Distributed systems
Lecture 15: Replication, quorums, consistency, CAP, and Amazon/Google case studies

Dr Robert N. M. Watson

Last time

- General issue of consensus:
  - How to get processes to agree on something
  - FLP says “impossible” in asynchronous networks with at least 1 (more) 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 `write(x, 5)` from \( S_4 \)
  – \( C_2 \) requests `read(x)` from \( S_3 \)
  – What should occur?
• With strong consistency, the distributed system behaves as if there is no replication present:
  – i.e. in above, \( C_2 \) should get the value 5
  – requires coordination between all servers
• With weak consistency, \( C_2 \) may get 3 or 5 (or ...?)
  – Less satisfactory, but much easier to implement
  – 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, or 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:
  - Must ensure get entire quorum, or cannot update
  - Hence need a commit protocol (as before)
- 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
- How might client-server traffic scale with \( Q_w/Q_r \)?

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
FIFO consistency

- As with group communication primitives, various ordering guarantees possible
- **FIFO consistency**: all updates originating at $S_i$ (on behalf of a client) occur in the same order at all replicas
  - As with FIFO multicast, can buffer for as long as we like!
  - But says nothing about how $S_i$’s updates are interleaved with $S_j$’s at another replica (may put $S_j$ first, or $S_i$, or mix)
- Still useful in some circumstances
  - E.g. single user accessing different replicas at disjoint times
  - I.e., client will see its writes **serialized**
  - Essentially primary replication with primary = last accessed
- E.g., sufficient for multiple mail clients interacting with the same mailbox independently (phone, tablet)

Eventual consistency

- FIFO consistency doesn’t provide very nice semantics:
  - E.g. $C_1$ writes $V_1$ of file $f$ to $S_1$
  - Later $C_1$ reads $f$ from $S_2$, and writes $V_2$
  - Much later, $C_1$ reads $f$ from $S_3$ and gets $V_1$ – changes lost!
- What happened?
  - $V_1$ arrived at $S_3$ after $V_2$, thus overwrote it (stooopid $S_3$)
- A desirable property in weakly consistent systems is that they **converge to a more correct state**
  - I.e. in the absence of further updates, every replica will eventually end up with the same latest version
- This is called **eventual consistency**
Implementing eventual consistency

• 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 $S_i$.
• If $V_i(O) \sim V_j(O)$ 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!
• Built around notion of a 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 Cᵢ writes a new value to x, a later read of x should see the update ... even if Cᵢ is now reading from another replica
  – Need Cᵢ 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

Session guarantees + availability

• There are many variations on session guarantees
  – All deal with allowable state on replica given history of accesses by a specific client
• Session guarantees are weaker than strong consistency, but stronger than ‘pure’ weak consistency:
  – But this means that they sacrifice availability
  – I.e. choosing not to allow a read or write if it would break a session guarantee means not allowing that operation!
  – ‘Pure’ weak consistency would allow the operation
• Can we get the best of both worlds?
Consistency, Availability & Partitions (CAP)

• Short answer: No ;-)  

• The **CAP Theorem** (Brewer 2000, Gilbert & Lynch 2002) says you can only guarantee two of:  
  – **Consistent data, Availability, Partition-tolerance**

• ... in a single system.

• In local-area systems, can sometimes drop partition-tolerance by using redundant networks

• In the wide-area, this is not an option:  
  – **Must choose between consistency & availability**  
  – Most Internet-scale systems ditch consistency

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

---

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’s MapReduce [2004]

- Specialized programming framework for scale
  - Run a program on 100’s to 10,000’s 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

Input: X: 5, X: 3, Y: 2, Y: 7

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

Shuffle

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

Output: Results: $X_{\text{sum}}: \text{8}, Y_{\text{sum}}: \text{9}$

End user can query results via distributed key/value store

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, 1>
  - **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, …

- `<larry.page", "websearch", 133746428"> → "cat pictures"
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></th>
<th>Dynamo</th>
<th>BigTable</th>
<th>Spanner</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Consistency</strong></td>
<td>eventual</td>
<td>weak(ish)</td>
<td>strong</td>
</tr>
<tr>
<td></td>
<td>high throughput, low latency</td>
<td>low throughput, high latency</td>
<td></td>
</tr>
<tr>
<td><strong>Availability</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Expressivity</strong></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