Atomic Commit and Concurrency Control

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

CS 240: Computing Systems and Concurrency
Lecture 16

Marco Canini

Let’s Scale Strong Consistency!

1. **Atomic Commit**
   – Two-phase commit (2PC)

2. Serializability
   – Strict serializability

3. Concurrency Control:
   – Two-phase locking (2PL)
   – Optimistic concurrency control (OCC)

Atomic Commit

- Atomic: All or nothing
- Either all participants do something (commit) or no participant does anything (abort)
- Common use: commit a transaction that updates data on different shards

The transaction

- *Definition:* A unit of work:
  - May consist of **multiple** data accesses or updates
  - Must **commit** or **abort** as a **single atomic unit**

- Transactions can either **commit,** or **abort**
  - When **commit,** all updates performed on data are made permanent, visible to other transactions

  - When **abort,** data restored to a state such that the aborting transaction never executed

Transaction examples

- Bank account transfer
  - A -= $100
  - B += $100

- Maintaining symmetric relationships
  - A FriendOf B
  - B FriendOf A

- Order product
  - Charge customer card
  - Decrement stock
  - Ship stock

Defining properties of transactions

- **Atomicity:** Either **all** constituent operations of the transaction complete successfully, or **none** do
- **Consistency:** Each transaction in isolation preserves a set of **integrity constraints** on the data
- **Isolation:** Transactions' behavior not impacted by presence of **other concurrent transactions**
- **Durability:** The transaction's **effects survive failure** of volatile (memory) or non-volatile (disk) storage

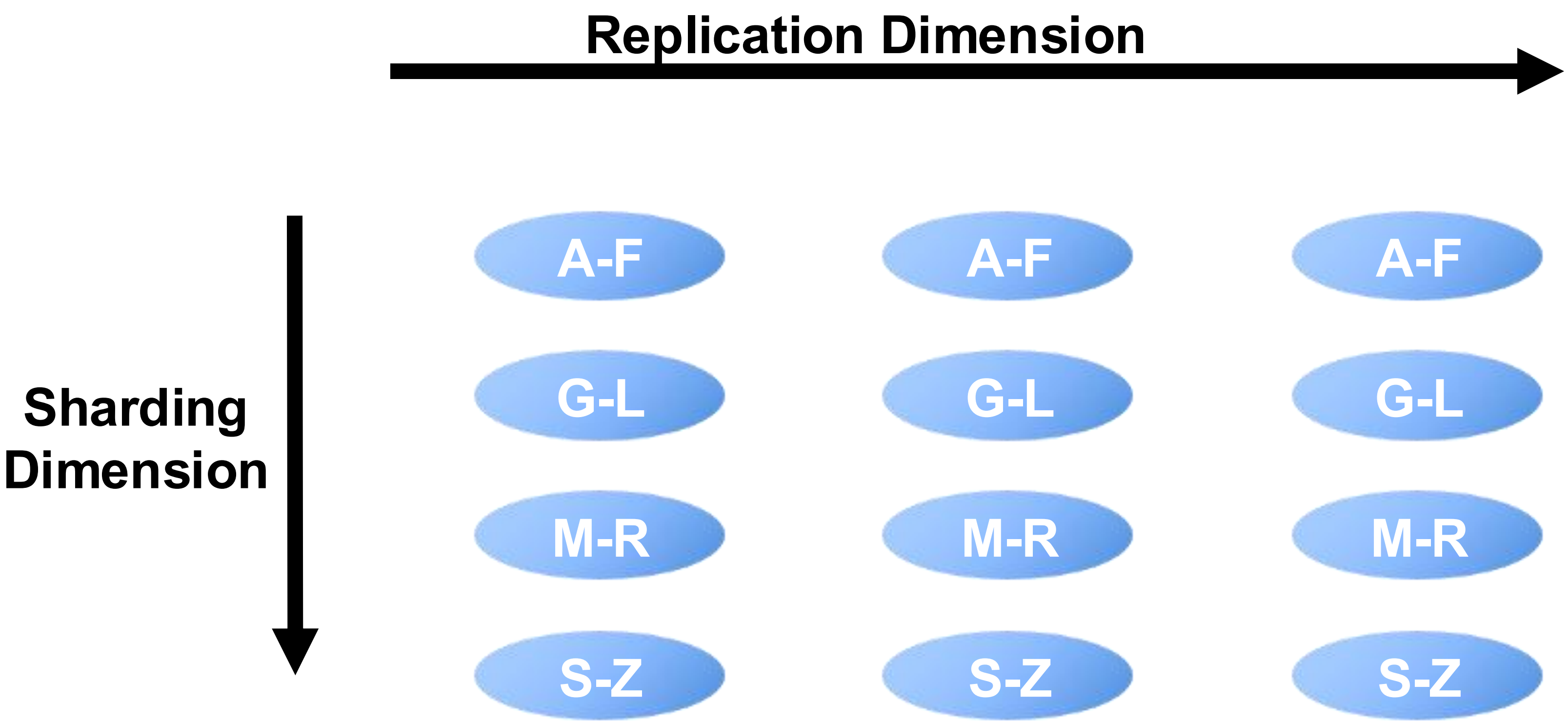
Relationship with replication

- Replication (e.g., RAFT) is about doing the same thing multiple places to provide fault tolerance

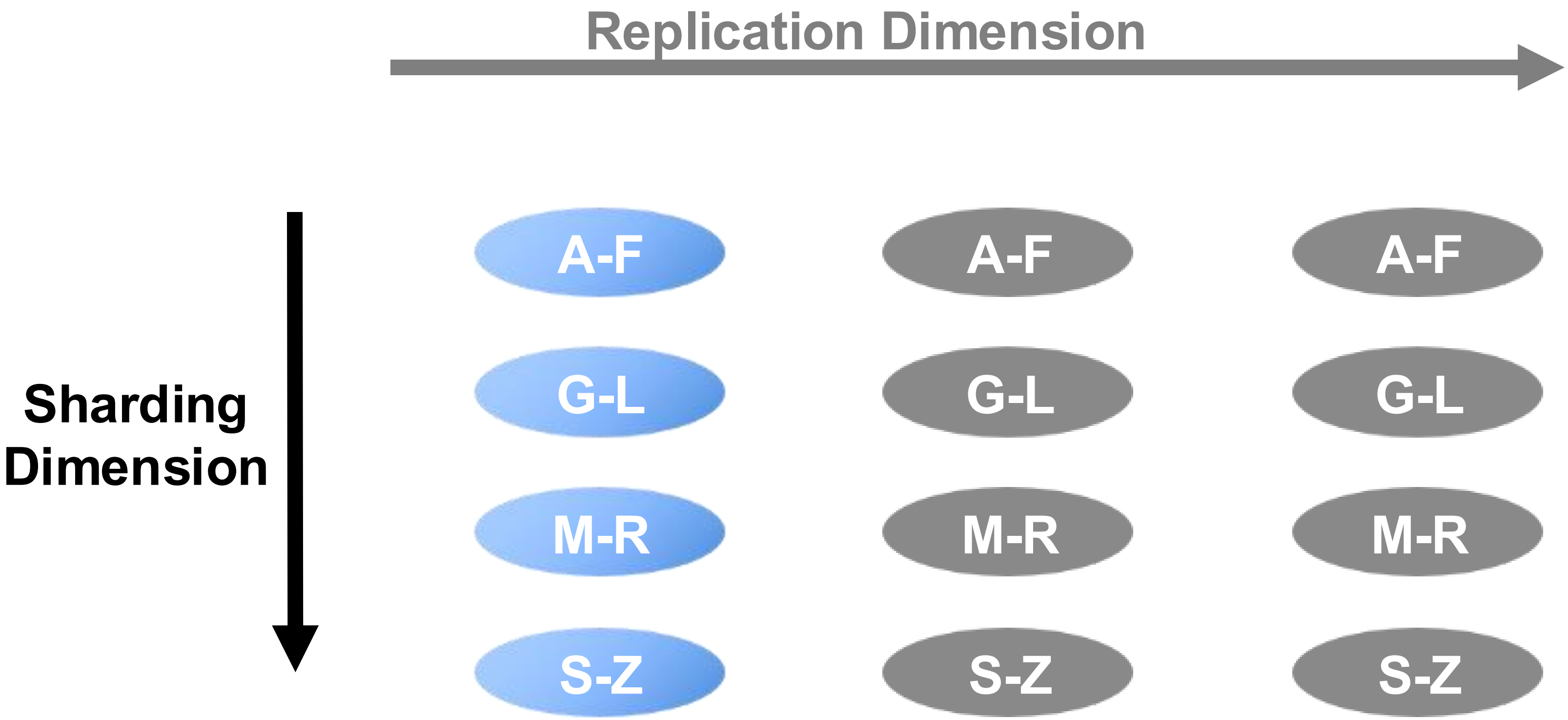
- Sharding is about doing different things multiple places for scalability

- Atomic commit is about doing different things in different places **together**

Relationship with replication



Focus on sharding for today



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();
   }
}
```

Atomic Commit

- Atomic: All or nothing

- Either all participants do something (commit) or no participant does anything (abort)

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 d

- 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

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

proces… 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

Atomic Commit

- Atomic: All or nothing
- Either all participants do something (commit) or no participant does anything (abort)
- Atomic commit is accomplished with the Two-phase commit protocol (2PC)

Let’s Scale Strong Consistency!

1. **Atomic Commit**
   – **Two-phase commit (2PC)**

2. Serializability
   – Strict serializability

3. Concurrency Control:
   – Two-phase locking (2PL)
   – Optimistic concurrency control (OCC)

Two-Phase Commit (almost)

Transaction Coordinator (TC)

I. Sends Prepare-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**

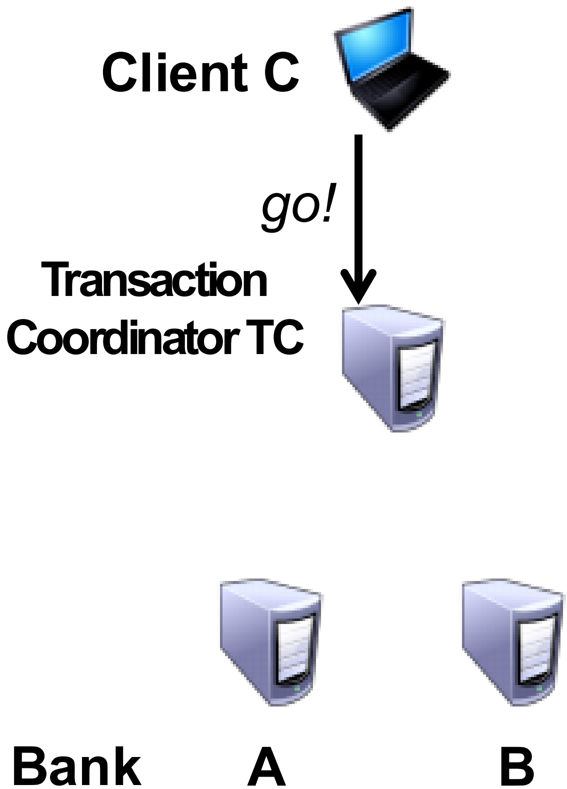
$decide_i$:= COMMIT

else

$decide_i$:= ABORT

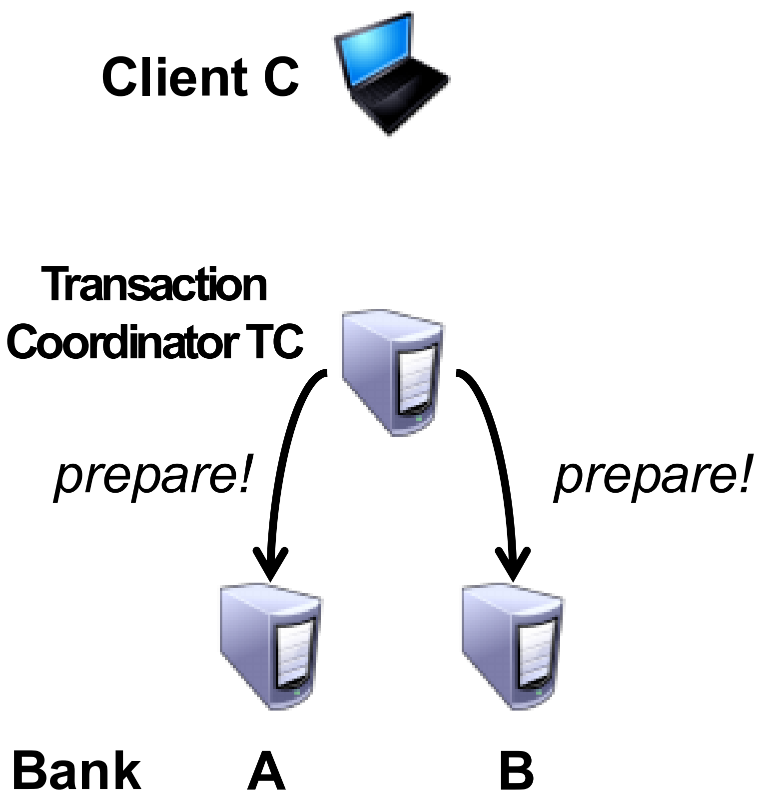
halt

Two-Phase Commit illustrated



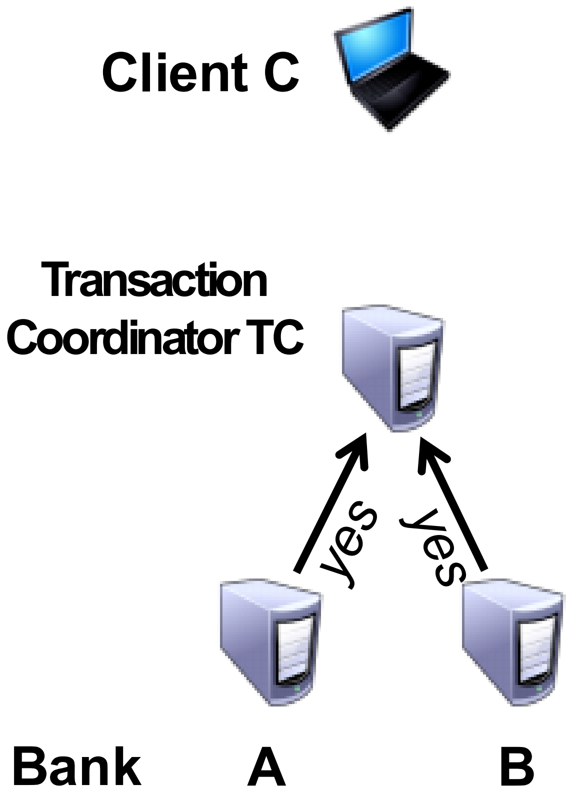
1. C → TC: *"go!"*

Two-Phase Commit illustrated



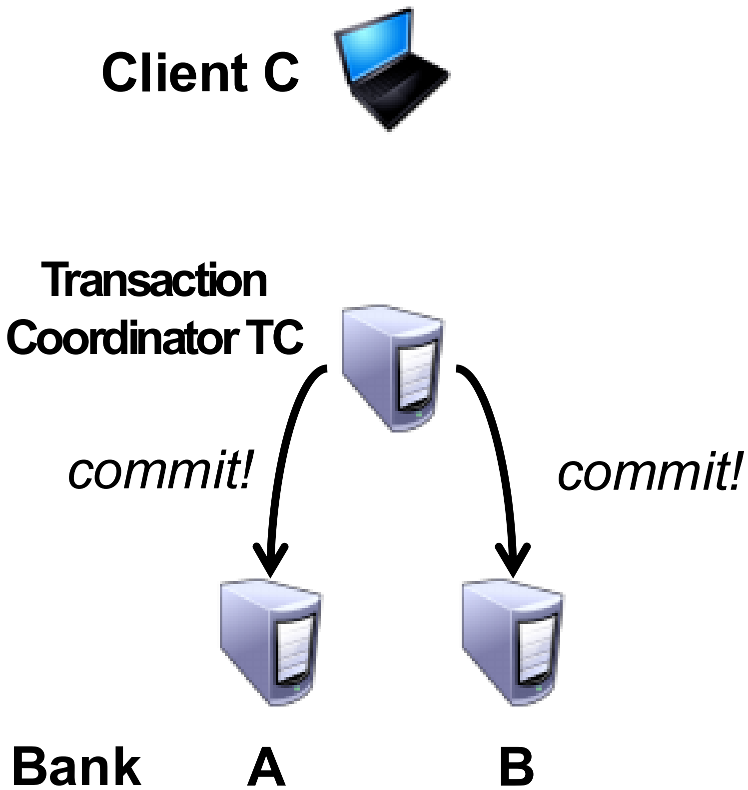
1. **C → TC:** *"go!"*
2. **TC → A, B:** *"prepare!"*

Two-Phase Commit illustrated



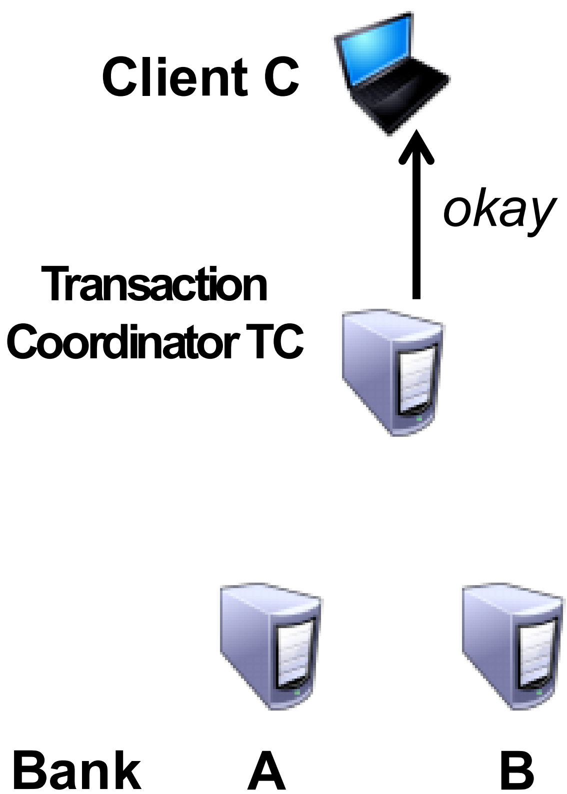
1. **C → TC:** *"go!"*
2. **TC → A, B:** *"prepare!"*
3. **A, B → TC:** vote *"yes"* or *"no"*

Two-Phase Commit illustrated



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*

Two-Phase Commit illustrated



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

Reasoning about two-phase 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 Prepare-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 Prepare-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**

Let’s Scale Strong Consistency!

1. Atomic Commit
   – Two-phase commit (2PC)

2. **Serializability**
   – **Strict serializability**

3. Concurrency Control:
   – Two-phase locking (2PL)
   – Optimistic concurrency control (OCC)

Two concurrent transactions

transaction **sum(A, B)**:
begin_tx
a ← read(A)
b ← read(B)
print a + b
commit_tx

transaction **transfer(A, B)**:
begin_tx
a ← read(A)
if a < 10 **then** *abort_tx*
else write(A, a−10)
b ← read(B)
write(B, b+10)
commit_tx

Isolation between transactions

- **Isolation:** **sum** appears to happen either completely before or completely after **transfer**
  - i.e., it appears that all operations of a transaction happened together
  - sometimes called *before-after atomicity*

- *Schedule* for transactions is an ordering of the operations performed by those transactions

Problem for concurrent execution: Inconsistent retrieval

- **Serial execution** of transactions—transfer then sum:

debit, credit

transfer: r_A w_A r_B w_B ©

sum: r_A r_B ©

- Concurrent execution resulting in ***inconsistent retrieval,*** result differing from any serial execution:

debit, credit

transfer: r_A w_A r_B w_B ©

sum: r_A r_B ©

Time →

© = commit

Isolation between transactions

- **Isolation: sum** appears to happen either completely before or completely after **transfer**
  – i.e., it appears that all operations of a transaction happened together
  – sometimes called *before-after atomicity*

- Given a schedule of operations:
  – *Is that schedule in some way "equivalent" to a serial execution of transactions?*

Equivalence of schedules

- Two **operations** from **different transactions** are ***conflicting*** if:

1. They **read** and **write** to the **same data item**
2. The **write** and **write** to the **same data item**

- Two **schedules** are ***equivalent*** if:

1. They contain the same transactions and operations
2. They **order** all **conflicting** operations of non-aborting transactions in the **same way**

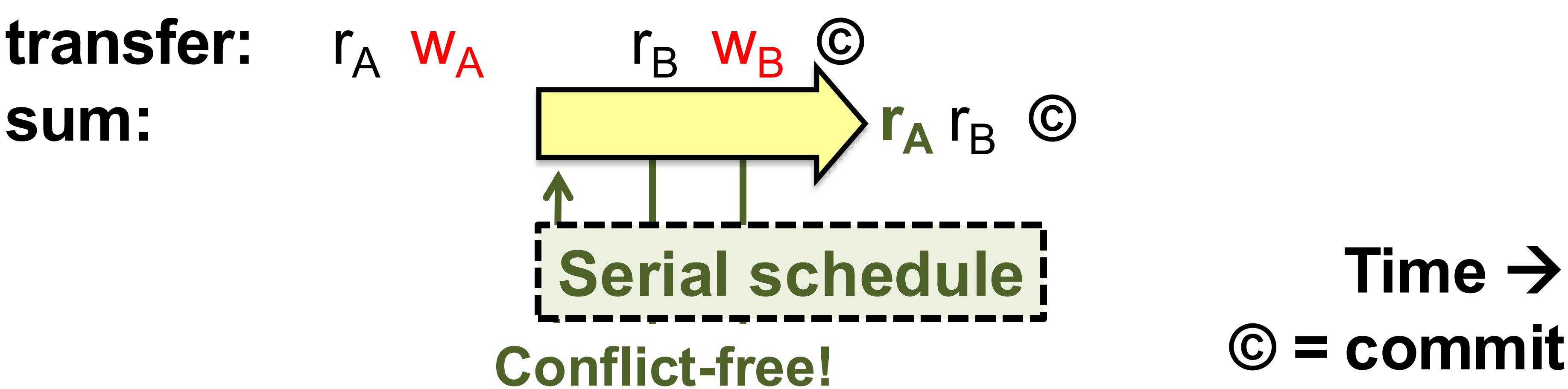
Serializability

- Ideal isolation semantics: *serializability*

- A schedule is ***serializable*** if it is equivalent to some serial schedule
  - *i.e.,* **non-conflicting** operations can be **reordered** to get a **serial** schedule

A serializable schedule

- Ideal isolation semantics: *serializability*
- A schedule is ***serializable*** if it is equivalent to some serial schedule
  - *i.e.*, **non-conflicting** operations can be **reordered** to get a **serial** schedule



A non-serializable schedule

- Ideal isolation semantics: *serializability*
- A schedule is ***serializable*** if it is equivalent to some serial schedule
  - *i.e.,* **non-conflicting** operations can be **reordered** to get a **serial** schedule

transfer: r_A w_A r_B w_B ©

sum: r_A r_B ©

But in a **serial schedule**, sum's reads either **both before** w_A or **both after** w_B

Conflicting ops

Time →

© = commit

Serializability versus linearizability

- **Linearizability:** a guarantee about **single** operations on **single** objects
  - Once write completes, all later reads (by wall clock) should reflect that write
- **Serializability** is a guarantee about **transactions** over **one or more** objects
  - Doesn't impose real-time constraints

- ***Strict serializability*** **= Serializability + real-time ordering**
  - Intuitively Serializability + Linearizability
  - Transaction behavior equivalent to some serial execution
    - **And** that serial execution **agrees with real-time**

Consistency Hierarchy

Strict Serializability — e.g., Spanner

↓

Linearizability — e.g., RAFT

↓

Sequential Consistency

↓

Causal+ Consistency — e.g., Bayou

↓

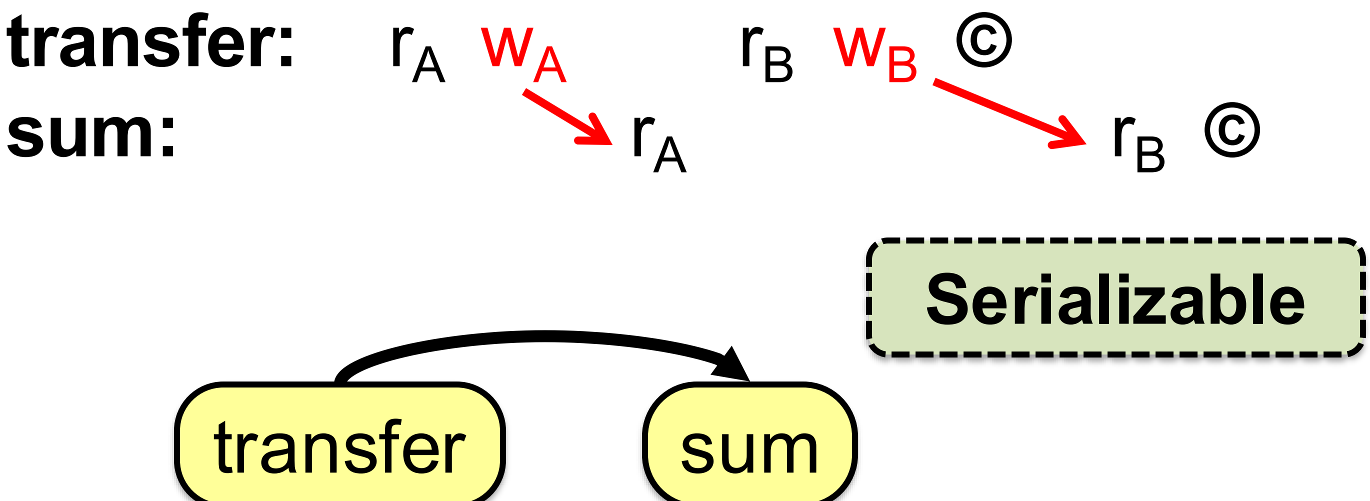
Eventual Consistency — e.g., Dynamo

Testing for serializability

- Each node ***t*** in the ***precedence graph*** represents a transaction ***t***
  - Edge from ***s*** to ***t*** if some action of ***s*** **precedes and conflicts with** some action of ***t***

Serializable schedule, acyclic graph

- Each node ***t*** in the ***precedence graph*** represents a transaction ***t***
  - Edge from ***s*** to ***t*** if some action of ***s*** **precedes and conflicts with** some action of ***t***



Time →

© = commit

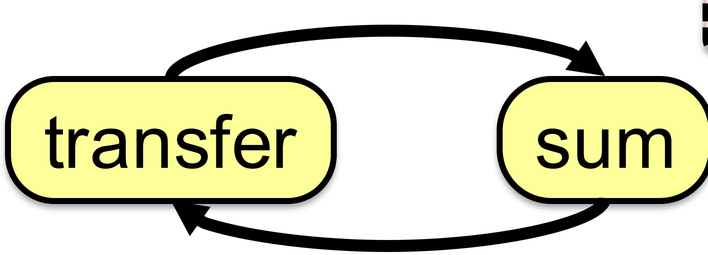
Non-serializable schedule, cyclic graph

- Each node ***t*** in the ***precedence graph*** represents a transaction ***t***
