Distributed Systems Reasoning

John Looney, Facebook - https://goo.gl/msPjCx
09:00 - Part 1
10:30 - coffee
10:50 - Part 2
12:40 - Lunch

- Sit at the front
- When you can add more colour, do so
- Speak up
Pipeline & Batch Systems (Part 1)

In which our heroes will:

- Learn what Orchestration means
- Learn how to achieve this with configuration & consensus
- Understand how to choose between data storage technologies
- Read and critique a design document
Orchestration: Finding, Ordering, Sharding

This needs to describe;
- data stores & Inputs
- units of processing (servers, pipeline stages)

And describe how
- ..data enters the system
- ..the system breaks data into parts
- ..those smaller parts are processed
- ..the processors communicate
- ..we know processing is done
What tech we'll discuss today...

- Terraform
- Zookeeper, etcd
- Kafka, Pubsub, SQS
- Apache Spark, Storm
- DNS, Consul
- Mesos, Kubernetes, AWS CE
Old School 'prescription'

- Pick a host to be a master
- Pick hosts to be mapping shards
- Pick hosts to be reducing shards
- Pick a storage location as destination for reduction data

```
/nfs/data/latest-in
(five files)
```

```
Mapper
i-8271
```

```
/nfs/data/tmp
```

```
Reducer
i-2512
```

```
/nfs/data/latest-out
```
Let's make this Dynamically Scaled!
Let's make this resilient!
Let's make this even MORE resilient!
Making Reliability Worse: Failover

- What if both masters are OK, just can't do network?
- What if masters are OK, but can't do heartbeats?
- What if the standby master takes over...and messes up?
- What if the standby master takes over, kills the old master, but it's running old software?
Lockservers; locks

Screw failover, outsource it to a Lockserver!

- Write locks; change the "locked" value
- Advisory locks; subscribe for updates
Lockservers; discovery

$ curl master.1.mapreduce.lockserver
HTTP/1.1 301 Moved Permanently
Content-Type: text/html
Location: https://i-3271:10001/

$ dig srv master._mapreduce.example.org
_master._mapreduce.example.org. 29 IN SRV 10 10 10001 i-3271
Lockservers; discovery

It's not just for masters. The slaves can use lockservers for checkins too!
Lockservers; failover

The King is dead!

Long live the King!
What happens when a Mapper can't talk to a master anymore?
Clients; self-resolution

```
$ echo $SHARD_TYPE
mapper
$ echo $SHARD_NUMBER
1
$ dig +short srv ${SHARD_TYPE}.${SHARD_NUMBER}._mapreduce.example.org
_mapper.1._mapreduce.example.org. 29 IN SRV 10 10 10002 i-1238
$ hostname
I-1231
$ if [ "$\{(\text{dig} +\text{short} \text{srv} \{SHARD\_TYPE\}\.\{SHARD\_NUMBER\}\.\_mapreduce.example.org|\text{cut} -f8 -d ' ')\}" != $\{(\text{hostname})\} ] ; then
    \text{reboot} -q
fi
```
So, what lockserver?

- Zookeeper (old, complex)
- Ghetto solutions, like locking a row in a database
- Npm lockserver.js
- Clustered Redis
- Etcd (the new hotness)
- Consul (complete solution)
What happens when the lockserver falls over?
Let's just add replicas! Though...they need to come to a consensus.
Let's talk about Consensus

Given a set processes, each chooses an initial value:

- All non-faulty processes **eventually** decide on a value
- A majority of processes decide on the **same** value
- The decision must have **proposed by one of the processes**

These three properties are referred to as **termination**, **agreement** and **validity**
Consensus Challenges

- Is it broken, or is it slow?
- Is it unresponsive, or was a message lost en-route?
- The FLP paper shows it's not actually possible always achieve consensus
  - We cannot be 100% sure of the initial state of a system
  - In an asynchronous system, ordering matters for changing *unsure* state to *sure*
  - In any attempt (round) at consensus, things may be *undecided*
  - *Undecided* last time doesn't mean be *decided* this time
Consensus; Requirements

- Given multiple servers, each can propose a value for the log entry
- All will agreed on a single value
- Only one value is chosen
- A server is not told a value is 'chosen' unless it definitely has been
- A value has to be chosen within a timeout
- All servers will be told about the value chosen, eventually
Consensus; Raft

