Distributed Data Management

Christoph Lofi
Institut für Informationssysteme
Technische Universität Braunschweig
http://www.ifis.cs.tu-bs.de
Summary

• Durability in DHTs
  – Replication

• Load balancing
  – Power of two choices
  – Virtual Servers

• Special Purpose Databases
  – Extreme requirements lead to non-standard databases
Sommerfest!

SOMMERFEST
09. Juli
Wirtschaftsinformatik
Informatik
ITIS
IST

BIER!
COCKTAILMASCHINE!
WÜRSTCHEN!
Profs als DJs!

Donnerstag, 09. Juli 2015
Alan-Turing-Allee – ab 18 Uhr
Geschlossene Gesellschaft – Eingeladen sind Studierende, Mitarbeiter und Professoren der Informatik, Studierende, Mitarbeiter und Professoren der Wirtschaftsinformatik.
Veranstalter sind die entsprechenden Fachgruppen, Institute sowie Departements.
10.0 Towards the Cloud

10.1 Trade-Offs
   – CAP Theorem
   – BASE transactions

10.2 Showcase: Amazon Dynamo
10.1 Trade-Offs

- In the following, we will examine some **trade-offs** involved when designing high performance **distributed and replicated** databases
  - **CAP Theorem**
    - “You can’t have a highly available partition-tolerant and consistent system”
  - **BASE Transactions**
    - Weaker than ACID transaction model following from the CAP theorem
The CAP theorem was made popular by Eric Brewer at the ACM Symposium of Distributed Computing (PODC)

- Started as a conjecture, was later proven by Gilbert and Lynch

- CAP theorem limits the design space for highly-available distributed systems
10.1 CAP-Theorem

• Assumption:
  – High-performance distributed storage system with replicated data fragments

• **CAP:** Consistency, Availability, Partition Tolerance

• Consistency
  – Not to be confused with ACID consistency
    • CAP is not about transactions, but about the design space of highly available data storage
  – Consistent means that all replicas of a fragment are always equal
    • Thus, CAP consistency is similar to ACID atomicity: an update to the system atomically updates all replicas
  – At a given time, all nodes see the same data

Distributed Data Management – Christoph Lofi – IfIS – TU Braunschweig
10.1 CAP-Theorem

• **Availability**
  – The data service is *available and fully operational*
  – Any node failure will allow the survivors to continue operation without any restrictions
  – Common problem with availability:
    **Availability most often fails when you need it most**
    • i.e. failures during busy periods because the system is busy
• **Partition Tolerance**
  
  – No set of *network failures* less than total network crash is allowed to cause the system to respond incorrectly
  
  – **Partition**
    
    • Set of nodes which can communicate with each other
    • The whole node set should always be one big partition
  
  – However, often multiple *partitions* may form
    
    • Assumption: short-term network partitions form very frequently
    • Thus, not all nodes can communicate with each other
    • Partition tolerant system must either
      
      – prevent this case of ever happening
      – or tolerate forming and merging of partitions without producing failures
10.1 CAP-Theorem

• Finally: The CAP theorem

  – “Any highly-scalable distributed storage system using replication can only achieve a maximum of two properties out of consistency, availability and partition tolerance”

    • Thus, only compromises are possible

  – In most cases, consistency is sacrificed

    • Availability and partition tolerance keeps your business (and money) running

    • Many application can live with minor inconsistencies
10.1 CAP-Theorem

• “Proof” of CAP Theorem

• Assume
  – Two nodes $N_1$ and $N_2$
  – Both share a piece of data $V$ with value $V_0$
  – Both nodes run some algorithm $A$ or $B$ which are safe, bug free, predictable and reliable
    • In this scenario:
      – $A$ writes new values of $V$
      – $B$ reads values of $V$
10.1 CAP-Theorem

• “Good” case:
  – $A$ writes new value $V_1$ of $V$
  – An update message $m$ is sent to $N_2$
  – $V$ is updated on $N_2$
  – $B$ reads correct value $V_1$ from $V$
10.1 CAP-Theorem

• Assume that the network **partitions**
  – No messages between $N_1$ and $N_2$ possible anymore
  – $V$ on $N_2$ is not updated, $B$ reads stale value $V_0$ from $V$
  • **Consistency violated**
10.1 CAP-Theorem

