CSE 541: Database Systems I

Distributed DBMS

Slides Credit: Tianzheng Wang

Scale-up vs. Scale-out

Scale-up:

- Single machine
- Scaling = add more CPU cores, more memory, more storage to the single node
  - Cost-of-ownership may be high
- Shared memory architecture
  - Easy to program
  - Low-overhead communication between worker threads/processes
- Single point of failure
  - Node down → cannot process requests

Scale-out:

- Multiple machines
- Scaling = add more machines
  - May help bring down cost-of-ownership
- Message passing/network based communication
  - Harder to program
  - Higher overhead with long/many network roundtrips
- Better fault tolerance
  - Multiple nodes can be used to serve requests

Scale-out: Assumptions

- Multiple nodes form a cluster
  - Interconnected by some network
  - Local or wide-area
  - Latency, bandwidth vary depending on type of network
    - E.g., 100Gb Infiniband, 100GbE vs. 1GbE

- Each node runs a copy of the DBMS
- A load balancing layer (middleware) may exist to direct requests to the right node

Scaling Out

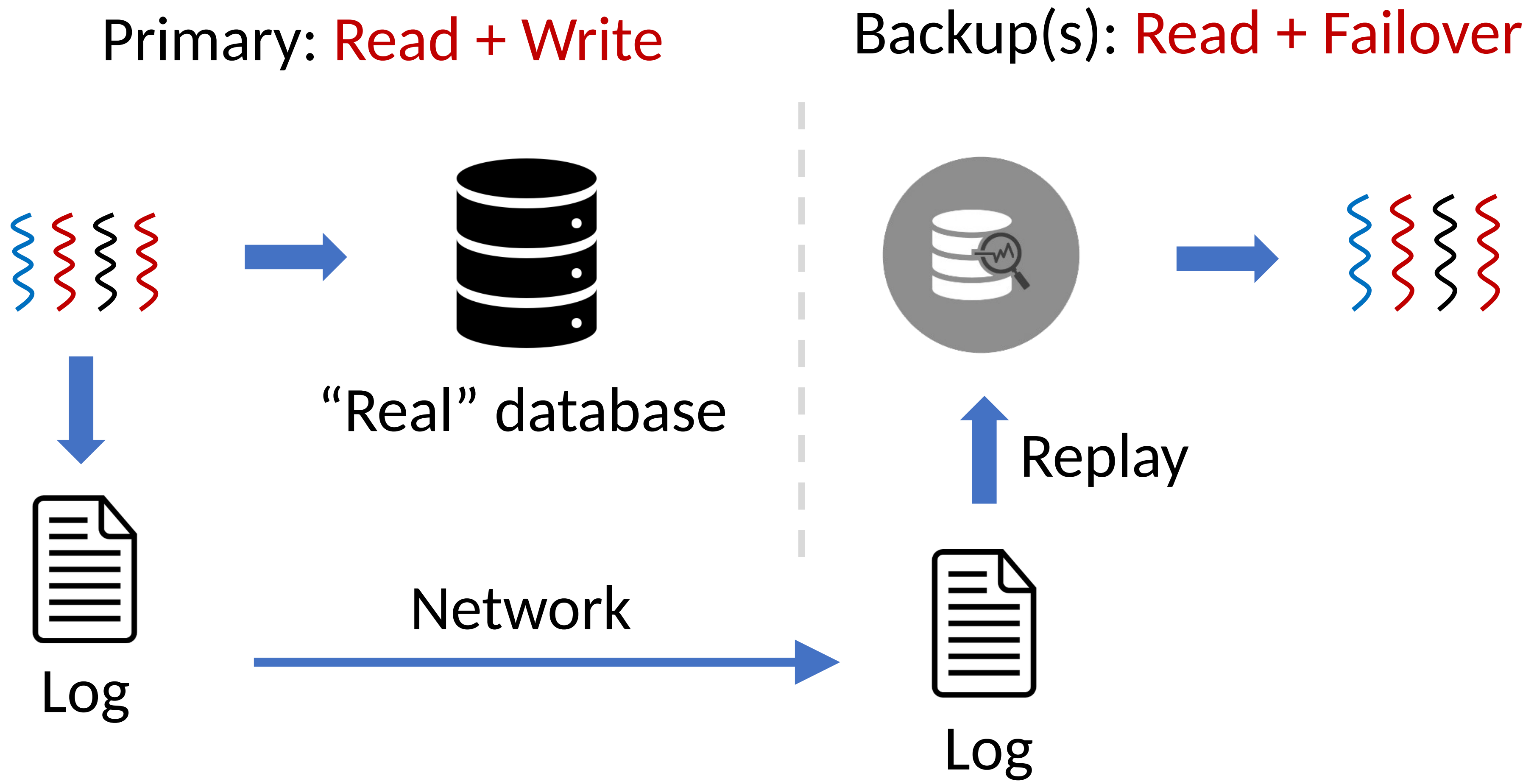
- Replication (high availability)
  - Replicating data to multiple nodes
  - If one node dies, another can take over to continue
  - “Single-master” – master == node that can accept writes
- Distributed transactions
  - Distributing data and queries/transactions on multiple machines
  - Provide more processing power
  - Handle larger data volume
  - Possibly with high availability
  - “Multi-master”