- There is 1 Leader, N-1 followers
- Changes to Log are sent to Leader
- If there is no Leader, an election is called
  - Each Follower asks all others to follow
- Heartbeats (~100ms) from a Leader postpones new elections
- Odd numbers of followers are most efficient
Consensus; Paxos

- All servers can Propose and Accept changes
- Complex proposal system, where each node can propose a change
  - If a majority accept, any subsequent proposal that conflicts is dropped
  - Must increment & persist proposal numbers
- Once a proposal is made, nodes broadcast if they Accept

Consensus; Paxos

- There are similarities to Raft, if every log addition was an election.
- Slower than Raft, but multi-master
- Multi-Paxos can use leader-election to make things go faster (just one proposer at a time, until Leader dies)
Replicated State Machine

Like a Finite State Machine, only...replicated
We need a better way to map input files to workers
Queues

Queue - where 'work items' can be 'leased' by a 'worker' for a period of time, and 'deleted' when done.

Master i-2716 -> Queue -> Work Item

Lease Requests

Mapper i-8271
Spot the Difference!

**Queue** *noun*  
[kyoō/]

Where 'work items' can be 'leased' by a 'worker' for a period of time, and 'deleted' when done.
Spot the Difference!

**Lockserver** *noun*

[lok sur-ver]

Where 'locks' can be 'locked' by a 'client' for a period of time, and 'released' when done.
Challenges of Scaling RSMs

- Batching - not fine-grained, longer latency
- Sharding - one shard can be slower - jitter/unordering
- Pipelining - extra resource tracking, some jitter
Unordered Queues: At Most Once

Queue gives each task, to exactly one worker, exactly once

- Worker fails, task is lost.
Unordered Queues: At Least Once

Queue gives each task to a worker, requests ack before timeout

- Worker #1 times out, task is given to Worker #2
- Worker #1 succeeded eventually, but wasn't reachable for a while
- Task is processed twice
Unordered Queues: Probably Exactly Once

Queue gives each task to a worker, says don't submit after timeout

- All Workers have a synchronised clock
- Worker #1 times out, task is given to Worker #2
- Worker #1 succeeded eventually, but wasn't reachable for a while
- Worker #1 notices that it's past the timeout, so drops the task
- Task is processed twice, saved once
Unordered Queues: Someone Else's Problem

Queue gives a task to multiple consumers, tells them to work it out

- Worker #1, #2 and #3 are given a task
- Worker #2 hits up a lockserver to lock the task
- Worker #2 times out. Lockserver expires the lock.
- Worker #1 grabs the lock, does the job, commits it.
- Worker #2 comes back, realises it's lost the lock, drops the job.
Ordered Queues: Pain And Suffering

- Makes no sense if you have multiple producers
- If you have multiple consumers, processing times can differ
- Ordered Queues can't be internally sharded without locking
- Properly implemented, they should have a deduplication key

Turns out, an ACID database table is best for ordered queues :(
Queues: PubSub & SQS

- Both provide AtLeastOnce semantics, maybe even ProbablyExactlyOnce
- SQS is one-queue-per-api call, PubSub 'subscribes' to multiple topics
- push/pull: SQS is pull, PubSub is both
- PubSub is like SNS/SQS/Kinesis in one
- SQS has 'FIFO' - ordering - if you want (300 qps max)
- SQS cleans up after 14 days, PubSub after 7

https://cloud.google.com/pubsub/docs/overview
https://aws.amazon.com/sqs/
Queues: Kafka, LogDevice

- Far more than a queue, more like a 'streaming log'
- Can be completely persistent, if you want
- Can mimic SQS or PubSub semantics
- Can also be basis for a stream-processing platform

https://kafka.apache.org/intro
Data Storage: CAP Theorem

- Consistency, Availability, Partition Tolerance (pick two)
  - Really 'sequential consistency' vs. 'high availability'
- We can kinda defeat PT it with Timing + Last Write Wins (see Spanner)
- We can kinda defeat Consistency with VectorClocks
- We can also defeat Availability with pre-prepared partitions
Data Storage

ACID

"All things to all people"

- **Atomicity**
  Transactions are 'all or nothing'

- **Consistency** (ugh)
  Refers to the application, not the DB

- **Isolation**
  Transactions don't step on toes

- **Durability**
  "Whatever you are having yourself"
Data Storage

BASE

"You call that a database?"

