Sharding (and replication)

<table>
<thead>
<tr>
<th>1</th>
<th>Pete</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>John</td>
</tr>
<tr>
<td>3</td>
<td>Al</td>
</tr>
<tr>
<td>4</td>
<td>Joe</td>
</tr>
</tbody>
</table>

- shard 1 \{ node 1 \} \{ replicas \}
- shard 2 \{ node 3 \} \{ node 4 \} consensus group
- shard 3 \{ node 5 \} how consistent?

Distributed consensus

- **Paxos**
  - majority rules, multiple rounds
- **Soft**
  - guarantees safety
  - does not guarantee progress

also assume fail-stop failures

- a replica follows protocol correctly
- or stops responding

Could also have byzantine failures

- replicas can lie, cheat, evil
PAXOS

Agree on a single result in network of unreliable processes.

Assumptions:

- non-byzantine (fail-stop) faults
- processors have arbitrary speed
- processors w/ stable storage may rejoin after failure

Safety requirements:

- only a value that has been proposed may be chosen (non-triviality)
- only a single value chosen (consistency)
- process never learns a value is chosen unless it has been (correctness)
- impossible, at least for asynchronous systems (fischer '85)

Roles

- clients
- proposers
- acceptors
- learners

Invariants:

1. P1: Accept first proposal received
   - works even if only one

2. P2: If proposal w/ value v is chosen, all higher-numbered chosen proposals choose v

Procedure:

1. Proposer issues PREPARE w/ proposal n and sends to requesters, returns:
   - PROMISE never again to accept a proposal < n, with
     - highest number <n it has accepted
     - Value v it has accepted previously

2. ACCEPT request w/ n, and value (possibly from prior proposal)
   - acceptor can accept a a proposal n iff it has not PROMISED w/ num > n
   - response is ACCEPTED
   - Majority of ACCEPTED is the winner. Safety properties satisfied

Opt:

- don't respond to PREPARE w/ num less than one already seen in a PREPARE (cause won't accept it anyway)
Progress:

- not guaranteed because p and q can duel w/ ever-increasing n

**Paxos ("Single Decree")**

<table>
<thead>
<tr>
<th>Client</th>
<th>Proposer</th>
<th>Acceptor</th>
<th>Learner</th>
</tr>
</thead>
<tbody>
<tr>
<td>X------</td>
<td>----------</td>
<td>----------</td>
<td>---------</td>
</tr>
<tr>
<td></td>
<td>X--------</td>
<td>----------</td>
<td>---------</td>
</tr>
<tr>
<td></td>
<td>X--------</td>
<td>----------</td>
<td>---------</td>
</tr>
<tr>
<td></td>
<td>X--------</td>
<td>----------</td>
<td>---------</td>
</tr>
<tr>
<td></td>
<td>X--------</td>
<td>----------</td>
<td>---------</td>
</tr>
<tr>
<td></td>
<td>X--------</td>
<td>----------</td>
<td>---------</td>
</tr>
<tr>
<td></td>
<td>X--------</td>
<td>----------</td>
<td>---------</td>
</tr>
</tbody>
</table>

**Dueling Proposers**

<table>
<thead>
<tr>
<th>Client</th>
<th>Leader</th>
<th>Acceptor</th>
<th>Learner</th>
</tr>
</thead>
<tbody>
<tr>
<td>X------</td>
<td>------</td>
<td>----------</td>
<td>---------</td>
</tr>
<tr>
<td></td>
<td>X------</td>
<td>----------</td>
<td>---------</td>
</tr>
<tr>
<td></td>
<td>X------</td>
<td>----------</td>
<td>---------</td>
</tr>
<tr>
<td></td>
<td>X------</td>
<td>----------</td>
<td>---------</td>
</tr>
<tr>
<td></td>
<td>X------</td>
<td>----------</td>
<td>---------</td>
</tr>
</tbody>
</table>

**Multi-Paxos (collapsed roles) ("Multiple Decrees")**

<table>
<thead>
<tr>
<th>Client</th>
<th>Servers</th>
</tr>
</thead>
<tbody>
<tr>
<td>X------</td>
<td>Request</td>
</tr>
<tr>
<td>X------</td>
<td>Prepare(N)</td>
</tr>
<tr>
<td>X------</td>
<td>Accept!(N, I, V)</td>
</tr>
<tr>
<td>X------</td>
<td>Response</td>
</tr>
<tr>
<td>X------</td>
<td>Request</td>
</tr>
<tr>
<td>X------</td>
<td>Accept!(N, I+1, W)</td>
</tr>
</tbody>
</table>
Example of Scalability Problem