High Availability

- Easy to implement and maintain
  - Better not to be tightly coupled with other components
- Fresh data access on replicas
- Fast failover speed
  - Related to freshness
- Little overhead on the primary server
- High resource utilization
  - Modern servers are high-end machines, better let replicas perform read-only analytics queries than sitting idle
- Provide safety guarantees – the very purpose of replication
  - Not losing committed work

HA through Log Shipping

- Replicate ("ship") log records between nodes



Log Shipping

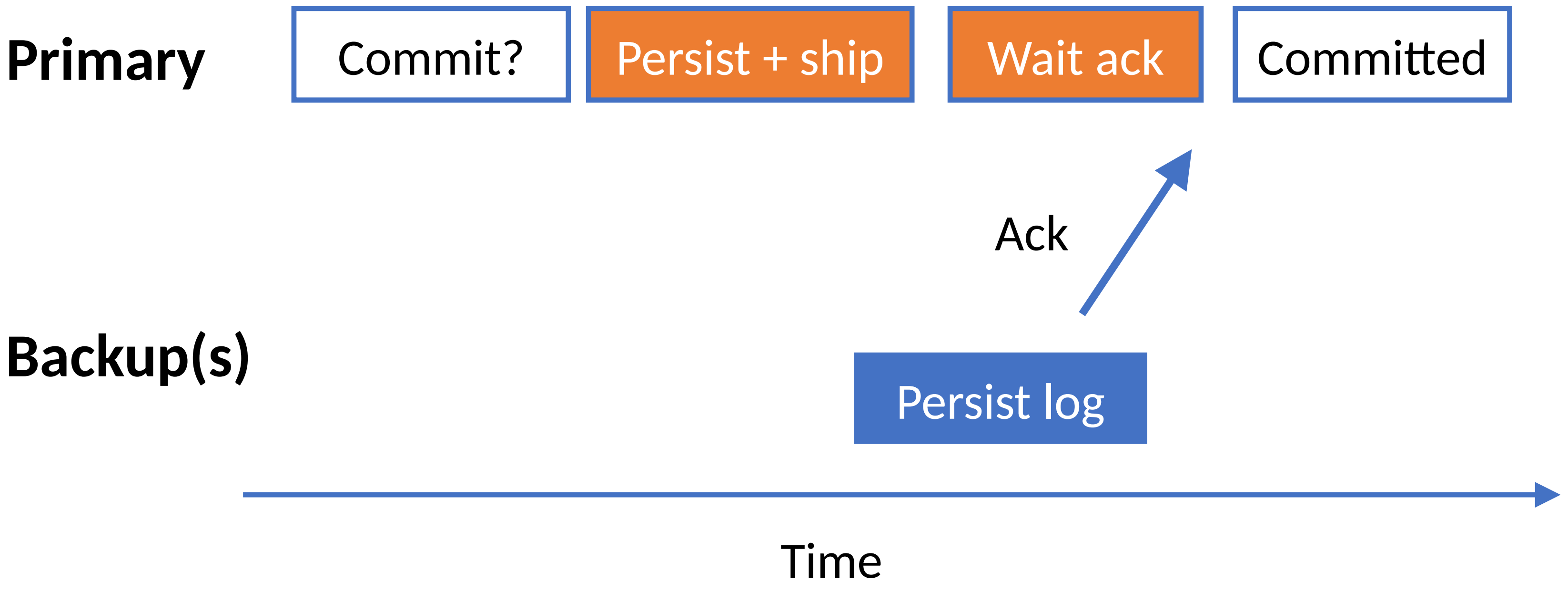
Log records can be logical or physical

- Physical log shipping
  - Log records contain real data/differentials
  - Log size may be big
  - More pressure on network
  - Replay is simple – works independent from concurrency control mechanisms
- Logical log shipping
  - Log records describe operations to be performed
  - Small log size
  - Low network bandwidth requirement
  - Replay is much harder – customization needed for concurrency control mechanism (need deterministic replay)

Log Shipping

Log shipping can be synchronous or asynchronous

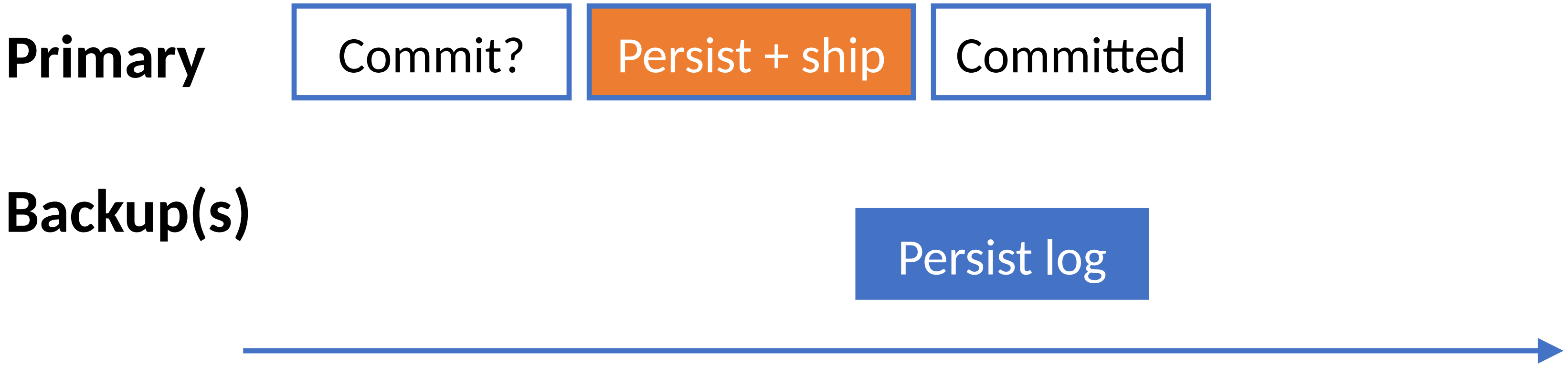
- Synchronous log shipping
  - Primary must wait for backup(s) to ack persistence