• How to deal with the situation?
• **Ensure consistency, drop availability**
  – Use *synchronous messages to update all replicas*
    • Treat updating all replicas as an transaction
    • e.g. as soon as \( V \) is updated, send update messages to all replicas
      – Wait for confirmation; lock \( V \) at all nodes until all replicas have confirmed
      – What if no confirmation is received? Short time partitioning event and wait? Node failure and waiting is futile?
  – This approach does definitely not scale
  – During synchronization, \( V \) is **not available**
    • Clients have to wait
    • Network partitions even increase synchronization time and thus decrease availability further
  – **Example**
    • Most traditional distributed databases
• **Ensure consistency, drop availability (alternative)**
  
  – Just use one single master copy of the value $V$
    • Naturally **consistent**, no locking needed
  
  – **But: No high availability**
    • As soon as the node storing $V$ fails or cannot be reached, it is unavailable
  
  – **Additionally:**
    • Possibly bad scalability, possibly bad latency
  
  – **Examples**
    • Non-replicating distributed database
    • Traditional Client-Server database
      – Is also partition tolerant as there is just one node
• **Drop consistency**, keep partition tolerance and availability
  – Base idea for **partition tolerance**
    • Each likely partition should have an own copy of any needed value
  – Base idea for **availability**
    • Partitions or failing nodes should not stop availability of the service
      – Ensure “always write, always read”
      – No locking!
    • Use asynchronous update messages to synchronize replicas
    • So-called “**eventual consistency**”
      – After a while, all replicas will be consistent; until then stale reads are possible and must be accepted
      – No guaranteed consistency
      – Deal with versioning conflicts! (Compensation? Merge Versions? Ignore?)
  – **Examples**
    • Most storage backend services in internet-scale business
      – e.g. Amazon (Dynamo), Google (BigTable), Yahoo (PNUTS), Facebook (Cassandra), etc.
Is the CAP-Theorem an insurmountable obstacle?

- Just consider: Availability, Consistency, and Partition Tolerance are not binary
  - There are different types and qualities of each
- Also: Partitioning events are rare
  - So, we should try to not sacrifice consistency and availability when partitions are fine
  - Maybe we can do partition recovery (and only sacrifice some availability and consistency during that time) instead of focusing only on 2 out of 3 properties?
    - Still, many early NoSQL systems go strongly for availability and partition tolerance…
• Most NoSQL will use special **partitioning modes** and **partition recover algorithms**

• General outline:
  – *(we will discuss that in more detail next week)*
• Accepting **eventual consistency** leads to new application and transaction paradigms

• **BASE transactions**
  – Directly follows from the CAP theorem
  – **Basic Availability**
    • Focus on availability – even if data is outdated, it should be available
  – **Soft-State**
    • Allow inconsistent states
  – **Eventual Consistent**
    • Sooner or later, all data will be consistent and in-sync
    • In the meantime, data is **stale** and queries return just approximate answers
    • Consistency here means replica consistency
10.1 BASE Transactions

• The transition between **ACID** and **BASE** is a continuum
  – You may place your application wherever you need it to between ACID and BASE

**ACID**
- Guaranteed Transactional Consistency
- Severe Scalability issues

You?

**BASE**
- High scalability and performance
- Eventually consistent, approximate query results
• “Buy-A-Book” transaction
  – Assume a store like Amazon
  – Availability counter for every book in store
  – User puts book with availability ≥ 1 into the shopping cart
    • Decrease availability by one
  – Continue shopping
  – Two options
    • User finally **buys**
      – Send invoice and get user’s money
      – **Commit**
    • User does not buy
      – **Rollback** (reset availability)
• Obviously, this transaction won’t work in Amazon when locks are used
  – But even shorter transactions will unavoidably lead to problems assuming million concurrent users
  – **Lock contention thrashing**
10.1 BASE Transactions

- **Consideration:** Maybe full ACID properties are not always necessary?
  - Allow the availability counter to be out-of-sync?
    - Use a cached availability which is updated eventually
  - Allow rare cases where a user buys a book while unfortunately the last copy was already sold?
    - Cancel the order and say you are very sorry…

- These considerations lead to the **BASE** transaction model!
  - Sacrifice transactional consistency for scalability and features!
• Example System: Amazon Dynamo


