Distributed Data Management

Christoph Lofi
Institut für Informationssysteme
Technische Universität Braunschweig
http://www.ifis.cs.tu-bs.de
10.0 Towards the Cloud

10.0 Special Purpose Database

10.1 Trade-Offs
   – CAP Theorem
   – BASE transactions

10.2 Showcase: Amazon Dynamo
• Traditional databases are usually all-purpose systems
  – e.g. DB2, Oracle, MySQL, …
  – Theoretically, general purpose DB provide all features to develop any data driven application
  – Powerful query languages
    • SQL, can be used to update and query data; even very complex analytical queries possible
  – Expressive data model
    • Most data modeling needs can be served by the relational model
– **Full transaction support**
  - Transactions are guaranteed to be “safe”
    - i.e. ACID transaction properties

– **System durability and security**
  - Database servers are resilient to failures
    - *Log files* are continuously written
      - Transactions running during a failure can be recovered
    - Most databases have support for constant *backup*
      - Even severe failures can be recovered from backups
    - Most databases support “*hot-standby*”
      - 2nd database system running simultaneously which can take over in case of severe failure of the primary system

  - Most databases offer basic *access control*
    - i.e. *authentication* and *authorization*
10.0 Special Purpose Databases

• In short, databases could be used as storage solutions in all kinds of applications
• Furthermore, we have shown distributed databases which also support all features known from classical all-purpose databases
  – In order to be distributed, additional mechanisms were needed
    • partitioning, fragmentation, allocation, distributed transactions, distributed query processor,....
However, classical all-purpose databases may lead to problems in extreme conditions

- Problems when being faced with massively high query loads
  - i.e. millions of transactions per second
  - Load to high for a single machine or even a traditional distrusted database
    - Limited scaling
- Problems with fully global applications
  - Transactions originate from all over the globe
  - Latency matters!
    - Data should be geographically close to users
  - Claims:
    - Amazon: increasing the latency by 10% will decrease the sales by 1%
    - Google: increasing the latency by 500ms will decrease traffic by 20%
– Problems with extremely high **availability** constraints

  • Traditionally, databases can be recovered using logs or backups
  • Hot-Standbys may help during repair time
  • But for some applications, this is not enough: **Extreme Availability** (Amazon)
    – “… must be available even if disks are failing, network routes are flapping, and several data centers are destroyed by massive tornados”
    – Additional availability and durability concepts needed!
• In extreme cases, specialized database-like systems may be beneficial
  – Specialize on certain query types
  – **Focus on a certain characteristic**
    • i.e. availability, scalability, expressiveness, etc…
  – Allow weaknesses and limited features for other characteristics
Typically, two types of queries can be identified in global businesses:

**OLTP queries**
- **Online Transaction Processing**
- Typical *business backend-data storage*
  - i.e. order processing, e-commerce, electronic banking, etc.
- Focuses on *data entry* and *retrieval*
- Usually, possible *transactions* are previously *known* and are only *parameterized* during runtime
- The *transaction load* is very high
  - Represents daily business
- Each *transaction* is usually very simple and local
  - Only few records are accessed in each transaction
  - Usually, only basic operations are performed
• OLAP queries
  – **OnLine Analytical Processing**
  – Business Intelligence Queries
    • i.e. complex and often multi-dimensional queries
  – Usually, only few OLAP queries are issued by business analysts
    • Not part of daily core business
  – Individual queries may need to access large amounts of data and uses complex aggregators and filters
    • Runtime of a query may be very high
In the recent years, discussing “NoSQL” databases have become very popular

- Careful: big misnomer!
  - Does not necessarily mean that no SQL is used
    - There are SQL-supporting NoSQL systems…
  - NoSQL usually refers to “non-standard” architectures for database or database-like systems
    - i.e. system not implemented as shown in RDB2
  - Not formally defined, more used as a “hype” word

- Popular base dogma: Keep It Stupid Simple!
The NoSQL movement popularized the development of **special purpose databases**

- In contrast to **general purpose systems** like e.g. DB2

NoSQL usually means one or more of the following

- Being massively **scalable**
  - Usually, the goal is unlimited linear scalability

- Being massively **distributed**

- Being extremely **available**

- Showing extremely high **OLTP performance**
  - Usually, not suited for OLAP queries
– Not being “all-purpose”
  • Application-specific storage solutions showing some database characteristics

– Not using the relational model
  • Usually, much simpler data models are used

– Not using strict ACID transactions
  • No transactions at all or weaker transaction models

– Not using SQL
  • But using simpler query paradigms

– Especially, not supporting “typical” query interfaces
  • i.e. JDBC
  • Offering direct access from application to storage system
In short:

– Most NoSQL focuses on building specialized high-performance data storage systems!
NoSQL and special databases have been popularized by different communities and are driven by different design motivations.

