Reliability: Replication + Topics in Distributed Coordination and Distributed Transactions

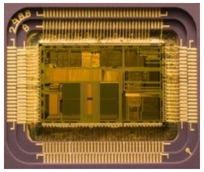
ACS, Yongluan Zhou

Do-it-yourself recap: Replication

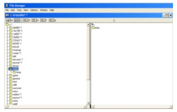
- **MAKE COPIES!! ☺**
  - State-machine replication
  - Asynchronous replication
    - Primary-Site
    - Peer-to-Peer
  - Synchronous replication
    - Read-Any, Write-All
    - Quorums

Replicated Interpreter

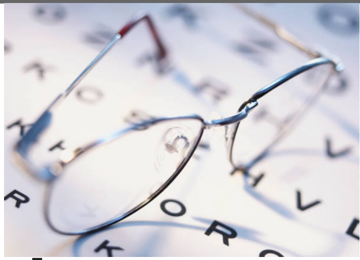
(loop (print (eval (read))))



Replicated Memory



- Techniques only good enough for a specific **failure model**
  - **Byzantine vs. crash vs. fail-stop**
  - What was the difference between synchronous and asynchronous replication?
  - What is the difference between the failure models above?



What should we learn today?

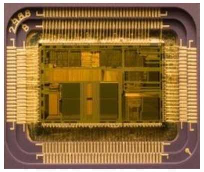
- Explain the difficulties of **guaranteeing atomicity in a replicated distributed system**
- Explain the **notion of state-machine replication** and the ISIS algorithm to **totally ordered multicast** among replicas
- Describe the implications of the **FLP impossibility result** and possible workarounds
- Explain mechanisms necessary for **distributed transactions**, such as distributed locking and distributed recovery
- Explain the operation of the **two-phase commit protocol** (**2PC**)
- Predict outcomes of 2PC under failure scenarios

Replication

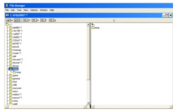
- **MAKE COPIES!!** ☺
  - State-machine replication
  - Asynchronous replication
    - Peer-to-Peer
    - Primary-Site
  - Synchronous replication
    - Read-Any, Write-All
    - Quorums

(loop (print (eval (read))))

Replicated Interpreter



Replicated memory



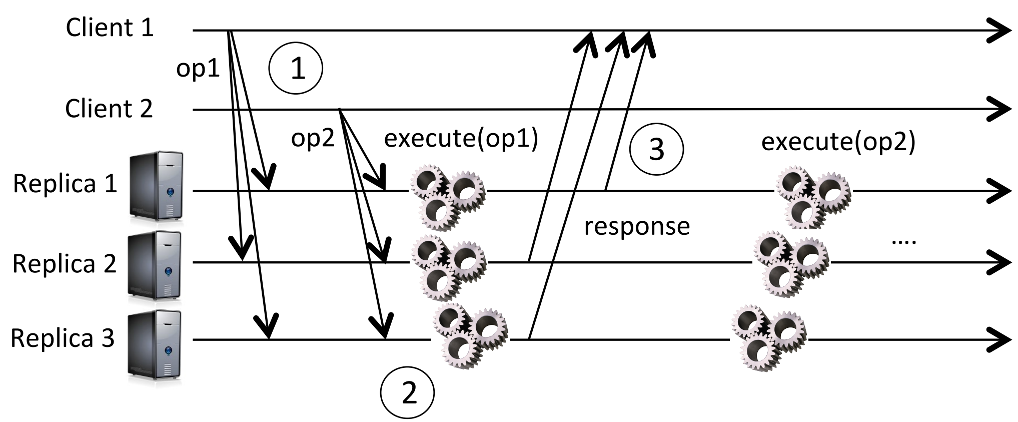
- Techniques only good enough for a specific **failure model**
  - Nuclear bomb
  - Component maliciously outputs random gibberish (**Byzantine**)
  - Components **crash** without telling you anything
  - Components are **fail-stop**

State Machines

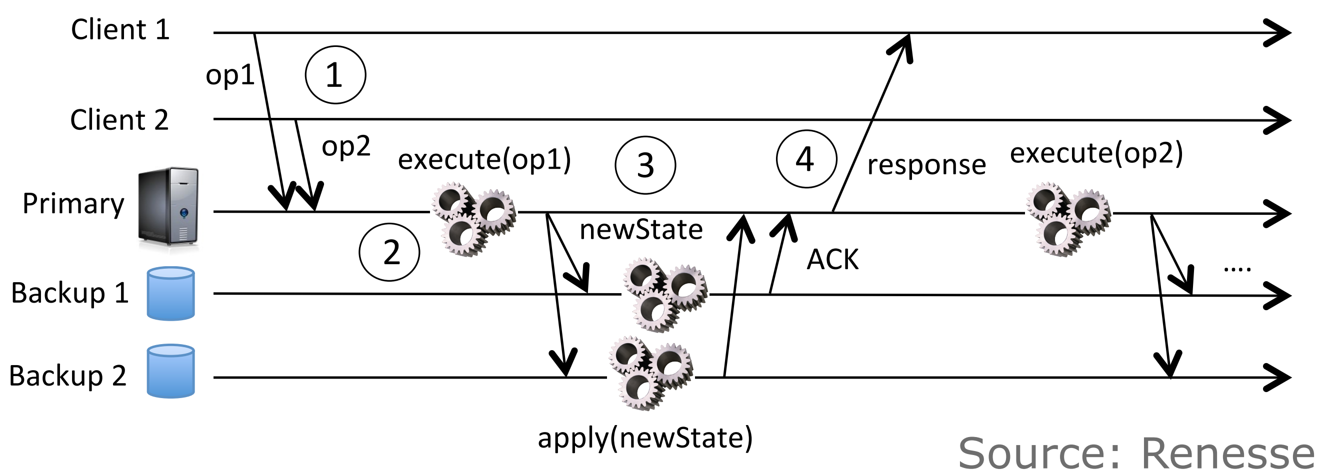
- A state machine consists of
  - State variables
    - encoding its state
  - Instructions
    - transforming its state

Active vs Passive Replication

- State Machine Replication is an active replication protocol



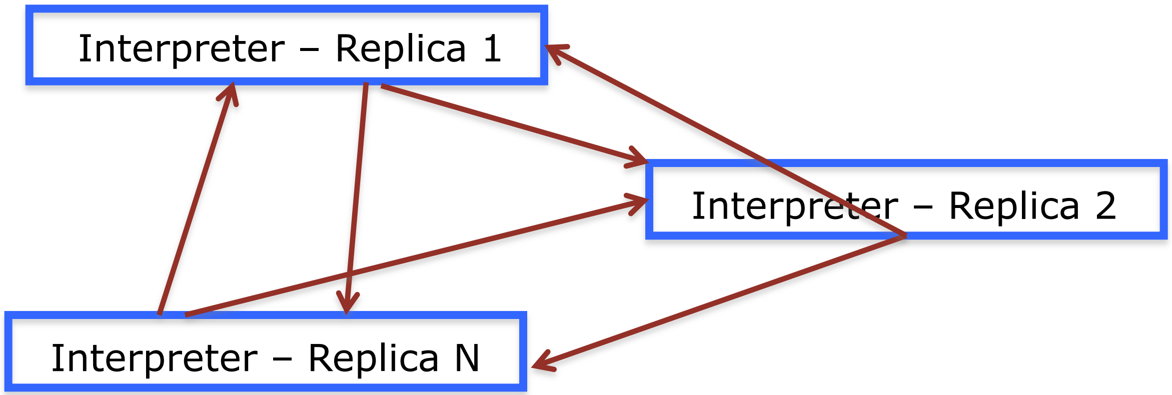
- Primary-Site Replication is a passive replication protocol



Source: Renesse et al.