Customer 2 buys widget 3 with store credit:

\[
W = \text{Widgets.READ}(3) \\
C = \text{Customers.READ}(2) \\
\text{IF (W.In Stock < 1)} \\
\text{\hspace{1cm} ABORT} \\
\text{IF (C.Store_Credit < W.price)} \\
\text{\hspace{1cm} ABORT} \\
\text{W.In Stock -= 1} \\
\text{C.Store_Credit -= W.price}
\]
### Example of Scalability Problem

<table>
<thead>
<tr>
<th>Widget ID</th>
<th>Price</th>
<th>In_Stock</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>99</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>325</td>
<td>12</td>
</tr>
<tr>
<td>3</td>
<td>79</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>199</td>
<td>7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Customer ID</th>
<th>Store_Credit</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>100</td>
</tr>
<tr>
<td>3</td>
<td>50</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>6</td>
<td>500</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>25</td>
</tr>
</tbody>
</table>

Customer 2 buys widget 3 with store credit:

\[
W = \text{Widgets.READ}(3) \\
C = \text{Customers.READ}(2) \\
\text{IF (W.In_Stock < 1) ABORT} \\
\text{IF (C.Store_Credit < W.price) ABORT} \\
W.In_Stock -= 1 \\
C.Store_Credit -= W.price
\]

### Example of Scalability Problem

<table>
<thead>
<tr>
<th>Widget ID</th>
<th>Price</th>
<th>In_Stock</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>99</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>325</td>
<td>12</td>
</tr>
<tr>
<td>3</td>
<td>79</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>199</td>
<td>7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Customer ID</th>
<th>Store_Credit</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>21</td>
</tr>
<tr>
<td>3</td>
<td>50</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>6</td>
<td>500</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>25</td>
</tr>
</tbody>
</table>

Customer 2 buys widget 3 with store credit:

\[
W = \text{Widgets.READ}(3) \\
C = \text{Customers.READ}(2) \\
\text{IF (W.In_Stock < 1) ABORT} \\
\text{IF (C.Store_Credit < W.price) ABORT} \\
W.In_Stock -= 1 \\
C.Store_Credit -= W.price
\]
Example of Scalability Problem

Customer 2 buys widget 3 with store credit:

\[
W = \text{Widgets.READ}(3) \\
C = \text{Customers.READ}(2) \\
\text{IF} \ (W.\text{In}_\text{Stock} < 1) \quad \text{ABORT} \\
\text{IF} \ (C.\text{Store}_\text{Credit} < W.\text{price}) \quad \text{ABORT} \\
W.\text{In}_\text{Stock} -= 1 \\
C.\text{Store}_\text{Credit} -= W.\text{price}
\]
**Example of Scalability Problem**

Customer 2 buys widget 3 with store credit:

\[
W = \text{Widgets.READ}(3) \\
C = \text{Customers.READ}(2) \\
\text{IF} \ (W.\text{In}\_\text{Stock} < 1) \\
\text{ABORT} \\
\text{IF} \ (C.\text{Store}\_\text{Credit} < W.\text{price}) \\
\text{ABORT} \\
W.\text{In}\_\text{Stock} -= 1 \\
C.\text{Store}\_\text{Credit} -= W.\text{price}
\]

**Replication Exacerbates the Problem**

Replicate transactions can run simultaneously:

\[
\text{No conflicting transactions can run!}
\]

Dreaded 2-phase commit protocol
Replication Exacerbates the Problem

No conflicting transactions can run!
Summary of Performance Issues

• Commit protocol like 2PC
  – Hurts performance
  – Helps with atomicity and isolation guarantees
• Synchronous replication
  – Hurts performance
  – Helps with consistency guarantees

But what if you want atomic, isolated MP Xacts and synchronous replication?

• Need some coordination
  – Costs latency
  – Costs throughput
  – State of the art increases these costs beyond what is strictly necessary
• Calvin
  – Uses determinism to move most of coordination outside of transaction boundaries
    • Drastically improves concurrency and throughput
    • Slightly improves latency
    • Simplifies and reduces monolithic nature of DBMS architecture
Replication Example: Traditional Mechanism

Customer 2 buys widget 3 with store credit:
W = Widgets.READ(3)
C = Customers.READ(2)
IF (W.In_Stock < 1) THEN ABORT
IF (C.Store_Credit < W.price) THEN ABORT
W.In_Stock -= 1
C.Store_Credit -= W.price

<table>
<thead>
<tr>
<th>Widget ID</th>
<th>Price</th>
<th>In_Stock</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>325</td>
<td>12</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Customer id</th>
<th>Store_Credit</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>100</td>
</tr>
<tr>
<td>6</td>
<td>500</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Widget ID</th>
<th>Price</th>
<th>In_Stock</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>79</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Customer id</th>
<th>Store_Credit</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>50</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
</tr>
</tbody>
</table>
Replication Example: Traditional Mechanism

Customer 2 buys widget 3 with store credit:
W = Widgets.READ(3)
C = Customers.READ(2)
IF (W.In_Stock < 1) THEN ABORT
IF (C.Store_Credit < W.price) THEN ABORT
W.In_Stock := 1
C.Store_Credit := W.price