- **Basically Available**
  Mostly

- **Soft State**
  Snapshots aren't helpful

- **Eventually Consistent**
  If it doesn't make sense, just wait
Data Storage; B-Trees vs. LSM

B-Trees
- Great for many small reads
- Good for updating-in-place
- Good for fast insertions
- Great for heavy use of indexes
- Described as OLTP

Oracle, MySQL, Postgres, NTFS

Log Structure Merging
- More suitable for scanning
- Underlying storage is just logs
- Random writes -> sequential writes
- Can be setup as 'Columnar'
- Occasional 'compactions'

Bigtable, Cassandra, HBase, Lucene, MyRocksDB
Data Storage; Weak vs Strong Isolation

Weak
- No Dirty Reads
- No Dirty Writes
- Snapshot Isolation
- Atomic Writes
- Explicit Locking
- Conflict Resolution

Strong
- Literally Serial Execution
- Two-Phase Locking
  - Per-Row locks
  - Predicate Locks
  - Index-Range Locks
- Serializable Snapshot Isolation
  - MVCC visibility
  - Abort-on-tripwire
- XA Transactions
Data Storage: Data Loss

How do we lose data?

- Disk loss
- Machine loss
- Switch loss
- Cluster loss
- Software bugs
- Security compromise
- Physics
- Chemistry
Data Storage: Data Loss

How do we lose data?

- Disk loss
- Machine loss
- Switch loss
- Cluster loss
- Software bugs
- Security compromise
- Physics
- Chemistry

How do we avoid data loss?

- Replication
- Replication + healthchecks
- Availability Zones
- Availability Zones
- Separate Backups
- Offsite Backups
- Background checksumming
- Scanning for correctable errors
Data Storage: Data Formats

- Columnar vs. Row
- Document vs. Cell Based
- Relational vs. NoSQL vs. Graph
Data Storage: Data Formats

- **Columnar vs. Row**
- **Document vs. Cell Based**
- **Relational vs. NoSQL vs. Graph**

**Row**
If gathering most of a row in every record
Finding a needle in a haystack

**Column**
Scanning in all of one or two columns.
Aggregations, etc.
Data Storage: Data Formats

- Columnar vs. Row
- **Document vs. Cell Based**
- Relational vs. NoSQL vs. Graph

**Cell**
Simple datatypes, with a fixed schema
Everyone is familiar with it from Excel to Oracle
Schema statically enforced on write

**Document**
Complex Datatypes, with looser schemas, like JSON, BSON, ProtocolBuffers, Avro etc.
Metadata is extracted from the Document.
Common in NoSQL, exotic in Relational DBs
Schema dynamically inferred on read
Data Storage: Database Types

- Columnar vs. Row
- Document vs. Cell Based
- **Relational vs. NoSQL vs. Graph**

**Relational**
Great for many-many relationships
Weak at scaling writes
The default since the 1970s

**NoSQL**
Great at storing 'child records' next to a parent
Weak at pulling out single-fields
Riak, Cassandra, Bigtable, Spanner, Dynamo

**Graph**
Stores vertices (data) and edges (relationships)
Queried declaratively, easy to optimise queries
Neo4J, Oracle, SAP Hana
Data Storage: SQL vs. GraphQL

MATCH (p:Product)-[:CATEGORY]-(l:ProductCategory)-[:PARENT*0..]-(:ProductCategory {name:"Dairy Products"})
RETURN p.name

SELECT p.ProductName
FROM Product AS p
JOIN ProductCategory pc ON (p.CategoryID = pc.CategoryID AND pc.CategoryName = "Dairy Products")
JOIN ProductCategory pc1 ON (p.CategoryID = pc1.CategoryID)
JOIN ProductCategory pc2 ON (pc2.ParentID = pc2.CategoryID AND pc2.CategoryName = "Dairy Products")
JOIN ProductCategory pc3 ON (p.CategoryID = pc3.CategoryID)
JOIN ProductCategory pc4 ON (pc4.ParentID = pc4.CategoryID)
JOIN ProductCategory pc5 ON (pc4.ParentID = pc5.CategoryID AND pc5.CategoryName = "Dairy Products");

Taken from https://neo4j.com/developer/guide-sql-to-cypher/
Design Review Time! (Optional)

1. Organise in Groups of 4
2. Read "Fast Recommendation Builder" Design; https://goo.gl/15wK5R
3. Make notes/improvements to the Design
4. Argue!
In which our heroes will discover the joy of working with...

