Consistency and Replication

• Today:
  – Consistency models
    • Data-centric consistency models
    • Client-centric consistency models

Computing Parable

• The Archery Teacher

• Courtesy: S. Keshav, U. Waterloo
Two-phase Locking

• Widely used concurrency control technique
• Scheduler acquires all necessary locks in growing phase, releases locks in shrinking phase
  – Check if operation on data item \( x \) conflicts with existing locks
    • If so, delay transaction. If not, grant a lock on \( x \)
  – Never release a lock until data manager finishes operation on \( x \)
  – One a lock is released, no further locks can be granted
• Problem: deadlock possible
  – Example: acquiring two locks in different order
• Distributed 2PL versus centralized 2PL
Strict Two-Phase Locking

- Strict two-phase locking.

Timestamp-based Concurrency Control

- Each transaction Ti is given timestamp ts(Ti)
- If Ti wants to do an operation that conflicts with Tj
  - Abort Ti if ts(Ti) < ts(Tj)
- When a transaction aborts, it must restart with a new (larger) time stamp
- Two values for each data item x
  - Max-rts(x): max time stamp of a transaction that read x
  - Max-wts(x): max time stamp of a transaction that wrote x
Reads and Writes using Timestamps

- **Read\textsubscript{i}(x)**
  - If $ts(T_i) < max\text{-}wts(x)$ then Abort $T_i$
  - Else
    - Perform $R_i(x)$
    - $Max\text{-}rts(x) = max(max\text{-}rts(x), ts(T_i))$

- **Write\textsubscript{i}(x)**
  - If $ts(T_i) < max\text{-}rts(x)$ or $ts(T_i) < max\text{-}wts(x)$ then Abort $T_i$
  - Else
    - Perform $W_i(x)$
    - $Max\text{-}wts(x) = ts(T_i)$

Why replicate?

- Data replication versus compute replication
- Data replication: common technique in distributed systems
- Reliability
  - If one replica is unavailable or crashes, use another
  - Protect against corrupted data
- Performance
  - Scale with size of the distributed system (replicated web servers)
  - Scale in geographically distributed systems (web proxies)
Replication Issues

- When to replicate?
- How many replicas to create?
- Where should the replicas located?

- Will return to these issues later (WWW discussion)
- Today: how to maintain consistency?
- Key issue: need to maintain consistency of replicated data
  - If one copy is modified, others become inconsistent

Object Replication

- Approach 1: application is responsible for replication
  - Application needs to handle consistency issues
- Approach 2: system (middleware) handles replication
  - Consistency issues are handled by the middleware
  - Simplifies application development but makes object-specific solutions harder
Replication and Scaling

- Replication and caching used for system scalability
- Multiple copies:
  - Improves performance by reducing access latency
  - But higher network overheads of maintaining consistency
  - Example: object is replicated $N$ times
    - Read frequency $R$, write frequency $W$
    - If $R << W$, high consistency overhead and wasted messages
    - Consistency maintenance is itself an issue
      - What semantics to provide?
      - Tight consistency requires globally synchronized clocks!
- Solution: loosen consistency requirements
  - Variety of consistency semantics possible

Data-Centric Consistency Models

- Consistency model (aka *consistency semantics*)
  - Contract between processes and the data store
    - If processes obey certain rules, data store will work correctly
    - All models attempt to return the results of the last write for a read operation
    - Differ in how “last” write is determined/defined
Strict Consistency

• Any read always returns the result of the most recent write
  – Implicitly assumes the presence of a global clock
  – A write is immediately visible to all processes
    • Difficult to achieve in real systems (network delays can be variable)

Sequential Consistency

• Sequential consistency: weaker than strict consistency
  – Assumes all operations are executed in some sequential order and each process issues operations in program order
    • Any valid interleaving is allowed
    • All agree on the same interleaving
    • Each process preserves its program order
    • Nothing is said about “most recent write”
Linearizability

• Assumes sequential consistency and
  – If TS(x) < TS(y) then OP(x) should precede OP(y) in the sequence
  – Stronger than sequential consistency
  – Difference between linearizability and serializability?
    • Granularity: reads/writes versus transactions

• Example:

<table>
<thead>
<tr>
<th>Process P1</th>
<th>Process P2</th>
<th>Process P3</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 1;</td>
<td>y = 1;</td>
<td>z = 1;</td>
</tr>
<tr>
<td>print (y, z);</td>
<td>print (x, z);</td>
<td>print (x, y);</td>
</tr>
</tbody>
</table>

Linearizability Example

• Four valid execution sequences for the processes of the previous slide. The vertical axis is time.

