Lecture 7: Key-Value Stores (Part 2)

CSCI4180

Patrick P. C. Lee

Consistency is Hard to Achieve

Initially, x = 0; y = 0

Process A

```
PUT("x", 1)
PUT("y", 1)
print GET("x")
```

Process B

```
while (GET("y") != 1) {
    sleep()
}
print GET("x")
```

- What are the outputs of Process A and Process B?
- In a single machine, we can use **memory barrier** to solve the problem
- What about in a distributed system? Any difficulty to put a "memory barrier"?

Outline

- NoSQL Origin
  - CAP Algorithm
- Case study: Amazon Dynamo

CAP Theorem

- CAP Conjecture by Eric Brewer in 2000
  - **C: consistency**, by which a shared and replicated data item appears as a single, up-to-date copy
  - **A: availability**, by which updates will always be eventually executed
  - **P: partition tolerance,** by which partitioning of nodes can be tolerated
  - Any networked system providing shared data can provide only two of the three properties
- It is later proven to be correct by Gilbert and Lynch in 2002
  - In a network subject to communication failures, it is impossible to realize an atomic read/write shared memory that guarantees a response to every request [Gilbert and Lynch, 2002]

Maarten van Steen Andrew S. Tanenbaum, “Distributed Systems”, 3rd edition, 2017 (Chapter 8)

CAP Theorem

- **C & P**: e.g., distributed database systems
  - Consistency?
    - When a user writes data, the same piece of data is written to all storage servers
  - Partition tolerance?
    - All storage nodes are guaranteed to have the same piece of data, even though part of the system is disconnected
  - Availability?
    - Cannot hold! Why?

