Distributed Systems

Catch-up Lecture:
Consistency Model Implementations

Slides redundant with Lec 11,12
Slide acks:  Jinyang Li, Robert Morris, Dave Andersen
Outline

• Last times: Consistency models
  • Strict consistency
  • Sequential consistency
  • Causal consistency
  • Eventual consistency
• How do you define them?
  • Today: Basic ideas for implementing them
    • Sequential consistency
    • Eventual consistency
  // Causal consistency (Promiscuous covered that)
Outline

• How do you define these consistency models:
  – Sequential consistency
    • All memory/storage accesses appear executed in a single order by all processes
  – Eventual consistency
    1. All replicas eventually become identical and no writes are lost
    2. All replicas eventually apply all updates in a single order

• Implementation summary:
  – Sequential consistency
    • Serialize all requests through a master, invalidate caches, wait for writes to complete → Ivy, a distributed shared mem system (DSM)
  – Eventual consistency
    • Perform reads/writes asynchronously, synch back with others later and solve conflicts at that time → file synchronizers
Distributed Shared Memory (DSM)

- Two models for communication in distributed systems:
  - message passing
  - shared memory

- Shared memory is often thought more intuitive to write parallel programs than message passing
  - Each machine can access a common address space

![Diagram of distributed shared memory with three machines (M1, M2, M3) and variables (var) connected to DSM]

```plaintext
read/write (load/store)
```

DSM
Example Application

M0:
\[ v0 = f0(); \]
\[ \text{done0} = 1; \]

M1:
\[ \text{while (done0 == 0) ;} \]
\[ v1 = f1(v0); \]
\[ \text{done1} = 1; \]

M2:
\[ \text{while (done1 == 0) ;} \]
\[ v2 = f2(v0, v1); \]

- What’s the intuitive intent?
  - M2 should execute \( f2() \) with results from M0 and M1
  - waiting for M1 implies waiting for M0
Naïve DSM System

• Each machine has a local copy of all of memory

• Operations:
  – **Read**: from local memory
  – **Write**: send update msg to each host (but *don't wait*)

• Fast: never waits for communication

**Question**: Does this DSM work well for our application?
Problem 1 with Naïve DSM

• M0's $v_0=\ldots$ and done0=… may be interchanged by network, leaving $v_0$ unset but done0=1

\[ v_0 = f_0() \]
\[ \text{done0}=1 \]

\[ v_0 \rightarrow \text{M0} \]
\[ \text{done0} \]
\[ v_1 = f_1(v_0) \]

whoops!
Problem 2 with Naïve DSM

- M2 sees M1's writes before M0's writes
  - I.e. M2 and M1 disagree on order of M0 and M1 writes

\[
\begin{align*}
v_0 &= f_0() \\
\text{done}_0 &= 1 \\
\text{done}_1 &= 1 \\
v_1 &= f_1(v_0) \\
v_2 &= f_2(v_0, v_1)
\end{align*}
\]
Naïve DSM Properties

- Naïve DSM is fast but has unexpected behavior

- Clearly, it's not sequentially consistent
  - So, how do we make it so, and what do we lose in performance because of that?
Sequential Consistency

- **Rules**: There exists a total ordering of ops s.t.
  - Rule 1: Each machine’s own ops appear in order
  - Rule 2: All machines see results according to total order (i.e. reads see most recent writes)

- We say that any runtime ordering of operations (also called a *history*) can be “explained” by a sequential ordering of operations that follows the above two rules
Does Seq. Consistency Avoid Problems?

• There is a total ordering of events s.t.:
  – Rule 1: Each machine’s own ops appear in order
  – Rule 2: All machines see results according to total order

• Problem 1: Can M1 ever see v0 unset but done0=1?
  – M0's execution order was v0=… done0=…
  – M1 saw done0=… v0=…
  – Each machine's operations must appear in execution order so cannot happen w/ sequential consistency

• Problem 2: Can M1 and M2 disagree on ops’ order?
  – M1 saw v0=… done0=… done1=…
  – M2 saw done1=… v0=…
  – This cannot occur given a single total ordering
Seq. Consistency Implementation Requirements

1. Each processor issues requests in the order specified by the program
   • Do not issue the next request unless the previous one has finished

2. Requests to an individual memory location are served from a single FIFO queue
   • Writes occur in a single order
   • Once a read observes the effect of a write, it’s ordered behind that write
Naive DSM violates R1,R2

- Read from local state
- Send writes to the other nodes, but do not wait

R1: a processor issues read before waiting for write to complete
R2: 2 processors issue writes concurrently, no single order
Case Study:
Ivy: Integrated shared virtual memory at Yale
Ivy distributed shared memory

• What does Ivy do?
  – Provide a shared memory system across a group of workstations

• Why shared memory?
  – Easier to write parallel programs with than using message passing
  – We’ll come back to this choice of interface in later lectures
Ivy architecture

- Each node caches read pages
  - Why?
- Can a node directly write cached pages?

If page not found in local memory, request from remote node

Each processor’s local memory keeps a subset of all pages
Ivy aims to provide sequential consistency

• How?
  – Always read a fresh copy of data
    • Must invalidate all cached pages before writing a page.
    • This simulates the FIFO queue for a page because once a page is written, all future reads must see the latest value
  – Only one processor (owner) can write a page at a time
Ivy implementation

• The ownership of a page moves across nodes
  – Latest writer becomes the owner
  – Why?