Multicast: Distributing the Dispatcher

- Replicas implement **multicast** operation → internalize dispatcher



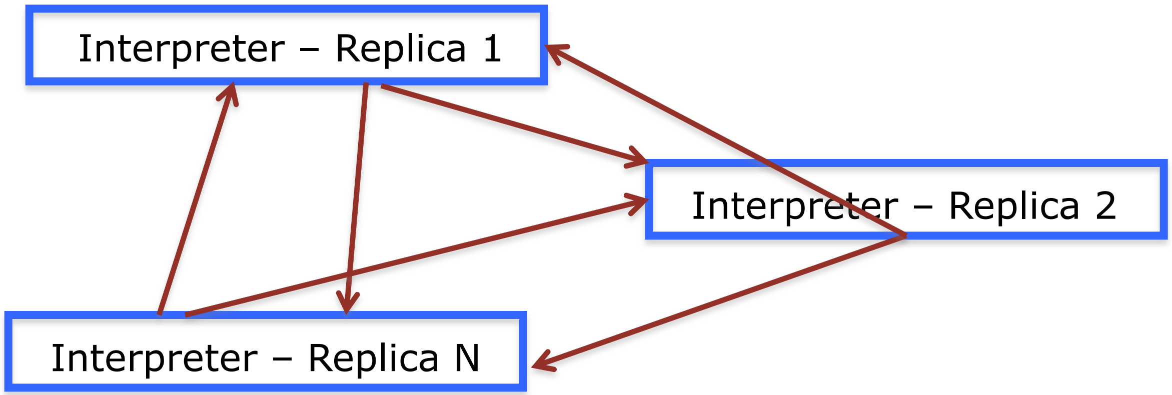
If replicas of interpreter get the **same inputs in the same order**, they will go to the same state and produce the same output

What might go wrong with the existence of faults of nodes or network links ?

What should the dispatchers ensure when sending the instructions to the replicas?

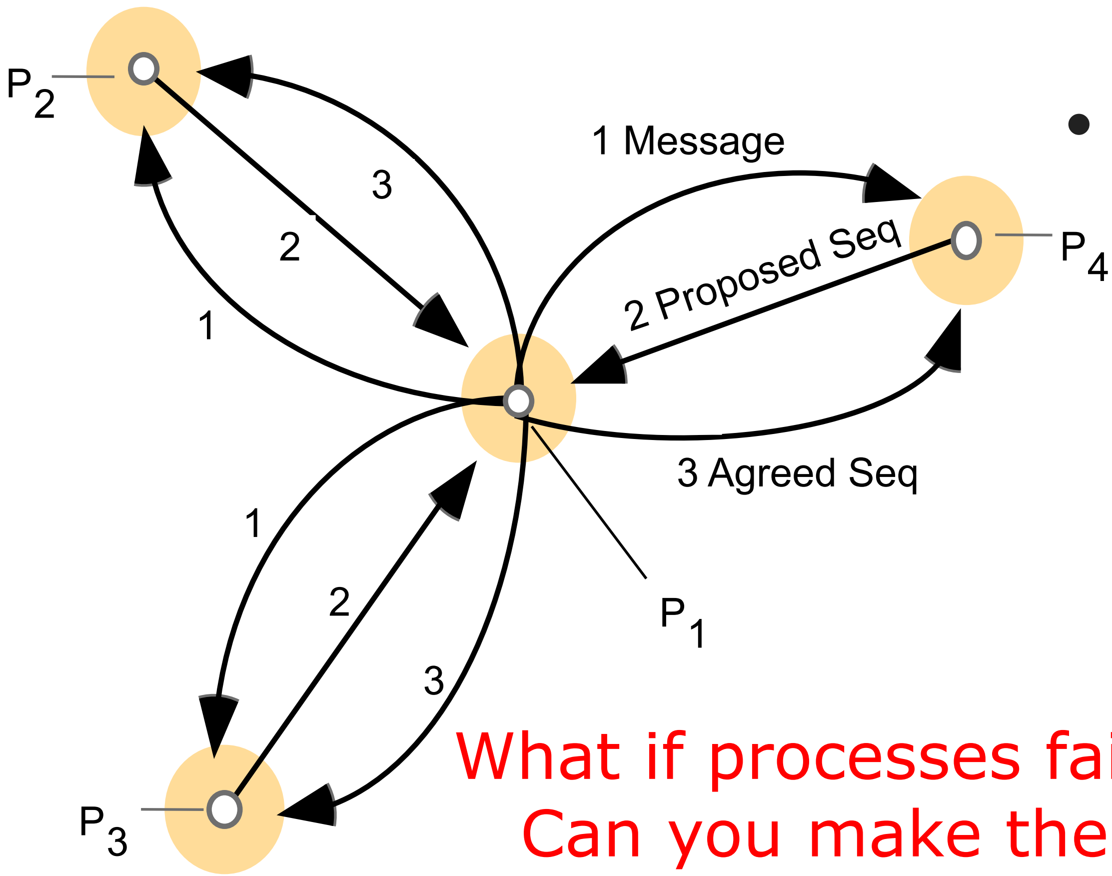
Multicast: Distributing the Dispatcher

- Replicas implement **multicast** operation → internalize dispatcher



- Must ensure **atomic** operation execution on all replicas
  - **All-or-nothing:** either in all correct replicas or none
    - Also called **Agreement**
  - **Before-or-after:** Equivalent to a total order
    - Also called **Order**
  - With at most *f* failures:
    - ***Fail-stop:*** requires *at least* N = *f* + 1 replicas
    - ***Byzantine:*** requires *at least* N = 2*f* + 1 replicas

Totally Ordered Multicast (ISIS)



- Assume for now no failures
- **Idea**
  - Process 1 sends message with identifier i to group
  - Every process p replies with proposed *seq#* = *max(accepted*$_p$*, proposed*$_p$*) + 1*
  - Process 1 selects maximum number and sends it to group
    - Note: ties in numbering broken by process numbers

What if processes fail? Can you make the multicast reliable?

Source: Coulouris et al (partial)

Reliable and Totally Ordered Multicast?

- If network **asynchronous** and assuming **crash** failures, **guaranteeing** reliable **and** totally ordered multicast is **IMPOSSIBLE**

- Fischer, Lynch, Patterson (FLP) result → Impossibility of Consensus
  - Set of processes with single binary variable
  - Want to decide outcome as 0 or 1 by just exchanging messages
  - **Intuition:** cannot make the difference between crashed process and process running very slowly
  - Adversary can **delay** consensus indefinitely
  - Does not mean that consensus cannot be reached in some cases!

Where to go from here?

- If network **asynchronous** and assuming **crash** failures, **guaranteeing** reliable **and** totally ordered multicast is **IMPOSSIBLE**

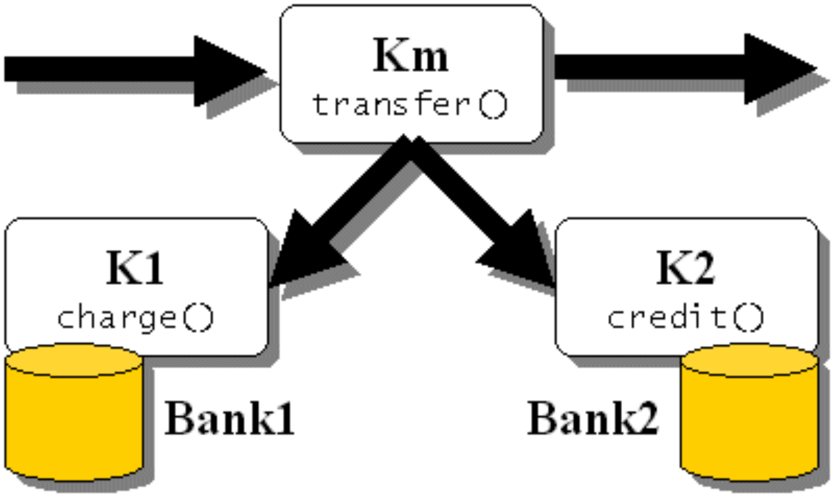
- **Solution 1:** Make model **fail-stop**, not crash
  - Instead of asynchronous system, make system behave as a (partially) synchronous one with reliable failure detector, e.g., timeout
  - Use failure detector to flag failed processes, no doubts
- **Solution 2:** Design **protocol that guarantees safety**, even if it cannot guarantee progress
  - Paxos example of such a protocol

Summary

- Many techniques for redundancy
- Replication widely used technique in practice
- Many flavors
  - State-machine replication
  - Asynchronous replication
    - Primary-Site
    - Peer-to-Peer
  - Synchronous replication
    - Read-Any, Write-All
    - Quorums
  - Tons of combinations of flavors possible!
    - E.g., primary-site + synchronous
  - Tons of variations in implementation according to failure model!
    - E.g., fail-stop, crash, Byzantine

Distributed Transactions

- Users should be able to write Xacts accessing multiple sites just like local Xacts
- Enforcing ACID calls for distributed locking, recovery, and commit protocols
- Hard to scale in number of sites in general
  - Use partitioning/replication techniques for trade-offs



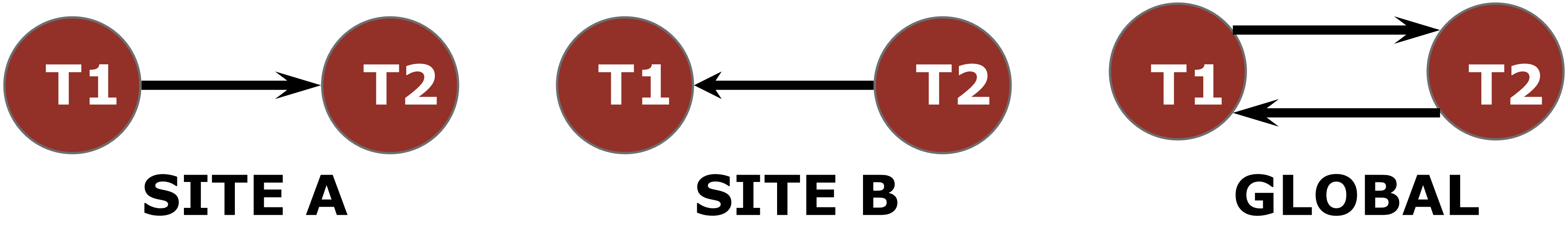
Source: Ramakrishnan & Gehrke (partial)

Distributed Locking

- How do we manage locks for objects across many sites?
- Centralized: One site does all locking
  - Vulnerable to single site failure
- Distributed: Locking for an object done at site where the object is stored

Distributed Deadlock Detection

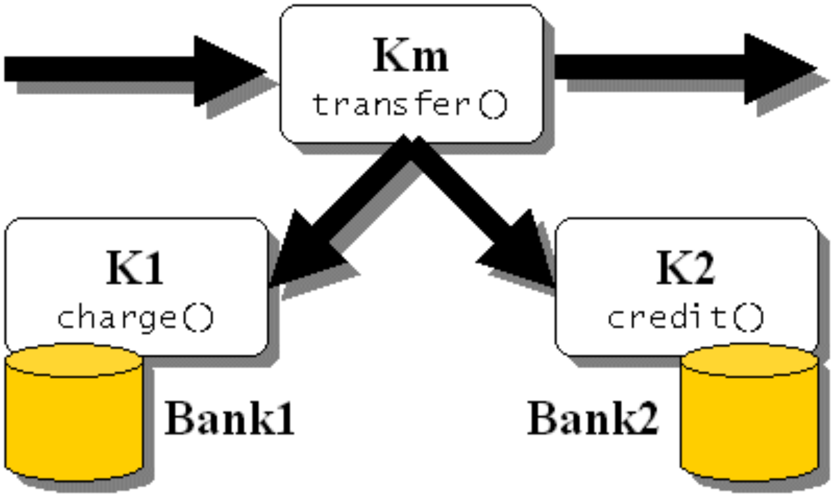
- Each site maintains a local waits-for graph
- A global deadlock might exist even if the local graphs contain no cycles:



- Three solutions
  - Centralized: send all local graphs to one site
  - Hierarchical: organize sites into a hierarchy and send local graphs to parent in the hierarchy
  - Timeout: abort Xact if it waits too long

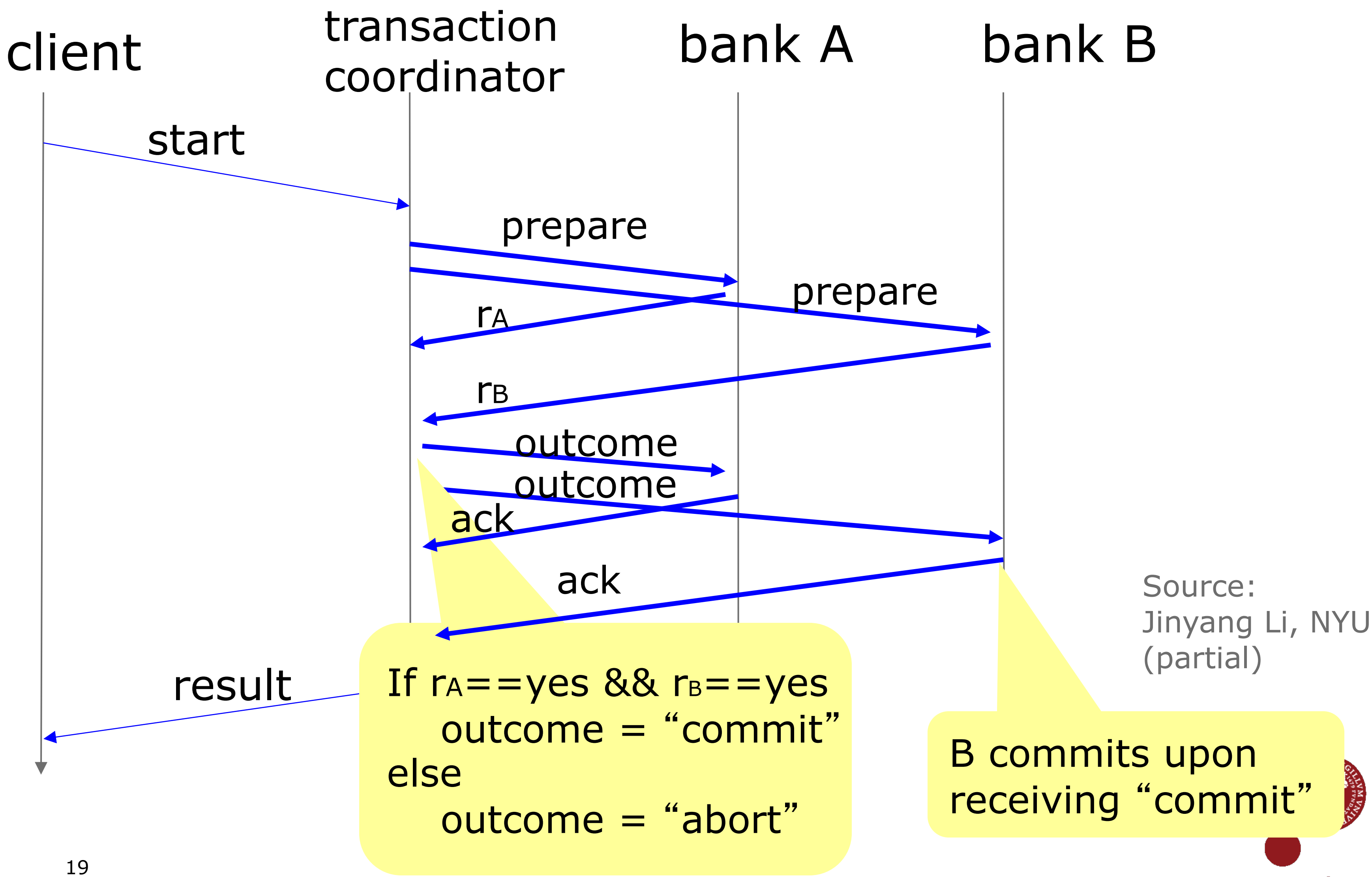
Distributed Recovery

- New issues:
  - New kinds of failure, e.g., links and remote sites
  - If “sub-transactions” of a Xact execute at different sites, all or none must commit
    - → Need a commit protocol to achieve this
- A log is maintained at each site, as in a centralized DBMS, and commit protocol actions are additionally logged



Source: Ramakrishnan & Gehrke (partial)

Two-Phase Commit (2PC)