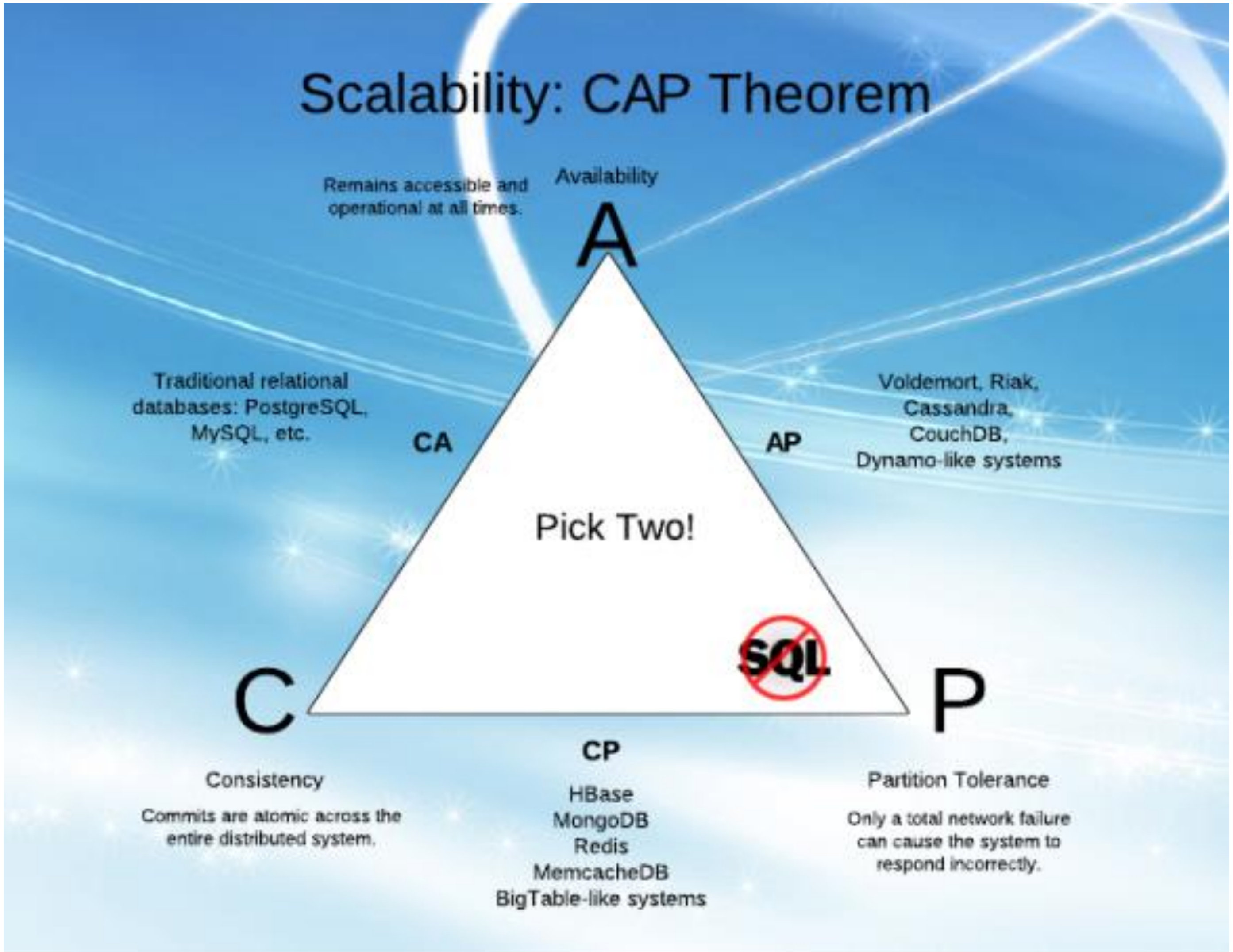
CAP Theorem

- **A & P**: e.g., DNS system
  - Availability?
    - YES! Every node in system will always be available no matter the <IP-name> pairs is valid or not
    - e.g., when using “**no-ip.org**”, it is always available, although the results may be invalid
    - Note that **invalid != inconsistent**!
  - Partition tolerance?
    - By design, every user (or machine) has its own DNS server on its own network, so network partitioning is not a problem.
  - Consistency?
    - Cannot hold. Why?

CAP Theorem

- **C & A**: e.g., single-node database
  - Consistency?
  - Availability?
  - Partition tolerance?
    - Cannot hold. Why?

CAP Theorem



https://ofirm.wordpress.com/2013/01/22/classical-big-data-reading-cap-theorem/#more-65

CAP in Reality

- “2 of 3” is misleading
  - It’s really between C and A only
- Reality:
  - Partitions are rare; no need to forfeit C or A
    - Partitions manifest themselves as high delays
  - Shouldn’t just give up either C or A for partition tolerance
  - Correct implementation: detect partitions, enter degraded mode to limit operations, initiate recovery when partitions are fixed
- Choice between C and A varies across operations and data → how to trade between C and A is critical

Brewer, “CAP Twelve Years Later: How the "Rules" Have Changed”, 2012

Outline

- NoSQL Origin
  - CAP Algorithm
- <u>Case study: Amazon Dynamo</u>

Amazon Dynamo

- Dynamo is designed as a **highly available, key-value storage** system
  - Used to power parts of Amazon storage services (e.g., S3)
- In an infrastructure with millions of components, something is always failing!
  - Failure is the normal case
- Good news: many services on Amazon's platform only need **primary-key** access
  - RDBMS is unnecessary

Assumptions and Requirements

- Simple read/write operations, uniquely identified by **keys**
  - Objects tend to be small (< 1MB)
- ACID gives poor availability
  - *Atomicity, Consistency, Isolation, Durability*
  - Use weaker consistency (C) for higher availability
- At least 99.9% of read/write operations must be performed within a few hundred milliseconds:
  - Avoid routing requests through multiple nodes

Assumptions and Requirements

- No security issues
  - Under single authority
- Applications can configure Dynamo for desired latency & throughput, balancing performance, cost, availability, durability guarantees
- Dynamo aims to be **always writable**
  - Rejecting updates is bad in customer experience (e.g., in shopping cart updates)
  - Push complexity of conflict resolution to reads

Design Principles

- Incremental scalability
  - System should be able to grow by adding a storage host (node) at a time
- Symmetry
  - Every node has the same set of responsibilities
- Decentralization
  - Favor decentralized techniques over central coordinators
- Heterogeneity
  - Workload partitioning should be proportional to capabilities of servers

Compared to BigTable

- Dynamo targets apps that only need key/value access with a primary focus on high availability
  - key-value store versus column-store (column families and columns within them)
  - Bigtable: distributed DB built on GFS
  - Dynamo: distributed hash table
  - Updates are not rejected even during network partitions or server failures

Consistency vs. Availability

- Strong consistency and high availability cannot be achieved simultaneously with high performance
- Replication of objects:
  - Good:
    - High availability
    - High read performance
  - Bad:
    - Need to update all of them to maintain consistency
- Dynamo enforces **eventual consistency**
  - Propagate updates to all replicas "eventually"
  - Trade weak consistency for availability and performance

Consistency and Availability

- In eventual consistency model, reads may return old data that is yet updated
- Who resolves conflicts?
  - Choices: the data store or the application
  - Data store
    - application-unaware, so choices limited simple policy, such as "last write wins”
  - Application
    - app is aware of the meaning of the data
    - can do application-aware conflict resolution
    - e.g., merge shopping cart versions to get a unified shopping cart.
    - fall back on "last write wins" if app doesn't want to bother

Dynamo's Design

Problem	Technique	Advantage
Partitioning	**Consistent hashing**	**Incremental scalability**
High availability for writes	Vector clocks	(Check the paper)
Handling temporary failures	**Sloppy quorum and hinted handoff**	**High availability and durability**
Recovering from permanent failures	Merkle trees	(Check the paper)
Membership and failure detection	Gossip-based protocol	(Check the paper)

- All techniques are known in literature
- We focus on **consistent hashing** and **sloppy quorum**

Dynamo's Storage

- Problem:
  - Key space of nodes = **{0, 1, …, n-1}**
  - Set of objects **$\{o_1, o_2, \ldots, o_k\}$**
  - Create a hash function **h(x)** such that
    - h(x) maps an element in $\{o_1, o_2, \ldots, o_k\}$ to number i
    - object x will be stored in node i
- Requirement of h(x):
  - Deterministic
  - Efficient
  - Evenly distributed among all nodes

Dynamo's Storage

- In Dynamo, a key is hashed with **MD5** to create a 128-bit identifier, combined with modulo operation

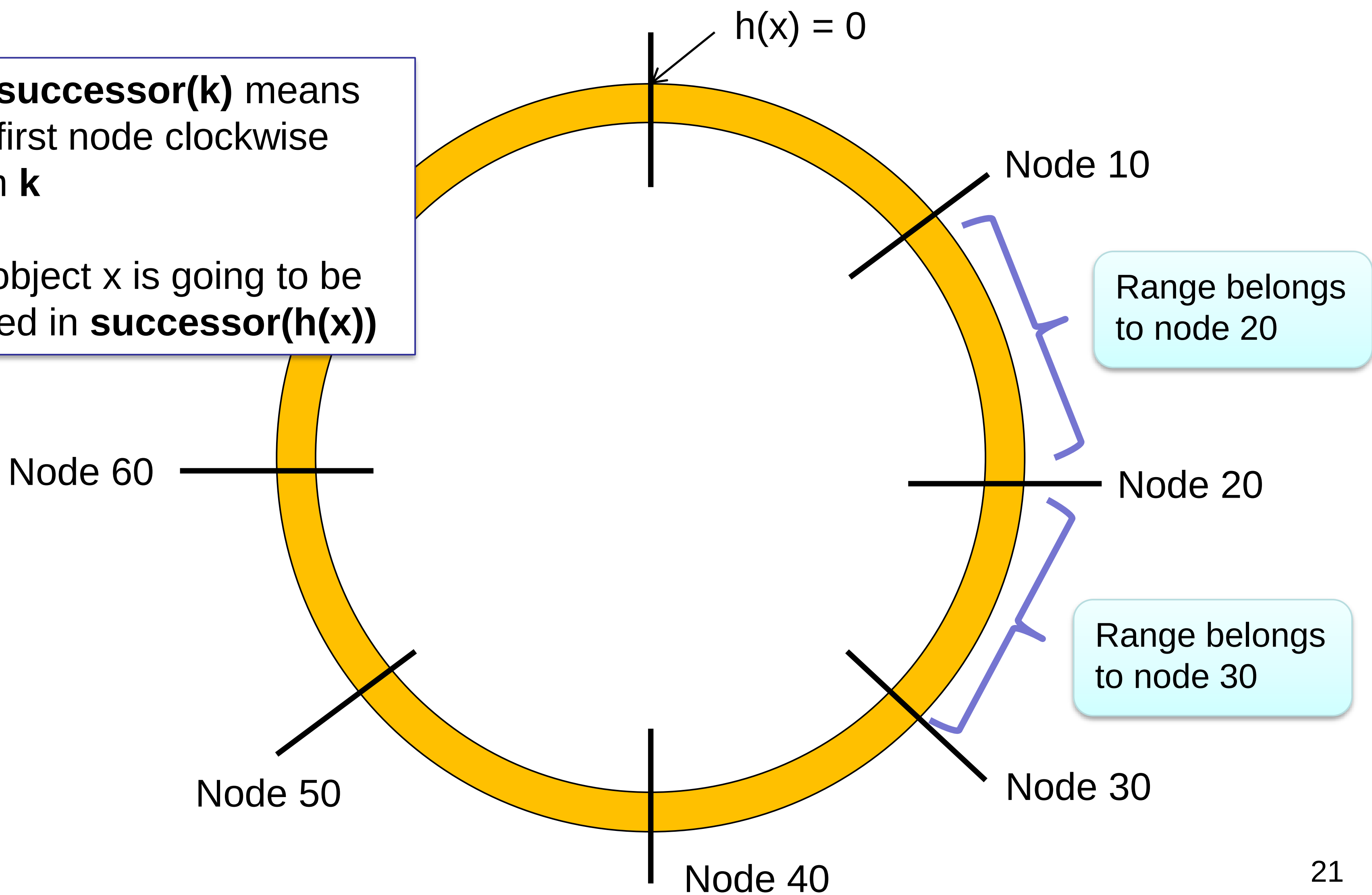
- **h(x) = MD5(x) mod n**

- Any pros and cons?

Consistent Hashing: Store Objects

Let **successor(k)** means the first node clockwise from **k**

An object x is going to be stored in **successor(h(x))**



Consistent Hashing: Node Join

- When a new server is added, only the data of the successor node is affected
- Example:
  - Node 26 is added
  - Data from node 30 is migrated to node 26
  - Update the routing tables (some routing protocol is needed)
- What about a node leave?

Stoica et al., “Chord: A scalable peer-to-peer lookup service for internet applications”, SIGCOMM’01

Consistent Hashing: Replication

- Replication: stores an object to N storage servers
  - 1st server is the one that is closest to h(x) in the clockwise direction, i.e., successor(h(x)).
  - Then, N-1 more servers, who are the clockwise successors of successor(h(x)), will store a redundant copy of object x.

- Example: h(x) = 15 and N = 3. Who will store the objects?

Consistency

- Replication → consistency
- Challenge:
  - To decide which storage server(s) to store a particular object, without choosing all servers.
  - Choosing all servers implies **strict consistency**!
- Strict consistency:
  - Any read on a data item x returns a value corresponding to the result of the most recent write on x

Client-Side Consistency

- **Strong consistency**
  - After the update reports a finished update, all subsequent access returns the updated value

- **Eventual consistency**
  - After the update reports a finished update, the system may report the old value
  - Nevertheless, the system will eventually report the updated value.

Server-Side Consistency

Configurable parameters

- **N** = the number of nodes that store replicas of the data
- **W** = the number of replicas that need to **acknowledge the receipt of update**
- **R** = the number of replicas that are **contacted when a data object is accessed**

Server-Side Consistency

- If $W + R > N$, then it is **strong consistency**

- If $N = 2$, $W = 2$, and $R = 1$, what will happen?

- If $N = 3$, $W = 2$, and $R = 2$, what may happen?
  - A write completes only when any two replica writes are confirmed
  - A read returns the values of any two replicas
  - Resolve conflict by data version

Versioning → Resolving Conflicts

- Not all updates may arrive at all replicas
- Application-based reconciliation
  - Each modification of data is treated as a new version
- Vector clocks are used for versioning
  - Capture causality between different versions of the same object
  - Vector clock is a set of (node, counter) pairs
  - Returned as a context from a get() operation

Server-Side Consistency

- If W + R <= N, then it is **eventual consistency**
- If N = 3, W = 2, and R = 1, what will happen?
  - A high reading-demand configuration
  - Strong consistency is sacrificed

Dynamo's Setting

- Common (N,R,W) in Dynamo: (3,2,2)
- However, strict quorum is vulnerable to partitions
- **Sloppy quorum**
  - All reads/writes are performed on the first N **healthy** nodes from the preference list, which may not always be the first N nodes in the consistent hashing ring
  - e.g., if node A is down, a replica is sent to node A'
- **Hinted handoff**
  - Node A' receives a hint that the update is temporary
  - A' sends back the replica to A if A is recovered, and removes its replica

Bonus

More on Eventual Consistency

- **Causal consistency**
  - If process A has communicated to process B that it has updated a data item, a subsequent access by process B will return the updated value, and a write is guaranteed to supersede the earlier write
- **Read-your-writes consistency**
  - Process A, after it has updated a data item, always accesses the updated value and will never see an older value
  - A special case of the causal consistency model

http://www.allthingsdistributed.com/2008/12/eventually_consistent.html

Bonus

More on Eventual Consistency

- **Session consistency**
  - A practical version of previous model, where read-your-write consistency is guaranteed in the same session
  - If a new session needs to be created, the guarantees do not overlap the sessions
- **Monotonic read consistency**
  - If a process has seen a particular value for the object, any subsequent accesses will never return any previous values
- **Monotonic write consistency**
  - Writes are serialized by the same process

http://www.allthingsdistributed.com/2008/12/eventually_consistent.html