• Challenge:
  – how to find the owner of a page?
  – how to ensure one owner per page?
  – How to ensure all cached pages are invalidated?
Ivy: centralized manager

<table>
<thead>
<tr>
<th>Page#, access</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1, read</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Page#, copy_set, owner</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1, {..}, {A}</td>
</tr>
</tbody>
</table>
Ivy: read

1. Page fault for p1 on C
2. C sends RQ(read request) to M
3. M sends RF(read forward) to A, M adds C to copy_set
4. A sends p1 to C, C marks p1 as read-only
5. C sends RC(read confirmation) to M
Ivy: write

1. Page fault for p1 on B
2. B sends WQ(write request) to M
3. M sends IV(invalidate) to copy_set = {C}
4. C sends IC(invalidate confirm) to M
5. M clears copy_set, sends WF(write forward) to A
6. A sends p1 to B, clears access
7. B sends WC(write confirmation) to M
Ivy invariants?

• Every page has exactly one current owner
• Current owner has a copy of the page
• If mapped r/w by owner, no other copies
• If mapped r/o by owner, identical to other copies
• Manager knows about all copies
Is Ivy Sequentially Consistent?

• Well, yes, but we’re not gonna prove it…

• Proof sketch:
  – Proof by contradiction that there is a schedule that cannot be explained by any sequential ordering that satisfies the two rules
  – This means that there are operations in the schedule that break one of the two rules
  – Reach contradiction on each of the two rules by using definition of reads/writes in Ivy

• For simplicity, let’s look instead at why Ivy doesn’t have the two problems we’ve identified for naïve DSM
Eventual Consistency (Overview)

• Allow stale reads, but ensure that reads will eventually reflect previously written values
  – Even after very long times

• Doesn’t order concurrent writes as they are executed, which might create conflicts later: which write was first?

• Used in Amazon’s Dynamo, a key/value store
  – Plus a lot of academic systems
  – Plus file synchronization ← familiar example, we’ll use this
Why Eventual Consistency?

- More concurrency opportunities than strict, sequential, or causal consistency
- Sequential consistency requires highly available connections
  - Lots of chatter between clients/servers
- Sequential consistency many be unsuitable for certain scenarios:
  - Disconnected clients (e.g. your laptop goes offline, but you still want to edit your shared document)
  - Network partitioning across datacenters
  - Apps might prefer potential inconsistency to loss of availability
Case-in-Point: Realizing Sequential Consistency

- All reads/writes to address X must be ordered by one memory/storage module responsible for X (see Ivy, Lec10)
- If you write data that others have, you must let them know
- Thus, everyone must be online all the time
Why (Not) Eventual Consistency?

✓ Support *disconnected* operations or network partitions
  – Better to read a stale value than nothing
  – Better to save writes somewhere than nothing

✓ Support for increased parallelism
  – But that’s not what people have typically used this for

☆ Potentially *anomalous* application behavior
  – Stale reads and *conflicting* writes…
Sequential vs. Eventual Consistency

• Sequential: **pessimistic** concurrency handling
  – Decide on update order as they are executed

• Eventual: **optimistic** concurrency handling
  – Let updates happen, worry about deciding their order later
  – May raise **conflicts**

  • Think about when you code offline for a while – you may need to resolve conflicts with other teammates when you commit
  • Resolving conflicts is not that difficult with code, but it’s really hard in general (e.g., think about resolving conflicts when you’ve updated an image)
Example Usage: File Synchronizer

• One user, many gadgets, common files (e.g., contacts)

• Goal of file synchronization
  1. All replica contents eventually become identical
  2. No lost updates
     – Do not replace new version with old ones
Operating w/o Total Connectivity

Client writes to its local replica

Sync w/ server resolves non-conflicting changes, reports conflicting ones to user

No sync between clients
Pair-wise Synchronization

Pair-wise sync resolves non-conflicting changes, reports conflicting ones to users

W(A)1

W(B)3

W(A)2
Prevent lost updates

- Detect if updates were sequential
  - If so, replace old version with new one
  - If not, detect conflict
How to Prevent Lost Updates?

- Strawman: use mtime to decide which version should replace the other

- Problems?
  1. If clocks are unsynchronized: new data might have older timestamp than old data
  2. Does not detect conflicts => may lose some contacts…
Strawman Fix

- Carry the entire modification history (a log)
- If history X is a prefix of Y, Y is newer
- If it’s not, then detect and potentially solve conflicts
H1:2 implies H1:1, so we only need one number per host
How to Deal w/ Conflicts?

• Easy: mailboxes w/ two different set of messages
• Medium: changes to different lines of a C source file
• Hard: changes to same line of a C source file

• After conflict resolution, add a new item to the history?
So, What’s Used Where?

- Sequential consistency
  - A number of both academic and industrial systems provide (at least) sequential consistency (some a bit stronger – linearizability)
  - Examples: Yale’s IVY DSM, Microsoft’s Niobe DFS, Cornell’s chain replication, ...

- Causal consistency – Promiscuous, many other DB replication systems

- Eventual consistency
  - Very popular for a while both in industry and in academia
  - Examples: file synchronizers, Amazon’s Dynamo, Bayou
Many Other Consistency Models Exist

• Other standard consistency models
  – Linearizability
  – Serializability
  – Monotonic reads
  – Monotonic writes
  – … read Tanenbaum 7.3 if interested (these are not required for exam)

• In-house consistency models:
  – AFS’s close-to-open
  – GFS’s atomic at-most-once appends