CS 555: Distributed Systems
[Replication & Consistency]

Shrideep Pallickara
Computer Science
Colorado State University

Frequently asked questions from the previous class survey

- Is there a preferred replication level for production systems?

Topics covered in this lecture

- Eventual Consistency
- Brewer's CAP theorem and implications for systems design
- Amazon Dynamo

EVENTUALLY CONSISTENT

Werner Vogels: Eventually Consistent. *ACM Queue* 6(6): 14-19 (2008)

Amazon systems use replication techniques ubiquitously

- Predictable performance
- Availability

Replication helps with these goals, but …

- **Not necessarily transparent**
- Under a number of **conditions** *consequences* of using replication techniques come to the fore
  - Network partitions
  - Node failures

Ideal world

- One consistency model
- When an update is made all observers see that update

Distribution transparency

- To the user of the system it *appears* as if there is only one system
  - Instead of a number of collaborating systems
- Approach taken in such systems?
  - Better to fail the complete system rather than break this transparency

In the mid-90s these practices were revisited

- Larger internet systems
- For the first time, **availability** was being considered the most important property

BREWER'S CAP CONJECTURE (AND LATER ON … THEOREM)

Brewer's CAP Theorem

- By Eric Brewer in 2000
- Three properties of shared-data systems
  1. Data **consistency**
  2. System **availability**
  3. Tolerance to network **partitions**
- Of these three only two can be achieved at a given time

Brewer's CAP: Consequences

- In large-scale distributed systems, network *partitions are common*
- So, consistency and availability cannot be achieved at the same time

What is the trade-off? [1/2]

- If your application **requires** consistency?
  - And some replicas are disconnected from the other replicas due to a network problem …
  - Then some replicas cannot process requests while they are disconnected:
    - They must either **wait** until the network problem is fixed, **or return an error**
    - Either way, they become **unavailable**

What is the trade-off? [2/2]

- If your application <u>does not</u> require consistency?
  - Then each replica can process requests independently
    - Even if it is disconnected from other replicas
  - The application can remain available in the face of a network problem, but its behavior is not consistent
- Thus, applications that don't require consistency can be **more tolerant of network problems**

Characterizing CAP correctly [1/3]

- CAP is sometimes presented as Consistency, Availability, Partition tolerance: pick 2 out of 3
  - Unfortunately, putting it this way is **misleading**
- Because network partitions are a kind of fault, they aren't something about which you have a choice:
  - They will happen whether you like it or not

Characterizing CAP correctly [2/3]

- At times when the network (and system) is working correctly, a system can provide both consistency and total availability
- When a network fault occurs, you have to choose between consistency or total availability

Characterizing CAP correctly [3/3]

- A better way of phrasing CAP would be
  - Either Consistent or Available when Partitioned
- A more reliable network needs to make this choice less often, but at some point the choice is inevitable!

CAP: Two choices on what to drop

- Relax consistency
  - To allow system to be **available under partitionable conditions**
- Make consistency a priority
  - And the system will be **unavailable under certain conditions**

The choices requires the developer to be aware of what is being offered by system

- If consistency is emphasized?
  - Developer must account for system unavailability
  - If a write fails?
    - Plan on *what will be done* with the data that must be written
- If availability is emphasized?
  - System may always accept writes but …
    - Under certain conditions a read will not reflect the results of a *recently completed* write

The C in ACID is a different kind of consistency {Atomicity, Consistency, Isolation and Durability}

- When a transaction is finished, the database is in a consistent state
- For e.g., when money is transferred between two accounts?
  - The total money in the two accounts should not change
- This kind of consistency is the **responsibility of the developer** writing the transaction
  - Database assists via managing integrity constraints

The "I" in ACID

- **Isolation**
- Ensures concurrent execution of transactions results in a final system state similar to what would be achieved if transactions were executed serially

Consistency: Two ways to look at this

- Client-side
  - How do clients observe updates?
- Server-side
  - How do updates flow through the system?
  - What guarantees can systems give with respect to updates?

CLIENT-SIDE CONSISTENCY

Client-side consistency [1/2]

- Consider a storage system
- Process **A** that writes and reads from the storage system
- Process **B** and **C** are independent of **A**
  - Write and read from the storage system too

Client-side consistency [2/2]

- How and when do observers (**A**, **B**, and **C**) see updates made to a data object?
- **Strong consistency**:
  - After update completes, any subsequent access by (**A**, **B**, or **C**) will return updated value
- **Weak consistency**:
  - No guarantee that subsequent accesses will return updated value
  - Number of conditions to be met before value is returned

The inconsistency window

- **Period** between
  - The *update*
    and
  - When any observer will **always see** the updated value

Eventual consistency

- A form of **weak consistency**
- Storage system guarantees that if no new updates are made to the object?
  - **Eventually** all accesses will return last updated value
- If no failures occur, size of the inconsistency window is determined by:
  - Communication delays, system load, and number of replicas