- Eventually Consistent Datastores
- Load Balancers
- Caches
Eventually Consistent Datastores

What's the problem?

- We can't tell when a node will come back
- We can't tell when a netsplit will end
- We can't tell if a node got a message or not
- We are in a hurry, and can't wait all day for confirmation
How do we get 'consistent'?

- Statement based replication
- Write-Ahead-Log replication
- Logical Log Replication
How do we get 'consistent'?

- Statement based replication
- **Write-Ahead-Log replication**
- Logical Log Replication
How do we get 'consistent'?

- Statement based replication
- Write-Ahead-Log replication
- Logical Log Replication
Cassandra & Tunable Consistency

- Choose how many nodes must take writes
- Choose how many nodes must ack writes
Cassandra & Tunable Consistency

- Choose how many nodes must take writes
- Choose how many nodes must ack writes
- Let's choose 4:2 (4 replicas, ack after 2 stored)

T=0 - Client sends data to a cassandra node
Cassandra & Tunable Consistency

- Choose how many nodes must take writes
- Choose how many nodes must ack writes
- **Let's choose 4:2 (4 replicas, ack after 2 stored)**

\[ t=1 \text{ Node sends data to other nodes} \]
Cassandra & Tunable Consistency

- Choose how many nodes must take writes
- Choose how many nodes must ack writes
- Let's choose 4:2 (4 replicas, ack after 2 stored)

\(t=3\) 1 node responds with 'ack'
Cassandra & Tunable Consistency

- Choose how many nodes must take writes
- Choose how many nodes must ack writes
- Let's choose 4:2 (4 replicas, ack after 2 stored)

\[ t=4 \text{ a second node responds with 'ack'} \]
Cassandra & Tunable Consistency

- Choose how many nodes must take writes
- Choose how many nodes must ack writes
- Let's choose 4:2 (4 replicas, ack after 2 stored)

\[ t=5 \] the client-facing node responds with 'ack', without waiting for other two nodes to ack.
Consistency problem #1: Replication Lag

1 Writer + X Readers
- Writer sends Binlogs to Readers
- Readers mutate their database
- Replication lag is ~5ms

Client sends data to the Writer
Consistency problem #1:

**Replication Lag**

1 Writer + X Readers
- Writer sends Binlogs to Readers
- Readers mutate their database
- Replication lag is ~5ms

MySQL Cluster

- Writer
- Reader1
- Reader2

DB Write

Binlog write

\[ t=5\text{ms} \]
Writer's binlog makes it to Reader1
Consistency problem #1: Replication Lag

1 Writer + X Readers
- Writer sends Binlogs to Readers
- Readers mutate their database
- Replication lag is ~5ms
- Client reads old data, joined with other data
  - Reads from Reader2

MySQL Cluster

DB write

DB read

t=6ms

Client reads back from Reader2
Consistency problem #2: Causality Violations

- Comments and Posts are stored on different partitions in a database
- A Post is created. Someone comments on the post.
- The comments are replicated to all shards of the partition
- One shard of the Post DB was slow
- A user read their list of comments, and the app threw a 500 because it couldn't join the comment with the missing post.
Consistency problem #2: Causality Violations

- Comments and Posts are stored on different partitions in a database.
- A Post is created. Someone comments on the post.
- The comments are replicated to all shards of the partition.
- One shard of the Post DB was slow.
- A user read their list of comments, and the app threw a 500 because it couldn't join the comment with the missing post.

Solution: Keep comments to a post in the same partition.
Consistency problem #3: Global split-brain

- We need data living in multiple continents
- We get regular net-splits
- During net-splits, we continue to accept writes
- After net-splits, try work out what the database should be
Consistency problem #3: Global split-brain

Netsplit happens

1. A moderator in the US marks a post as 'unacceptable' with a reason
2. A moderator in the EU marks a post as 'illegal' with a reason
3. The EU appserver sets the 'last updated by' as the EU moderator
4. The US appserver sets the 'last updated by' as the US moderator

Netsplit finishes

What should we do with the post & 'last updated by'?
Consistency problem #3: Global split-brain

Some options...

