CSE 124
Mar 1, 2017

Winter 2017, UCSD
Prof. George Porter
Each replica must handle write load of entire system?
Gossip Architecture

Client → FE → Replica

Client → FE → Replica

Client → FE → Replica

Service
Update Ordering Requirements

• Total Order
  – Bulletin board: all messages assigned globally unique message identifier
  – For messages $r1$, $r2$, either $r1$ appears before $r2$ at all replicas or $r1$ appears after $r2$ at all replicas

• Causal Order
  – Bulletin board: message replies appear after original posting
  – For messages $r1$, $r2$, $r1$ appears before $r2$ if $r1$ happens before $r2$
Happens Before

- Captures potential causal ordering (information flow)
- \( a \rightarrow b \) if \( a \) takes place before \( b \) in same process
- \( \text{Send}(m) \rightarrow \text{recv}(m) \)
- Transitivity holds
- Need tie-breaker such as IP address or MAC address
Implementing Total Ordering

• Use *sequencer*
  – Send updates to centralized site, assign monotonically increasing identifier, distribute to all replicas
  – Single point of failure, contention

• Distributed total ordering
  – Front end sends update to all replicas
  – Each replica proposes unique id
  – Front end picks highest value
  – Transmits final value back to replicas
  – 3 messages/replica overhead
Network Partitions

- Some failure (either network or host) keeps replicas from communicating with one another
- How to proceed with read/write transactions in case where not all replicas can be contacted?
Network Partitions

• Some failure (either network or host) keeps replicas from communicating with one another
• How to proceed with read/write transactions in case where not all replicas can be contacted?
  – Optimistic versus pessimistic techniques
• Optimistic: proceed as normal, resolving conflicts later
• Pessimistic: ensure that replicated database can eventually be restored to consistent state without user intervention
  – Assumes that partition will eventually be repaired
  – What if the network is never fully connected?
Quorum (Voting) Consensus Techniques

• Allow updates to be completed even when only a subset of replicas are available
  – Or have the latest version of a data item
  – May need to bring some set of replicas up to date to proceed

• One technique is the weighted voting scheme
  – Assign a number of votes to each replica
  – Determine a write quorum $W$ and a read quorum $R$
    • $W > \text{half the total votes}, R+W > \text{total number of votes}$
  – Ensures that there is some overlap between read and write quorums
    • Read operation guaranteed access to one site with latest version
Quorum Example

Write quorum

Read quorum
Quorum Consensus

- Write operations can be propagated in background to replicas not in quorum
  - Assumes eventual repair of any network partition
- Operations are slowed by the necessity of first gathering a quorum
  - Though previously, all writes had to go to all replicas
    - With quorum system, must only contact subset of replicas
Quorum Example

- 5 replicas, read quorum: 3, write quorum: 3
  - \( R+W > 5 \) votes ensures overlap between any read/write quorum

- How does this perform for reads?
- How does this perform for writes?
Managing concurrency

• What happens if W=1, R=5, and two clients concurrently update a variable?
  – Might choose different and orthogonal write quorums

• Solution 1:
  – Synchronization outside of the system

• Solution 2:
  – Choose \( W > (N/2) \) where \( N = \# \) nodes
Byzantine Failures

• Normally consider failures to be *fail-stop*
  – Meaning that a process/machine goes down without doing anything incorrect
  – In real world, corrupted memory means that failures are not always fail stop

• Consider worst-case scenario where machines/processes fail and become malicious
  – Can send arbitrary messages
  – Can lie about their identity
Consensus and Byzantine Failure

• Byzantine Generals problem:
  – $N$ Generals encamped outside of enemy city
  – Would like to come to consensus as to whether they should attack at dawn
  – Use messengers to communicate
    • Assume messages are not lost
    • Can identify sender of each message (but no signatures)
  – What happens when one or more of the generals fails (becomes malicious)?
Byzantine Consensus Goals

• Termination
  – Eventually, each correct process sets its decision variable

• Agreement
  – The decision value of all correct processes is the same

• Integrity
  – If the commander is correct, then all correct processes decide on the value the commander proposed
Byzantine Consensus

Neither G2 nor G3 can determine that G1 is bad; something is wrong but attack or retreat?
Byzantine Consensus

G2 realizes that someone is faulty, but who?

