SE252:Lecture 25, Apr 16

IL05: Performance & Consistency *(BASE & Eventual Consistency)*

Yogesh Simmhan

Today's Lecture based on © Ken Birman's CS5412 Spring 2012 (Cloud Computing), Lecture 10 & 8: Logical Clocks & ACID vs BASE

Deadlines

- 23/24 Apr, Thu/Fri: Final Project Report/Demo (20%)
  - See slide from Lectures 21+22.
- 23 Apr, Thu: Research Summary (10%)
  - See next page slide
- 25 Apr, Sat: Homework C (10%)
  - Posted online today
- 27 Apr, Mon 2-5PM: Final Exam (15%)
  - Full syllabus
  - Review lectures, homework, text book & citations to external sources.

Research Review Comments

- Report should be self-contained
  - Do not use phrases, incomplete sentences
  - Start a section with a summary of the section's goals.
  - Avoid "numbered lists". Instead have *self contained* paragraphs
  - Offer your critique and "cite" the section number
  - Use IEEE/ACM format, Include references

Recall that clouds have tiers

- Focus has been on client systems and the network, and the way that the cloud has reshaped both
- Looked superficially at the tiered structure of clouds
  - **Tier 1**: Very lightweight, responsive "web page builders" that can also route (or handle) "web services" method invocations. Limited to "soft state".
  - **Tier 2**: (key,value) stores and similar services that support tier 1. Basically, various forms of caches.
  - **Inner tiers**: Online services that handle requests not handled in the first tier. These can store persistent files, run transactional services. But we shield them from load.
  - **Back end**: Runs offline services that do things like indexing the web overnight for use by tomorrow morning's tier-1 services.

Replication

- A central feature of the cloud
- To handle more work, make more copies
  - In the first tier, which is highly elastic, data center management layer pre-positions inactive copies of virtual machines for the services we might run
    - » Exactly like installing a program on some machine
  - If load surges, creating more instances just entails
    - » Running more copies on more nodes
    - » Adjusting the load-balancer to spray requests to new nodes
  - If load drops… just kill the unwanted copies!
    - » Little or no warning. Discard any "state" they created locally.

Replication is about keeping copies

- The term may sound fancier but the meaning isn't

- Whenever we have many copies of something we say that we've replicated that thing
  - But usually replica does connote "identical"
  - Instead of *replication* we use the term *redundancy* for things like alternative communication paths *(e.g. if we have two distinct TCP connections from some client system to the cloud)*
  - **Redundant** things might not be identical. Replicated things usually play identical roles and have equivalent data.

Things we can replicate in a cloud

- **Files** or other forms of data used to handle requests
  - If all our first tier systems replicate the data needed for end-user requests, then they can handle all the work!
  - Two cases to consider
    - » In one the data itself is "**write once**" like a photo. Either you have a replica, or don't
    - » In the other the **data evolves over time**, like the current inventory count for the latest iPad in the Apple store
- **Computation**
  - Here we replicate some *request* and then the work of computing the answer can be spread over multiple programs in the cloud
  - We benefit from parallelism by getting a faster answer
  - Can also provide fault-tolerance

Many things "map" to replication

- As we just saw, data (or databases), computation
- Fault-tolerant request processing
- Coordination and synchronization *(e.g. "who's in charge of the air traffic control sector over Paris?")*
- Parameters and configuration data
- Security keys and lists of possible users and the rules for who is permitted to do what
- Membership information in a DHT or some other service that has many participants

So… focus on replication!

- If we can get replication right, we'll be on the road to a highly assured cloud infrastructure
- Key is to understand what it means to correctly replicate data at cloud scale…
- … then once we know what we *want* to do, to find scalable ways to implement needed abstraction(s)

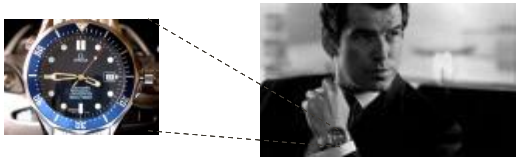
Concept of "consistency"

- *We would say that a replicated entity behave in a consistent manner if mimics the behavior of a non-replicated entity*
  - E.g. if I ask it some question, and it answers, and then you ask it that question, your answer is either the same or reflects some update to the underlying state
  - Many copies but act like just one

- An inconsistent service is one that seems "broken"

Consistency lets us ignore implementation

A <u>consistent</u> distributed system will often have many components, but users observe behavior indistinguishable from that of a single-component reference system



Reference Model

Implementation

Dangers of Inconsistency

My rent check bounced?
That can't be right!