– Amazon is one of the specialized storage solutions used at Amazon

• Among S3, SimpleDB, Elastic Block Storage, and others

• In contrast to the other service, it is only used internally
10.2 Dynamo
• Amazon infrastructure
  – Amazon uses a fully service oriented architecture
    • Each function used in any Amazon system is encapsulated in a service
      – i.e. shopping cart service, session management service, render service, catalog service, etc.
    • Each service is described by a service level agreement
      – Describes exactly what the service does
      – Describes what input is needed
      – Gives quality guarantees
• Services usually use other services
  – e.g. the page render service rendering the Amazon personalized start accesses roughly 150 simpler services
  – Services may be **stateful** or **stateless**
    • **Stateless**: Transformation, Aggregation, etc.
    • **Stateful**: Shopping cart, session management, etc.
  – **Dynamo** is a data storage service which mainly drives stateful services
    • Notably: shopping cart and session management
    • There are also other storage services
Service Level Agreements (SLA) are very important for Amazon

- Most important: latency requirements
- Goal: 99.9% of all users must have an internal page render response times below 300ms
  - Not average response times, but guaranteed maximum latency for nearly all customers!
  - It should not matter what the user does, how complex his history is, what time of day it is, etc.

- Most lower-tier services have very strict (and even tighter) SLA requirements
  - Final response is generated by aggregating all service responses
    - e.g. often, response times below 1ms for deep core services
10.2 Dynamo

• Furthermore, Amazon is a very big company
  – Up to 6 million sales per day
    • For each sale, there are hundreds of page renders, data accesses, etc.
    • Even more customers who just browse without buying!
  – **Globally** accessible and **operating**
    • Customers are from all over the world
  – **Highly scalable** and distributed systems necessary
    • Amazon Shopping uses several 10,000s servers
  – **Amazon services must always be available**
Hard learned lessons in early 2000: **RDBMS are not up for the job**

- Most features not needed
- Bad scalability
- Can’t guarantee extremely low latency under load
- High costs
- Availability problems
10.2 Dynamo

- **Dynamo** is a low-level distributed storage system in the Amazon service infrastructure.

- Requirements:
  - Very strict 99.9th percentile *latency*
    - No query should ever need longer than guaranteed in the SLA
  - Must be “*always writable*”
    - At no point in time, write access to the system is to be denied
  - Should support *user-perceived consistency*
    - i.e. technically allows for inconsistencies, but will eventually lead to an consistent state again
      - User should in most cases not notice that the system was in an inconsistent state
10.2 Dynamo

- **Low cost of ownership**
  - Best run on commodity hardware

- **Incremental scalability**
  - It should be easy to incrementally add nodes to the system to increase performance

- **Tunable**
  - During operation, trade-offs between costs, durability, latency, or consistency should be tunable
• Observation
  – Most services can efficiently be implemented only using **key-value stores**
    • e.g. shopping cart
      – key: session ID; value: blob containing cart contents
    • e.g. session management
      – key: session ID; value: meta-data context
  – No complex data model or queries needed!
• Further assumptions
  – All nodes in a Dynamo cluster are non-malicious
    • No fraud detection or malicious node removal necessary
  – All nodes are altruistic
    • No personal agenda; will participate in the system as long as able
  – Each service can set up its own dynamo cluster
    • Scalability necessary, but cluster don’t need to scale infinitely
10.2 Dynamo - Design

• Dynamo Implementation Basics
  – Overall, strong similarities to typical DHT implementations (e.g. Chord)
  – Build a distributed storage system on top of a DHT
    • Just provide \texttt{put()} and \texttt{get()} interfaces
  – Hashes nodes and data onto a 128-Bit address space ring using MD5
    • Consistent hashing similar to Chord
    • Nodes take responsibility of their respective anti-clockwise arc
### 10.2 Dynamo - Design

- **Assumption:** usually, nodes don’t leave or join
  - Only in case of hardware extension or node failure

- **Assumption:** ring will stay manageable in size
  - e.g. 10,000 nodes, not millions or billions

- **Requirement:** each query must be answered as fast as possible (low latency)

- **Conclusion:** For routing, each node uses a **full finger table**
  - Ring is **fully connected**
    - Maintenance overhead low due to ring’s stability
  - Each request can be routed within **one single hop**
    - No varying response time as in multi-hop systems like Chord!
– For load-balancing, each node may create additional virtual server instances

• Virtual servers may be created, merged, and transferred among nodes
  – Virtual servers are transferred using a large file binary transfer
    » Transfer not on record level

