$ := ABORT
    halt
```

Reasoning about atomic commit

- Satisfies AC-1 to AC-4

- But not AC-5 (at least “as is”)
  - A process may be waiting for a message that may never arrive
    - Use Timeout Actions
  - No guarantee that a recovered process will reach a decision consistent with that of other processes
    - Processes save protocol state in DT-Log

Timeout actions

Where do hosts **wait** for messages?

II. p_i is waiting for VOTE-REQ from **TC**

III. TC waits for “yes” or “no” from participants

IV. p_i (who voted YES) waits for “commit” or “abort” from **TC**

Timeout actions

II. p_i is waiting for VOTE-REQ from **TC**

- Since it is has not cast its vote yet, can decide ABORT and halt

Timeout actions

III. TC waits for “yes” or “no” from participants

- **TC** hasn’t yet sent any commit messages, so can **safely** ABORT after a timeout
- Send ABORT to all participants which voted YES, and halt

Timeout actions

IV. p_i (who voted YES) waits for “commit” or “abort” from **TC**

- Can it unilaterally abort?
- Can it unilaterally commit?
- p_i cannot decide: must run a **termination protocol**

Termination protocol

- Consider **B** (**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**. Then:
  1. (No reply from **A**): no decision, **B** waits for **TC**
  2. **A** received commit or abort from **TC**: **B** agrees with **TC**'s decision
  3. **A** hasn't voted yet or voted *no:* both **abort**
     - **TC** can't have decided to commit
  4. **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 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 DT-Log** to record *“commit!” and “yes”* to stable storage

Recovery protocol with non-volatile state

- If everyone rebooted and is reachable, TC can just check for **commit** record on DT-Log 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

Reconfiguration and View Change Protocols