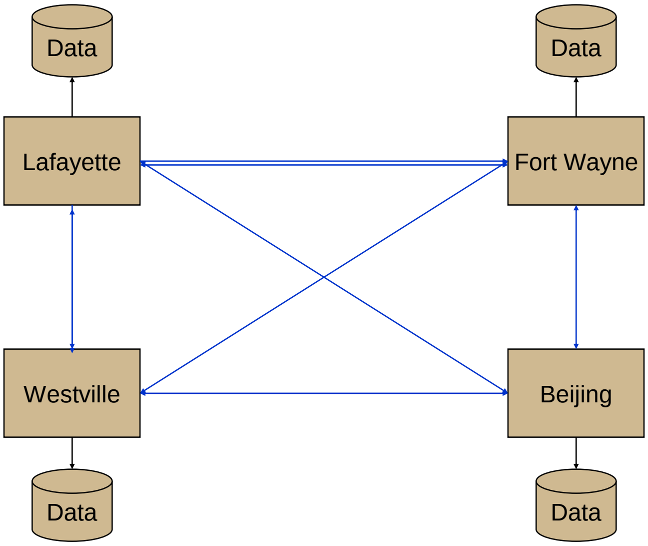
CS 44800: Introduction To Relational Database Systems

Distributed Databases

Prof. Chris Clifton

9 December 2021

Distributed Database



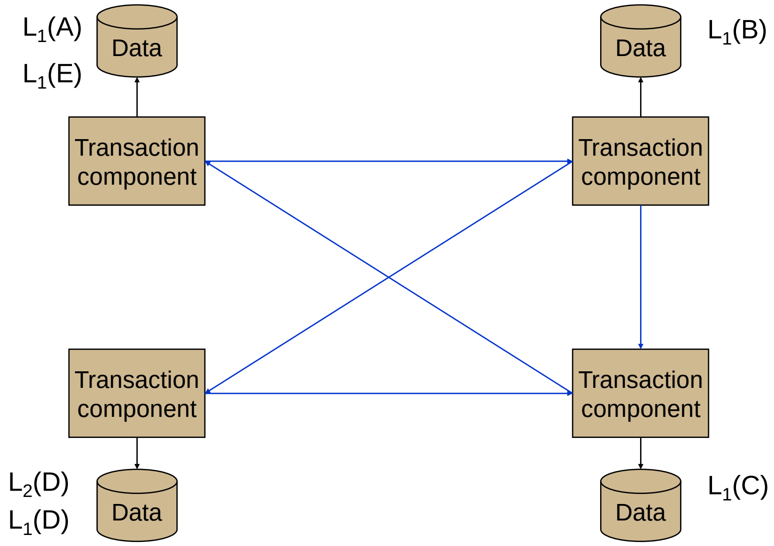
Distributed Database: Why?

- Performance
  - – Put the data close to the users
  - – Parallelism
- Resilience
  - – Fewer failures that can stop users from reaching the data
- Redundancy
  - – Copies of data to handle media failure
  - – Continue running when one machine fails

Distributed Database: Challenges

- Where’s the data?
  - – Easy – Block ID includes location
  - – Hard – What if we want to move the data?
  - – Harder – What about multiple copies of the data?
- Query Processing
  - – Query may access multiple sites
- Concurrency Control
  - – *Distributed* transactions
- Failure/recovery
  - – Is the fail-stop model still appropriate?

Distributed Transaction: Locking



$L_1(A)$
$L_1(E)$
Data

Data
$L_1(B)$

Transaction component

Transaction component

Transaction component

Transaction component

$L_2(D)$
$L_1(D)$
Data

Data
$L_1(C)$

Distributed Locking

- All locks are local!
  - Checking for locks happens at each site
  - Wait happens at each site
  - No need to communicate
- 2-phase locking is global
  - No transaction component can release a lock until all are done obtaining the lock
- We get serializability!

Distributed Concurrency Control: Challenges

- Deadlock detection/prevention: Deadlocks can be distributed
  - T_1 waiting for T_2 at site A
  - T_2 waiting for T_1 at site B
- *Replication*
  - What if we want to have multiple copies of the data?

Replicated Data

- Thus far, we have assumed that there is only a single copy of each data item.
- This copy is placed at one of the sites, which is responsible for concurrency control and recovery for that data item.
- However, for a data item that is accessed often from different sites, this could lead to a significant amount of communication.
- Moreover, when a sites fails, all data residing on that site becomes unavailable.

PURDUE
UNIVERSITY

Department of Computer Science

Replication

- To increase availability of data, and to reduce communication for remote data, data can be replicated.
- From the user's point of view, replication (like distribution, physical and logical organization of data), should be transparent.
- I.e. the user should not be aware that some (or all) data items are replicated, and should see no difference in performance.
- The user can be a programmer or an end user.

PURDUE
UNIVERSITY

Department of Computer Science

1 Copy Serializability

- The correctness definition for replicated databases is therefore that it should behave as though all transactions are executed in a ***serial manner on a single copy database***.
- This is the notion of one copy serializability, I.e. 1SR.
- The user must be given a one copy view of the database.
- How is this achieved?
- Read-only is easy. For writes we must manage carefully!

Department of Computer Science

Write-All approach

- This is the obvious first solution:
  - Reads can be satisfied by any copy in the system,
  - Writes must all modify *every* copy of the data item being written.
- This is a very effective solution – it completely eliminates the problem of multiple copies, and gives each txn the correct view.
  - Lock each copy
  - If someone reading a copy, we can't get write lock
- Very poor in terms of performance and progress:
  - Failures have a crippling effect on transactions!