→ G3 in this case
Byzantine Consensus

- In the absence of authentication:
  - $3m+1$ servers required to detect the presence of $m$ traitors
- Consider mission critical system trying to reach consensus
  - $N$ replicated servers performing the same calculation
  - Each server votes for the decision
  - Majority wins
  - Assume “good” (non-faulty) servers all come to the same conclusion
Byzantine Generals

- 4 processes,
  1 faulty
Problem Scenario

• Replicated Service
  – Perhaps across the Internet, perhaps in mobile computing environment

• Clients can apply updates to any replica (read any replica)

• Replicas may never be fully connected
  – How to achieve eventual consistency?
Logical Clocks

- Assign a monotonically increasing logical “timestamp” to all updates
- On message exchange, set local logical clock to be “one more” than remote logical clock value
Consistency through Pair-wise Message Exchange

- Each replica maintains *logical time vector* of length \( n \)
  - Where \( n \) is the number of replicas
  - Entry \( ltv_j[i] \) is the last known logical time for server \( i \) known by server \( j \)

- Periodically, a pair of replicas come into contact with one another ➔ exchange messages
  - Analogous to gossip exchange
  - Logical time vector states which updates the replicas are missing relative to one another
    - Exchange necessary updates
    - Update logical time vectors
Anti-Entropy: Simple Example

- Each of R1, R2, and R3 have seen 5 different updates (for total of 15 independent updates)
Anti-Entropy: Simple Example

- R1 and R2 perform anti-entropy
  - R1, R2 have 10 updates, R3 has 5 updates
Anti-Entropy: Simple Example

- R2 and R3 perform anti-entropy, has 15 updates
  - Its own updates at times 1-5, 10 remote updates at time 6
Anti-Entropy: Simple Example

- What happens when R1 and R3 perform anti-entropy?
  - What if R1 accepts two new updates in the interim?
Anti-Entropy: Simple Example

• On Anti-Entropy between R1 and R3, R1 only propagates writes between time 6 and 8, R3 propagates all writes (after 0)

Two new updates
Anti-Entropy: Simple Example

- Anti-Entropy maintains causal order (happens-before) but (as described) does not impose a total order.
Transactions
ACID Property

• Atomicity
  – All or nothing (even in the face of failure)

• Consistency
  – Take system from one consistent state to another

• Isolation
  – No interference from concurrent transactions: do not see the intermediate (uncommitted) results of any transaction

• Durability
  – Once committed all effects are saved in permanent storage
Serial Equivalence

- One approach to avoiding concurrency problems is to execute transactions in *serial* order
  - Do not begin transaction $n+1$ until transaction $n$ either commits or aborts
  - Results in unacceptable performance
  - What if transactions are not accessing the same data items?
- Another approach is to ensure that transactions execute in *serial equivalent* order
  - Interleave operations only when it is safe to do so
  - Final result same as if transactions performed in serial order
- How to achieve serial equivalence?
Achieving Serial Equivalence

• Approaches to serial equivalence
  – Locking
  – Optimistic concurrency control (check on commit)
  – Timestamps

• Achieve serial equivalence by locking necessary data items
  – Avoiding deadlock
  – Breaking deadlock
  – Granularity of locks
  – Read/write (shared/exclusive) locks
  – Two phase commit for distributed lock management
Two-Phase Locking

• Serial equivalence requires that all data accesses be serialized w/respect to other transactions
  – If two transactions conflict, then *all* operations within transactions must be (logically) done either before or after other transaction

• Two-phase locking: do not acquire any new locks after releasing any lock
  – Growing phase, followed by shrinking phase

• Strict two-phase locking
  – Must worry about aborted transactions (dirty writes)
  – Do not release locks until transaction aborts/commits
Lock Manager

• Want to automate the process of acquiring/releasing locks
  – Avoid forcing programmers to get it right every time
  – Use API to force all data accesses go through lock manager

