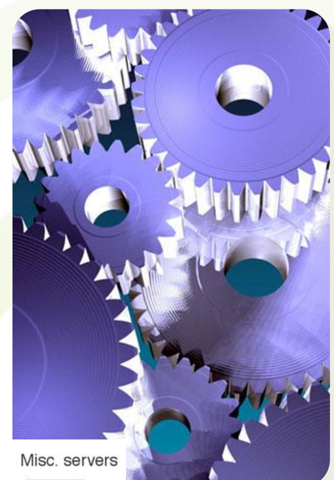
Distributed Storage

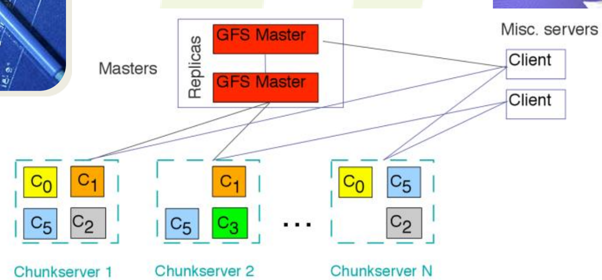
Wolf-Tilo Balke & Pierre Senellart
IFIS, Technische Universität Braunschweig
IC2, Telecom ParisTech

Introduction

Basics in Distributed Storage

- Context, Motivation & Applications
- The world of DHTs
- Dynamo & BigTable





Special Purpose Databases

- 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**

Special Purpose Databases

- **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 **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**

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,….

Special Purpose Databases

- 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%

Special Purpose Databases

- 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!

Special Purpose Databases

- 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

Special Purpose Databases

- Typically, two types of queries can be identified in global businesses
- **OLTP queries**
  - **O**n**L**ine **T**ransaction **P**rocessing
  - 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

Special Purpose Databases

- **OLAP queries**
  - **O**n**L**ine **A**nalytical **P**rocessing
  - 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

Special Purpose Databases

- 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: **K**eep **I**t **S**tupid **S**imple!

Special Purpose Databases

- 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 scalability
  - Being massively **distributed**
  - Being highly **available**
  - Showing extremely high **OLTP performance**
    - Usually, not suited for OLAP queries

Special Purpose Databases

- 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

Special Purpose Databases

- In short:
  - Most NoSQL focuses on building specialized **high-performance** data storage systems!

Special Purpose Databases

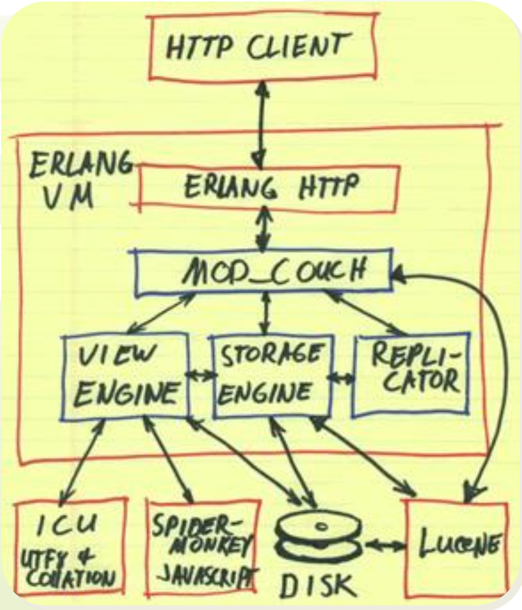
- NoSQL and special databases have been popularized by different **communities** and a driven by different design motivations
- Base motivations
  - **Extreme Requirements**
    - Extremely high availability, extremely high performance, guaranteed low latency, etc.
  - **Alternative data models**
    - Less complex data model suffices
    - Non-relational data model necessary
  - **Alternative database implementation techniques**
    - Try to maintain most database features but lessen the drawbacks

Special Purpose Databases

- 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

Special Purpose Databases

- 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



Special Purpose Databases

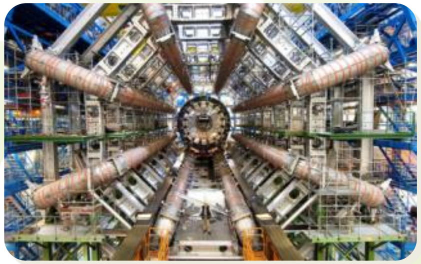
– **Key-Value stores**
  - Each record consist of just a **key-value pair**
  - Very simple data and query capabilities
    – **Put** and **Get**
  - Usually implemented on top of a **Distributed Hash Table**
  - **Example:**
    – MemcacheDB and **Amazon Dynamo**

– 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**!

Special Purpose Databases

- 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



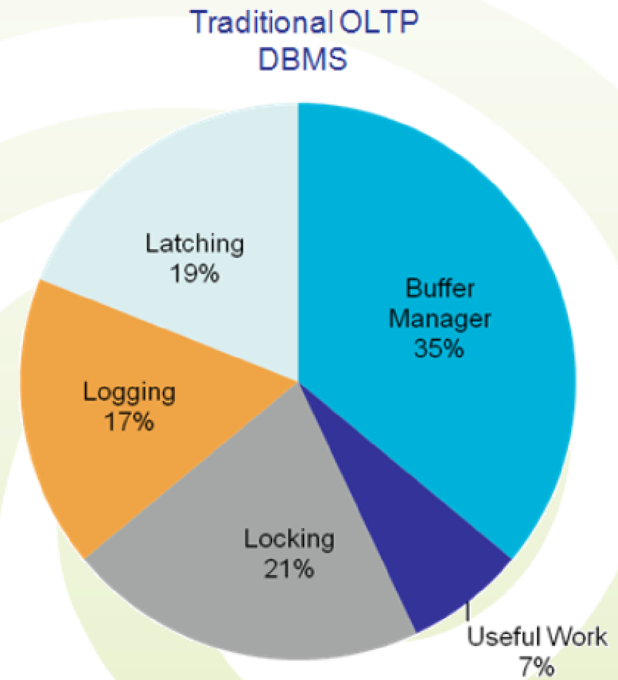
Special Purpose Databases

- **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)

Special Purpose Databases

– **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



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.
  - This kind of systems will be covered in one of the next weeks

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

CAP-Theorem

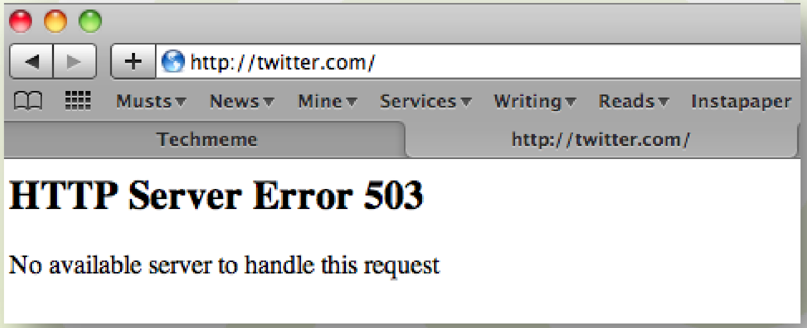
- The **CAP theorem** was made popular by **Eric Brewer** at the ACM Symposium of Distributed Computing (PODC)
  - Started as a conjecture, was later proven by Gilbert and Lynch
    - Seth Gilbert, Nancy Lynch. ***"Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services".*** ACM SIGACT News, *2002*
  - CAP theorem limits the design space for highly-available distributed systems

CAP-Theorem

- Assumption:
  - High-performance distributed storage system with replicated data fragments
- **CAP: C**onsistency, **A**vailability, **P**artition 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

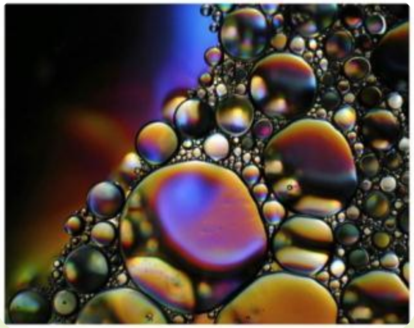
CAP-Theorem

- **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



CAP-Theorem

- **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

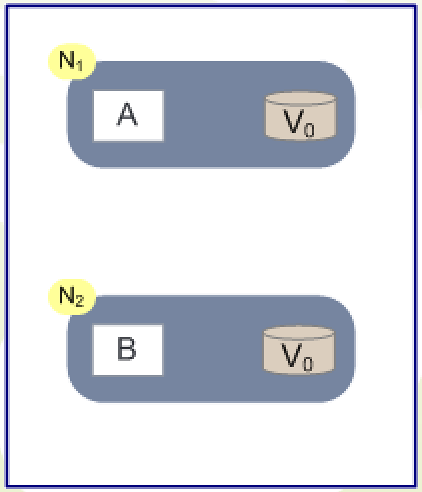


CAP-Theorem

- 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 life with minor inconsistencies

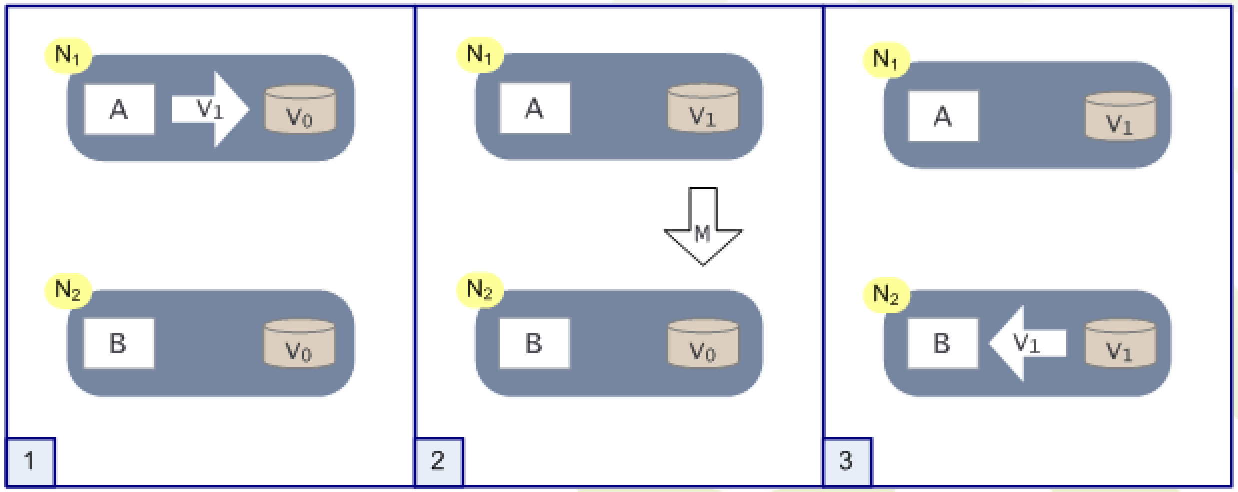
CAP-Theorem

- “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



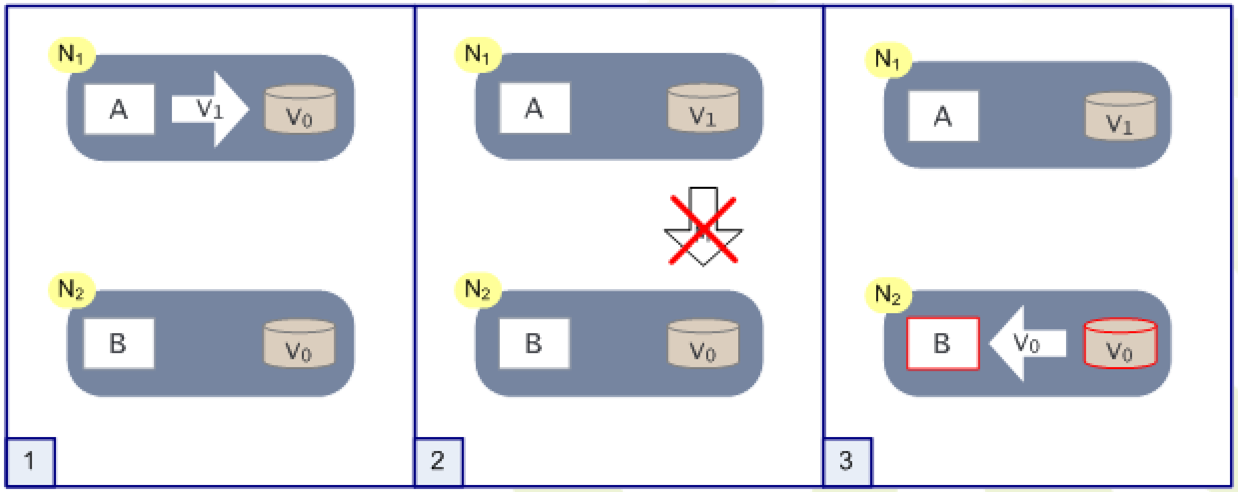
CAP-Theorem

- "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



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



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

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 additionally partition tolerant as there is just one node

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 real 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.

CAP-Theorem

- Accepting **eventual consistency** leads to new application and transaction paradigms
- **BASE transactions**
  - Directly follows from the CAP theorem
  - **B**asic **A**vailability
    - Focus on availability – even if data is outdated, it should be available
  - **S**oft-State
    - Allow inconsistent states
  - **E**ventual Consistent
    - Sooner or later, all data will be consistent and in-sync
    - In the meantime, data is **stale** and queries return just approximate answers

BASE Transactions

- **“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**
      – Write invoice and get user’s money
      – **Commit**
    - User does not buy
      – **Rollback** (reset availability)

BASE Transactions

- Obviously, this transaction won't work in Amazon when locks are used
  - But even smaller transactions will unavoidably lead to problems assuming million concurrent users
  - **Lock contention thrashing**

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 user and say you are very sorry…
- These consideration lead to the **BASE** transaction model!
  – Sacrifice transactional consistency for scalability and features!

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

You?

ACID

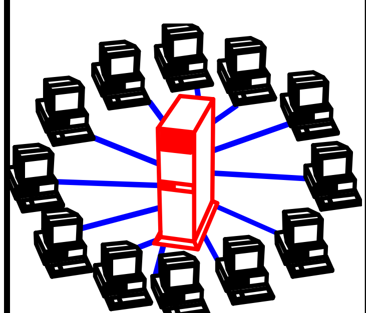
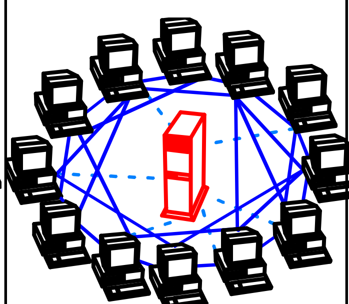
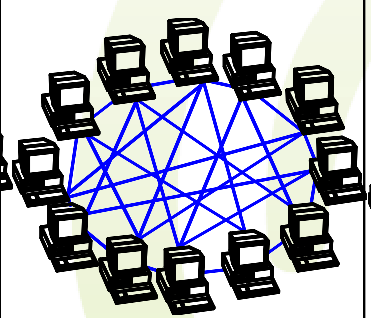
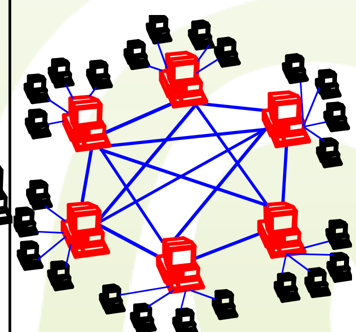
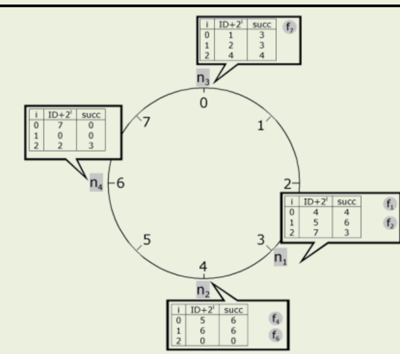
BASE

+ Guaranteed Transactional Consistency
- Severe Scalability issues

+ High scalability and performance
- Eventually consistent, approximate answers

The P2P Paradigm

Detour

Client-Server	*Peer-to-Peer*			
1. Server is the central entity and only provider of service and content. → Network managed by the Server 2. Server as the higher performance system. 3. Clients as the lower performance system Example: WWW	1. Resources are shared between the peers 2. Resources can be accessed directly from other peers 3. Peer is provider and requestor (Servent concept)			
	Unstructured P2P			***Structured P2P***
	Centralized P2P	*Pure P2P*	*Hybrid P2P*	*Pure P2P (DHT Based)*
	1. All features of Peer-to-Peer included 2. Central entity is necessary to provide the service 3. Central entity is some kind of index/group database Example: Napster	1. All features of Peer-to-Peer included 2. Any terminal entity can be removed without loss of functionality 3. → No central entities Examples: Gnutella 0.4, Freenet	1. All features of Peer-to-Peer included 2. Any terminal entity can be removed without loss of functionality 3. → dynamic central entities Example: Gnutella 0.6, JXTA	1. All features of Peer-to-Peer included 2. Any terminal entity can be removed without loss of functionality 3. → No central entities 4. Connections in the overlay are "fixed" Examples: Chord, CAN
				

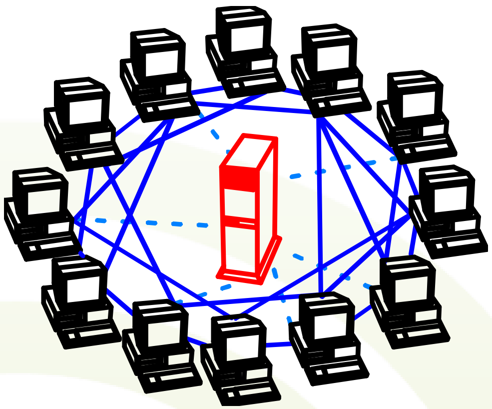
1st Gen.

2nd Gen.

Unstructured P2P

Detour

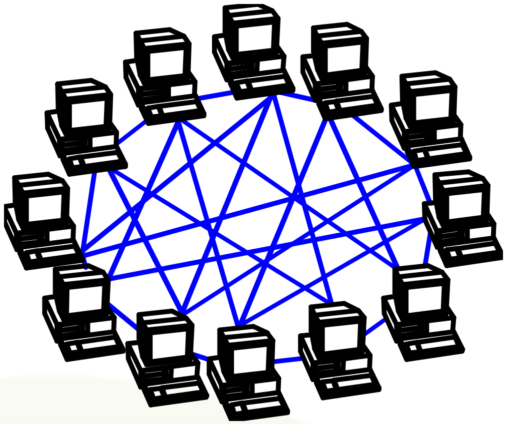
- In **centralized P2P systems**, a **central server** is used to **index** all available data
  - During bootstrap, peers provide a content list to the server
  - Any search request is resolved by the server
- **Advantages**
  - Search complexity of ***O(1)*** – "just ask the server"
  - Complex and fuzzy queries are possible
  - Simple and fast
- **Problems**
  - