Distributed systems
Lecture 15: Replication, quorums, consistency, CAP, and Amazon/Google case studies
Michaelmas 2019
Dr Martin Kleppmann
(With thanks to Dr Robert N. M. Watson and Dr Steven Hand)
Last time

• General issue of **consensus**:
  – How to get processes to agree on something
  – FLP says “impossible” in **asynchronous networks** with at least 1 failure ... but in practice we’re OK!
  – General idea useful for **leadership elections, distributed mutual exclusion**: relies on being able to detect failures

• **Distributed transactions**:
  – Need to commit a set of “sub-transactions” across multiple servers – want **all-or-nothing semantics**
  – Use **atomic commit** protocol like 2PC

• **Replication**:
  – Performance, load-balancing, and **fault tolerance**
  – Introduction to **consistency**
Replication and consistency

• More challenging if clients can perform updates
• For example, imagine \( x \) has value 3 (in all replicas)
  – \( C_1 \) requests \texttt{write}(x, 5) from \( S_4 \)
  – \( C_2 \) requests \texttt{read}(x) from \( S_3 \) (after \( C_1 \)’s request has completed)
  – What should occur?
• With strong consistency/linearizability, the system behaves as if there was no replication:
  – That is, \( C_2 \) should read the value 5
  – Requires coordination between all servers
• With weak consistency, \( C_2 \) may get 3 or 5 (or ...?)
  – Harder to reason about, but better performance
  – Recall close-to-open consistency in NFS
Achieving strong consistency

• Goal: impose **total order** on updates to some state \( x \)
  – Ensure update propagated to replicas **before** later reads

• Simple **lock-step** solution for replicated object:
  1. When \( S_i \) receives update for \( x \), locks \( x \) at all other replicas
  2. Make change to \( x \) on \( S_i \)
  3. Propagate \( S_i \)’s change to \( x \) to all other replicas
  4. Other servers send ACK to \( S_i \)
  5. After ACKs received, instruct replicas to unlock \( x \)
  6. Once \( C_j \) has ACK for its write to \( S_i \), any \( C_k \) will see update

• Need to handle failure (of replica, of network)
  – Add step to tentatively apply update, and only actually apply (“commit”) update if all replicas agree

• We’ve reinvented distributed transactions & 2PC!
Quorum systems

• Transactional consistency works, but:
  – High overhead, and
  – Poor availability during update (worse if crash!)

• An alternative is a quorum system:
  – Imagine there are $N$ replicas, a write quorum $Q_w$, and a read quorum $Q_r$
    – Constraint on writes: $Q_w > N/2$
    – Constraint on reads: $(Q_w + Q_r) > N$

• To perform a write, must update $Q_w$ replicas
  – Ensures a majority of replicas have new value

• To perform a read, must read $Q_r$ replicas
  – Ensures that we read at least one updated value
Example

- Seven replicas \((N=7)\), \(Q_w = 5\), \(Q_r = 3\)
- All objects have associated version \((T, S)\)
  - \(T\) is logical timestamp, initialized to zero
  - \(S\) is a server ID (used to break ties)
- Any write will update at least \(Q_w\) replicas
- Performing a read is easy:
  - Choose replicas to read from until get \(Q_r\) responses
  - Correct value is the one with highest version
Quorum systems: writes

• Performing a write is trickier:
  – What if write succeeds on some replicas, but not on the full write quorum?
  – Atomic commit across entire quorum?

• In fact, transactional consistency is a quorum protocol with $Q_w = N$ and $Q_r = 1$!
  – But when $Q_w < N$, additional complexity since must bring replicas up-to-date before updating

• Quorum systems are good when expect failures
  – Additional work on update, additional work on reads...
  – ... but increased availability during failure

• What about reads happening concurrently with writes?
  – May or may not see concurrently written value
Consistency, Availability & Partitions (CAP)

- **CAP Theorem** (Brewer 2000, Gilbert & Lynch 2002):
  - Must choose between strong **Consistency** and total **Availability** in the presence of a network **Partition**
  - “consistency” = linearizability (∼ always return latest value)
  - “availability” = any replica must be able to handle request
  - “partition” = two replicas cannot communicate
  - (Can have both Consistency and Availability while there is no network partition)

- **NB**: this doesn’t mean things are always inconsistent, just that they’re not **guaranteed** to be consistent

- Different parts of a system can make different choices
  - Many Internet-scale systems ditch consistency
  - Also affects performance, not just availability
Weak consistency

• Maintaining strong consistency has costs:
  – Need to coordinate updates to all (or $Q_w$) replicas
  – Slow... and will block other accesses for the duration

• **Weak consistency** systems provides fewer guarantees:
  – E.g. $C_1$ updates (replica of) object $Y$ at $S_3$
  – $S_3$ lazily propagates changes to other replicas

• We can do this by reducing quorum parameters
  – $Q_r$: Clients can potentially read **stale** value from other $S_x$
  – $Q_w$: Writes may **conflict**: >1 $Y$ values w/same timestamp

• Considerably more **efficient** and more **available**:
  – Less waiting for replicas on read and write...
  – ... hence is also more available (i.e. fault tolerant)

• But it can be harder to reason about possible outcomes
Eventual consistency

• All replicas **eventually** converge to the same value
• Servers $S_i$ keep a **version vector** $V_i(O)$ for each object $O$
  – For each update of $O$ on $S_i$, increment $V_i(O)[i]$
  – (essentially a **vector clock** as a per-object version number)
• Servers synchronize pair-wise from time to time
  – For each object $O$, compare $V_i(O)$ to $V_j(O)$
  – If $V_i(O) < V_j(O)$, $S_i$ gets an up-to-date copy from $S_j$
  – If $V_j(O) < V_i(O)$, $S_j$ gets an up-to-date copy from $S_i$
• But if $V_i(O) \parallel V_j(O)$ we have a **write conflict**:
  – Concurrent updates have occurred at 2 or more servers
  – Must apply some kind of reconciliation method
  – (similar to revision control systems, and equally painful)
• Coda filesystem (next lecture) uses this approach
Amazon’s Dynamo [2007]

- Storage service used within Amazon’s web services
- Designed to prioritize availability above consistency:
  - SLA to give bounded response time 99.99% of the time
  - If customer wants to add something to shopping basket and there’s a failure... still want addition to ‘work’
  - Even if get (temporarily) inconsistent view... fix later!
- To maximise availability, uses so-called sloppy quorum:
  - Have $N, Q_w, Q_r$ as we saw earlier... but don’t actually require that $Q_w > N/2$, or that $(Q_w + Q_r) > N$
  - Instead make tunable: lower $Q$ values = higher availability; and higher read (or write) throughput
  - Also let system continue during failure
  - Application must handle (reconcile) inconsistency
Session guarantees

• Eventual consistency seems great, but how can you program to it?
  – Need to know something about guarantees to the client
• These are called session guarantees:
  – Not system wide, just for one (identified) client
  – Client must be a more active participant
  – E.g. client maintains version vectors of objects it reads/writes
• Example: Read Your Writes (RYW):
  – If $C_i$ writes a new value to $x$, a later read of $x$ should see the update ... even if $C_i$ is now reading from another replica
  – Need $C_i$ to remember highest ID of any update it made
  – Only read from a server if it has seen that update
• E.g., Webmail: Exchange stale message read/delete flags between sessions for greater scalability
A Google datacentre

- **MapReduce**
  - Scalable distributed computation model
- **BigTable**
  - Distributed storage with weak consistency
- **Spanner**
  - Distributed storage with strong consistency
- Many spiffy distributed systems at Google
  - E.g., **Dapper**: trace RPCs and distributed events
Google: architecture overview

- Parallel data processing: **MapReduce**
- Structured storage: **BigTable**
- Fast data analytics: **Dremel**
- Web serving: **GWS**
- Cross-datacenter RDBMS: **Spanner**
- Distributed locking: **Chubby**
- Cluster management and scheduling: **Borg / Omega**
- Distributed storage: **Colossus**
- RPCs
Google’s MapReduce [2004]

• **Specialized** programming framework for scale
  – Run a program on 100s to 10,000s machines

• Framework takes care of:
  – Parallelization, distribution, load-balancing, scaling up (or down) & fault-tolerance
  – **Locality**: compute close to (distributed) data

• Programmer implements two methods
  – `map(key, value) → list of <key’, value’> pairs`
  – `reduce(key’, value’) → result`
  – Inspired by functional programming
  – Reduce data movement by computing close to data source
  – E.g., for every word, count documents using word(s):
    – Extract words from local documents in `map()` phase
    – Aggregate and generate sums in `reduce()` phase
MapReduce: for each key, sum values

Map

Shuffle

Reduce

Output

Perform Map() query against local data matching input spec; write new keys/values (e.g., 5 instances of X found here)

Aggregate gathered results for each intermediate key using Reduce() (e.g., $X_{\text{sum}} = \sum(X_i)$)

End user can query results via distributed key/value store

Results: $X_{\text{sum}}: 8, Y_{\text{sum}}: 9$
MapReduce example programs

• **Sorting** data is trivial (map, reduce both identity function)
  – Works since the shuffle step essentially sorts data

• **Distributed grep** (search for words)
  – **map**: emit a line if it matches a given pattern
  – **reduce**: just copy the intermediate data to the output

• **Count URL access frequency**
  – **map**: process logs of web page access; output <URL, n>
  – **reduce**: add all values for the same URL

• **Reverse web-link graph**
  – **map**: output <target, source> for each link to target in a page
  – **reduce**: concatenate the list of all source URLs associated with a target. Output <target, list(source)>
MapReduce: pros and cons

• **Extremely simple**, and:
  – Can **auto-parallelize** (since operations on every element in input are independent)
  – Can **auto-distribute** (since rely on underlying Colossus/BigTable distributed storage)
  – Gets **fault-tolerance** (since tasks are idempotent, i.e. can just re-execute if a machine crashes)

• Doesn’t really use **any** of the sophisticated algorithms we’ve seen (except storage replication)
  – Limited to batch jobs and computations that are expressible as a `map()` followed by a `reduce()`
Google’s BigTable [2006]

• “Three-dimensional” structured key-value store:
  – `<row key, column key, timestamp> → value`

• Effectively a distributed, sorted, sparse map
  • Versioned web contents by URL, user activity history, web logs, ..
Google’s BigTable [2006]

- Distributed **tablets** (~1 GB max) hold subsets of **map**
  - Adjacent rows have user-specifiable locality
  - E.g., store pages for a particular website in the same tablet
- On top of **Colossus**, which handles replication and fault tolerance: *only one (active) server per tablet!*
- Reads & writes within a row are **transactional**
  - Independently of the number of columns touched
  - **But:** no cross-row transactions possible
- META0 tablet is “root” for name resolution
  - Filesystem meta stored in BigTable itself
- Use **Chubby** to elect master (META0 tablet server), and to maintain list of tablet servers & schemas
  - 5-way replicated **Paxos consensus** on data in **Chubby**
Google’s Spanner [2012]

- **BigTable** insufficient for some consistency needs
- Often have transactions across >1 datacenters
  - May buy app on Play Store while travelling in the U.S.
  - Hit U.S. server, but customer billing data is in U.K.
- **Spanner** offers **transactional consistency**: full RDBMS power, ACID properties, at global scale!
- Wide-area consistency is hard
  - Due to long delays and clock skew
- Secret sauce: **hardware-assisted clock sync**
  - Using GPS and atomic clocks in datacenters
  - Use global timestamps and **Paxos** to reach consensus
  - Still have a period of uncertainty for write TX: **wait it out!**
Comparison

<table>
<thead>
<tr>
<th>Consistency</th>
<th>Dynamo</th>
<th>BigTable</th>
<th>Spanner</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>eventual</td>
<td>weak(ish)</td>
<td>strong</td>
</tr>
<tr>
<td>Availability</td>
<td>high throughput, low latency</td>
<td>high throughput, low latency</td>
<td>low throughput, high latency</td>
</tr>
<tr>
<td>Expressivity</td>
<td>simple key-value</td>
<td>row transactions</td>
<td>full transactions</td>
</tr>
</tbody>
</table>
Summary + next time

- Strong, weak, and eventual consistency
- Quorum replication
- Session guarantees
- CAP theorem
- Amazon/Google case studies

- Distributed-system security
  - Access control, capabilities, RBAC, single-system sign on
- Distributed storage system case studies
  - NASD, AFS3, and Coda