• Inside transaction, on data access:
  – If object not locked, server locks it
  – If conflicting lock, block transaction (Wait on condition var)
  – If non-conflicting lock, acquire shared lock
  – For read/write access consider lock promotion

• On transaction commit/abort:
  – Release all locks, signal appropriate waiting transactions
  – Starvation?
What happens when a transaction gets stuck?
Deadlocks

• Deadlock occurs when a loop is created in a logical \textit{waits for} graph
  – Transaction T acquires A, Transaction U acquires B
  – Transaction T waits for B, Transaction U waits for A
  – Neither transaction able to make forward progress

• Loop can be arbitrarily long
  – T\rightarrow U\rightarrow V\rightarrow W\rightarrow \ldots \rightarrow Z\rightarrow T
  – Typically, loops are short (one hop)
## Deadlock Example

<table>
<thead>
<tr>
<th>Transaction T</th>
<th>Transaction U</th>
</tr>
</thead>
<tbody>
<tr>
<td>Withdraw(A, 4); Deposit(B, 4)</td>
<td>Withdraw(C, 3); Deposit(B, 3)</td>
</tr>
<tr>
<td>Balance=A.Read()</td>
<td>Balance = C.Read()</td>
</tr>
<tr>
<td>A.Write(balance-4)</td>
<td>C.Write(balance-3)</td>
</tr>
<tr>
<td>Balance=B.read()</td>
<td>Balance=B.Read()</td>
</tr>
</tbody>
</table>

- **Balance**
  - A.Read()
  - B.read()

- **Operations**
  - T: Withdraw(A, 4); Deposit(B, 4)
  - U: Withdraw(C, 3); Deposit(B, 3)
## Deadlock Example

<table>
<thead>
<tr>
<th>Transaction T</th>
<th>Transaction U</th>
</tr>
</thead>
<tbody>
<tr>
<td>Withdraw(A, 4); Deposit(B, 4)</td>
<td>Withdraw(C, 3); Deposit(B, 3)</td>
</tr>
<tr>
<td><strong>Balance</strong>=A.Read()</td>
<td><strong>Balance</strong>=C.Read()</td>
</tr>
<tr>
<td>A.Write(balance-4)</td>
<td>C.Write(balance-3)</td>
</tr>
<tr>
<td><strong>Balance</strong>=B.read()</td>
<td><strong>Balance</strong>=B.Read()</td>
</tr>
<tr>
<td>B.Write(balance+4)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Wait U</td>
</tr>
</tbody>
</table>
# Deadlock Example

<table>
<thead>
<tr>
<th>Transaction T</th>
<th>Transaction U</th>
</tr>
</thead>
<tbody>
<tr>
<td>Withdraw(A, 4); Deposit(B, 4)</td>
<td>Withdraw(C, 3); Deposit(B, 3)</td>
</tr>
<tr>
<td>Balance=A.Read()</td>
<td>Balance = C.Read()</td>
</tr>
<tr>
<td>A.Write(balance-4)</td>
<td>C.Write(balance-3)</td>
</tr>
<tr>
<td>Balance=B.read()</td>
<td>Balance=B.Read()</td>
</tr>
<tr>
<td>B.Write(balance+4)</td>
<td>B.Write(balance+3)</td>
</tr>
</tbody>
</table>

- **Wait U**
- **Wait T**

---

**Deadlock**
- Cycle: \( T \rightarrow B \rightarrow U \rightarrow B \rightarrow T \)
Deadlock Prevention

• Gather *all* locks at beginning of transaction
  – Can we still run into problems?
Deadlock Prevention

• Gather *all* locks at beginning of transaction
  – Can we still run into problems? Yes, if locks are acquired in different order

<table>
<thead>
<tr>
<th>Transaction T</th>
<th>Transaction U</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lock A</td>
<td>Lock B</td>
</tr>
<tr>
<td>Lock B (Wait)</td>
<td>Lock A (Wait)</td>
</tr>
</tbody>
</table>
Deadlock Prevention

• Gather *all* locks at beginning of transaction
  – Can we still run into problems? Yes, if locks are acquired in different order
  – Solution: gather locks in canonical order
• Won’t work in many circumstances
  – Cannot predict locking requirements of interactive apps
  – Unnecessarily reduces concurrency
    • E.g., acquire read (write) lock for long-running transaction
    • Prevent *any* other transaction from acquiring write (read) lock
Deadlock Detection