Quorum Consensus Protocol

Quorum consensus protocol for locking

- Each site is assigned a weight; let S be the total of all site weights
- Choose two values **read quorum** $\mathrm{Q_R}$ and **write quorum** $\mathrm{Q_W}$
  - Such that $Q_r + Q_w > S$ and $2 * Q_w > S$
- Each read must lock enough replicas that the sum of the site weights is $\geq Q_r$
- Each write must lock enough replicas that the sum of the site weights is $\geq Q_w$
- Can choose Q_r and Q_w to tune relative overheads on reads and writes

PURDUE UNIVERSITY
Department of Computer Science

Google Spanner

- SQL-based query language
  - MapReduce based execution
- (Dyamically) replicated data
  - High availability
- Read/write consistency
  - *Timestamp* based serializability

PURDUE UNIVERSITY
Department of Computer Science

Google Spanner Data Replication

- Data divided into *Zones*
  - Replication across zones
  - May be thousands of servers in a zone
  - Placement in a zone dynamic (location proxies)
  - Similar to BigTable (Servers)
- Internally: *tablet* abstraction
  - Maps (key, timestamp) → string
- Lock Table at each replica

Overview

- Feature: Lock-free distributed read transactions
- Property: External consistency of distributed transactions
  - First system at global scale
- Implementation: Integration of concurrency control, replication, and 2PC
  - Correctness and performance
- Enabling technology: TrueTime
  - Interval-based global time

PURDUE UNIVERSITY
Department of Computer Science

Concurrency Control

- Three types of transactions
  - Read-write
  - Snapshot Transactions
    - Pre-declared as having no writes
  - Snapshot reads
    - Weak consistency guarantee
    - “sufficiently up to date”
- All data timestamped

Consistency: Read/Write

- Read-write uses strict two-phase locking
  - Locks held until commit
- Timestamp assigned after all locks acquired
  - Timestamps assigned by “leader” at each site
  - All writes have that timestamp
- Replicas track “safe time” – maximum timestamp at which a replica is up-to-date
  - Infinite if no transactions operating on object
  - Otherwise timestamp of first completed (but not committed) transaction
- Serializability is timestamp order
  - If T_2 starts after T_1 commits, must have later timestamp

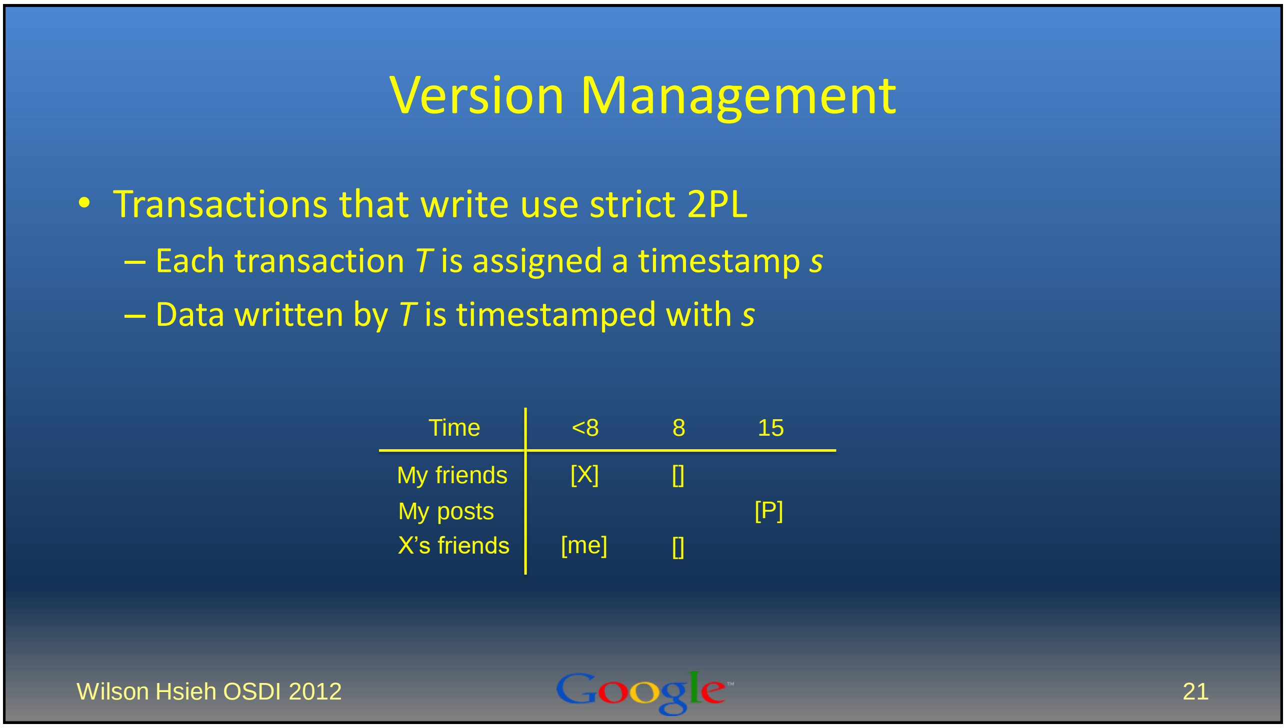
---

Consistency: Reads

- Read transactions assigned a timestamp
  - Only read data written before that timestamp
  - Can’t read data if timestamp > safe time

$$T_{write} < T_{read} < T_{safe}$$

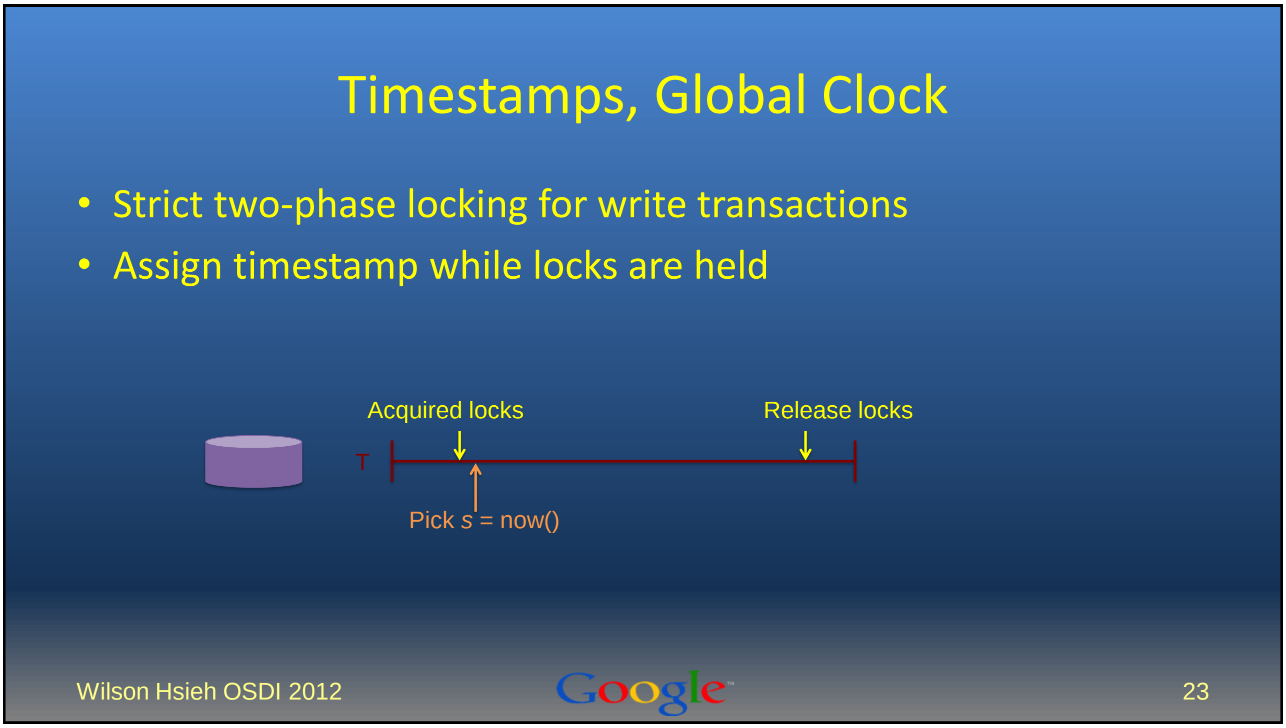
- What to assign as a timestamp?
  - Current time means replicas may be past “safe”
  - Can assign “old” timestamps, more replicas are okay
  - *Read transactions may serialize before they actually start*



Version Management

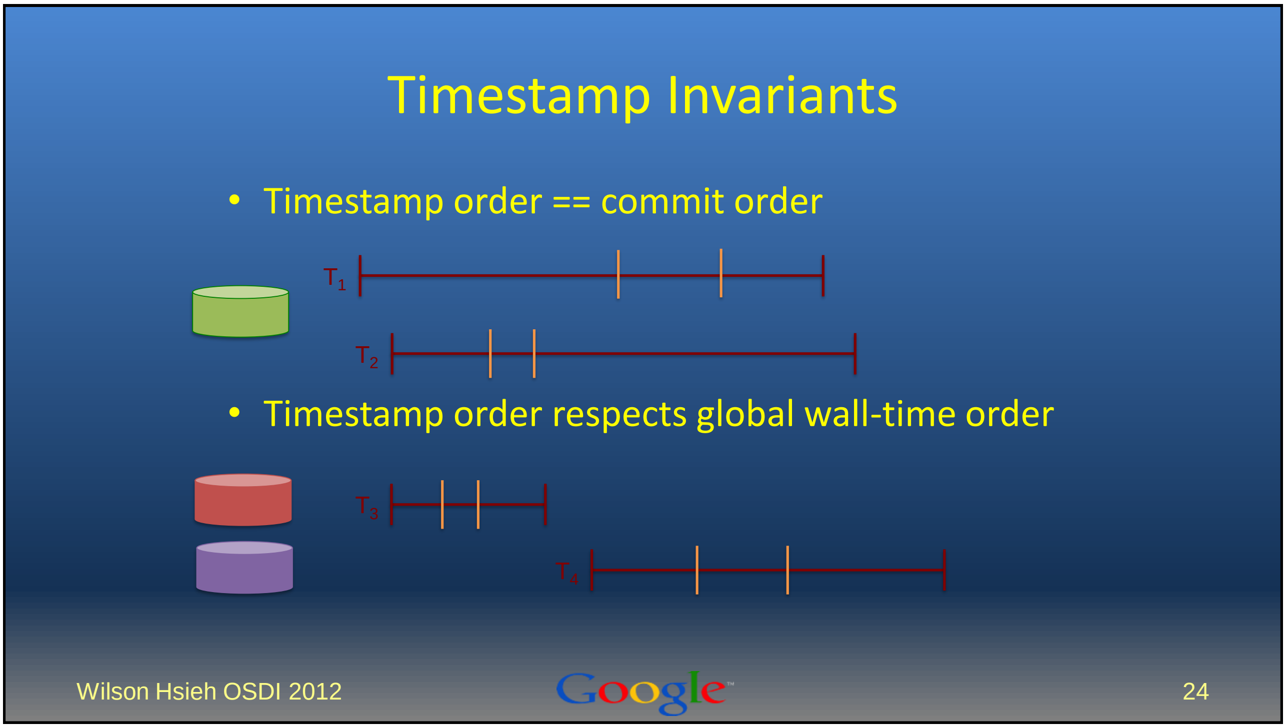
- Transactions that write use strict 2PL
  - Each transaction *T* is assigned a timestamp *s*
  - Data written by *T* is timestamped with *s*

Time	<8	8	15
My friends	[X]	[]	
My posts			[P]
X's friends	[me]	[]	



Timestamps, Global Clock

- Strict two-phase locking for write transactions
- Assign timestamp while locks are held



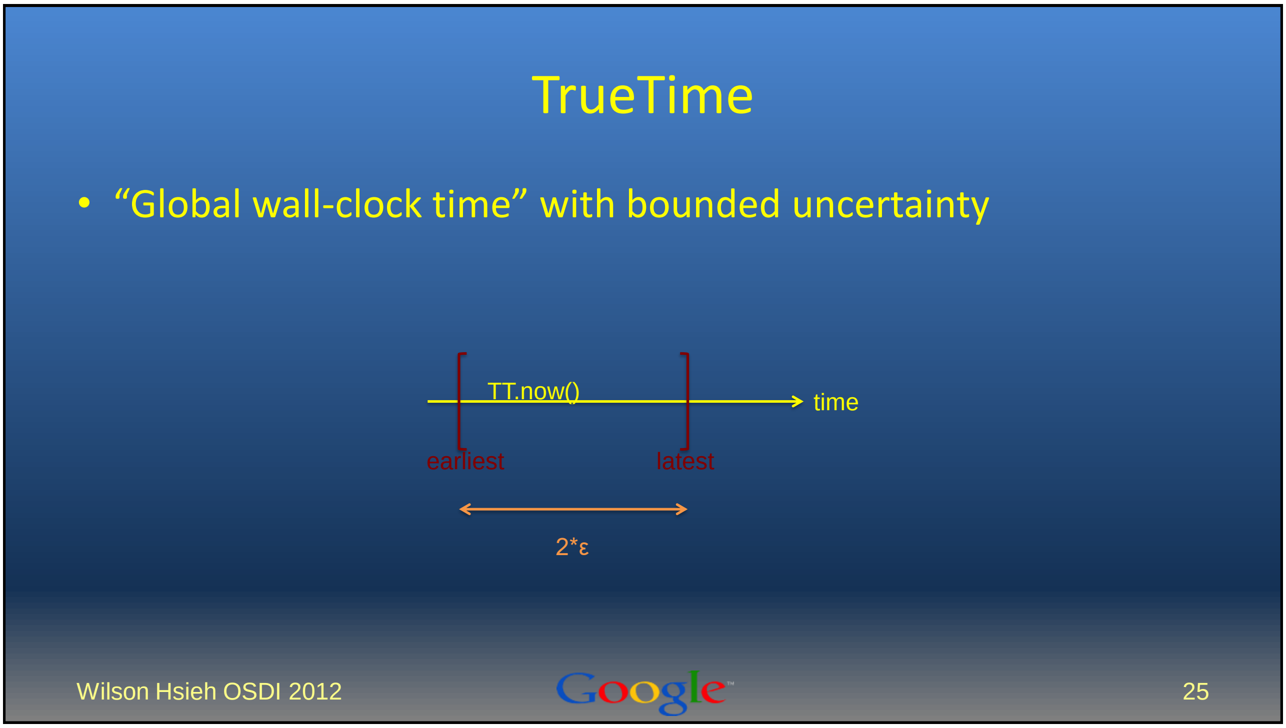
Timestamp Invariants

- Timestamp order == commit order

- Timestamp order respects global wall-time order

Wilson Hsieh OSDI 2012

24

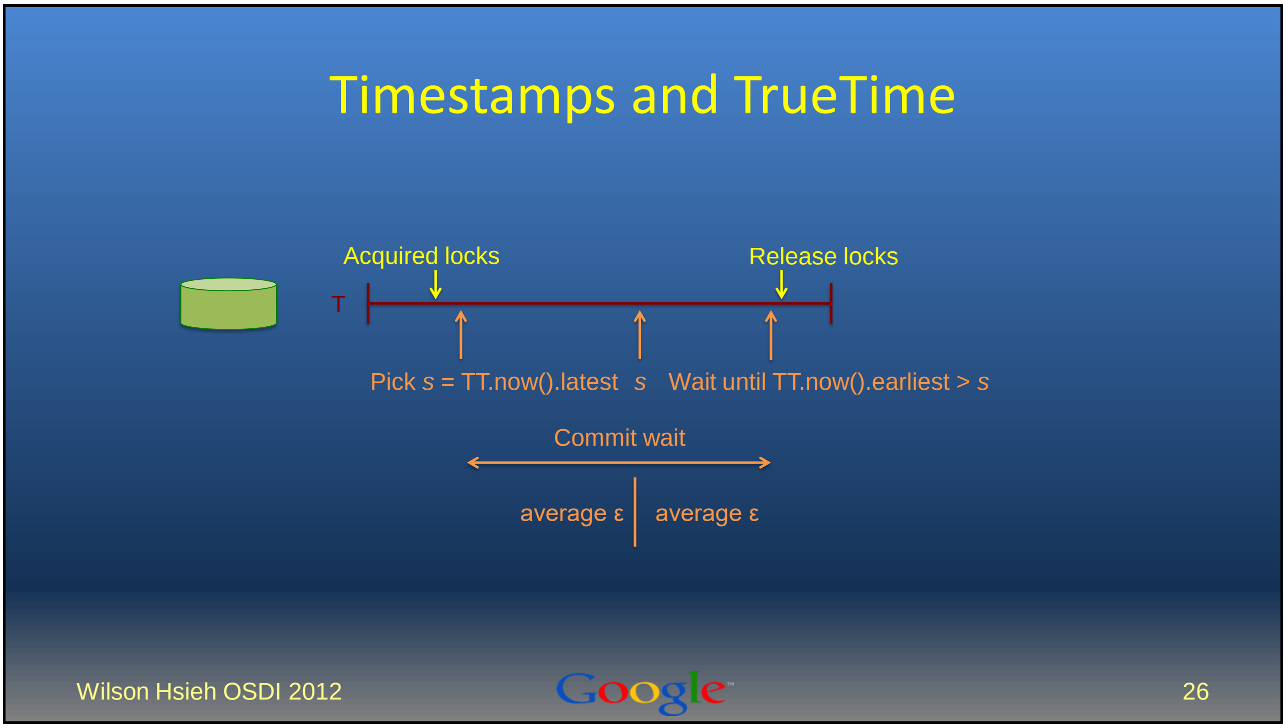


TrueTime

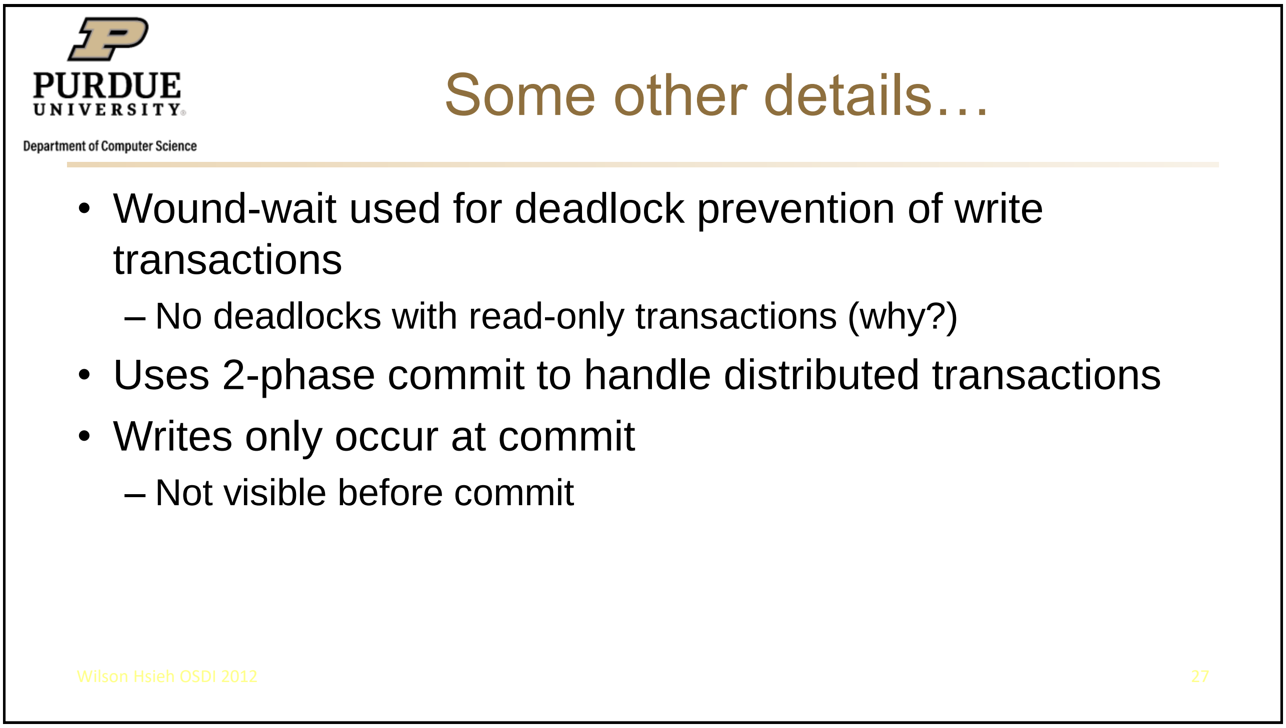
- "Global wall-clock time" with bounded uncertainty

Wilson Hsieh OSDI 2012

25



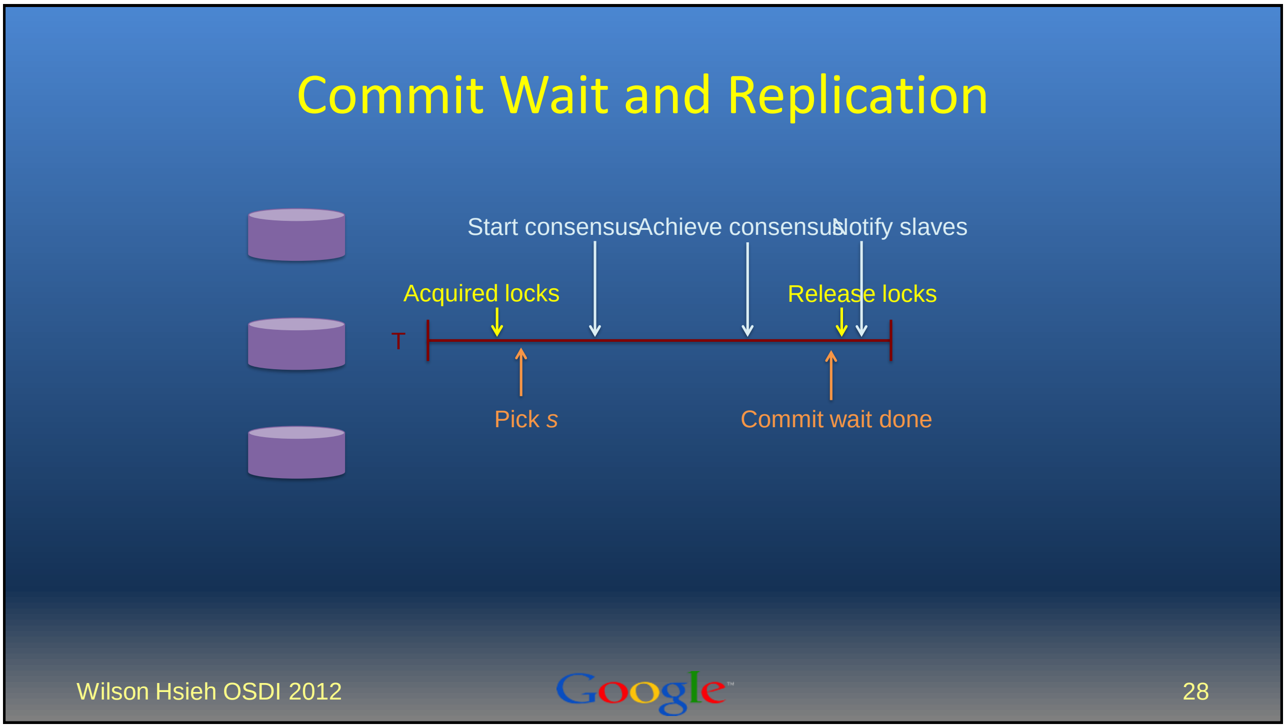
Timestamps and TrueTime



PURDUE UNIVERSITY
Department of Computer Science

Some other details…

- Wound-wait used for deadlock prevention of write transactions
  - – No deadlocks with read-only transactions (why?)
- Uses 2-phase commit to handle distributed transactions
- Writes only occur at commit
  - – Not visible before commit



Commit Wait and Replication

Start consensus

Achieve consensus

Notify slaves

Acquired locks

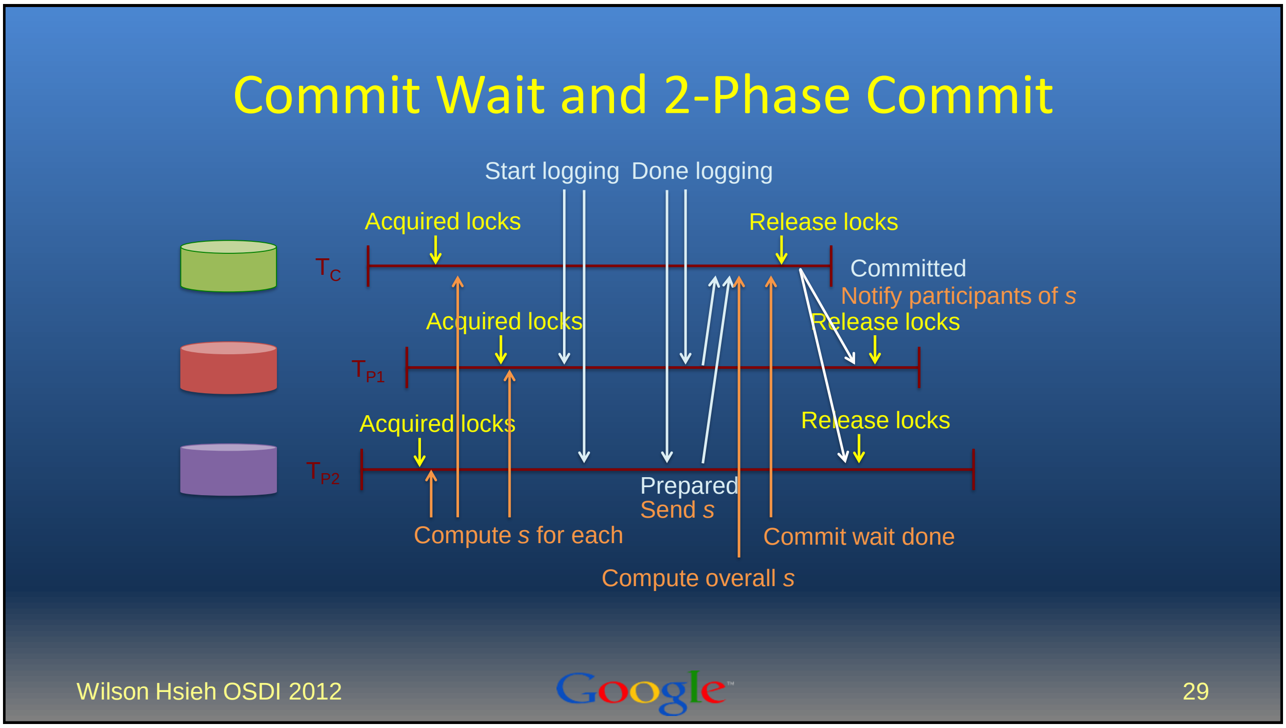
Release locks

T

Pick *s*

Commit wait done

Wilson Hsieh OSDI 2012



Commit Wait and 2-Phase Commit

Start logging

Done logging

Acquired locks

Release locks

T_C

Committed

Notify participants of *s*

Acquired locks

Release locks

T_{P1}

Acquired locks

Release locks

T_{P2}

Prepared

Send *s*

Compute *s* for each

Commit wait done

Compute overall *s*

Wilson Hsieh OSDI 2012

Write-All-Available

- Allow a transaction to proceed even though failures make it impossible to write all copies of the data.
- Allow the transaction to simply write to every site that is available. Those that are down can be ignored.
- Thus some copies of the data may be out of sync, i.e., may not contain the latest updates.

Example

- Consider the following execution. Note that multiple copies are marked using the upper case subscripts.
  $w_0[x_A]\ w_0[x_B]\ w_0[y_C]\ c_0\ r_1[y_C]\ w_1[x_A]\ c_1\ r_2[x_B]\ w_2[y_C]\ c_2$
- T_2 reads copy x_B from T_0, even though it should have read from T_1.
- Thus the above history is not equivalent to $T_0T_1T_2$.
- Is it equivalent to some other serial one-copy history?
- NO! $w_0[y_C] < r_1[y_C] < w_2[y_C]$, there is no other equivalent serial execution.
- This is interesting, because the execution actually seems to be a serial execution of the transactions!!!

Example (contd.)

- So what has gone wrong?
- The problem is that the write by T_1 into x, did not update all copies of x – x_B in particular.
- This could only mean that site B must have been down when T_1 wrote x, and must have recovered before T_2 read x.
- I.e. the failures must have been as such:

$w_0[x_A]\ w_0[x_B]\ w_0[y_C]\ c_0\ r_1[y_C]\ fail_B\ w_1[x_A]\ c_1\ \ Recover_B\ r_2[x_B]\ w_2[y_C]\ c_2$

- Thus the problem is that T_2 read a copy at a site that had failed and upon recovery did not re-sync with the other sites!
  - Recovery necessary to get concurrency control right!

Handling Failures with Majority Protocol

- The majority protocol with version numbers
  - Each replica of each item has a **version number**
  - Locking is done using majority protocol, as before, and version numbers are returned along with lock allocation
  - Read operations read the value from the replica with largest version number
  - Write operations
    - Find highest version number like reads, and set new version number to old highest version + 1
    - Writes are then performed on all locked replicas and version number on these replicas is set to new version number
- Read operations that find out-of-date replicas may optionally write the latest value and version number to replicas with lower version numbers
  - no need to obtain locks on all replicas for this task

Reducing Read Cost

- Quorum consensus can be used to reduce read cost
  - But at increased risk of blocking of writes due to failures
- Use primary copy scheme:
  - perform all updates at primary copy
  - reads only need to be done at primary copy
  - But what if primary copy fails
    - Need to ensure new primary copy is chosen
      - Leases can ensure there is only 1 primary copy at a time
    - New primary copy needs to have latest committed version of data item
      - Can use consensus protocol to avoid blocking

PURDUE UNIVERSITY | Department of Computer Science

Distributed Deadlock Handling

PURDUE UNIVERSITY
Department of Computer Science

Timestamp Ordering

- The TM assigns each txn, T_i, a unique timestamp, $ts(Ti)$.
- No two txns share a timestamp.
- A TO scheduler enforces:
- TO Rule: if $p_i[x]$ and $q_j[x]$ are conflicting operations, then the DM processes $p_i[x]$ before $q_j[x]$ iff $ts(T_i) < ts(T_j)$.

PURDUE UNIVERSITY
Department of Computer Science

Serializability

- Theorem: If *H* is a history representing an execution produced by a TO scheduler, then *H* is serializable.
- Proof: Consider SG(*H*).
- If $T_i \rightarrow T_j$ is an edge in SG(*H*), then there must exist conflicting operations $p_i[x]$ and $q_j[x]$ in *H* such that $p_i[x] < q_j[x]$.
- Hence by the TO rule, $ts(T_i) < ts(T_j)$.
- If there is a cycle $T1 \rightarrow T2 \rightarrow \ldots \rightarrow Tn \rightarrow T1$ in SG(*H*), then by induction, $ts(T_1) < ts(T_1)$!!!

PURDUE UNIVERSITY
Department of Computer Science

Basic TO

- For each operation, we pass it to the DM as long as it is not too late!
- An operation is too late if a conflicting operation with a larger timestamp has already been sent to the DM.
- If an operation is too late, the earlier operation cannot be undone, then the txn is aborted.
- The aborted txn is restarted with a new timestamp – why?
- This avoids cyclic restart.