• Multiple central controllers manage virtual server creation and transfers (Many-to-Many)
For **durability**, replicas are maintained for each key-value entry

- Replicas are stored at the clockwise successor nodes
- For each data item, there is a **coordinator**
- Each node maintains a so-called **preference list** of nodes which may store replicas
  - More or less a renamed **successor list**
  - Preference list is usually longer than number of desired replicas

- Both techniques combined allow for **flexible**, **balanced**, and **durable** storage of data
Eventual Consistency

- After a `put()` operation, updates are propagated asynchronously
  - Eventually, all replicas will be consistent
  - Under normal operation, there is a hard upper bound until constancy is reached
- However, certain failure scenarios may lead to extended periods of inconsistency
  - e.g. network partitions, severe server outages, etc.
- To track inconsistencies, each data entry is tagged with a version number
10.2 Dynamo - Consistency

• Partition Mode
  – Just keep on working, use version number to track changes to data

• Partition Recovery
  – Use quorums to ensure some read and write consistency
  – Use anti-entropy repair protocol
    • Find inconsistent data
    • Try to determine total order of version numbers of former partitioned data (and operations)
      – If possible: Just merge the effects of operations
      – If not possible: Conflict, let application resolve error
• Clients can send any `put()` or `get()` request to any Dynamo node
  – Typically, each client chooses a Dynamo node which is used for the whole user session
  – Responsible node is determined by either
    • Routing requests through a set of *generic load balancers*, which reroute it to a Dynamo node to balance the load
      – Very simple for clients, additional latency overhead due to additional intermediate routing steps
    • Or the *client* uses a partition-aware client library
      – i.e. Client determines independently which node to contact by e.g. hashing
      – Less communication overhead and lower latency; programming clients is more complex
10.2 Dynamo – Requests

• Request Execution
  – Read / Write request on a key
    • Arrives at a node (coordinator)
      – Ideally the node responsible for the particular key
      – Else forwards request to the node responsible for that key and that node will become the coordinator
    • The first $N$ healthy and distinct nodes following the key position are considered for the request
      – Nodes selected from preference lists of coordinating node
    • Quorums are used to find correct versions
      – $R$: Read Quorum
      – $W$: Write Quorum
      – $R + W > N$
10.2 Dynamo – Requests

– Writes
  - Requires generation of a **new data entry version** by coordinator
  - Coordinator writes locally
  - Forwards to $N$ healthy nodes, if $W - 1$ respond then the write was successful
    - Called **sloppy quorum** as only healthy nodes are considered, failed nodes are skipped
    - Not all contacted nodes must confirm
  - Writes may be buffered in memory and later written to disk
    - Additional risks for durability and consistency in favor for performance

– Reads
  - Forwards to $N$ healthy nodes, as soon as $R - 1$ nodes responded, results are forwarded to user
    - Only unique responses are forwarded
  - Client handles merging if multiple versions are returned
    - Client notifies Dynamo later of the merge, old versions are freed for later Garbage Collection
10.2 Dynamo – Requests

• Tuning dynamo
  – Dynamo can be tuned using three major parameters
    • \( N \): Number of contacted nodes per request
    • \( R \): Number of Read quorums
    • \( W \): Number of Write quorums

<table>
<thead>
<tr>
<th>( N )</th>
<th>( R )</th>
<th>( W )</th>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>2</td>
<td>2</td>
<td>Consistent durable, interactive user state (typical)</td>
</tr>
<tr>
<td>( n )</td>
<td>1</td>
<td>( n )</td>
<td>High performance read engine</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Distributed web cache (not durable, not consistent, very high performance)</td>
</tr>
</tbody>
</table>
10.2 Dynamo - Consistency

• Theoretically, the same data can reside in **multiple versions** within the system
  – Multiple causes
    • **No failure**, asynchronous update in progress
      – Replicas will be eventual consistent
      – In rare case, branches may evolve
    • **Failure**: ring partitioned or massive node failure
      – Branches will likely evolve
  – In any case, a client just continues operation as usual
    • As soon as the system detects conflicting version from different branches, **conflict resolution** kicks in
Version Conflict Resolution

- Multiple possibilities
  - Depends on application! Each instance of Dynamo may use a different resolution strategy

- Last-write-wins
  - The newest version will always be dominant
  - Changes to older branches are discarded

- Merging
  - Changes of conflicting branches are optimistically merged
• **Example Merging**
  