Log Shipping

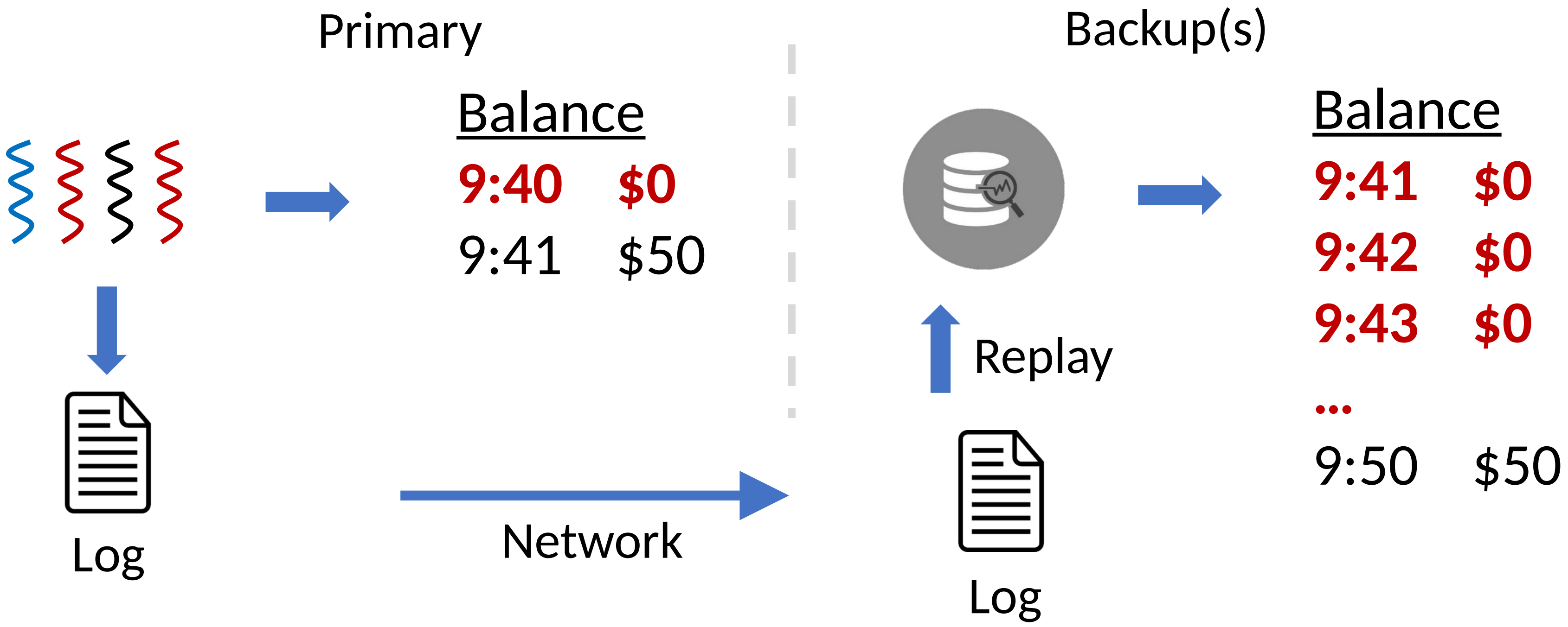
Log shipping can be synchronous or asynchronous

- Asynchronous log shipping
  - Primary keeps sending log records to backups
  - Primary does not wait for backup(s) to ack persistence



Primary may commit (i.e., user/client sees result) before log persisted in replica(s) – no safety guarantee + stale reads

Asynchronous Log Shipping: Stale



- Default for a lot of systems to avoid having network and/or I/O on the critical path

➔ Safety and freshness traded for primary speed

Safety Guarantees in Log Shipping

1-safe: Transaction commits as soon as its log records are persisted on the primary server

2-safe: Transaction cannot commit until its log records are persisted both on the primary and backup server

- Original definition considered 2-node case
- Can be extended to “commit when log is persisted in the primary and a majority of replicas”

Very-safe: Same as 2-safe, but does not allow transaction to commit if the backup is down

Synchronous log shipping needed for 2-safe and very-safe

- Network and I/O on the critical path

Asynchronous log shipping can achieve 1-safe

- Network off the critical path, but not safe

Freshness

How early/late can replicas see the most recent updates done on the primary?

- If a read query started after an update on the primary, then it should see the updated, new content
  - Alternative 1: block and wait for replay if not seeing the latest content
  - Alternative 2: finish log replay before acknowledging primary
- Also depends on whether the replay itself is fast

Tradeoffs

Replay speed vs. utilization:

- Faster replay requires typically more threads (CPU cycles) dedicated to replaying the log
  - Decrease effective utilization (CPU cycles/threads dedicated for read queries)

Transaction processing vs. long-running queries:

- Standby server dedicated much resource on handling long-running queries
  - Some of them might be very important, mission critical
  - If a failover now is needed, should the long-running queries be killed to make room for transactions?

Ease of implementation/maintenance vs. replication speed:

- Physical vs. logical logging and log shipping

Multi-Master

Allow multiple nodes to process both read and write transactions, i.e., multiple primary nodes

- There might be some read-only replicas as well

Benefits:

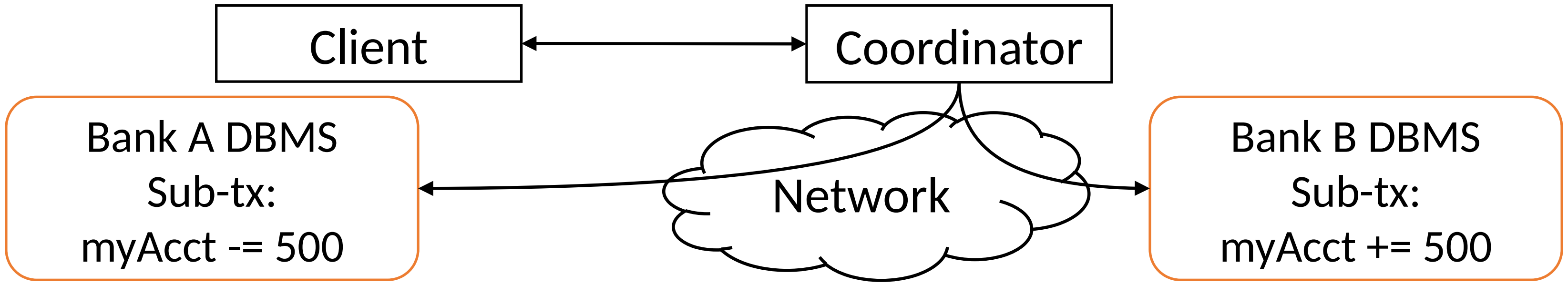
- More capacity (processing power) for read/write transactions
- Higher availability
  - Zero recovery time objective (RTO)
  - When one primary node goes down, client can immediately switch to another and retry the transaction
    - Same effect as a dead connection

Distributed Transactions

Transactions that access multiple resource managers distributed across a network

- Resource manager: manages resources (e.g., DBMS)
- Consists of multiple sub-transactions
- Each sub-transaction runs on a different DBMS site

Example: Transfer $500 from Bank A to B



- Desired outcome: all sub-tx succeed, or none

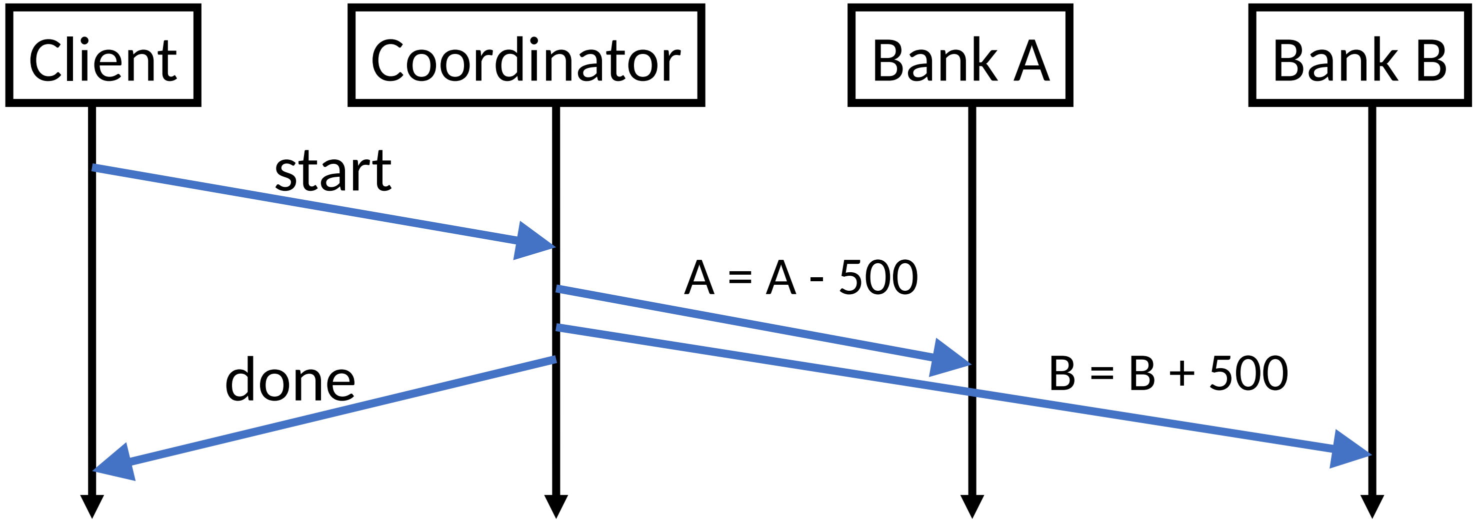
Distributed Transactions

Desired outcome:

- Individual DBMS
  - Support ACID locally for each sub-transaction
  - Sub-transactions execute normally like other transactions there
- Additionally: Global atomicity
  - Either all sub-transactions commit or abort
  - Enforced by the transaction coordinator
- Think of sub-transactions as “actions” in the transaction concept

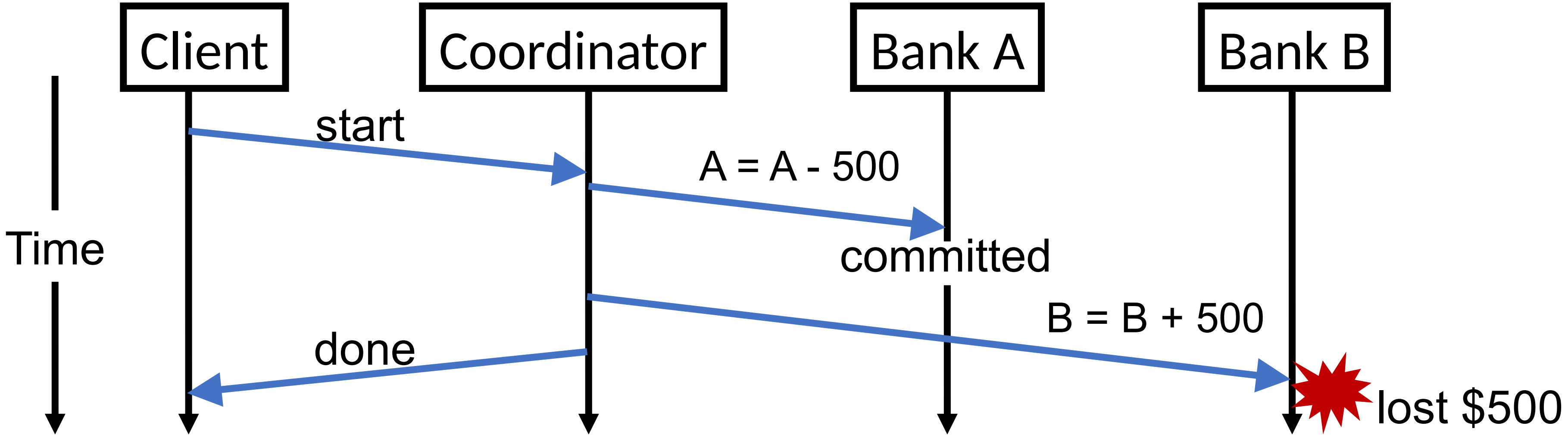
Distributed Transactions

- What can go wrong?
  - A does not have enough money
  - B's account no longer exists
  - B has crashed
  - Coordinator crashes



Distributed Transactions

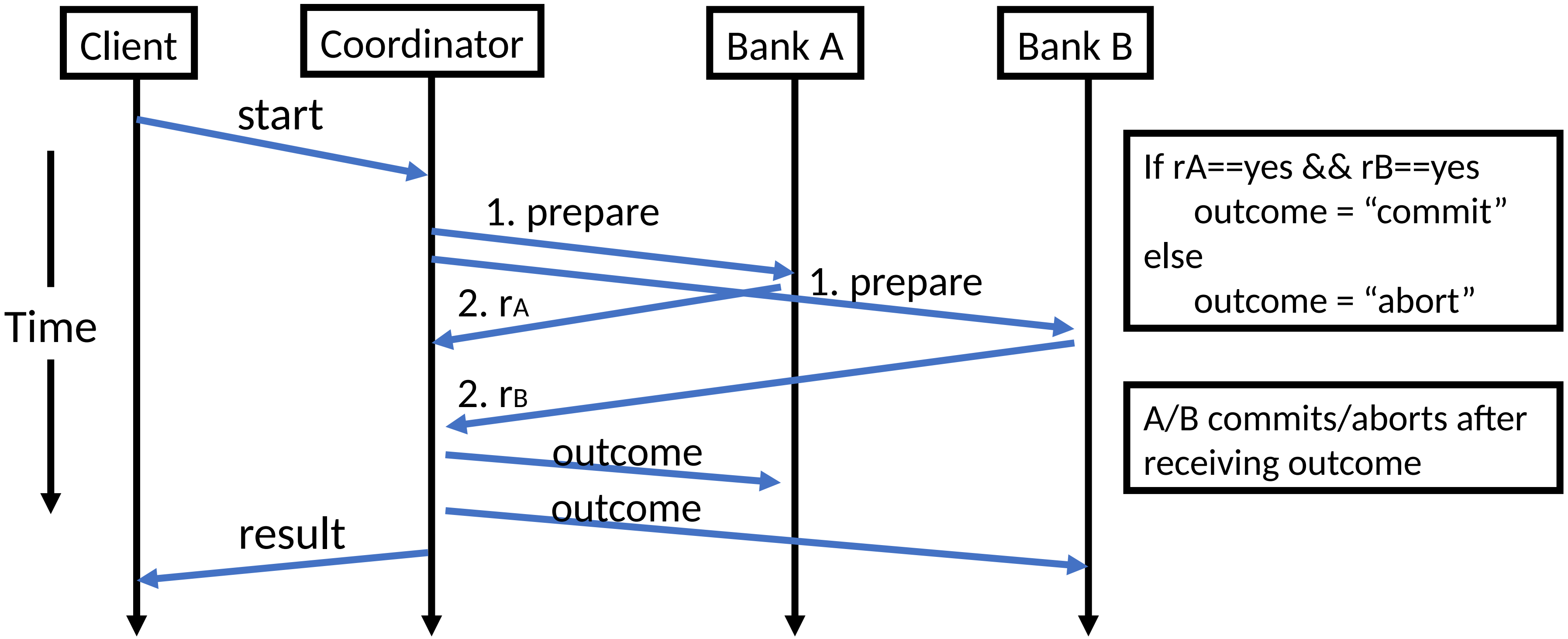
- Coordinating activities of participants (naïve way)
  - Coordinator waits for the result of every sub-transaction
  - Impossible to rollback already committed sub-transactions



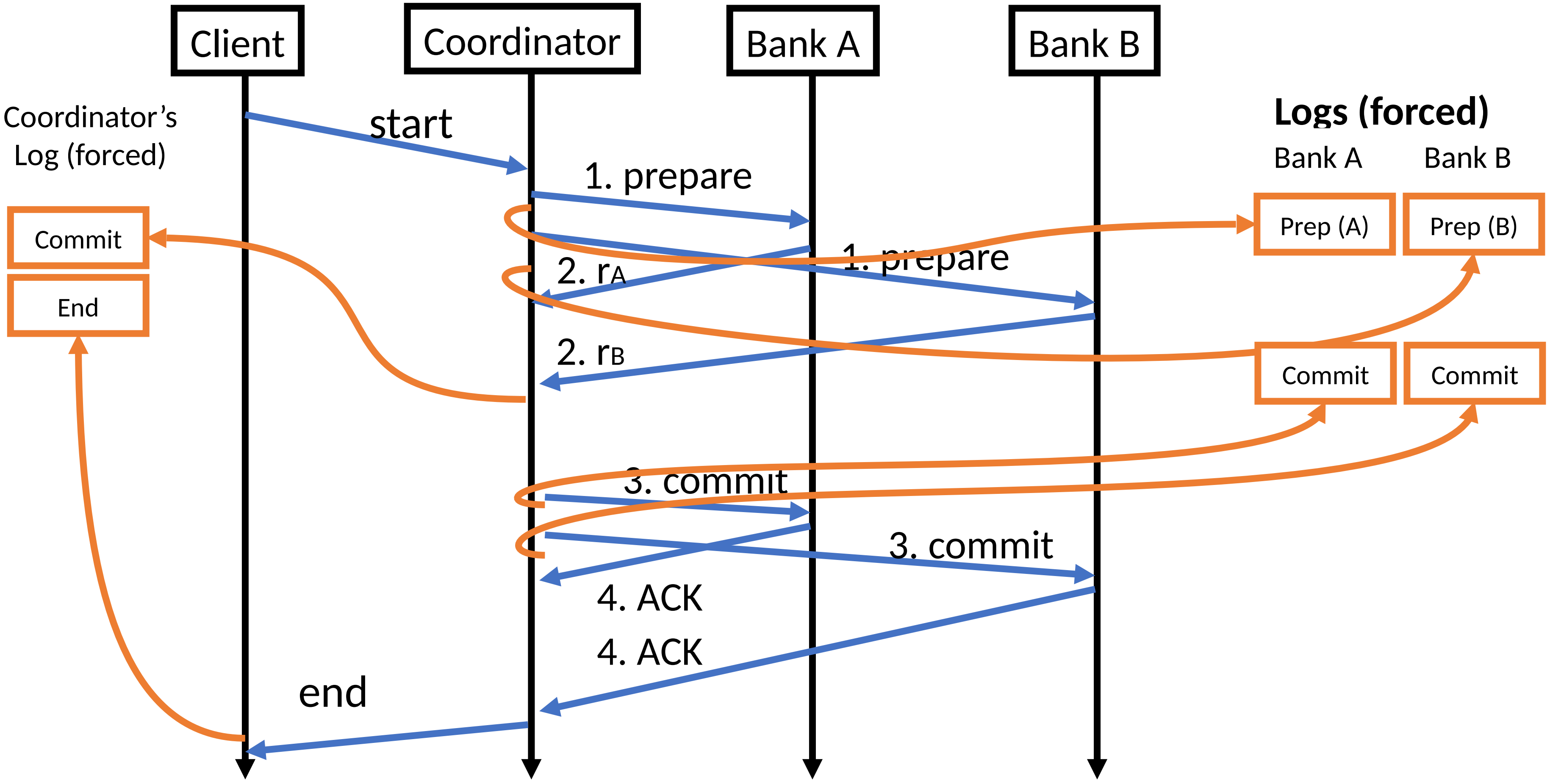
  - Manually rollback at A, or continue at B by super-user
- Need some protocol for uniform commitment

Two-Phase Commit (2PC)

- Basic Idea (a successful 2PC, no failures)
- Commit if everyone can, otherwise abort

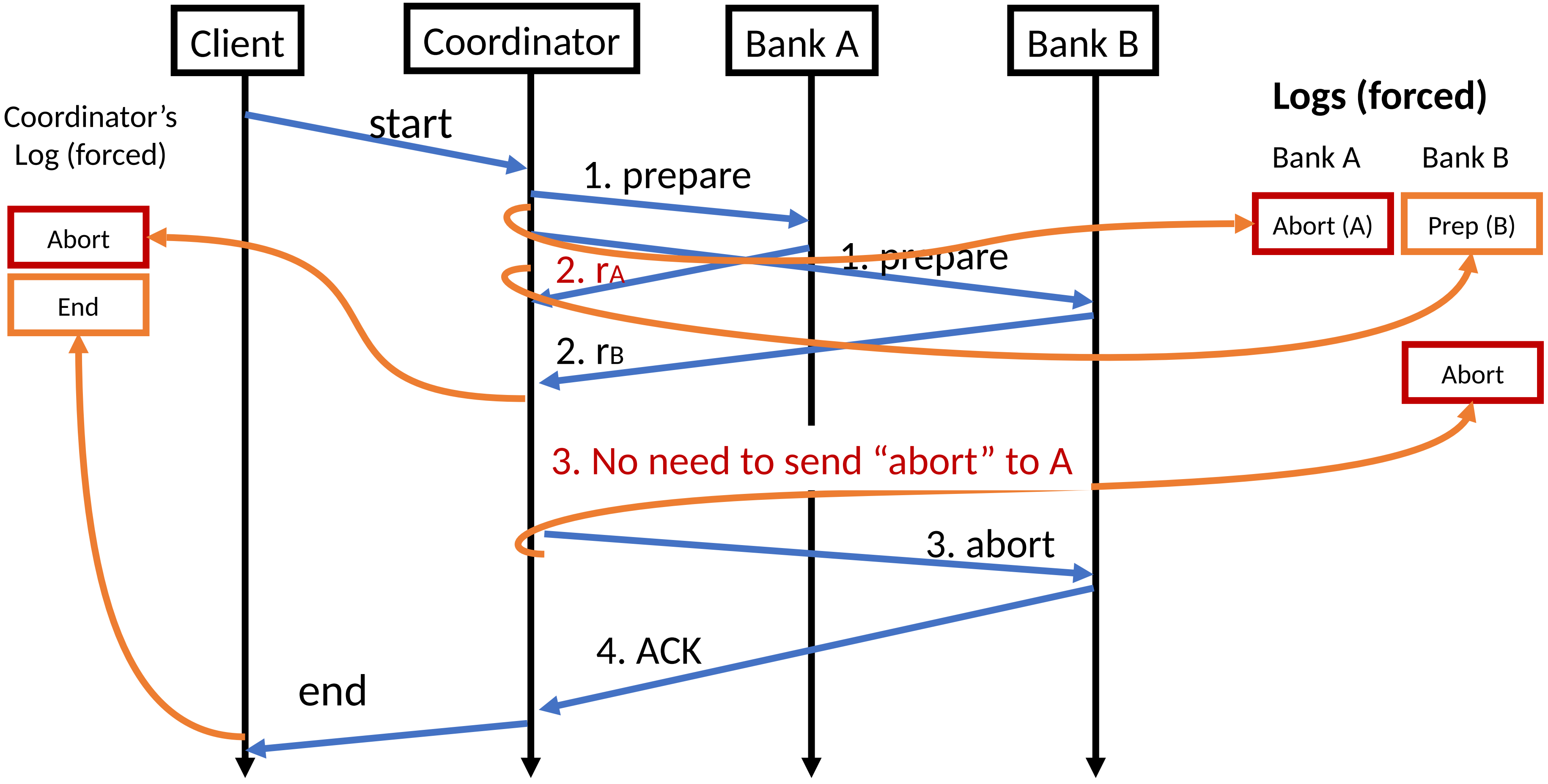


Two-Phase Commit (2PC)



Two-Phase Commit (2PC)

