Lecture 13 – Distributed Transactions and Replication
References


• Chapters 8 and 9 in Principles of Transaction Processing. Second Ed. Phil Bernstein and Eric Newcomer.

• Chapter 22 in Ramakrishnan and Gehrke
Distributed Transactions
Distributed Transactions

• Concurrency control

• Failure recovery
  – Transaction must be committed at all sites or at none of the sites!
    • No matter what failures occur and when they occur
  – Two-phase commit protocol (2PC)
Distributed Concurrency Control

- Different techniques are possible
  - Pessimistic, optimistic, locking, timestamps
- Common implementation: distributed two-phase locking
  - Simultaneously hold locks at all sites involved
- Deadlock detection techniques
  - Global wait-for graph (not very practical)
  - Timeouts
- If deadlock: abort least costly local transaction
Two-Phase Commit: Motivation

1) User decides to commit
2) COMMIT
3) COMMIT
4) Coordinator crashes

What do we do now?

Subordinate 1
Subordinate 2
Subordinate 3

But I already aborted!
Two-Phase Commit Protocol

• One coordinator and many subordinates
  – Phase 1: prepare
    • All subordinates must flush tail of write-ahead log to disk before ack
    • Must ensure that if coordinator decides to commit, they can commit!
  – Phase 2: commit or abort
  – Log records for 2PC include transaction and coordinator ids
  – Coordinator also logs ids of all subordinates

• Principle
  – When a process makes a decision: vote yes/no or commit/abort
  – Or when a subordinate wants to respond to a message: ack
  – First force-write a log record (to make sure it survives a failure)
  – Only then send message about decision
2PC: Phase 1, Prepare

1) User decides to commit

2) PREPARE

3) Force-write: prepare

4) YES

Coordinator

Subordinate 1

Subordinate 2

Subordinate 3
2PC: Phase 2, Commit

1) Force-write: commit

2) COMMIT

3) Force-write: commit
4) ACK
5) Commit transaction and “forget” it

4) ACK

2) COMMIT

5) Write: end, then forget transaction

Transaction is now committed!

Subordinate 1

Subordinate 2

Subordinate 3
2PC with Abort

1) User decides to commit

2) PREPARE

3) Force-write: prepare

4) YES

5) Abort transaction and “forget” it

2) PREPARE

3) Force-write: abort

4) NO

5) Abort transaction and “forget” it

CSE 544 - Magda Balazinska, Winter 2015
2PC with Abort

1) Force-write: abort

2) ABORT

3) Force-write: abort
5) Abort transaction and “forget” it

4) ACK

5) Write: end, then forget transaction
Coordinator State Machine

- All states involve waiting for messages

Subordinate State Machine

- INIT and PREPARED involve waiting

- INIT
  - R: Prepare
  - FW: Abort
  - S: No vote

- PREPARED
  - R: Abort
  - FW: Abort
  - S: Ack

- ABORTING
  - Abort and forget

- COMMITTING
  - Commit and forget

R: Prepare
FW: Prepare
S: Yes vote

R: Commit
FW: Commit
S: Ack
Handling Site Failures

• Approach 1: no site failure detection
  – Can only do retrying & blocking

• Approach 2: timeouts
  – Since unilateral abort is ok,
  – Subordinate can timeout in init state
  – Coordinator can timeout in collecting state
  – Prepared state is still blocking

• 2PC is a blocking protocol
Site Failure Handling Principles

• Retry mechanism
  – In prepared state, periodically query coordinator
  – In committing/aborting state, periodically resend messages to subordinates
• If doesn't know anything about transaction respond “abort” to inquiry messages about fate of transaction
• If there are no log records for a transaction after a crash then abort transaction and “forget” it
Site Failure Scenarios

INIT
- Receive: Commit
- Send: Prepare

COLLECTING
- R: No votes
- FW: Abort
- S: Abort
- R: Yes votes
- FW: Commit
- S: Commit