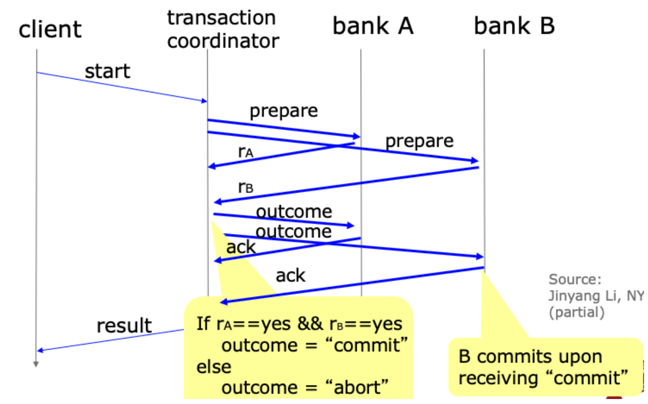


Comments on 2PC

- Two rounds of communication: first, voting; then, termination
  - Both initiated by coordinator
- **Any site** can decide to **abort** an Xact
- Every msg reflects a **decision** by the sender
  - To ensure that this decision survives failures, it is first **recorded in the local log**.
- All commit protocol log recs for an Xact contain Xactid and Coordinatorid
  - The coordinator's abort/commit record also includes ids of all subordinates/cohorts.

Source: Ramakrishnan & Gehrke (partial)

Discussion of Failures Scenarios



- Coordinator times out waiting for subordinate's "yes/no" response
  - Can coordinator unilaterally decide to commit?
  - Can coordinator unilaterally decide to abort?
- If subordinate *i* responded with "no" …
  - Can it unilaterally abort?
- If subordinate *i* responded with "yes" …
  - Can it unilaterally abort?
  - Can it unilaterally commit?

Source: Jinyang Li, NYU (partial)

Restart After a Failure at a Site

- If we have a commit or abort log rec for Xact T, but not an end rec, must redo/undo T
  - If this site is the coordinator for T, keep sending **commit/abort** msgs to subs until **acks** received
- If we have a prepare log rec for Xact T, but not commit/abort, this site is a subordinate for T
  - Repeatedly contact the coordinator to find status of T, then write **commit/abort** log rec; redo/undo T; and write **end** log rec
- If we don't have even a prepare log rec for T, unilaterally abort and undo T
  - This site may be coordinator! If so, subs may send msgs

Blocking

- If coordinator for Xact T fails, subordinates who have voted yes cannot decide whether to commit or abort T until coordinator recovers
- T is blocked

Link and Remote Site Failures

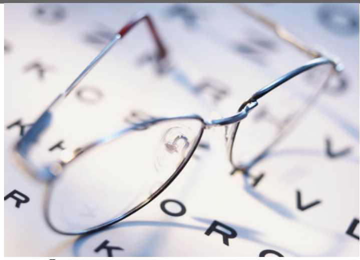
- If a remote site does not respond during the commit protocol for Xact T, either because the site failed or the link failed

- If the current site is the coordinator for T, should abort T
- If the current site is a subordinate, and has not yet voted yes, it should abort T
- If the current site is a subordinate and has voted yes, it is blocked until the coordinator responds

2PC with Presumed Abort

- When coordinator aborts T, it undoes T and removes it from the Xact Table immediately
  - Doesn't wait for **acks**; "presumes abort" if Xact not in Xact Table. Names of subs not recorded in **abort** log rec
- Subordinates do not send acks on abort
- If subxact does not do updates, it responds to prepare msg with reader instead of yes/no
- Coordinator subsequently ignores readers
- If all subxacts are readers, 2nd phase not needed

Source: Ramakrishnan & Gehrke



What should we learn today?

- Explain the difficulties of guaranteeing atomicity in a replicated distributed system
- Explain the notion of state-machine replication and the ISIS algorithm to totally ordered multicast among replicas
- Describe the implications of the FLP impossibility result and possible workarounds
- Explain mechanisms necessary for distributed transactions, such as distributed locking and distributed recovery
- Explain the operation of the two-phase commit protocol (2PC)
- Predict outcomes of 2PC under failure scenarios