- Bank A aborts



Handling Failures in 2PC

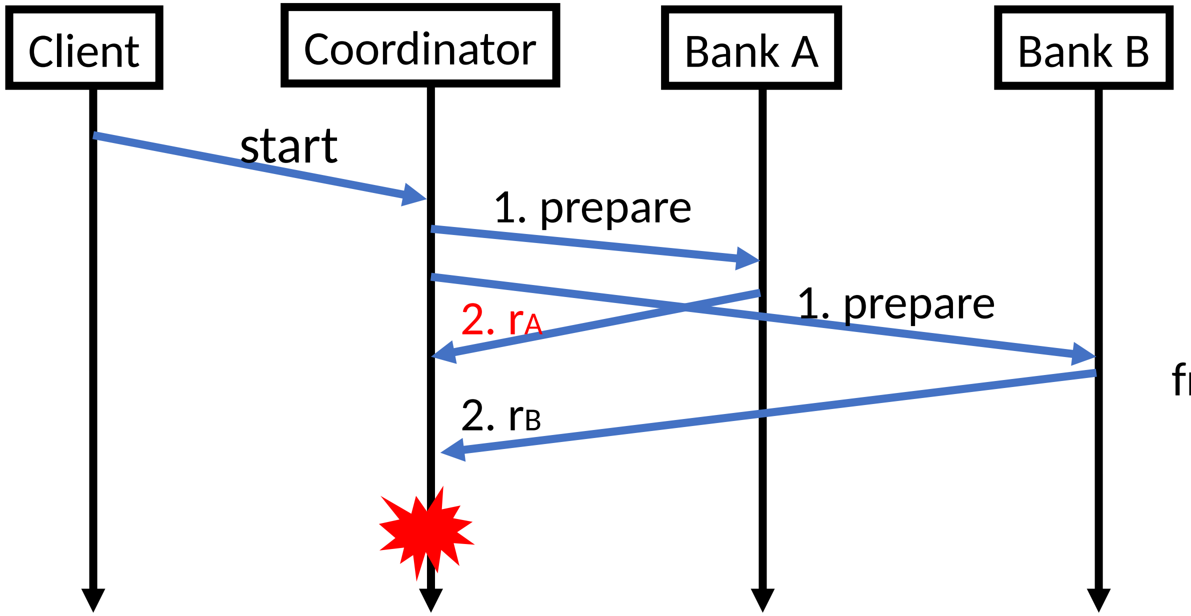
Basically by reading logs

- Assumption: no node crashes forever
- DBMS sites: failure at “prepared” state
  - Upon restart, contact the coordinator to find out the final decision
  - Continue with the decision as if in normal processing
- Coordinator: failure at normal processing (no commit rec)
  - Abort the transaction
- Coordinator: failure after written the commit record
  - Send final decision (commit/abort) to DBMS sites that have not ACKed
  - Wait for ACKs and end the transaction

Problem of 2PC

System blocks if coordinator fails

- Sub-transactions/node block until they receive the final decision from coordinator – other jobs may stall
- Permanent coordinator failure results in (1) sub-tx that can never finish and (2) data loss

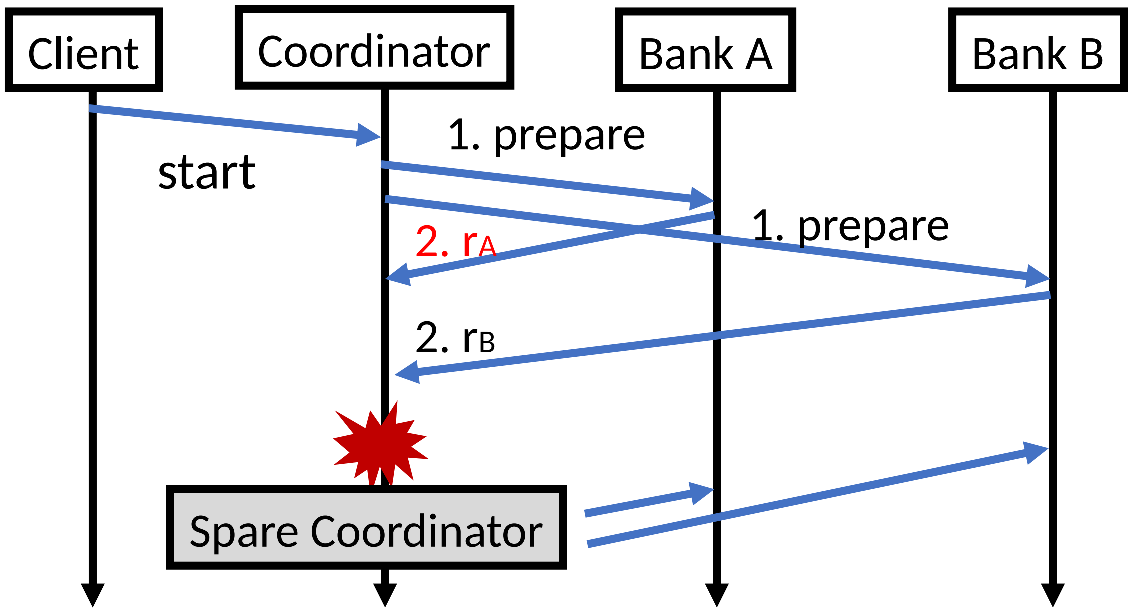


A and B wait for results from coordinator, *holding locks*

Problem of 2PC

System blocks if coordinator fails

- Workaround: add a spare coordinator
- Usually called non-blocking commit, 3PC
  - Correctness unknown, no complete algorithm



- Inconsistency arises if the old coordinator later is up and running
- Both claim to be the current coordinator but with different decisions
- Difficult to avoid and implement [Gray and Lamport, 2004]