ABORTING
- R: ACKS
- W: End

COMMITTING
- R: ACKS
- W: End

PREPARED
- R: Prepare
- FW: Prepare
- S: No vote
- R: Abort
- FW: Abort
- S: Ack
- R: Commit
- FW: Commit
- S: Ack

END
- Abort and forget
- Commit and forget
Observations

• Coordinator keeps transaction in transactions table until it receives all acks
  – To ensure subordinates know to commit or abort
  – So acks enable coordinator to “forget” about transaction

• After crash, if recovery process finds no log records for a transaction, the transaction is presumed to have aborted

• Read-only subtransactions: no changes ever need to be undone nor redone
Presumed Abort Protocol

• Optimization goals
  – Fewer messages and fewer force-writes
• Principle
  – If nothing known about a transaction, assume ABORT
• Aborting transactions need no force-writing
• Avoid log records for read-only transactions
  – Reply with a READ vote instead of YES vote
• Optimizes read-only transactions
2PC State Machines (repeat)

**INIT**
- Receive: Commit
- Send: Prepare

**COLLECTING**
- R: No votes
- FW: Abort
- S: Abort
- R: Yes votes
- FW: Commit
- S: Commit

**ABORTING**
- R: ACKS
- W: End

**COMMITTING**
- R: ACKS
- W: End

**END**

**PREPARED**
- R: Prepare
- FW: Abort
- S: No vote
- R: Prepare
- FW: Commit
- S: Yes vote

**ABORTING**
- R: Abort
- FW: Abort
- S: Ack

**COMMITTING**
- R: Commit
- FW: Commit
- S: Ack

Abort and forget
Commit and forget
Presumed Abort State Machines

**COLLECTING**
- Receive: Commit
- Send: Prepare
- R: Yes votes
- RW: Commit
- FW: Commit
- S: Commit
- R: No votes
- W: Abort
- S: Abort

**COMMITTING**
- R: AACKS
- W: End

**COMMITTING**
- R: Prepare
- FW: Commit
- S: Ack

**PREPARED**
- R: Prepare
- FW: Prepare
- S: Yes vote

**ABORTING**
- R: Commit
- FW: Commit
- S: Abort

**END**
- W: Abort
- S: Abort

**INIT**
- Receive: Commit
- Send: Prepare
- R: Yes votes
- RW: Commit
- FW: Commit
- S: Commit
- R: No votes
- W: Abort
- S: Abort

**INIT**
- R: Prepare
- FW: Prepare
- S: No vote

**INIT**
- R: Prepare
- FW: Prepare
- S: Yes vote

**INIT**
- R: Commit
- FW: Commit
- S: Ack

20
Presumed Abort for Read-Only

INIT

Receive: Commit
Send: Prepare

COLLECTING

R: Read
Forget

END

DONE

INIT

R: Prepare
S: Read vote

Forget
Replication
Outline

• Goals of replication

• Three types of replication
  – Eager replication
  – Lazy replication
  – Two-tier replication
Goals of Replication

• Goal 1: availability
• Goal 2: performance

But, it’s easy to build a replicated system that reduces performance and availability
Eager Replication

• Also called synchronous replication
• All updates are applied to all replicas (or to a majority) as part of a single transaction (need two phase commit)
  – E.g., triggers on tables apply updates to replicas within transaction
• Main goal: as if there was only one copy
  – Maintain consistency
  – Maintain one-copy serializability
  – I.e., execution of transactions has same effect as an execution on a non-replicated db
• Transactions must acquire global locks
Eager Master

• One master for each object holds primary copy
  – The “Master” is also called “Primary”
  – To update object, transaction must acquire a lock at the master
  – Lock at the master is global lock

• Master propagates updates to replicas synchronously
  – Updates propagate as part of the same distributed transaction
  – For example, using triggers
Crash Failures

- What happens when a secondary crashes?
  - Nothing happens
  - When secondary recovers, it catches up

- What happens when the master/primary fails?
  - Blocking would hurt availability
  - Must chose a new primary: run election
Network Failures

- Network failures can cause trouble...
  - Secondaries think that primary failed
  - Secondaries elect a new primary
  - But primary can still be running
  - Now have two primaries!
Majority Consensus

• To avoid problem, only majority partition can continue processing at any time

• In general,
  – Whenever a replica fails or recovers...
  – a set of communicating replicas must determine...
  – whether they have a majority before they can continue
Eager Group

- **With n copies**
  - Exclusive lock on x copies is global exclusive lock
  - Shared lock on s copies is global shared lock
  - Must have: $2x > n$ and $s + x > n$

- **Majority locking**
  - $s = x = \lceil \frac{(n+1)}{2} \rceil$
  - No need to run any reconfiguration algorithms

- **Read-locks-one, write-locks-all**
  - $s=1$ and $x = n$, high read performance
  - Need to make sure algo runs on quorum of computers
Eager Replication Properties

• Favours **consistency** over availability
  – Only majority partition can process requests
  – There appears to be a single copy of the db

• **High runtime overhead**
  – Must lock and update at least majority of replicas
  – Two-phase commit
  – Runs at pace of slowest replica in quorum
  – So overall system is now slower
  – Higher deadlock rate (transactions take longer)
Lazy Replication

- Also called asynchronous replication
- Also called optimistic replication

- Main goals: availability and performance

- Approach
  - One replica updated by original transaction
  - Updates propagate asynchronously to other replicas
Lazy Master

• One master holds primary copy
  – Transactions update primary copy
  – Master asynchronously propagates updates to replicas, which process them in same order (e.g. through log shipping)
  – Ensures single-copy serializability

• What happens when master/primary fails?
  – Can lose most recent transactions when primary fails!
  – After electing a new primary, secondaries must agree who is most up-to-date
Lazy Group

- Also called multi-master
- Best scheme for availability
- Cannot guarantee one-copy serializability!

```
R1
Init: x=1
Update x=2
```

```
R2
Init: x=1
Update x=3
```
Lazy Group

• Cannot guarantee one-copy serializability!
• Instead guarantee convergence
  – Db state does not reflect any serial execution
  – But all replicas have the same state
• Detect conflicts and reconcile replica states
• Different reconciliation techniques are possible
  – Manual
  – Most recent timestamp wins
  – Site A wins over site B
  – User-defined rules, etc.
Detecting Conflicts Using Timestamps

R1
Init: $x=1$ at $T_0$
Update at $T_1$: $x=2$
$x=2$ at $T_1$

R2
Init: $x=1$ at $T_0$
$x=2$ at $T_1$

$x=2$, Old: $T_0$ New: $T_1$
Detecting Conflicts Using Timestamps

Init: $x=1$ at $T_0$
Update at $T_1$: $x=2$

Conflict!
Reconciliation rule $T_2 > T_1$, so $x=3$

Init: $x=1$ at $T_0$
Update at $T_2$: $x=3$

Conflict!
Reconciliation rule $T_2 > T_1$, so $x=3$
Lazy Group Replication Properties

• Favours availability over consistency
  – Can read and update any replica
  – High runtime performance

• Weak consistency
  – Conflicts and reconciliation
Two-Tier Replication

- Benefits of lazy master and lazy group
- Each object has a master with primary copy
- When disconnected from master
  - Secondary can only run tentative transactions
- When reconnects to master
  - Master reprocesses all tentative transactions
  - Checks an acceptance criterion
  - If passes, we now have final commit order
  - Secondary undoes tentative and redoes committed
Conclusion

• Distributed transactions are very important
  – Necessary for scalability (throughput and global services)
  – But ACID properties require expensive 2PC protocol

• Replication is a very important problem
  – Fault-tolerance (various forms of replication)
  – Caching (lazy master)
  – Warehousing (lazy master)
  – Mobility (two-tier techniques)

• Replication is complex, but basic techniques and trade-offs are very well known
  – Eager or lazy replication
  – Master or no master
  – For eager replication: use quorum