Base motivations:

- **Extreme Requirements**
  - Extremely high availability, extremely high performance, guaranteed low latency, etc.
  - e.g. global web platforms

- **Alternative data models**
  - Less complex data model suffices
  - Non-relational data model necessary
  - e.g. multi-media or scientific data

- **Alternative database implementation techniques**
  - Try to maintain most database features but lessen the drawbacks
  - e.g. “traditional” database applications, e.g. VoltDB
Motivation: Extreme Requirements

- Extreme Availability
  - No disaster or failure should ever block the availability of the database
  - Usually achieved by strong global replication
    - i.e. data is available in multiple sites with completely different location and connections

- Guaranteed low latency
  - Distances from users to data matters in term of latency
    - e.g. crossing the Pacific from east-coast USA to Asia easily amounts for 500ms latency
  - Data should be close to users
    - e.g. global allocation considering the network layer’s performance

- Extremely high throughput
  - Some systems need to handle extremely high loads
    - e.g. Amazon’s four million checkouts during holidays
      » Each checkout was preceded by hundreds of queries
Community: **Alternative Data Models**

- This is where the NoSQL originally came from
- **Base idea:**
  - Use a very simple data model to improve performance
  - No complex queries supported
- e.g. **Document stores**
  - Data consist of key-value pairs and additional document payload
    - e.g. payload represents text, video, music, etc.
  - Often supports IR-like queries on documents
    - e.g. ranked full text searches
  - Examples
    - CouchDB, MongoDB
Key-Value stores

- Each record consists of just a **key-value pair**
- Very simple data and query capabilities
  - **Put** and **Get**
- Usually implemented on top of a **Distributed Hash Table**
- **Example:**
  - MemcacheDB and Amazon Dynamo

Both document and key-value stores offer **low-level, one-record-at-a-time** data interfaces

- **XML** stores, **RDF** stores, Object-Oriented Databases, etc.
  - Not important in current context as most implementations have neither high performance nor are scalable
  - Those use the opposite philosophy of “classic” NoSQL: do it more complex!
• Community: *Alternative Database Implementation*

• **OLTP Overhead Reduction**
  
  – Base observation: most time in traditional OLTP processing is spent in overhead tasks
    
    • Four major overhead sources equally attribute to most of the used time

  – **Base idea**
    
    • Avoid overhead all those sources of unnecessary overhead
### 10.0 Special Purpose Databases

**Logging**
- “Traditional” databases write everything twice
  - Once to tables, once to log
  - Log is also forced to disk ⇒ performance issues

**Locking**
- For ensuring transactional consistency, usually locks are used
- Locks force other transaction to wait for lock-release
- Strongly decreases maximum number of transactions!

**Latching**
- Updates to shared data structures (e.g. B-tree indexes) are difficult for multiple threads
- Latches are used (a kind of short-term lock for shared data structures)
Buffer Management

- Disk-based systems have problems randomly accessing small bits of data
- Buffer management locates the required data on disk and caches the whole block in memory
- While increasing the performance of disk based systems, it still is a considerable overhead by itself
10.0 Special Purpose Databases

• Current trend for overhead avoidance
  – **Distributed single-thread** minimum-overhead **shared-nothing** parallel **main-memory** databases (OLTP)
    • e.g. VoltDB (Stonebraker et al.),
  – **Sharded row stores** (mostly **OLAP**)
    • e.g. Greenplum, MySQL Cluster, Vertica, etc.
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
10.1 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
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
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 $V$
• 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
10.1 CAP-Theorem

• 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
10.1 CAP-Theorem

• **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.
10.1 CAP-Theorem

- 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
• “Buy-A-Book” transaction
  – Assume a store like Amazon
  – Availability counter for every book in store
  – User puts book with availability $\geq 1$ 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**
• **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 consideration lead to the **BASE** transaction model!
  – Sacrifice transactional consistency for scalability and features!
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

BASE

+ High scalability and performance
- Eventually consistent, approximate query results

You?
10.2 Dynamo

• 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
• 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
10.2 Dynamo

- 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
• 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**
9.3 Dynamo

• 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

Distributed Data Management – Christoph Lofi – IfS – TU Braunschweig
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
– **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 `put()` and `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
- 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
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
• 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>
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
10.2 Dynamo - Consistency

- **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
– 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 – Vector Clocks

• 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
• 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.
• Vector clock
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
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\}
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)
10.2 Dynamo – Consistency

• 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.
• Test results for response requirement is 300ms for any request (read or write)
• 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 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**
• Wonderful Cloudy Future
  – What is the Cloud?
  – Software as a Service?
  – IT as utility?

• More cloud technology
  – Towards more complex cloud data models
  – Google BigTable
  – Facebook Cassandra