- Inconsistency causes bugs
  - Clients would never be able to trust servers… a free-for-all

- Weak or "best effort" consistency?
  - Common in today's cloud replication schemes
  - But strong security guarantees demand consistency
  - *Would you trust a medical electronic-health records system or a bank that used "weak consistency" for better scalability?*

ACID & BASE

ACID

- A model for correct behaviour of databases
  - *Name was coined (no surprise) in California in 60's*
- **Atomicity**: even if "transactions" have multiple operations, does them to completion (commit) or rolls back so that they leave no effect (abort)
- **Consistency**: A transaction that runs on a correct database leaves it in a correct ("consistent") state
- **Isolation**: It looks as if each transaction ran all by itself. Basically says "we'll hide any concurrency"
- **Durability**: Once a transaction commits, updates can't be lost or rolled back

ACID eases development

- No need to worry about a transaction leaving some sort of partial state
  - For example, showing Tony as retired and yet leaving some customer accounts with him as the account rep
- Transaction can't glimpse a partially completed state of some concurrent transaction
  - Eliminates worry about transient database inconsistency that might cause a transaction to crash
- Serial & Serializable Execution
  - Offers concurrency while hiding side-effects
- But costs are not small
  - $O(n^2)..O(n^5)$ for replicated ACID, *n* is replica set size

Jim Gray, Pat Helland, Patrick E. O'Neil, Dennis Shasha: The Dangers of Replication and a Solution. SIGMOD 1996: 173-182

BASE

- Basically Available Soft-State Services with Eventual Consistency
  - Methodology for transforming transactional application into more concurrent & less rigid
  - Guide programmers to a cloud solution that performs much better
- Doesn't guarantee ACID properties
  - Uses the CAP Theorem

BASE: An ACID Alternative, DAN PRITCHETT, May/June 2008 ACM QUEUE

BASE

- **Basically Available**
  - *Goal is to promote rapid responses.*
  - Partitioning faults are rare in data centers
    - » Crashes force isolated machines to reboot
  - Need rapid responses even when some replicas on critical path can't be contacted
    - » Fast response even if some replicas are slow or crashed

BASE

- **Soft State Service**
  - Runs in first tier. *Can't store permanent data.*
  - Restarts in a "clean" state after a crash
  - To remember data:
    - » Replicate it in memory in enough copies to never lose all in any crash
    - » Pass it to some other service that keeps "hard state"

BASE

- **Eventual Consistency**
  - OK to send “optimistic” answers to external client
    - » Send reply to user before finishing the operation
  - Can use cached data (without staleness check)
  - Can guess the outcome of an update
  - Can skip locks, hoping no conflicts happen
  - *Later, if needed, correct any inconsistencies in an offline cleanup activity*

- Developer ends up thinking hard and working hard!

Amazon's Dynamo DB

- Key-Value Store
  - Simple Get() & Put() operations on objects with unique ID. No queries.
- Highly Available
  - Even the slightest outage has significant financial consequences
- Service Level Agreements
  - Guaranteeing response in 300ms for 99.9% of requests at a peak load of 500 req/sec

Dynamo: Amazon's Highly Available Key-value Store, Giuseppe DeCandia, et al, SOSP, 2007

Design Choices

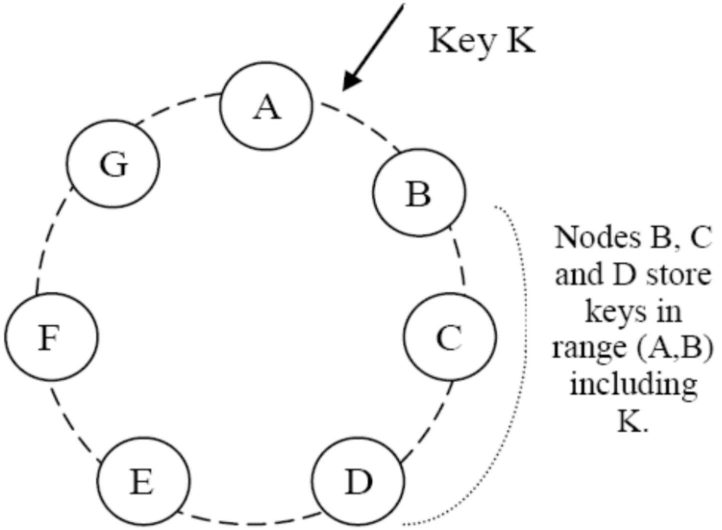
- Sacrifice strong consistency for availability
  - "always writeable". No updates are rejected.
  - Conflict resolution is executed during ***read*** instead of ***write***, i.e. "always writeable".
- Incremental scalability & decentralization
  - Symmetry of responsibility
  - Heterogeneity in capacity
- All nodes are trusted