Replication Example: Deterministic Mechanism

Customer 2 buys widget 3 with store credit:
W = Widgets.READ(3)
C = Customers.READ(2)
IF (W.In_Stock < 1) THEN ABORT
IF (C.Store_Credit < W.price) THEN ABORT
W.In_Stock := 1
C.Store_Credit := W.price
Replication Example: Deterministic Mechanism

Customer 2 buys widget 3 with store credit:
W = Widgets.READ(3)
C = Customers.READ(2)
IF (W.In_Stock < 1) THEN ABORT
IF (C.Store_Credit < W.price) THEN ABORT
W.In_Stock = 1
C.Store_Credit = W.price

Customer 2 buys widget 3 with store credit:
W = Widgets.READ(3)
C = Customers.READ(2)
IF (W.In_Stock < 1) THEN ABORT
IF (C.Store_Credit < W.price) THEN ABORT
W.In_Stock = 1
C.Store_Credit = W.price

Why the previous mechanism doesn’t work without a deterministic DBMS

T1: Customer 2 buys widget 3 with store credit:
W = Widgets.READ(3)
C = Customers.READ(2)
IF (W.In_Stock < 1) THEN ABORT
IF (C.Store_Credit < W.price) THEN ABORT
W.In_Stock = 1
C.Store_Credit = W.price

T2: Customer 6 buys widget 3 with store credit:
W = Widgets.READ(3)
C = Customers.READ(6)
IF (W.In_Stock < 1) THEN ABORT
IF (C.Store_Credit < W.price) THEN ABORT
W.In_Stock = 1
C.Store_Credit = W.price
Why the previous mechanism doesn’t work without a deterministic DBMS

T1: Customer 2 buys widget 3 with store credit:
W = Widgets.READ(3)
C = Customers.READ(2)
IF (W.In_Stock < 1) THEN ABORT
IF (C.Store_Credit < W.price) THEN ABORT
W.In_Stock -= 1
C.Store_Credit -= W.price

T2: Customer 6 buys widget 3 with store credit:
W = Widgets.READ(3)
C = Customers.READ(6)
IF (W.In_Stock < 1) THEN ABORT
IF (C.Store_Credit < W.price) THEN ABORT
W.In_Stock -= 1
C.Store_Credit -= W.price

DBMS has diverged

T1 Grabs Lock First
T2 Grabs Lock First

Traditional Database Systems Are Nondeterministic

• Traditional database systems guarantee “serializability”
  – Concurrent execution equivalent to execution in some serial order
    • Which serial order is determined by nondeterministic events
Why Nondeterminism?

- Building DBMS on top of:
  - OSes that enable threads to be scheduled arbitrarily
    - Arbitrary transaction interleaving
  - Networks that deliver messages with arbitrary delays (and potentially arbitrary orders)
  - Hardware that can fail arbitrarily
- Only natural to allow the state of the database to be dependent on these nondeterministic events
- *Database users have accepted the nondeterministic serializability guarantee*

Architecture of a Deterministic DBMS

DBMSs guarantee serial equivalence to order of txns in log
Failure in a Traditional DBMS

Customer 2 buys widget 3 with store credit:
W = Widgets.READ(3)
C = Customers.READ(2)
IF (W.In_Stock < 1) THEN ABORT
IF (C.Store_Credit < W.price) THEN ABORT
W.In_Stock -= 1
C.Store_Credit -= W.price
Failure in a Traditional DBMS

Customer 2 buys widget 3 with store credit:

\[
W = \text{Widgets.READ}(3);
C = \text{Customers.READ}(2);
\]

\[
\text{IF (W.In_Stock < 1) THEN ABORT;}
\text{IF (C.Store_Credit < W.price) THEN ABORT;}
\]

\[
W.In_Stock -= 1;
C.Store_Credit -= W.price;
\]

---

Failure in a Deterministic DBMS

\[
W = \text{Widgets.READ}(3); C = \text{Customers.READ}(2);
\]

\[
\text{IF (W.In_Stock < 1) THEN ABORT;}
\text{IF (C.Store_Credit < W.price) THEN ABORT;}
\]

\[
W.In_Stock -= 1;
C.Store_Credit -= W.price;
\]

Distributed Replicated Log
Failure in a Deterministic DBMS

Customer 2 buys widget 3 with store credit:

\[
\begin{align*}
W &= \text{Widgets.\text{READ}(3)} \\
C &= \text{Customers.\text{READ}(2)} \\
\text{IF (W.In\_Stock < 1) THEN ABORT} \\
\text{IF (C.Store\_Credit < W.price) THEN ABORT} \\
W.In\_Stock &:= 1 \\
C.Store\_Credit &:= W.price
\end{align*}
\]

2PC assumes failures cause aborts

Customer 2 buys widget 3 with store credit:

\[
\begin{align*}
W &= \text{Widgets.\text{READ}(3)} \\
C &= \text{Customers.\text{READ}(2)} \\
\text{IF (W.In\_Stock < 1) THEN ABORT} \\
\text{IF (C.Store\_Credit < W.price) THEN ABORT} \\
W.In\_Stock &:= 1 \\
C.Store\_Credit &:= W.price
\end{align*}
\]
2PC assumes failures cause aborts

Customer 2 buys widget 3 with store credit:
\[ W = \text{Widgets.READ}(3) \]
\[ C = \text{Customers.READ}(2) \]
IF (\( W.\text{In}_\text{Stock} \) < 1) THEN ABORT
IF (\( C.\text{Store}_\text{Credit} \) < \( W.\text{price} \)) THEN ABORT
\( W.\text{In}_\text{Stock} := 1 \)
\( C.\text{Store}_\text{Credit} := W.\text{price} \)

No need for 2PC when Deterministic

\[ W = \text{Widgets.READ}(3); C = \text{Customers.READ}(2) \]
IF (\( W.\text{In}_\text{Stock} \) < 1) THEN ABORT
IF (\( C.\text{Store}_\text{Credit} \) < \( W.\text{price} \)) THEN ABORT
\( W.\text{In}_\text{Stock} := 1 \)
\( C.\text{Store}_\text{Credit} := W.\text{price} \)
Failure in a Deterministic DBMS

```
W = Widgets.READ(3); C = Customers.READ(2)
IF (W.In_Stock < 1) THEN ABORT
IF (C.Store_Credit < W.price) THEN ABORT
W.In_Stock = W.In_Stock - 1
C.Store_Credit = C.Store_Credit - W.price
```

Distributed Replicated Log

Not going to abort

Commit

Not going to abort

Summary of Commit Protocol if you have a Deterministic DBMS

- If data-dependent abort is impossible
  - Commit as soon as it is writing into distributed log
    - Very low latency
    - No commit protocol
      - Increased concurrency due to lower contention footprint

- If data-dependent abort is possible
  - Commit as soon as transaction finishes with all code that could potentially cause an abort
    - More latency than above, but much less than 2PC
  - Communication necessary to find out if abort will happen
    - Can be overlapped with transaction execution (unlike 2PC)
    - 1 round of communication instead of 2 rounds
      - Reduces contention footprint
Architecture of a Deterministic DBMS

Distributed Replicated Log

DBMSs guarantee serial equivalence to order of txns in log

How to guarantee equivalence to running log in serial order

• Every xact immediately requests all locks it will need (in order of log)
  • If it doesn’t know what it will need
    – Run enough of the xact to find out, but do not change the database state
    – Reissue xact to the preprocessor with lock requirements included as parameter
    – Run enough of the new xact to find out if it locked the correct items (database state might have changed in the meantime)
      • If so, then xact can proceed as normal
      • If not, reissue again to the preprocessor and repeat as necessary
  • Both deterministic and deadlock-free

T1: Customer 2 buys widget 3 with store credit:
W = Widgets.READ(3)
C = Customers.READ(2)
IF (W.In_Stock < 1) THEN ABORT
IF (C.Store_Credit < W.price) THEN ABORT
W.In_Stock = 1
C.Store_Credit = W.price

T2: Customer 6 buys widget 3 with store credit:
W = Widgets.READ(3)
C = Customers.READ(6)
IF (W.In_Stock < 1) THEN ABORT
IF (C.Store_Credit < W.price) THEN ABORT
W.In_Stock = 1
C.Store_Credit = W.price

T3: Customer 4 buys widget 3 with store credit:
W = Widgets.READ(3)
C = Customers.READ(4)
IF (W.In_Stock < 1) THEN ABORT
IF (C.Store_Credit < W.price) THEN ABORT
W.In_Stock = 1
C.Store_Credit = W.price
**Additional Advantages**

- Alleviates contention problem in distributed DBMSs
  - Optimistic protocols result in many aborts
  - Pessimistic protocols result in many deadlocks
  - Deterministic protocol never aborts due to contention and never deadlocks

- Simplifies DBMS design and makes it modular
  - Traditional systems have tight interactions between lock manager, recovery manager, access manager, and transaction manager
  - In Calvin:
    - Lock manager totally separate from the rest of DBMS
    - Logging can be literally outside of the DBMS
    - Recovery manager can also be outside DBMS
      - May want to integrate checkpointing with DBMS for performance
  - No ARIES
  - No 2PC

**Architecture of a Deterministic DBMS**

Ordering of replication log is a distributed process does not interfere w/ ongoing transactions can be executed on different, smaller fault-tolerant set of nodes

DBMSs guarantee serial equivalence to order of txns in log