1. x = 1;
   print ((y, z);
   y = 1;
   print (x, z);
   z = 1;
   print (x, y);
   Prints: 001011
   Signature: 001011 (a)

2. x = 1;
   y = 1;
   z = 1;
   print (y, z);
   print (x, z);
   x = 1;
   print (y, z);
   print (x, z);
   Print (y, z);
   Print (x, y);
   Prints: 101011
   Signature: 101011 (b)

3. y = 1;
   z = 1;
   print (x, y);
   print (x, z);
   x = 1;
   print (y, z);
   print (x, y);
   Print (y, z);
   Prints: 110111
   Signature: 110111 (c)

4. y = 1;
   x = 1;
   z = 1;
   print (x, z);
   print (y, z);
   print (x, y);
   Print (y, z);
   Print (x, y);
   Prints: 111111
   Signature: 111111 (d)
Causal consistency

- Causally related writes must be seen by all processes in the same order.
  - Concurrent writes may be seen in different orders on different machines

Other models

- FIFO consistency: writes from a process are seen by others in the same order. Writes from different processes may be seen in different order (even if causally related)
  - Relaxes causal consistency
  - Simple implementation: tag each write by (Proc ID, seq #)
- Even FIFO consistency may be too strong!
  - Requires all writes from a process be seen in order
- Assume use of critical sections for updates
  - Send final result of critical section everywhere
  - Do not worry about propagating intermediate results
    - Assume presence of synchronization primitives to define semantics
Other Models

Use granularity of critical sections, instead of individual read/write

- Weak consistency
  - Accesses to synchronization variables associated with a data store are sequentially consistent
  - No operation on a synchronization variable is allowed to be performed until all previous writes have been completed everywhere
  - No read or write operation on data items are allowed to be performed until all previous operations to synchronization variables have been performed.

- Entry and release consistency
  - Assume shared data are made consistent at entry or exit points of critical sections

<table>
<thead>
<tr>
<th>Consistency</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strict</td>
<td>Absolute time ordering of all shared accesses matters.</td>
</tr>
<tr>
<td>Linearizability</td>
<td>All processes must see all shared accesses in the same order. Accesses are furthermore ordered according to a (nonunique) global timestamp</td>
</tr>
<tr>
<td>Sequential</td>
<td>All processes see all shared accesses in the same order. Accesses are not ordered in time</td>
</tr>
<tr>
<td>Causal</td>
<td>All processes see causally-related shared accesses in the same order.</td>
</tr>
<tr>
<td>FIFO</td>
<td>All processes see writes from each other in the order they were used. Writes from different processes may not always be seen in that order</td>
</tr>
</tbody>
</table>

(b)
Client-centric Consistency Models

- Assume read operations by a single process $P$ at two different
  local copies of the same data store
  - Four different consistency semantics
- **Monotonic reads**
  - Once read, subsequent reads on that data items return same or more recent values
- **Monotonic writes**
  - A write must be propagated to all replicas before a successive write by the same process
  - Resembles FIFO consistency (writes from same process are processed in same order)
- **Read your writes**: read(x) always returns write(x) by that process
- **Writes follow reads**: write(x) following read(x) will take place on same or more recent version of x

Eventual Consistency

- Many systems: one or few processes perform updates
  - How frequently should these updates be made available to other read-only processes?
- Examples:
  - DNS: single naming authority per domain
  - Only naming authority allowed updates (no write-write conflicts)
  - How should read-write conflicts (consistency) be addressed?
  - NIS: user information database in Unix systems
    - Only sys-admins update database, users only read data
    - Only user updates are changes to password
Eventual Consistency

• Assume a replicated database with few updaters and many readers
• Eventual consistency: in absence of updates, all replicas converge towards identical copies
  – Only requirement: an update should eventually propagate to all replicas
  – Cheap to implement: no or infrequent write-write conflicts
  – Things work fine so long as user accesses same replica
  – What if they don’t:

Epidemic Protocols

• Used in Bayou system from Xerox PARC
• Bayou: weakly connected replicas
  – Useful in mobile computing (mobile laptops)
  – Useful in wide area distributed databases (weak connectivity)
• Based on theory of epidemics (spreading infectious diseases)
  – Upon an update, try to “infect” other replicas as quickly as possible
  – Pair-wise exchange of updates (like pair-wise spreading of a disease)
  – Terminology:
    • Infective store: store with an update it is willing to spread
    • Susceptible store: store that is not yet updated
• Many algorithms possible to spread updates
Spreading an Epidemic

- Anti-entropy
  - Server \( P \) picks a server \( Q \) at random and exchanges updates
  - Three possibilities: only push, only pull, both push and pull
  - Claim: A pure push-based approach does not help spread updates quickly (Why?)
    - Pull or initial push with pull work better
- Rumor mongering (aka gossiping)
  - Upon receiving an update, \( P \) tries to push to \( Q \)
  - If \( Q \) already received the update, stop spreading with prob \( 1/k \)
  - Analogous to “hot” gossip items => stop spreading if “cold”
  - Does not guarantee that all replicas receive updates
    - Chances of staying susceptible: \( s = e^{-(k+1)(1-s)} \)

Removing Data

- Deletion of data items is hard in epidemic protocols
- Example: server deletes data item \( x \)
  - No state information is preserved
    - Can’t distinguish between a deleted copy and no copy!
- Solution: death certificates
  - Treat deletes as updates and spread a death certificate
    - Mark copy as deleted but don’t delete
    - Need an eventual clean up
      - Clean up dormant death certificates