Inconsistency in Distributed Systems

CS 475, Spring 2019
Concurrent & Distributed Systems
Recurring Problem: Replication

- Replication solves some problems, but creates a huge new one: consistency

Set A=5  “OK”!

Read A  “6”!

OK, we obviously need to actually do something here to replicate the data… but what?
Sequentially Consistent DSM

CPU 1

thread0() → CPU Cache

CPU 2

thread1() → CPU Cache

FIFO queue

Memory

1s?

7ns

100ns

1s?
Ivy Architecture

Each node keeps a cached copy of each piece of data it reads.

If some data doesn’t exist locally, request it from remote node.
• Ivy never copies the actual values until a replica reads them (unlike HW4)
  • Invalidate messages are probably smaller than the actual data!
• Ivy only sends update (invalidate) messages to replicas who have a copy of the data (unlike HW4)
  • Maybe most data is not actively shared
• Ivy requires the lock server to keep track of a few more bits of information (which replica has which data)
  • With near certainty Ivy is a lot faster :)
Today

• Consistency in distributed systems - can we have it all? If not, what can we get?
• Relaxed consistency models
• Reminders:
  • HW3 graded by end of week
  • HW4 is out!
Sequential Consistency

Set A = 5
“OK”!
Read A
“5”!

Set A = 5
“OK!”
Our protocol for sequential consistency does NOT guarantee that the system will be available!
Consistent + Available

Set $A=5$

"OK"!

Read $A$

"5"!

Set $A=5$

Assume replica failed

$A$ replica failed

$A$

$5$

$7$

$6$

$7$
Still broken...

Set $A=5$

“OK”!

Read $A$

“6”!

Set $A=5$

Assume replica failed

5 7

6 7
Network Partitions

- The communication links between nodes may fail arbitrarily
- But other nodes might still be able to reach that node
CAP Theorem

- Pick two of three:
  - Consistency: All nodes see the same data at the same time (sequential consistency)
  - Availability: Individual node failures do not prevent survivors from continuing to operate
  - Partition tolerance: The system continues to operate despite message loss (from network and/or node failure)
- You can not have all three, ever

Our goals as system builders

A property of the environment
CAP Theorem vs FLP

- FLP: Can not guarantee both liveness and agreement assuming messages may be delayed but are eventually delivered
- CAP: Can not guarantee consistency, availability, partition-tolerance assuming messages may be dropped
- Nice comparison: http://the-paper-trail.org/blog/flp-and-cap-arent-the-same-thing/
CAP Theorem

• C+A: Provide strong consistency and availability, assuming there are no network partitions
• C+P: Provide strong consistency in the presence of network partitions; minority partition is unavailable
• A+P: Provide availability even in presence of partitions; no sequential consistency guarantee, \textit{maybe can guarantee something else}
Still broken...

The robot devil will return in lecture 25
Relaxing Consistency

• We can relax two design principles:
  • How stale reads can be
  • The ordering of writes across the replicas
Allowing Stale Reads

<table>
<thead>
<tr>
<th>P1</th>
<th>W(X) 0</th>
<th>R(X)</th>
<th>R(X)</th>
<th>R(X)</th>
</tr>
</thead>
<tbody>
<tr>
<td>P2</td>
<td>W(X) 1</td>
<td>R(X)</td>
<td>W(X) 0</td>
<td>R(X)</td>
</tr>
<tr>
<td>P3</td>
<td>R(X)</td>
<td>R(X)</td>
<td>R(X)</td>
<td>R(X)</td>
</tr>
</tbody>
</table>
Allowing Stale Reads

Java’s memory model is “relaxed” in that you can have stale reads.
Relaxing Consistency

- Intuition: less constraints means less coordination overhead, less prone to partition failure
Naïve DSM

- Assume each machine has a complete copy of memory
- Reads from local memory
- Writes broadcast update to other machines, then immediately continue

```java
class Machine1 {
    DSMInt x = 0;
    DSMInt y = 0;
    
    static void main(String[] args) {
        x = 1;
        if(y==0)
            System.out.println("OK");
    }
}

class Machine2 {
    DSMInt x = 0;
    DSMInt y = 0;
    
    static void main(String[] args) {
        y = 1;
        if(x==0)
            System.out.println("OK");
    }
}
```
Naïve DSM

- Assume each machine has a complete copy of memory
- Reads from local memory
- Writes broadcast update to other machines, then immediately continue

```java
class Machine1 {
    DSMInt x = 0;
    DSMInt y = 0;

    static void main(String[] args) {
        x = 1;
        if (y == 0)
            System.out.println("OK");
    }
}

class Machine2 {
    DSMInt x = 0;
    DSMInt y = 0;

    static void main(String[] args) {
        y = 1;
        if (x == 0)
            System.out.println("OK");
    }
}
```
Naïve DSM

