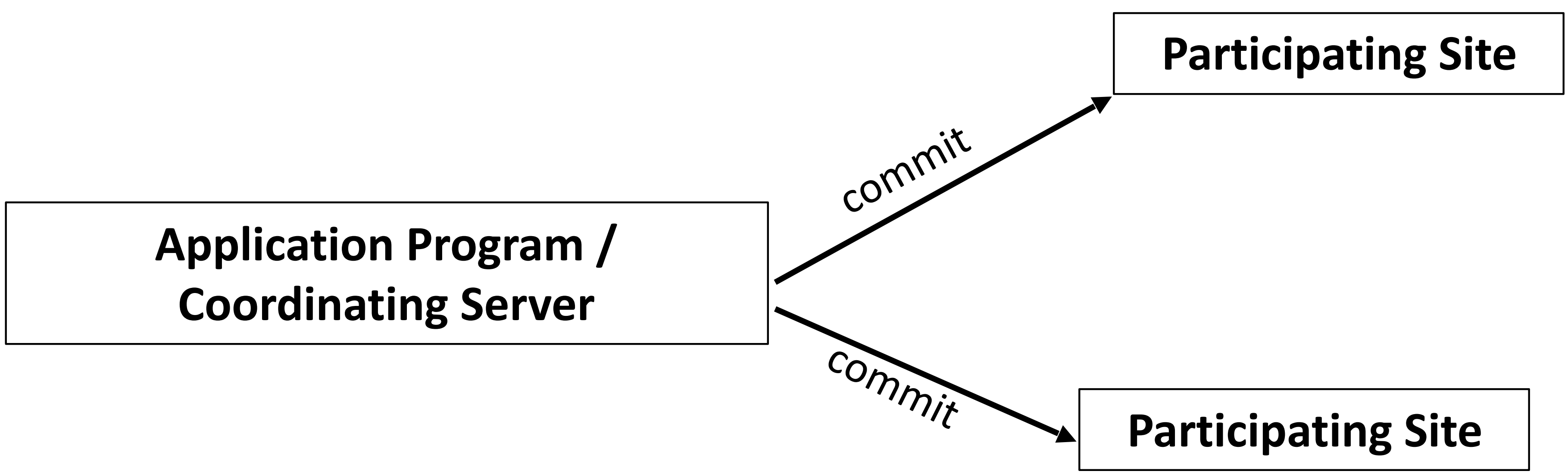
Distributed Transactions

CMPT 431

A Distributed Transaction

- A transaction is distributed across n processes.
- Each process can decide to commit or abort the transaction
- A transaction must commit on all sites or abort on all sites



Example

- Transfer money from bank A to bank B.
- Debit at A, credit at B, tell client "ok".
- We want both to do it, or both not to do it.
- We **never** want only one to act.
- We'd rather have nothing happen

A Naïve Approach

- *Client*, *Bank A*, *Bank B*, *transaction coordinator TC*
- Client sends transaction request to TC
- TC tells A and B to perform debit and credit
- A and B report “ok” to TC
- TC responds “ok” to the client

How Can This Fail?

- There's not enough money in A's bank account
- B's bank account no longer exists
- The network link to B is broken
- A or B has crashed
- TC crashes between sending the messages

What Do We Want to Happen?

- If A commits, B does not abort
- If A aborts, B does not commit
- A and B eventually decide one way or the other

Properties of Atomic Commitment

- <u>Property 1:</u> All participants that decide reach the same decision
- <u>Property 2:</u> If any participant decides **commit**, then all participants must have voted **YES**
- <u>Property 3:</u> If all participants vote **YES** and no failures occur, then all participants decide **commit**
- <u>Property 4:</u> Each participant decides at most once (a decision is irreversible)

A Distributed Transaction

 # Coordinator executes:

send [T_START: transaction, Dc, participants] to all participants

Dc is *compute delay* – time required to finish transaction

 # All participants (including the coordinator) execute:

upon (receipt of T_START: transaction, Dc, participants]

Cknow = local_clock

 # Perform operations requested by transaction

if(willing and able to make updates permanent) then

vote = YES

else vote = NO

#Decide commit or abort for the transaction

atomic_commitment(transaction, participants)

Components of Atomic Commitment

- Normal execution
  - The steps executed while no failures occur
- Termination protocol
  - When a site fails, the correct sites should still be able to decide on the outcome of pending transactions.
  - They run a termination protocol to decide on all pending transactions.
- Recovery
  - When a site fails and then restarts it has to perform recovery for all transactions that it has not yet committed
  - Single site recovery: safe to abort all transactions that were active at the time of the failure
  - Distributed system: might have to ask around; maybe an active transaction was committed in the rest of the system, so you have to commit it as well

Two-Phase Commit Protocol (2PC)

- An atomic commitment protocol
  - Phase 1: Decide **commit** or **abort**
  - Phase 2: Get the final decision from the coordinator, and execute the final decision
- We will study the protocol in a ***synchronous system***
  - Assume a message arrives within interval **δ**
  - Assume we can compute **Dc** – local time required to complete the transaction
  - Assume we can compute **Db** – additional delay associated with broadcast

Two-Phase Commit Protocol (I)

 # Executed by coordinator

procedure atomic_commitment(transaction, participants)

send [VOTE_REQUEST] **to all** participants

set-timeout-to local_clock + 2δ

wait-for (receipt of [vote: vote] messages from all participants)

if (all votes are YES) **then**

broadcast (**commit**, participants)

else broadcast (**abort**, participants)

on-timeout

broadcast (**abort**, participants)

Two-Phase Commit Protocol (II)

 # Executed by all participants (including the coordinator)

```
    set-timeout-to Cknow + Dc + δ
    wait-for (receipt of [VOTE_REQUEST] from coordinator)
        send [vote: vote] to coordinator
        if (vote = NO) then
            decide abort
        else
            set-timeout-to Cknow + Dc + 2δ + Db
            wait-for (delivery of decision message)
                if(decision message is abort) then
                    decide abort
                else decide commit
            on-timeout
                What should we do?
    on-timeout
        decide abort
```

Options:

1. Wait forever
2. Run a termination protocol

The Need for Termination Protocol

- If a participant
  - Voted YES
  - Sent its decision to coordinator, but….
  - Received no *final* decision from coordinator
- A ***termination protocol*** must be run
  - Participants cannot simply decide to abort
  - If they already said they would commit, they cannot change their minds
  - ***The coordinator might have sent "commit" decisions to some participants and then crashed***
  - Since those participants might have committed, no other participant can decide "abort"
- A termination protocol will try to contact other participants to find out what they decided, and try to reach a decision

Termination Protocol (for B if it voted “YES”)

- B sends "status" request message to A
  - Asks if A knows whether transaction should commit
- If B doesn't hear reply from A
  - No decision, wait for coordinator
- If A received "commit" or "abort" from coordinator
  - B decides the same way
  - Can't disagree with the coordinator...
- If A hasn't voted yes/no yet
  - B and A both abort
  - Coordinator can't have decided "commit", so it will eventually hear from A or B

Termination Protocol (cont.)

- If A voted "no”
  - B and A both abort
  - Coordinator can't have decided "commit"
- If A voted "yes"
  - No decision possible!
  - Coordinator might have decided "commit". Or coordinator might have timed out and aborted. A and B must wait for the coordinator

- Does this protocol guarantee correctness?
- Does it guarantee termination?
  - No, A and B will block in case where decision is impossible

Blocking vs. Non-Blocking Atomic Commitment

- Blocking Atomic Commitment: correct participants may be prevented from terminating the transaction due to failures of other part of the system
- Non-Blocking Atomic Commitment: transactions terminate consistently at all participating sites even in the presence of failures

Blocking Nature of Two-Phase Commit

- Scenario that leads to blocking in the termination protocol:
  - The coordinator crashes during the broadcast of a decision
  - Several participants received the decision from coordinator, applied it, and then crashed
  - All other (not crashed) participants voted “YES”, so they cannot abort
  - ***Correct participants cannot decide until faulty participants recover***

Atomic Commitment Problem

- Can we say that a two-phase commit will EVENTUALLY terminate in an asynchronous system?
- No. Termination protocol may block
- But it is still used in asynchronous systems under certain assumptions ***but with no guarantees about termination***:
  - Communication is reliable
  - Processes can crash
  - Processes eventually recover from failure
  - Processes can log their state in stable storage
  - Stable storage survives crashes and is accessible upon restart

2PC in Asynchronous Systems

- 2PC can be implemented in an asynchronous system with ***reliable communication channels***
- This means that a message ***eventually*** gets delivered…
- But we cannot set bounds on delivery time
- So the process might have to wait forever…
- Therefore, ***you cannot have non-blocking atomic commitment in an asynchronous system***
- What if a participant whose message is waited on has crashed?
- The expectation is that the participant will properly *recover* and continue the protocol
- So now let's look at ***distributed recovery***

Distributed Recovery

- Remember single-site recovery:
  - transaction log records are kept on stable storage
  - upon reboot the system "undoes" updates from active or uncommitted transactions
  - "replays" updates from committed transactions
- In a distributed system we cannot be sure whether a transaction that was active at the time of crash is:
  - Still active
  - Has committed
  - Has aborted
  - Maybe it has executed more updates while the recovering site was crashing and rebooting

Crash Before Local Decision

- Suppose a site crashes during the execution of transaction, before it reaches local decision (YES or NO)
- The transaction could have completed at other sites
- What are the options?
- Option 1: The crashed site restores its state with help of other participants (restore the updates made while it was crashing and recovering)
- Option 2: The crashed site realizes that it crashed (by keeping the crash count), and sets local decision to NO

Crash After Local Decision

- Actions performed by recovering site:
  - For each transaction that was active before the crash, try to decide unilaterally based on log records (if the coordinator message had been received, decide based on that message)
  - If no coordinator message was received: ask others what they have decided
- Actions performed by other participants
  - Send the decision
  - Or a “don’t know” message

Logging for Distributed Recovery

- Coordinator: forces **"commit" decision** to log before informing any participants
  - *Like redo rule for single-site logging*
- Participant: forces **its vote** (YES or NO) to disk before sending the vote to coordinator –
  - This way it knows that it must reach decision in agreement with others
- Participant: forces **final decision** (received from coordinator) to the log, then responds to the coordinator
  - Once the coordinator receives responses from all participants, it can remove its own decision log record

Distributed Concurrency Control

- Multiple servers execute transactions, they share data distributed across sites
- A lock on data may be requested by many different servers
- Distributed concurrency control methods:
  - Centralized two-phase locking (C-2PL)
  - Distributed two-phase locking (D-2PL)
  - Optimistic concurrency control

