Consistency

CS 475, Spring 2018
Concurrent & Distributed Systems
Review: 2PC, Timeouts when Coordinator crashes

- What if the bank doesn’t hear back from coordinator?
- If bank voted “no”, it’s OK to abort
- If bank voted “yes”
  - It can’t decide to abort (maybe both banks voted “yes” and coordinator heard this)
  - It can’t decide to commit (maybe other bank voted yes)
- Hence, 2PC does NOT guarantee liveness, but it will guarantee safety
Review: 3PC, Has an answer for every CRASH timeout

Coordinator  Participants (A,B,C,D)

Soliciting votes

Timeout causes abort

Commit authorized (if all yes)

Timeout causes abort

Done

Status: Uncertain

Timeout causes abort

Status: Prepared to commit

Timeout causes commit

Status: Committed

This only works if a timeout == crash!
Review: Partitions

Coordinator

Participant B: Prepared to commit
Participant C: Prepared to commit
Participant D: Prepared to commit
Coordinator: Soliciting Authorization

Network Partition!!!

Timeout behavior: Abort
Timeout behavior: Commit!
Review: FLP

• Why can’t we make a protocol for consensus/agreement that can tolerate node failures or network delays that may be transient?
• To tolerate a failure, you need to assume that **eventually** the failure will be resolved, and the network will deliver the delayed packets
• But the messages might be delayed **forever**
• Hence, your protocol would not come to a result, until **forever** (it would not have the **liveness** property)
Review: Partition Tolerant Consensus

• Key idea: if you always have an odd number of nodes...
• There will always be a minority partition and a majority partition
• Give up processing in the minority until partition heals and network resumes
• Majority can continue processing
• Really, really really complicated to design (Paxos) and implement (ZooKeeper, Raft) an algorithm that does this
Announcements

- HW4 is out!
- Today:
  - Consistency & Memory Models
  - Strict Consistency
  - Sequential Consistency
  - Distributed Shared Memory
- Additional readings:
  - Tannenbaum 7-7.2
Review: Properties of Transactions

- Traditional properties: ACID
  - Atomicity: transactions are “all or nothing”
  - Consistency: Guarantee some basic properties of data; each transaction leaves the database in a valid state
  - Isolation: Each transaction runs as if it is the only one; there is some valid serial ordering that represents what happens when transactions run concurrently
  - Durability: Once committed, updates cannot be lost despite failures
Consistency

- The problem of consistency arises whenever some data is replicated.
- That data exists in (at least) two places at the same time.
- What is a "valid" state?
Set A=5  "OK"!  Read A  "5"!

Set A=5  "OK!"

"OK!"
Consistency

• Why do we think the prior slide was consistent?
  • Whenever we read, we see the most recent writes
• Does that come for free?
  • No, remember NFS
  • What do you need to do ensure that each read reflects the most recent write?
• Even programs running on a single computer have to obey some consistency model
Quiz: What’s the output?

class MyObj {
    int x = 0;
    int y = 0;

    void thread0()
    {
        x = 1;
        if(y==0)
            System.out.println("OK");
    }

    void thread1()
    {
        y = 1;
        if(x==0)
            System.out.println("OK");
    }
}
class MyObj {
    int x = 0;
    int y = 0;

    void thread0() {
        x = 1;
        if (y == 0) {
            System.out.println("OK");
        }
    }

    void thread1() {
        y = 1;
        if (x == 0) {
            System.out.println("OK");
        }
    }
}

Java Memory Model: Threads are allowed to cache reads and writes
Java Memory Model

CPU 1
- thread0()
- CPU 1 Cache
- 7ns
- 100ns

CPU 2
- thread1()
- CPU 2 Cache

Main Memory
Quiz: What’s the output?

class MyObj {
    volatile int x = 0;
    volatile int y = 0;

    void thread0() {
        x = 1;
        if(y==0)
            System.out.println("OK")
    }

    void thread1() {
        y = 1;
        if(x==0)
            System.out.println("OK")
    }
}
class MyObj {
    volatile int x = 0;
    volatile int y = 0;

    void thread0() {
        x = 1;
        if (y == 0)
            System.out.println("OK");
    }

    void thread1() {
        y = 1;
        if (x == 0)
            System.out.println("OK");
    }
}
Volatile Keyword

CPU 1

thread0() \( \rightarrow \) CPU 1 Cache \( \rightarrow \) Main Memory

CPU 2

thread1() \( \rightarrow \) CPU 2 Cache \( \rightarrow \) Main Memory

\( 7\text{ns} \)

\( 100\text{ns} \)
Consistency

• This is a consistency model!
  • Constraints on the system state that are observable by applications
  • “When I write $y=1$, any future reads must say $y=1$”
  • ... except in Java, if it’s a non-volatile variable
• Clearly, this often comes at a cost (see simple example with volatile...)