PURDUE UNIVERSITY | Department of Computer Science

Distributed Failure and Recovery

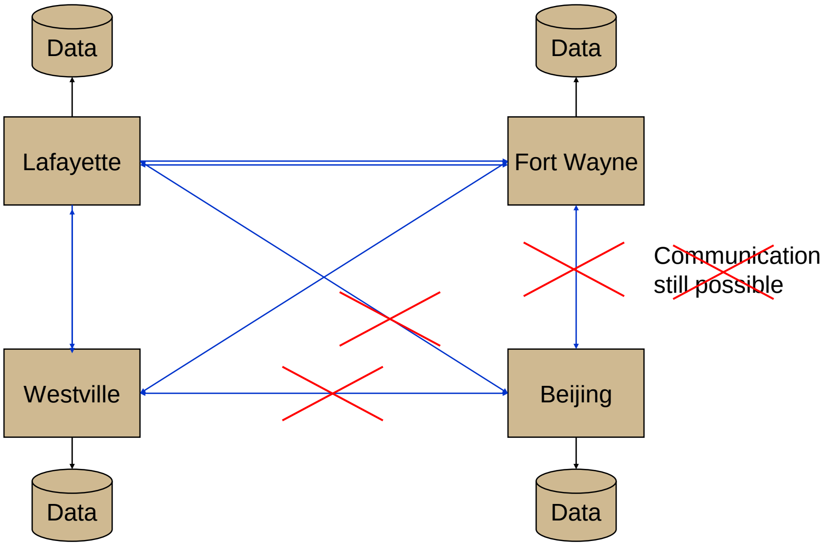
PURDUE UNIVERSITY
Department of Computer Science

Failure Model

- Fail-stop: Entire system stops when anything fails
  - – Defeats the purpose
- *Individual* sites fail-stop
  - – Challenge: Multi-site transactions
- New problem: Link Failure
  - – Both machines still running
  - – But can't communicate

PURDUE UNIVERSITY
Department of Computer Science

Link Failure Model:
Partition



~~Communication still possible~~

Solution: Fail-Stop Model

- One partition continues
  - The other stops
- Which one?
  - Partition that has majority
    - Can be slow to determine majority
  - “Leader”
    - If leader not in partition, elect a new leader
    - Requires majority vote
    - Leader must ensure its partition has a majority before other partition could elect a new leader

Limitation: *Transient* Partition

- One side checks, can’t communicate
  - The other side is able to when it checks
- No “perfect” solution
  - See “Byzantine Generals” problem
- Requires *some* single point of failure
  - *Make that single point extremely reliable*

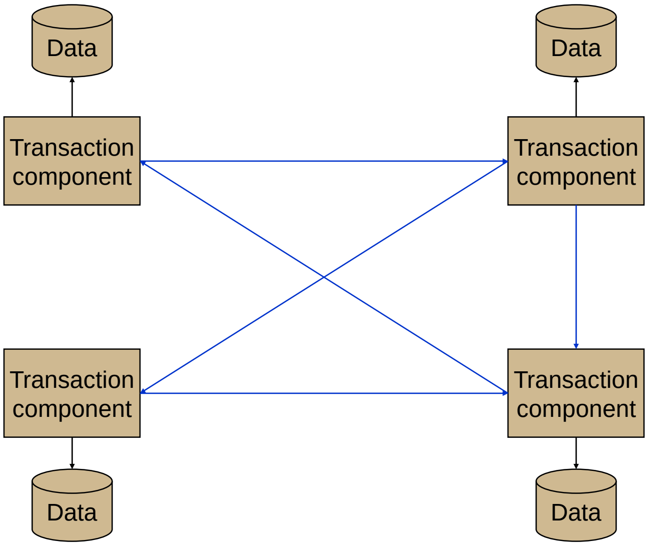
PURDUE UNIVERSITY
Department of Computer Science

Distributed Failure/Recovery

- We’re back to fail-stop:
  - – Does everything work as before?
- Problem: Distributed Transactions

PURDUE UNIVERSITY
Department of Computer Science

What is a Distributed Transaction?



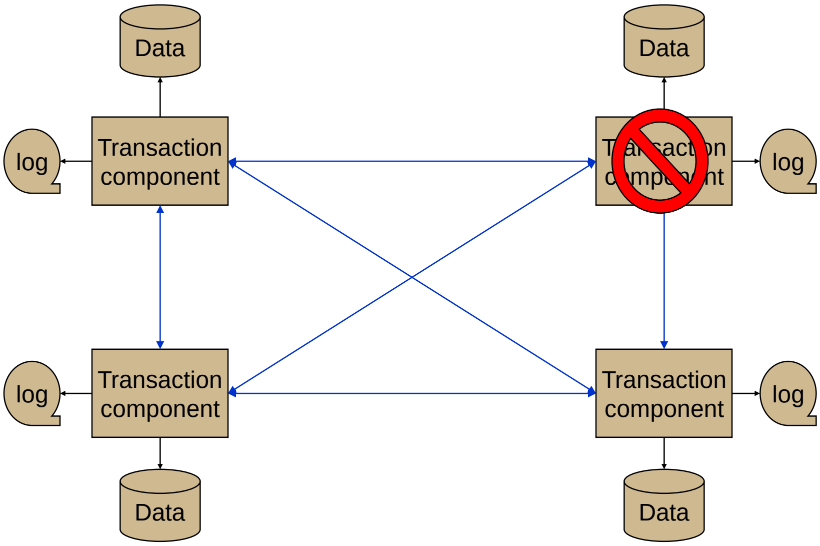
Why are Distributed Transactions Hard?

- **A**tomic
  - Different parts of a transaction may be at different sites
  - How do we ensure all or none committed?
- **C**onsistent
  - Failure may affect only part of transaction
- **I**solated
  - Commitment must occur “simultaneously” at all sites
- **D**urable
  - Not much different when other problems solved
  - Makes “delayed commit” difficult

Distributed Failure/Recovery

- We’re back to fail-stop:
  - Does everything work as before?
- Problem: Distributed Transactions
- Simplifying assumption: No data replication
  - Locking handled at local site
  - Transaction ensures 2-phase locking
- *Concurrency control still works*
  - Ignore the difficulty of deadlock detection/prevention

Committing a Distributed Transaction



Atomic Commit Protocols

- The steps in an Atomic Commit Protocol (ACP) are as follows:
  - TM gets a commit operation from the txn.
  - ACP needs to arrive at a single, consistent decision to commit or abort based upon the state of the txn at each site i.e.
    - Scheduler
    - DM (ensure that redo rule is satisfied) if there were only read operations at a site, ACP doesn't need to consult DM
  - Can do this by polling all sites.
  - Send the decision to each site.

