Outline

• Announcements:
  – HW7+8 worth 10 points
  – Syllabus updated

• Today:
  – Replication consistency
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
    • Need to coordinate to capture the city
  – Use messengers to communicate
    • Can be caught
  – Generals might be traitors
Scenario: Coordinating activities

- General Zod will attack if General Tarkin agrees, otherwise will retreat.
- If timeout fires, then Zod retreats.
- If Tarkin responds, then Zod attacks.
What about unreliable networks?

• General Zod will attack if General Tarkin agrees, otherwise will retreat **AND VICE VERSA**

![Diagram](image)

• Tarkin only attacks when Zod agrees to attack
  – Thus after Tarkin gets Zod’s ACK
  – Otherwise, maybe Zod didn’t get the ‘Yes’ response?
Reaching agreement

• Consider any protocol $P$ that solves the Byzantine Generals problem
  – How many messages exchanged in $P$?
• Impossibility result!
• How to prove?
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
Goal

• Suppose process $P_i$ has a private value $V_i$
• We’d like an algorithm that allows each nonfaulty $P$ to construct:
  $X_i = (A_{i1}, A_{i2}, ..., A_{in})$ s.t.
  • If $P_j$ is nonfaulty, $A_{ij} = V_j$
  • If $P_i$ and $P_j$ are nonfaulty, $X_i = X_j$
Properties of solutions

• Correct algorithm requires:
  – $N \geq 3m + 1$ (for $m$ faulty nodes)
  – A worst-case delay proportional to $(m+1)$ message passing delays
  – # messages exchanged is large

• A general solution is quite complex, so let’s look at an example using 4 nodes
4-node example

- 4 processes, 1 faulty
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 being 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)

• 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 about 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 → U → V → W → ... → Z → 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>

- **A** deposits 4, then **B** withdraws 4.
- **C** deposits 3, then **A** reads, writes, and reads again.
- **C** reads, writes, and reads again.

_A timeline of events is shown below._
## 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>Wait U</td>
</tr>
</tbody>
</table>

`time`
### 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() Read A</td>
<td>Balance = C.Read() Read C</td>
</tr>
<tr>
<td>A.Write(balance-4)</td>
<td>C.Write(balance-3)</td>
</tr>
<tr>
<td>Balance=B.read() Read B</td>
<td>Balance=B.Read() Read B</td>
</tr>
<tr>
<td>B.Write(balance+4) Wait U</td>
<td>B.Write(balance+3) Wait T</td>
</tr>
</tbody>
</table>

... | ...

... | ...

... | ...

Deadlock
Waits-For Graph

- 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
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 *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 \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?
  - 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 *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 *DoCommit* to each worker
  – Otherwise, send *AbortTransaction* to each worker
  – On success, each worker sends *HaveCommitted* to coordinator
Two-Phase Commit

1. CanCommit?
2. Yes
3. DoCommit
4. HaveCommit

1. Coord: Prepared to Commit
2. Worker: Prepared to Commit (uncertain)
3. Coord: Committed
4. Worker: Committed (done)