Eventual consistency variations

- Causal consistency
- Read-your-writes consistency
- Session consistency
  - As long as session exists, system guarantees read-your-writes consistency
  - Guarantees *do not overlap* sessions
- Monotonic read consistency
- Monotonic write consistency

RDBMS implement replication in different modes

- **Synchronous**
  - Replica update is part of the transaction
- **Asynchronous**
  - Updates arrive at the backup in a delayed manner
    - **Log shipping**
  - If primary fails before the logs were shipped?
    - Reading from promoted backup will produce old, inconsistent values

Other RDBMS approaches to improve speed

- RDBMSs have also started to provide ability to read from backup
  - Classic case of eventual consistency
- Size of the inconsistency window in such a setting?
  - Periodicity of the log shipping

SERVER SIDE CONSISTENCY

Server-side consistency

- Based on how updates flow through the system
- **N**: Number of nodes that store replicas of data
- **W**: Number of replicas that need to acknowledge receipt of update before it completes
- **R**: Number of replicas that are contacted when data object is accessed through read operation

W+R > N?

- The write-set and read-set overlap
  - Possible to guarantee strong consistency
- Primary-backup RDBMS
  - With synchronous replication
    - N=2, W=2 and R =1
    - Client always reads a consistent answer
  - With asynchronous replication
    - N=2, W=1 and R=1
    - Consistency cannot be guaranteed

In distributed storage systems the number of replicas is higher than two

- Systems that focus on fault tolerance use **N**=3
  - With **W**=2 and **R**=2
- Systems that serve very high read loads
  - Replicate data beyond what is needed for fault tolerance
  - **N** can 10s to 100s of nodes
  - **R** will be set to 1
    - A single read will return the result
  - For consistency **W**=**N** for updates
    - Decreases the probability of write succeeding

For systems concerned about fault tolerance but not consistency

- **W**=1
  - Minimal durability
- Rely on lazy (epidemic) techniques to update other replicas

Configuring values of N, R and W

- Depends on the **common case**
- **Performance path** that needs to be optimized
- If **R**=1 and **N**=**W** ?
  - We optimize for the read case
- If **W**=1 and **R**=**N** ?
  - We optimize for a very fast write
  - Durability is not guaranteed
  - If **W** < (**N**+1)/2 there is a possibility of conflicting writes when the write-sets do not overlap

Weak/eventual consistency

- Also arises when **W**+ **R** <= **N**
  - Possibility that the read and write set will not overlap
- If it's deliberate and not based on failure cases?
  - Hardly makes sense to set **R** to anything but 1

Weak/eventual consistency: Two common cases where R=1

- Massive replication for read scaling
- When data access is more complicated
  - In simple <key, value> systems easy to compare versions to determine latest written value
  - When set of objects are returned, reasoning gets more complicated

When partitions occur

- Some nodes cannot reach a set of other nodes
- With a classic majority quorum approach
  - Partition that has **W** nodes of the replica set continues to take updates
  - The other partition becomes unavailable

For some applications unavailability of partitions is unacceptable

- Important that clients, that reach a partition, can progress
- Merge operation is executed when partition heals
- Amazon shopping-cart?
  - **Write-always** system
  - Customer can continue to put items in the cart even when original cart lives on other partitions

DYNAMO: AMAZON'S HIGHLY AVAILABLE KEY-VALUE STORE

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, Werner Vogels: *Dynamo: Amazon's Highly Available Key-value Store*. SOSP 2007: 205-220

Lesson learned at Amazon: Reliability & Scalability depends on

- Application *state* &
- How it is *manage*d

Amazon architecture

- Service oriented architecture (**SOA**)
  - Decentralized
  - Loosely-coupled
- Hundreds of services up and running
- Needs storage scheme that is **always** available
  - E.g. Shopping cart service
    - Must be able to read/write from its data store

Amazon's operational requirements

- Performance
- Scalable
- Reliability
  - **Financial** consequences
  - Impacts customer *trust*

Storage technologies at Amazon

- Simple Storage Service (S3)
- SimpleDB
  - Distributed database
  - Written in *Erlang*
- Dynamo

Dynamo: Highlights

- Completely decentralized system
- Provides a **key-value** store
- Underlying technology for several core services
- **Scales** to extreme peak loads
  - Holiday shopping period
  - No downtimes
  - 3 million checkouts per day

Data from 2007

But why not use an RDBMS?

- Would lead to **inefficiencies**
- Scaling and availability issues
  - ACID Guarantees
  - Difficult to scale-out
  - Difficult to have **smart partitioning**
- Most services only need *primary-key* access
  - No need for complex querying and management
- $$$$$?

The contents of this slide-set are based on the following references

- Werner Vogels: Eventually Consistent. ACM Queue 6(6): 14-19 (2008)
- Martin Kleppmann. Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems. 1st Edition. O'Reilly Media. 2017. [Chapter 9]
- Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, Werner Vogels: *Dynamo: Amazon's Highly Available Key-value Store*. SOSP 2007: 205-220