Distributed Concurrency Control: Notation

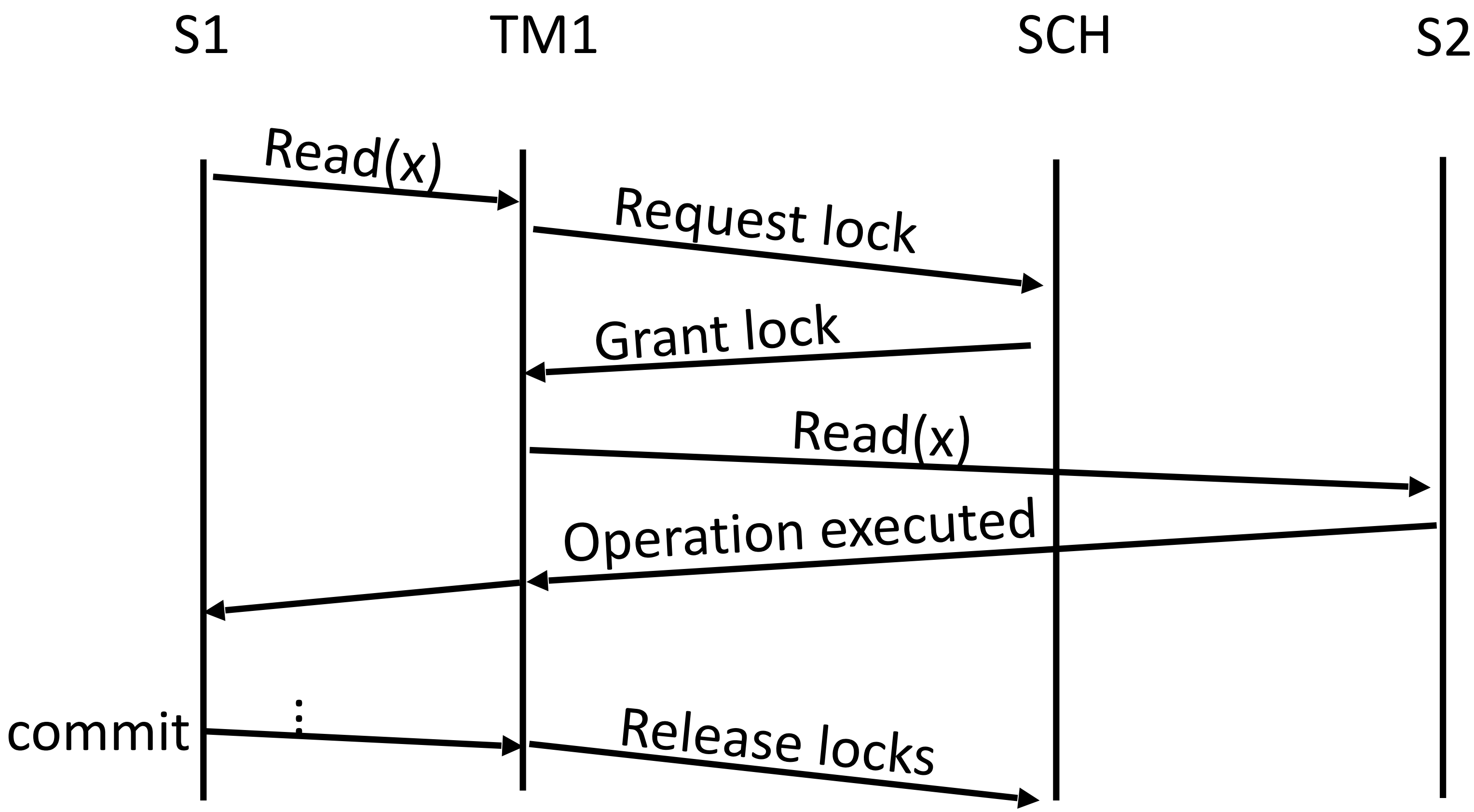
- S1, S2, … - servers performing a distributed transaction
- T_i – a transaction
- O_{ij} – an operation that's part of T_i
- $O_{ij}(X)$ – an operation requiring a lock on X

- SCH - global lock scheduler
- TM1, TM2, … - transaction managers – one for each server

Centralized 2PL

- Let S1 be the server to which transaction T_i was submitted
- Let S2 be the server maintaining data X
- For each operation $O_{i1}(X)$, TM1 first requests the corresponding lock from SCH (the central 2PL scheduler)
- Once the lock is granted, the operation is forwarded to the server S2 maintaining X.

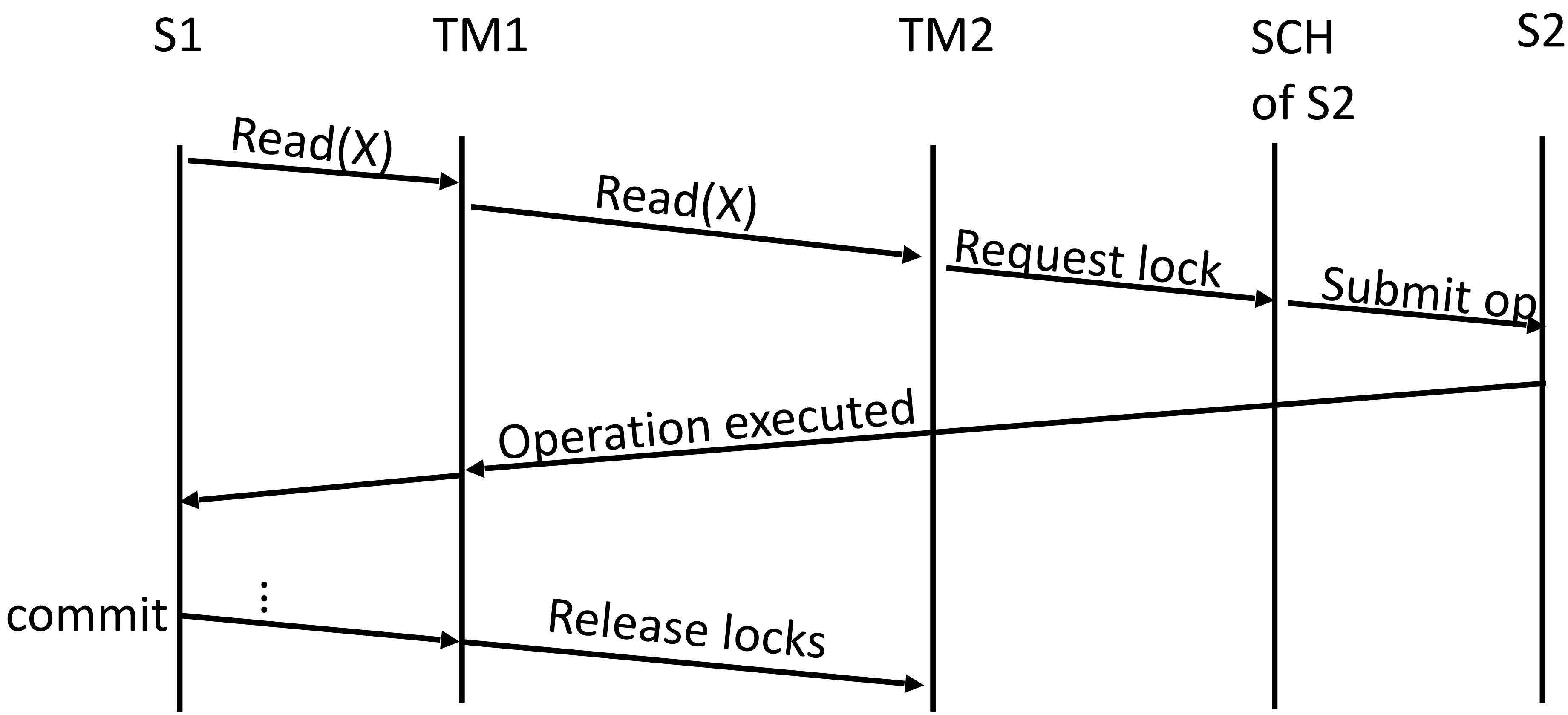
Centralized 2PL



Distributed 2PL

- Each server has its own local 2PL scheduler SCH
- Let S1 be the server to which transaction T_i was submitted:
  - For each operation $O_{i1}(X)$, TM1 forwards the operation to the TM2 of server S2 maintaining X
  - The remote site first acquires a lock on X and then submits the execution of the operation.

Distributed 2PL



Optimistic Concurrency Control

- Locking is conservative
  - Locking overhead even if no conflicts
  - Deadlock detection/resolution (especially problematic in distributed environment)
  - Lock manager can fail independently
- Optimistic concurrency control
  - Perform operation first
  - Check for conflicts only later (e.g., at commit time)

Optimistic Concurrency Control

- Working Phase:
  - If first operation on X, then load last committed version from DB and cache
  - Otherwise read/write cached version
  - Keep **WriteSet** containing objects written
  - Keep **ReadSet** containing objects read
- Validation Phase
  - Check whether transaction conflicts with other transactions
- Update Phase
  - Upon successful validation, cached version of updated objects are written back to DB (= changes are made public)
- Validation can be eager or lazy
  - Eager: check for conflicts as objects are accessed
  - Lazy: check for conflicts at commit time

Distributed Deadlock Resolution

- Similar remedies as for single-site deadlock resolution:
  - Prevention (lock ordering)
  - Avoidance (abort transaction that waits for too long)
  - Detection (maintain a wait-for graph, abort transactions involved in a cycle)
- Deadlock avoidance and detection require keeping dependency graphs, or wait-for-graphs (WFGs)
- WFGs are more difficult to construct in a distributed system (takes more time, must use vector clocks or distributed snapshots)
- Deadlock managers can fail independently

Summary

- Atomic commitment
  – Two-phase commit
  – Non-blocking implementation possible in a synchronous system with reliable communication channel
  – Possible in an asynchronous system, but not guaranteed to terminate (blocking)
- Distributed recovery
  – Keep state on stable storage
  – When reboot, ask around to recover the most current state
- Distributed concurrency control
  – Centralized lock manager
  – Distributed lock manager
  – Optimistic concurrency control