Strict Consistency

- Each operation is stamped with a global (wall-clock, aka absolute) time
- Rules:
  - Each read sees the latest write
  - All operations on one CPU have time-stamps in execution order
Strict Consistency

class MyObj {
    volatile int x = 0;
    volatile int y = 0;

    void thread0() {
        x = 1;
        if(y==0)
            System.out.println("OK");
    }

    void thread1() {
        y = 1;
        if(x==0)
            System.out.println("OK");
    }
}
Sequential Consistency

- Strict consistency is often not practical
  - Requires globally synchronizing clocks
- Sequential consistency gets close, in an easier way:
  - There is some total order of operations so that:
    - Each CPUs operations appear in order
    - All CPUs see results according to that order
      (read most recent writes)
Sequential Consistency

- There is some *total order* of operations so that:
  - Each CPUs operations appear in order
  - All CPUs see results according to that order (read most recent writes)
- Consider this case, noting that there are **no locks** to enforce the ordering

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

**Sequentially consistent. NOT strictly consistent**

W(X)b, R(X)b, R(X)b, W(X)a, R(X)a, R(X)a
Sequential Consistency

- There is some *total order* of operations so that:
  - Each CPUs operations appear in order
  - All CPUs see results according to that order (read most recent writes)
- Consider this case, noting that there are **no locks** to enforce the ordering

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

**Not sequentially consistent**
Sequential Consistency: Activity

- [http://b.socrative.com](http://b.socrative.com), room CS475
Distributed Shared Memory

thread0() stack
heap

thread1() stack
heap

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

• Gets even more funny when we add a third host
  • Many more interleaving possible
• Definitely not sequentially consistent
• Who is at fault?
  • The DSM system?
  • The app?
• The developers of the app, if they thought it would be sequentially consistent.
Sequentially Consistent DSM

• How do we get this system to behave similar to Java’s volatile keyword?

• We want to ensure:
  • Each machine’s own operations appear in order
  • All machines see results according to some total order (each read sees the most recent writes)

• We can say that some observed runtime ordering of operations can be “explained” by a sequential ordering of operations that follow the above rules
Sequentially Consistent DSM

- Each node must see the most recent writes before it reads that same data
- Performance is not great:
  - Might make writes expensive: need to wait to broadcast and ensure other nodes heard your new value
  - Might make reads expensive: need to wait to make sure that there are no pending writes that you haven’t heard about yet
Sequentially Consistent DSM

• Each processor issues requests in the order specified by the program
  • Can’t issue the next request until previous is finished
• Requests to an individual memory location are served from a single FIFO queue
  • Writes occur in single order
  • Once a read observes the effect of a write, it’s ordered behind that write
Sequentially Consistent DSM

CPU 1
- thread0()
- 7ns
- CPU 1 Cache
- 100ns
- Main Memory
- 1s?

CPU 2
- thread1()
- CPU 2 Cache
- FIFO queue

GMU CS 475 Spring 2018
Ivy DSM

- Integrated shared Virtual memory at Yale
- Provides shared memory across a group of workstations
- Might be easier to program with shared memory than with message passing
  - Makes things look a lot more like one huge computer with hundreds of CPUs instead of hundreds of computers with one CPU
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 provides sequential consistency

• Support multiple readers, single writer semantics
• Write invalidate update protocol
• If I write some data, I must tell everyone who has cached it to get rid of their cache
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 remote node.

cached data
x=1

invalidate x

cached data
x=0

read x

read x

Read X

cached data
x=0

cached data
x=1

Read X
Ivy Implementation

• Ownership of data moves to be whoever last wrote it
• There are still some tricky bits:
  • How do we know who owns some data?
  • How do we ensure only one owner per data?
  • How do we ensure all cached data are invalidated on writes?
• Solution: Central manager node
Ivy invariants

• Every piece of data has exactly one current owner
• Current owner is guaranteed to have a copy of that data
• If the owner has write permission, no other copies can exist
• If owner has read permission, it’s guaranteed to be identical to other copies
• Manager node knows about all of the copies
• Sounds a lot like HW4? :)}
Each node keeps a cached copy of each piece of data it reads.

- **cached data**
  - x = 1

**Write X = 1**

**Update x = 1**

- Each node always has a copy of the most recent data.

**Read X**

- cached data
  - x = 0

**Read X**
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 remote node.

Read $X$

cached data

$x=1$

invalidate $x$

read $x$

cached data

$x=0$

cached data

$x=0$

cached data

$x=1$
Ivy vs HW4

• 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 :)

Sequential Consistency

Set A=5

“AOK”!

Read A

“5”!

Set A=5

“AOK!”

5 7

5 7
Availability

- 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

Assume replica failed
Still broken...

Set A=5

“OK”!

Assume replica failed

Set 5

Read A

“6”!

A  B

5  7

A  B

6  7

J. Bell

GMU CS 475 Spring 2018
Network Partitions

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

```
Set A=5

"OK"!

Read A

"6"!
```

Assume replica failed

```
5 7

6 7
```
CAP Theorem

• Pick two of three:
  • Consistency: All nodes see the same data at the same time (strong 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**
  • If you relax your consistency guarantee (we’ll talk about in a few weeks), you might be able to guarantee THAT…
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 strong consistency guarantee.
Still broken...

The robot devil will return in lecture 23