  – User browses Amazon’s web catalog and adds a **book** to the shopping cart
    
    • Page renderer service stores new cart to Dynamo
      
      – Current session has a preferred Dynamo node
    
    • Shopping cart is replicated in the cart-service Dynamo instance
  
  – Dynamo **partitions** due to large-scale network outages

  – User adds a **CD** to his cart
    
    • Updated cart is replicated within the current partition
10.2 Dynamo - Consistency

– Page renderer service *looses connection* to the whole partition containing preferred Dynamo node
  • Switches to another node from the other partition
    – That partition contains only stale replicas of the cart, missing the CD
– User adds a *watering can* to his cart
  • Dynamo is “always write”
  • Watering can is just added to an old copy of the cart (only book)
– Partitioning event ends
  • Both partitions can contact each other again
  • Conflict detected
  • Both carts are simply merged
  • In the best case, the user did not even notice that something was wrong
10.2 Dynamo – Replica Synchronization

• Each node is responsible for multiple virtual servers
  – Each virtual server is responsible for hash range
  – The same virtual server assigned to multiple nodes for durability issues

• **Anti-Entropy Protocol** (i.e. replica synchronization)
  – Required to deal with severe problems like partitioning and node outages
  – Dynamo uses Merkle trees to detect replica inconsistencies
    • Read: Detects IF there is an inconsistency, not which version is correct!
Merkle Tree:

- Merkle Trees are **Hash Trees**
- Used for checking integrity of large “files”
  - Used in multiple P2P protocols (Gnutella, LimeWire, DC++), NoSQL systems (Cassandra, Riak, Dynamo), file systems (ZFS), revisions control systems (GIT), and digital signature schemes

Core Idea

- Leaf nodes are hashes of data blocks
- Inner nodes are hashes of the concatenation of its leafs
- Often used instead of hash lists
  - Advantage: One can check the integrity of a partially transferred tree / data block set
• Merkle Tree
Dynamo uses Merkle Trees to find consistency problems

- Each node maintains one Merkle tree per virtual server (and therefore per key range)
  - Assumption: Usually, there won’t be problems
  - Assumption: Network traffic is expensive, local computation is not
- Exchange only the root of the Merkle tree to check for inconsistencies between nodes
  - If roots are the same, everything is in sync
  - If not, recursively compare differing children down to leaves to find the problem

Compare to using hash lists $O(n)$:

- Merkle Tree: Worst Case $O(N)$, Average Case $O(\log(N))$
- Hash Lists: Worst Case $O(N)$, Average Case $O(N)$
• Now, we know that there is an inconsistency…
• But, how can we resolve it?
  – Use version numbers!
  – Newer version overrides older version
• Problem:
  – How can we have decentralized version numbers with partitioning?
  – Time stamps are notoriously unreliable!
• Version numbers are stored using **vector clocks**
  – Addressed problem: Detect conflicts using version numbers without central authority
  – Vector clocks are used to generate **partially ordered labels** for events in distributed systems
    • Designed to detect causality violations (e.g. conflicting branches)
    • Developed in 1988 independently by Colin Fridge and Friedmann Mattern
10.2 Dynamo – Vector Clocks

• Base idea vector clocks
  – Each node / process maintains an individual logical clock
    • Initially, all clocks are 0
    • A global clock can be constructed by concatenating all logical clocks in an array
  – Every node stores a local “smallest possible values” copy of the global clock
    • Contains the last-known logical clock values of all related other nodes
Every time a node raises an event, it increases its own logical clock by one within the vector.

Each time a message is to be sent, a node increases its own clock in the vector and attaches the whole vector to the message.

Each time a node receives a message, it increments its own logical clock in the vector.

- Additionally, each element of the own vector is updated with the maximum of the own vector and the received vector.
- Conflicts can be detected if messages are received with clocks which are not in total order in each component.
10.2 Dynamo – Vector Clocks