- Assume each machine has a complete copy of memory
- Reads from local memory
- Writes broadcast update to other machines, then immediately continue

```java
class Machine1 {
    DSMInt x = 1;
    DSMInt y = 0;

    static void main(String[] args) {
        x = 1;
        if (y == 0)
            System.out.println("OK");
    }
}

class Machine2 {
    DSMInt x = 0;
    DSMInt y = 1;

    static void main(String[] args) {
        y = 1;
        if (x == 0)
            System.out.println("OK");
    }
}
```

Is this correct?
Naïve DSM

- It definitely is not sequentially consistent
- Are there any guarantees that it provides though?
  - Reads can be stale
  - Writes can be re-ordered
  - Not really.
- Can we come up with something more clever though with SOME guarantee?
  - (Not as is, but with some modifications maybe it's...)}
Causal Consistency

• An execution is **causally-consistent** if all **causally-related** read/write operations are executed in an order that reflects their causality.

• Reads are fresh ONLY for writes that they are dependent on.

• Causally-related writes appear in order, but not in order to others.

• Concurrent writes can be seen in different orders by different machines.

• Compare to sequential consistency: **every machine** must see the same order of operations!
Causal Consistency

Causally Consistent. \( W(X)_b \) and \( W(X)_c \) are not related, hence could have happened one either order. 
\( W(X)_a \) and \( W(X)_B \) ARE causally related and must occur in this order.

<table>
<thead>
<tr>
<th></th>
<th>( W(X)_a )</th>
<th>( W(X)_b )</th>
<th>( W(X)_c )</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>( W(X)_a )</td>
<td>( W(X)_c )</td>
<td></td>
</tr>
<tr>
<td>P2</td>
<td>( R(X)_a )</td>
<td>( W(X)_b )</td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td>( R(X)_a )</td>
<td>( R(X)_c )</td>
<td>( R(X)_b )</td>
</tr>
<tr>
<td>P4</td>
<td>( R(X)_a )</td>
<td>( R(X)_b )</td>
<td>( R(X)_c )</td>
</tr>
</tbody>
</table>
## Causal Consistency

<table>
<thead>
<tr>
<th></th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
</tr>
</thead>
<tbody>
<tr>
<td>W(X)</td>
<td>a</td>
<td>R(X)</td>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>W(X)</td>
<td>b</td>
<td>R(x)</td>
<td>b</td>
<td>a</td>
</tr>
<tr>
<td>R(x)</td>
<td>b</td>
<td>R(x)</td>
<td>a</td>
<td>b</td>
</tr>
</tbody>
</table>

**NOT Causally Consistent.** X couldn’t have been b after it was a

<table>
<thead>
<tr>
<th></th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
</tr>
</thead>
<tbody>
<tr>
<td>W(X)</td>
<td>a</td>
<td>W(X)</td>
<td>b</td>
<td></td>
</tr>
<tr>
<td>W(X)</td>
<td>b</td>
<td>R(x)</td>
<td>b</td>
<td>a</td>
</tr>
<tr>
<td>R(x)</td>
<td>b</td>
<td>R(x)</td>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>R(x)</td>
<td>a</td>
<td>R(x)</td>
<td>b</td>
<td></td>
</tr>
</tbody>
</table>

**Causally Consistent.** X can be a or b concurrently
Why Causal Consistency?

• It is clearly weaker than sequential consistency
  • (Note that anything that is sequentially consistent is also causally consistent)
• Many more operations for concurrency
  • Parallel (non-dependent) operations can occur in parallel in different places
    • Sequential would enforce a global ordering
  • E.g. if W(X) and W(Y) occur at the same time, and without dependencies, then they can occur without any locking
• Still requires some perhaps complicated implementation - each client must know what is related to what.
Eventual Consistency

• Allow stale reads, but ensure that reads will **eventually** reflect the previously written values
  • Eventually: milliseconds, seconds, minutes, hours, years…
• Writes are NOT ordered as executed
  • Allows for conflicts. Consider: Dropbox
• Git is eventually consistent
Eventual Consistency

• More concurrency than strict, sequential or causal
  • These require **highly available** connections to send messages, and generate lots of chatter
• Far looser requirements on network connections
  • Partitions: OK!
  • Disconnected clients: OK!
  • Always available!
• Possibility for conflicting writes :(
Review: Ivy Architecture

Each node keeps a cached copy of each piece of data it reads.

Write X=1

If some data doesn’t exist locally, request it from a remote node.

invalidate x

All of these messages…
All of the clients must always be online!
Relax!

Read X

cached data

Read X

cached data

Read X

cached data

x=0

x=0

x=1

x=1

x=0

x=1

x=1

x=0
Sequential vs Eventual Consistency

• Sequential: “Pessimistic” concurrency control
  • Assume that everything could cause a conflict, decide on an update order as things execute, then enforce it

• Eventual: “Optimistic” concurrency control
  • Just do everything, and if you can’t resolve what something should be, sort it out later
  • Can be tough to resolve in general case
Eventual Consistency: Distributed Filesystem

When everything can talk, it’s easy to synchronize, right?

Goal: Everything eventually becomes synchronized.
No lost updates (don’t replace new version with old)
Eventual Consistency: Distributed Filesystem

When everything can talk, it’s easy to synchronize, right?

Goal: Everything eventually becomes synchronized.
No lost updates (don’t replace new version with old)

Fix: Add coordinating sync server
Eventual Consistency: Distributed Filesystem

- Role of the sync server:
  - Resolve conflicting changes, report conflicts to user
  - Do not allow sync between clients
  - Detect if updates are sequential
  - Enforce ordering constraints
Detecting Conflicts

Do we just use timestamps?

\[ t=0 \]
write \( x = a \)

\[ t=1 \]
write \( x = b \)
Detecting Conflicts

Do we just use timestamps?

\[ t=0 \]
write \( x = a \)

\[ t=1 \]
write \( x = b \)

\textbf{NO, what if clocks are out of sync?}
\textbf{NO does not actually detect conflicts}
Detecting Conflicts

Solution: Track version history on clients

\[
v=0
\]
write \( x = a \)

\[
v=0
\]
write \( x = b \)

Still doesn’t tell us what to do with a conflict
Client-Centric Consistency

- What can we guarantee in disconnected operation?
- Monotinic-reads: any future reads will return the same or newer value (never older)
- Monotonic-writes: A processes’ writes are always processed in order
- Read-you-writes
- Writes follow reads
Eventually Consistent + Available + Partition Tolerant

Set A=5

“OK”!

Read A

“6”!

Assume replica failed

5 7

5 7
Choosing a consistency model

• Sequential consistency
  • All over - it’s the most intuitive

• Causal consistency
  • “Increasingly useful” but not really widely used - still pay coordination cost, unclear what the performance benefits are

• Eventual consistency
  • Very popular in industry and academia
  • File synchronizers, Amazon’s Bayou and more
Example: Facebook

- Problem: >1 billion active users
- Solutions: Thousands of servers across the world
- What kind of consistency guarantees are reasonable? Need 100% availability!
- If I post a story on my news feed, is it OK if it doesn’t immediately show up on yours?
  - Two users might not see the same data at the same time
  - Now this is “solved” anyway because there is no “sort by most recent first” option anyway
Example: Airline Reservations

- Reservations and flight inventory are managed by a GDS (Global Distribution System), who acts as a middle broker between airlines, ticket agencies and consumers [Except for Southwest and Air New Zealand and other oddballs]
- GDS needs to sell as many seats as possible within given constraints
- If I have 100 seats for sale on a flight, does it matter if reservations for flights are reconciled immediately?
- If I have 5 seats for sale on a flight, does it matter if reservations are reconciled immediately?
Example: Airline Reservations

- Result: Reservations can be made using either a strong consistency model or a weak, eventual one.
- Most reservations are made under the normal strong model (reservation is confirmed immediately).
- GDS also supports “Long Sell” - issue a reservation without confirmed availability, need to eventually reconcile it.
- Long sells require the seller to make clear to the customer that even though there’s a confirmation number it’s not confirmed!
Filesystem consistency

- What consistency guarantees do a filesystem provide?
- read, write, sync, close
- On sync, guarantee writes are persisted to disk
- Readers see most recent
- What does a network file system do?
Network Filesystem Consistency

• How do you maintain these same semantics?
• (Cheat answer): Very, very expensive
  • EVERY write needs to propagate out
  • EVERY read needs to make sure it sees the most recent write
  • Oof. Just like Ivy.
Consistency Takeaways

• Strong consistency (sequential or strict) comes at a tradeoff: performance, availability

• Weaker consistency also has a tradeoff (weaker consistency)

• But: applications can make these design choices clear to end-users
  • Facebook
  • Dropbox

• Next week: examples of two systems that involve replication and handle consistency differently: DNS, NFS
This work is licensed under a Creative Commons Attribution-ShareAlike license

- This work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-sa/4.0/
- You are free to:
  - Share — copy and redistribute the material in any medium or format
  - Adapt — remix, transform, and build upon the material
  - for any purpose, even commercially.
- Under the following terms:
  - Attribution — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.
  - ShareAlike — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.
  - No additional restrictions — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.