• Concurrent thread keeps track of waits-for graph
  – Typically lock manager
  – For each successful lock operation, track resources held by each transaction
    • Not strictly necessary
  – For each conflicting lock operation (condition var wait), track transaction waits for relationship
  – On lock release, delete edges corresponding to signaled transactions
Deadlock Detection

- Search for loops in \textit{waits for} graph on adding edge
  - Abort one transaction in the loop
  - Release all locks associated with transaction
    - Releasing locks signals blocked transaction in loop, break deadlock
  - Aborted transaction must restart
    - Starvation
- Which transaction to abort?
Deadlock Detection

• Search for loops in *waits for* graph on adding edge
  – Abort one transaction in the loop
  – Release all locks associated with transaction
    • Releasing locks signals blocked transaction in loop, break deadlock
  – Aborted transaction must restart
    • Starvation

• Which transaction to abort?
  – Assign priorities to transactions?
  – Shortest running transaction?
Preventing Deadlocks Through Timeouts

• Assign a timeout with each lock
  – When timeout expires, lock becomes vulnerable
• If no other transaction waiting for lock, continue as normal
• If other transaction waiting, abort transaction holding vulnerable lock
• Pros/cons?
Preventing Deadlocks Through Timeouts

• Assign a timeout with each lock
  – When timeout expires, lock becomes vulnerable
• If no other transaction waiting for lock, continue as normal
• If other transaction waiting, abort transaction holding vulnerable lock

• Pros/cons
  – Pros: simpler to implement?
  – Cons: abort transaction when no deadlock exists, what value to set for timeout (lightly loaded vs. heavily loaded system)
Distributed Transactions
Distributed Transactions

• Client makes atomic request that accesses resources at multiple databases
  – Potentially spread across wide area

• Distributed transaction can be:
  – *Simple*: Client explicitly accesses resources at multiple sites
  – *Nested*: transaction spawns one-or more sub-transactions
Simple Distributed Transaction
Nested Distributed Transaction

Client

T

X

Y

Z
Distributed Transaction

• In distributed transactions, some server must be responsible for either committing or aborting transaction at all sites

• Typically, first server contacted becomes *coordinator*
  – Client tracks identity of coordinator
  – Tells each additional server (*workers*) of coordinator identity
  – Workers responsible for registering with coordinator

• Transaction abort/commit through coordinator
  – Client commit request relayed to workers
  – Worker abort request relayed to other workers
Distributed Transaction Commit Protocol

• One option: client communicates to coordinator desire to abort or commit
  – Coordinator communicates decision to all workers
  – One-phase protocol
  – Problems?
Distributed Transaction Commit Protocol

• One option client communicates to coordinator desire to abort or commit
  – Coordinator communicates decision to all workers
  – One-phase protocol
  – Does not allow individual servers to voice their opinion
    • Failure, concurrency control (deadlock)
Two-Phase Commit

• Phase one (voting)
  – Coordinator sends \textit{CanCommit} request to each worker
  – Each worker replies with vote (yes/no)
• Phase two (complete based on vote)
  – If no failures and everyone votes yes, coordinator sends \textit{DoCommit} to each worker
  – Otherwise, send \textit{AbortTransaction} to each worker
  – On success, each worker sends \textit{HaveCommitted} to coordinator
Two-Phase Commit

1. Coord: Prepared to Commit
2. Worker: Prepared to Commit (uncertain)
3. Coord: Committed
4. Worker: Committed (done)
Extra Slides
Transactions with Replicated Data

• For increased availability (performance?), consider transactional data replicated at multiple sites

• How do read/write operations work for distributed transactions?

• How to proceed with two-phase commit?
Transactions with Replicated Data

• For increased availability (performance?), consider transactional data replicated at multiple sites

• How do read/write operations work for distributed transactions?
  – Read-one/Write-all

• How to proceed with two-phase commit?
  – Nested two-phase commit where workers pass request to replica managers before replying back to the coordinator

• But what about network partitions?