- Vector clock

Time

A:0

B

B:0

C:0

C:1

A:1

B:2

C:1

A:2

B:2

C:1

A:3

B:3

C:3

A:4

B:5

C:5

B:1

C:1

B:2

C:1

B:3

C:1

B:4

C:1

B:5

C:1

B:3

C:2

B:3

C:3

B:5

C:4

B:5

C:5

Causality

Effect
Example problem to be solved

- **Alice, Ben, Cathy, and Dave** are planning to meet next week for dinner.
- The planning starts with **Alice** suggesting they meet on **Wednesday**.
- Later, **Dave** discuss alternatives with **Cathy**, and they decide on **Thursday** instead.
- **Dave** also exchanges email with **Ben**, and they decide on **Tuesday**.
- When **Alice** pings everyone again to find out whether they still agree with her **Wednesday** suggestion, she gets mixed messages:
  - **Cathy** claims to have settled on **Thursday** with **Dave**.
  - **Ben** claims to have settled on **Tuesday** with **Dave**.
  - **Dave** can't be reached - no one is able to determine the order in which these communications happened.
- Neither **Alice, Ben, nor Cathy** know whether **Tuesday or Thursday** is the correct choice.
10.2 Dynamo – Vector Clocks

- Problem can be solved by tagging each choice with a vector clock
  - Alice says, "Let's meet Wednesday,"
    - Message 1: date = Wednesday; vclock = \{A: 1\}
  - Now Dave and Ben start talking. Ben suggests Tuesday
    - Message 2: date = Tuesday; vclock = \{A: 1, B: 1\}
  - Dave replies, confirming Tuesday
    - Message 3: date = Tuesday; vclock = \{A: 1, B: 1, D: 1\}
  - Now Cathy gets into the act, suggesting Thursday (independently of Ben or Dave, in response to initial message)
    - Message 4: date = Thursday; vclock = \{A: 1, C: 1\}
10.2 Dynamo – Vector Clocks

– **Dave** now received **two conflicting messages**
  - Message 3: date = Tuesday; vclock = \{A: 1, B: 1, D: 1\}
  - Message 4: date = Thursday; vclock = \{A: 1, C: 1\}
  - **Dave should resolve this conflict somehow**
  - **Dave agrees with Thursday** and confirms only to **Cathy**
    - Message 5: date = Thursday; vclock = \{A: 1, B: 1, C: 1, D: 2\}

– **Alice asks all her friends for their latest decision and receives**
  - **Ben**: date = Tuesday; vclock = \{A: 1, B: 1, D: 1\}
  - **Cathy**: date = Thursday; vclock = \{A: 1, B: 1, C: 1, D: 2\}
  - **No response from Dave**
  - But still, Alice knows by using the vector clocks that Dave intended to overrule Ben
    - She also knows that Dave is a moron and did not inform Ben of this decision (> “application decision” required)
• Dynamo (continued)
  – Eventual Consistency through asynchronous replica updates
  – To detect diverging branches and inconsistencies, vector clocks are used
    • Each data entry is tagged with a minimal vector clock
      – i.e. array has length one if only one node performs updates
      – For each additional node performing updates, the length of the vector increases
    • After a vector grows larger than 10 entries, the oldest ones are removed
      – Keeps the vector clock size capped
      – Some inconsistencies cannot be detected anymore
      – Has usually no practical impact as very strange (and unlikely) network failures are needed to generate vector clocks of size $\geq 10$
– Version branches may evolve (due to partitioning)
  • Version graph is only partially ordered in the worst case
– As soon as conflicting versions are detected (usually during replication update or client read), a reconciliation process is started
  • e.g. merge, discard old ones, etc.

Figure 3: Version evolution of an object over time.
• Test results for response requirement is 300ms for any request (read or write)
10.2 Dynamo - Evaluation

• Load distribution

Figure 6: Fraction of nodes that are out-of-balance (i.e., nodes whose request load is above a certain threshold from the average system load) and their corresponding request load. The interval between ticks in x-axis corresponds to a time period of 30 minutes.
10.2 Dynamo - Evaluation

- **Consistency vs. Availability**
  - 99.94% of values have one version
  - 0.00057% of values have two versions
  - 0.00047% of values have three versions
  - 0.00009% of values have four versions

- **Server-driven or Client-driven coordination**
  - **Server-driven**
    - uses load balancers
    - forwards requests to desired set of nodes
  - **Client-driven 50% faster**
    - requires polling of Dynamo membership updates
    - the client is responsible for determining the appropriate nodes to send the request to

- **Successful responses (without time-out) 99.9995%**
  - Configurable ($N, R, W$)
10.2 Discussion of Dynamo

- **How to distribute and locate data?**
  - Dynamo uses horizontal partitioning based on hash values of primary key
  - Queries can directly determine node responsible, routing using full finger tables is $O(1)$