- Last Write Wins
  - Variants like 'based on userID, not date' or 'based on webserver IP'
- Notify both Admins of the conflict, and hold changes
- Force writes through one writer
- Partition by post ID, with forced-writer
- Transactions
- Dedicated "conflict handler"
  - On read, or on write
- Operational Transformations instead of 'updates'
Handling Scaling; Sharding & Partitioning

- Share data, and the load it attracts over more nodes
- Reduce hotspots where possible
- Round-Robin inbound items of data is naïve
- More partitions (shards) == more fanout
- More replicas == more bandwidth & reliability
Load Balancing; What's The Point?

- Spread the load, evenly.
- Make good use of all nodes.
- Spot broken nodes.
Load Balancing; Which node?

How do we choose the next destination?

Server #1: 22ms median, 100ms 99th%
Server #2: 20ms median, 70ms 99th%
Server #2: 30ms median, 190ms 99th%
Load Balancing; Spreading Load

Why do servers respond differently to requests?

Server #1
30% CPU

Server #2
65% CPU

Server #2
25% CPU

22ms median
100ms 99th%

30ms median
190ms 99th%

20ms median
70ms 99th%
Load Balancing; What's The Point?

What changed?

Seems CPU was such a good proxy here...
Load Balancing; Troubleshooting Time!

Ah. Slow database. What are you going to do?
Ah. Slow database. What are you going to do?
Global load balancers can't just go on response times to or CPU of the last node in the chain.

A backend could report the max of many metrics, or any of it's children's metrics.
Load Balancing; Common Failure Modes

- Thundering Herd & Lukewarm Caches
- Death Ray of Doom
- Dirty Deeds, Done Dirt Cheap
- Deep Healthchecking
Load Balancing; Common Software

- **AWS ELB (L3)**
  - Dumb packet switcher, HTTP1.x only

- **Front-End Proxies**
  - Nginx
  - Apache etc.

- **Full L4 balancers:**
  - Good for routing URLs around
  - Maybe some protocol-specific magic
## Load Balancing; Layer-4 balancers

<table>
<thead>
<tr>
<th>AWS ALB (L4)</th>
<th>IPVS</th>
</tr>
</thead>
<tbody>
<tr>
<td>More even connection balancing than ELB</td>
<td>Linux Kernel-Space load balancing</td>
</tr>
<tr>
<td>Can route to ECS services as well as ip:ports</td>
<td>Simple, high-throughput forwarding</td>
</tr>
<tr>
<td>Very basic control over balancing choices</td>
<td>No SSL termination etc.</td>
</tr>
</tbody>
</table>

**HAProxy**
- Good variable/state exporting
- User Space & rock solid

**IPVS**
- Supports VS-DR (Direct Routing)
- Supports UDP & VRRP
Caches; Overview

- Trade-off a storage resource for cpu, network or memory saving
- Usually at every layer of the stack
  - Caches compound
- The choice of eviction algorithm dictates how they behave under-stress
  - First In, First Out
  - Last In, First Out
  - Least Recently Used
  - Time-Aware
  - Least-Frequent, Recently Used
Caches; Distributed Coherence
Caches; Distributed Coherence

When Yellow or Green will get back a different answer for the same value!

Critical if you are doing transactions where one item depends on the previous one!
Cache Snooping

A message queue that all caches read from, to get updated important values can be useful.

If it's not already cached, it might not be set from the queue, as Green's cache has done. Choose from:

- Write Invalidate
- Write Update (as seen here)

Scalability depends on frequency of writes. Partitioning is key.
A directory of cache leases is kept

Caches that want to write to the cache get a 'lease' on a subset of the dataset. Only they can write to the dataset.

Always 'Write-Invalidate'
Cache Capacity Planning

How do you choose a cache size?

- Single-level caches are easy
  - load test them, decide on cost of cache vs scaled service
- Multi-level caches are sums of multiple curves
  - each layer load-tested
- It's never acceptable to guess, unless the cache doesn't matter
- Test your cold-caches!
Design Review Time! (Optional)

1. Organise in Groups of 4

2. Make a Copy of the "Fast Recommendation Service" Design at https://goo.gl/z87ov4

3. Make notes/improvements to the Design

4. Argue!