Techniques

Problem	Technique	Advantage
Partitioning	Consistent Hashing	Incremental Scalability
High Availability for writes	Vector clocks with reconciliation during reads	Version size is decoupled from update rates.
Handling temporary failures	Sloppy Quorum and hinted handoff	Provides high availability and durability guarantee when some of the replicas are not available.
Recovering from permanent failures	Anti-entropy using Merkle trees	Synchronizes divergent replicas in the background.
Membership and failure detection	Gossip-based membership protocol and failure detection.	Preserves symmetry and avoids having a centralized registry for storing membership and node liveness information.

Partitioning

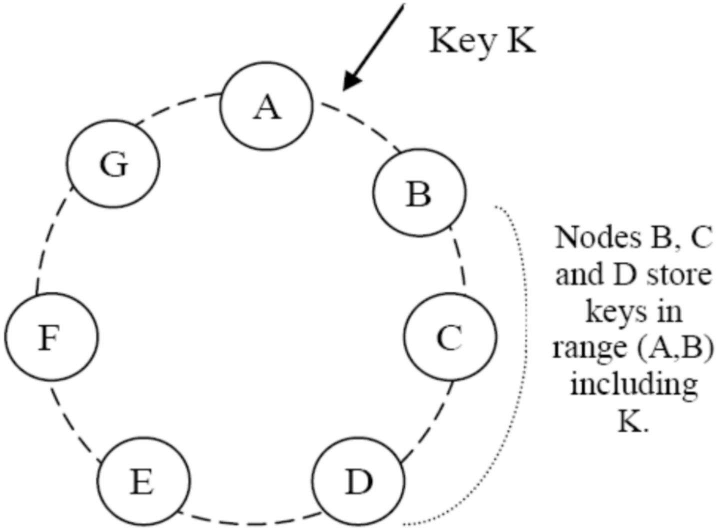
- Consistent hashing
  - Output range of hash func. on key is a fixed "ring"
  - Hash value corresponds to a virtual node
- Each physical node responsible for multiple virtual nodes



- Adapt to capacity of physical nodes
- Incrementally add/remove nodes
  - Node unavailable: Load balance on available ones
  - Node joins: Accepts range of virtual nodes from existing ones

Replication

- Each data item is replicated at N hosts.
  - "*preference list*": The list of nodes responsible for storing a particular key.
  - Skip virtual nodes present on same physical node
- Gossip protocol
  - Propagates changes among nodes



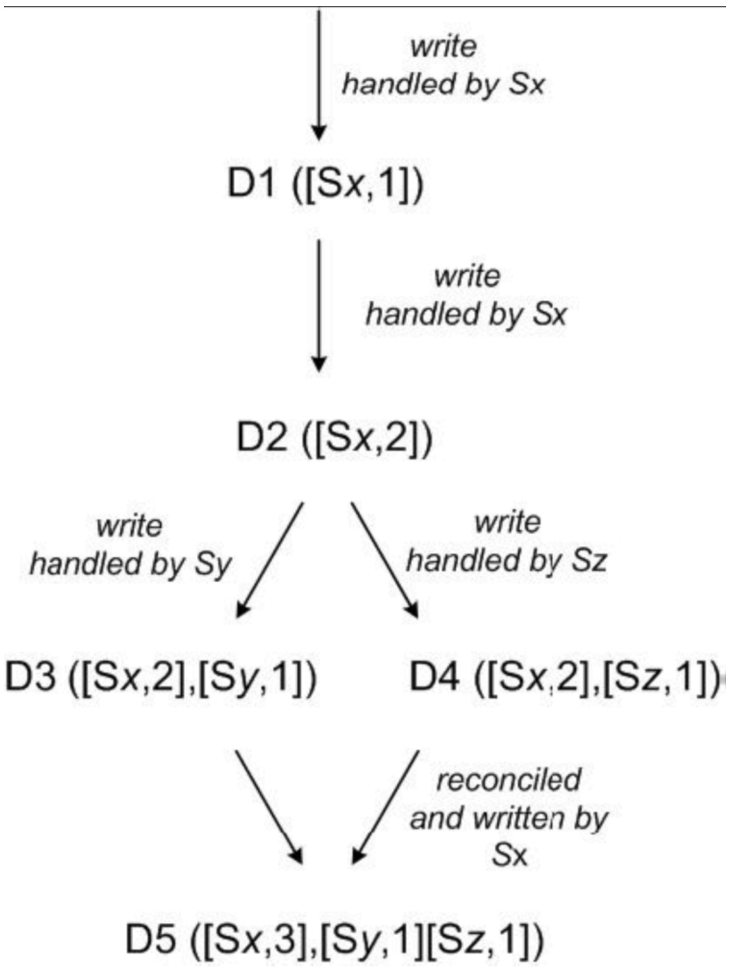
D stores (A, B], (B, C], (C, D]