- **How to make data durable?**
  - Multiple replicas using preference lists

- **How to deal with load balancing?**
  - Virtual servers
  - Application-side load balancer, system-wide load balancers

- **How to deal with inconsistencies between replicas?**
  - Eventual consistency, delayed replication, sloppy quorums
  - Use anti-entropy protocol to repair inconsistencies resulting from outages or partitioning events

- **How to deal with user transactions?**
  - Well….not at all…

- **How to deal with complex data models or query operators?**
  - Also, Dynamo does not do this
10.2 Dynamo - Summary

- Dynamo is not the Holy Grail of Data Storage

**Strength**
- Highly available
- Guaranteed **low latencies**
- Incrementally scalable
- Trade-offs between properties can be **tuned dynamically**

**Limitations**
- **No infinite scaling**
  - Due to fully meshed routing and heavy load on new node arrival (virtual server transfer)
- Does **not support real OLTP** queries
- Each application using dynamo must provide **conflict resolution strategies**
10.3 Project Voldemort

• Open Source Implementation of Dynamo
  – From DHT Key-Value Store to Database
  – Backed by LinkedIn

• Interesting about Voldemort:
  – “Schema” using JSON
    • Still, queries only using key

• Voldemort supports cross-datacenter replication
  – Allowing quorum reads/writes within a single geographic area
  – Purely asynchronous writes to datacenters in a different geographic area

• http://www.project-voldemort.com/voldemort/design.html
10.3 Project Voldemort

Logical Architecture

- Client API
- Conflict Resolution
- Serialization / Compression
- Routing
  - Consistency mechanisms (Read repair / Hinted handoff)
- Storage Engine
  - (BDB / MySQL / Memory / Read-only)

Network Client & Server
  - (HTTP / Sockets / NIO)

Requests \rightarrow Responses
10.3 Project Voldemort

Physical Architecture Options

- **3-Tier, Server-Routed**
  - Frontend
  - Backend Service
  - Voldemort Cluster
  - Partition-aware Routing

- **3-Tier, Client-Routed**
  - Frontend
  - Backend Service
  - Voldemort Cluster
  - Partition-aware Routing

- **2-Tier, Frontend-Routed**
  - Frontend
  - Backend Service
  - Vdmtn Inst.
  - Additional routing
Google was founded in 1998 by the Stanford Ph.D. candidates Larry Page and Sergey Brin

- Headquarter in Mountain View, CA, USA
- Named after the number Googol
- More than 20,000 employees
• **Initial mission**
  – “to organize the world's information and make it universally accessible and useful”
    • and “Don’t be evil”

• **Originally, Google became famous for their search engine**
  – **Initial Idea: Google PageRank**
    • Not all web pages are equally important
    • Link structure can be used to determine site’s importance
    • Resulting search engine showed much higher result quality than established products (e.g. Yahoo, Altavista, etc.)
      – Rise of Google as one of the big **internet pioneers** starts
Currently, Google offers a multitude of services

- Google Search
- Google+
- Google Mail
- Google Maps
- Google Earth
- Google Documents
- Picasa Web Album
- etc.

Thus, Google hosts and actively uses several Petabytes of data!
• (some slides from the start 😊 ) … or how to build one of the **most powerful data centers** out of **crappy hardware**
  – Google has jealously guarded the design of its data centers for a long time
    • In 2007 & 2009 some details have been revealed

• **The Google Servers**
  – Google only uses custom build servers
  – Google is the world 4\textsuperscript{th} largest server producer
    • They don’t even sell servers…
    • In 2007, it was estimated that Google operates over 1,000,000 servers over 34 major and many more minor data centers
11.0 Google Servers

- Each server rack holds 40 to 80 commodity-class x86 PC servers with custom Linux
  - Each server runs slightly outdated hardware
  - Each system has its own 12V battery to counter unstable power supplies
  - No cases used, racks are setup in standard shipping containers and are just wired together
    - More info: http://www.youtube.com/watch?v=Ho1GEyftpmQ
10.4 Google Servers

• Challenges to the data center software
  – Deal with hardware failures while avoiding any data loss and ~100% global uptime
  – Decrease maintenance costs to minimum
  – Allow flexible extension of data centers
  – Solution:
    • Use cloud technologies
    • **GFS** (Google File System) and **Google Bigtable Data System**
    • **NEXT WEEK!**
• More cloud technology
  – Google GFS
  – Google BigTable