ACP Requirements

- AC1: All processes that reach a decision reach the same one.
- AC2: A process cannot reverse its decision after it has reached one.
- AC3: The *Commit* decision can only be reached if *all* processes voted *Yes*.
- AC4: If there are no failures and all processes voted *yes*, then the decision will be to commit.
- AC5: Consider any execution containing only failures that the ACP is designed to tolerate. At any point in this execution, if all existing failures are repaired and no new failures occur for sufficiently long, then all processes will eventually reach a decision.

Key Issues

- Commitment
  - Standard techniques preserve properties when commit occurs
  - Distributed systems need commit *protocols* so we know when commit has occurred
- Failures
  - Standard techniques support durability for commit/abort
  - What happens if a site fails during commitment?

PURDUE
UNIVERSITY
Department of Computer Science

Two-Phase Commit (Lamport ’76, Gray ’79)

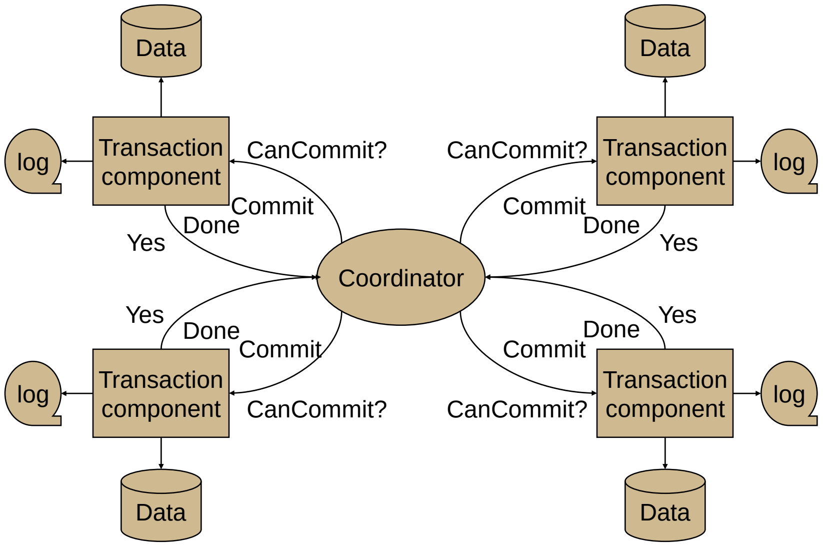
- Assumes central coordinator
  - Coordinator initiates protocol
  - Participants: entities with actions to be committed/aborted
- Phase 1:
  - Coordinator asks if participants can commit
  - Participants respond yes/no
- Phase 2:
  - If all votes yes, coordinator sends Commit
    - Otherwise send Abort
  - Participants send Have Committed / Have Aborted

PURDUE
UNIVERSITY
Department of Computer Science

2 Phase Commit Protocol (Lamport ’76, Gray ’79)

1. *Coord* sends *VOTE_REQ* to all *participants*.
2. Each *P* sends a msg back with its vote: *YES* or *NO*. If it votes *NO*, it decides *ABORT* and stops.
3. The Coord collects all votes.
   - If all are *YES* and its own vote is *YES*, it decides *COMMIT* and sends *COMMIT* msgs to each participant. Stop
   - Otherwise, it decides *ABORT* and send *ABORT* msgs to all participants that voted *YES.* Stop*.*
4. Each participant that voted *YES* waits for the coord’s decision, decides accordingly and stops.

Two-Phase Commit



Complications

- If no failures take place this ACP works fine.
- However, if there are failures, we need to specify what happens when:
  - There is a timeout while waiting for a message; or
  - A site crashes and then recovers during the ACP?
- Timeout actions:
  - Participant waiting for a *VOTE_REQ*: unilaterally abort.
  - Coord waiting for a vote: decide *ABORT* and send msg to all sites that voted *yes*.

Cooperative Termination Protocol

- Process *P* sends a *decision_REQ* message to every participant, *Q. P* learns of the other participants from the *VOTE_REQ* message sent by the Coord.
- *Q* does the following:
  - If *Q* has already decided, then it send its decision to *P*
  - If *Q* has not yet voted, then it can unilaterally abort and send *ABORT* to P.
  - If *Q* is also uncertain then it cannot help *P* – both are blocked.

Handling site failure in 2PC

- We use a distributed transaction log to record necessary information about termination protocols, in order to recover correctly.
- The DT log can be a part of the regular log too.
- It works as follows:
  - When *Coord* sends a *VOTE_REQ*, it writes a *start-2PC* record (before or after sending message).
  - If a participant votes *yes*, it writes a *yes* record before sending the vote. This record contains the identities of the coordinator and other participants (as given by the initial message of the coord).

PURDUE
UNIVERSITY
Department of Computer Science

DT Log

- If the participant votes *no,* it writes an *abort* record, either before or after sending the vote.
- Before the *Coord* sends a commit decision, it writes a *commit* record.
- When the *Coord* sends abort, it writes the *abort* record to the log
- After receiving commit(abort), a participant writes a *commit(abort)* record to its log.

PURDUE
UNIVERSITY
Department of Computer Science

Recovery

- When a site recovers, the fate of a distributed txn is determined as follows.
- If the DT log contains a *start-2PC* record, then the recovering site, *s*, was the coordinator
  - if it also contains a *commit*_ or *abort*_ record, then the coord had reached a decision before failure.
  - if neither is found, the coord can now unilaterally decide_ ABORT.
- If the DT log doesn't contain the *start-2PC* record, then the site was a participant. There are three cases:

PURDUE UNIVERSITY
Department of Computer Science

Recovery (contd.)

- The DT log contains a *commit* or *abort* record – I.e. participant had reached a decision.
- The DT log does not contain a *yes* record: either the participant failed before voting, or voted *NO*. It can therefore unilaterally decide to *ABORT*.
- The DT log contains a *yes* record, but no *commit* or *abort* record: participant failed during the uncertainty period – use the termination protocol to determine fate.

PURDUE UNIVERSITY
Department of Computer Science

3PC

- The problem with 2PC is that the coordinator sends *Commit* messages while the participants are uncertain.
- Thus participants can *decide* commit while some other participants are uncertain.
- 3PC avoids this by sending *pre-Commit* messages instead of *Commit* messages, thereby moving every participant out of the uncertainty period before any participant commits.
- After coord receives ack for *pre-Commits*, it sends commit, allowing participants to commit.