  - Edge from ***s*** to ***t*** if some action of ***s*** **precedes and conflicts with** some action of ***t***

transfer: r_A w_A r_B w_B ©

sum: r_A r_B ©

Non-serializable



Time →

© = commit

Testing for serializability

- Each node ***t*** in the ***precedence graph*** represents a transaction ***t***
  - Edge from ***s*** to ***t*** if some action of ***s*** **precedes and conflicts with** some action of ***t***

In general, a schedule is **serializable** if and only if its **precedence graph** is **acyclic**

Let's Scale Strong Consistency!

1. Transactions and Atomic Commit review

2. Serializability
   – Strict serializability

3. **Concurrency Control:**
   – **Two-phase locking (2PL)**
   – Optimistic concurrency control (OCC)

Concurrency Control

- Concurrent execution can violate serializability

- We need to **control** that concurrent execution so we do things a single machine executing transactions one at a time would
  - Concurrency control

Concurrency Control Strawman #1

- **Big Global Lock**
  - Acquire the lock when transaction starts
  - Release the lock when transaction ends

- Provides strict serializability
  - Just like executing transaction one by one because we are doing exactly that

- No concurrency at all
  - Terrible for performance: one transaction at a time

Locking

- Locks maintained on each shard
  - Transaction requests lock **for a data item**
  - Shard **grants** or **denies** lock

- **Lock types**
  - ***Shared:*** Need to have before read object
  - ***Exclusive:*** Need to have before write object

	Shared (S)	**Exclusive (X)**
Shared (S)	Yes	No
Exclusive (X)	No	No

Concurrency Control Strawman #2

- Grab locks **independently**, for each data item (*e.g.*, bank accounts A and B)

transfer: ◢$_A$ r_A w_A ◣$_A$ ◢$_B$ r_B w_B ◣$_B$ ©

sum: ◿$_A$ r_A ◺$_A$ ◿$_B$ r_B ◺$_B$ ©

Permits this **non-serializable** interleaving

Time →

© = commit

◢ / ◿ = **eXclusive- / Shared-lock;** ◣ / ◺ = **X- / S-unlock**

Two-phase locking (2PL)

- **2PL rule:** Once a transaction has **released** a lock it is **not allowed to obtain** any other locks
  - **Growing phase** when transaction acquires locks
  - **Shrinking phase** when transaction releases locks
- In practice:
  - Growing phase is the entire transaction
  - Shrinking phase is during commit

2PL provides strict serializability

- **2PL rule:** Once a transaction has **released** a lock it is **not allowed to obtain** any other locks

transfer: ◢$_A$ r_A w_A ◣$_A$ ◢$_B$ r_B w_B ◣$_B$ ©

sum: ◿$_A$ r_A ◺$_A$ ◿$_B$ r_B ◺$_B$ ©

2PL **precludes** this **non-serializable** interleaving

Time →

© = commit

◢ / ◿ = X- / S-lock; ◣ / ◺ = X- / S-unlock

2PL and transaction concurrency

- **2PL rule:** Once a transaction has **released** a lock it is **not allowed to obtain** any other locks

transfer: ◿$_A$ r$_A$ ◢$_A$ w$_A$ ◿$_B$ r$_B$ ◢$_B$ w$_B$ ✻©

sum: ◿$_A$ r$_A$ ◿$_B$ r$_B$ ✻©

2PL **permits** this **serializable, interleaved** schedule

Time →

© = commit

◢ / ◿ = **X- / S-lock;** ◣ / ◺ = **X- / S-unlock;** ✻ = **release all locks**

2PL doesn't exploit all opportunities for concurrency

- **2PL rule:** Once a transaction has **released** a lock it is **not allowed to obtain** any other locks

transfer: r_A w_A r_B w_B ©
sum: r_A r_B ©

2PL **precludes** this **serializable, interleaved** schedule

Time →
© = commit
(locking not shown)

Issues with 2PL

- What do we do if a lock is unavailable?
  – Give up immediately?
  – Wait forever?

- Waiting for a lock can result in **deadlock**
  – Transfer has A locked, waiting on B
  – Sum has B locked, waiting on A

- Many ways to detect and deal with deadlocks
  – e.g., centrally detect deadlock cycles and **abort involved transactions**

Lets Scale Strong Consistency!

1. Atomic Commit
   – Two-phase commit (2PC)

2. Serializability
   – Strict serializability

3. Concurrency Control:
   – Two-phase locking (2PL)
   – **Optimistic concurrency control (OCC)**

2PL is pessimistic

- Acquire locks to **prevent** all possible violations of serializability

- **But leaves a lot of concurrency on the table that is okay and available**

- More Concurrency Control Algorithms
  - Optimistic Concurrency Control (OCC)
  - Multi-Version Concurrency Control (MVCC)