Data Versioning & Consistency

- **Put()** may return to its client before the update is applied at all replicas
- **Get()** may return many versions of same object
- Challenge
  - *Distinct version sub-histories need to be reconciled.*
- Solution
  - *Uses vector clocks to capture causality between different versions of the same object.*

Consistency using Logical (Vector) Clocks

- Vector clock: List of (node, counter) pairs.
  - Every version of every object is associated with one vector clock.
- If the counters on the first object's clock are less-than-or-equal to all of the nodes in the second clock, then the first is an ancestor of the second and can be forgotten.
  - *i.e. first object happened before second object*



Consistency & Quorum

- Writes are successful if ‘w’ replicas can be updated (w<N)
- Reads return ‘r’ replica values (r<N)
- Reads & writes dictated by slowest replica
  - Set **r+w > N**
- If get() has multiple replica versions, return causally “unrelated” versions
  - *i.e. remove partial ordered & only return causally unordered versions for reconciliation*
- Client writes the reconciled version back

Vogels: World-Wide Failure Sensing

- Vogels wrote a paper in which he argued that we really could do much better
  - In a cloud computing setting, the cloud management system often "forces" slow nodes to crash and restart
    - » Used as a kind of all-around fixer-upper
    - » Also helpful for elasticity and automated management

  - So in the cloud, management layer is a fairly trustworthy partner, if we were to make use of it
    - » We don't make use of it, however, today

The Postman Always Rings Twice

- Suppose the mailman wants a signature
  - He rings and waits a few seconds
  - Nobody comes to the door… should he assume you've died?
- Hopefully not
- Vogels suggests that there are many reasons a machine might timeout and yet not be faulty

Causes of delay in the cloud

- Scheduling can be sluggish
- A node might get a burst of messages that overflow its input sockets and triggers message loss, or network could have some kind of malfunction in its routers/links
- A machine might become overloaded and slow because too many virtual machines were mapped on it
- An application might run wild and page heavily

Vogels suggests?

- He recommended that we add some kind of failure monitoring service as a standard network component
- Instead of relying on timeout, even protocols like remote procedure call (RPC) and TCP would ask the service and it would tell them
- It could do a bit of sleuthing first… e.g. ask the O/S on that machine for information… check the network…

Why clouds *don't* do this

- In the cloud our focus tends to be on keeping the "majority" of the system running
  - No matter what the excuse it might have, if some node is slow it makes more sense to move on
  - Keeping the cloud up, as a whole, is way more valuable than waiting for some slow node to catch up
  - End-user experience is what counts!

- So the cloud is casual about killing things
- ... and avoids services like "failure sensing" since they could become bottlenecks

Also, most software is buggy!

- A mix of "Bohrbugs" and "Heisenbugs"
  - Bohrbugs: Boring and easy to fix. Like Bohr model of the atom
  - Heisenbugs: They seem to hide when you try to pin them down (caused by concurrency and problems that corrupt a data structure that won't be visited for a while). Hard to fix because crash seems unrelated to bug
- Studies show that pretty much all programs retain bugs over their full lifetime.
  - So if something is acting strange, it may be failing!

Worst of all... timing is flakey

- At cloud scale, with millions of nodes, we can trust timers at all
- Too many things can cause problems that manifest as timing faults or timeouts
- Again, there are some famous models... and again, none is ideal for describing real clouds

Things we just can't do

- We can't detect failures in a trustworthy, consistent manner
- We can't reach a state of "common knowledge" concerning something not agreed upon in the first place
- We can't guarantee agreement on things (election of a leader, update to a replicated variable) in a way certain to tolerate failures

ILO 5: Performance & Consistency on Clouds

- *Describe* and *compare* different performance metrics for evaluating Cloud applications and
- *demonstrate* their use for application measurement.
- *Explain* the distinctions between Consistency, Availability and Partitioning (CAP theorem), and
- *discuss* the types of Cloud applications that exhibit these features.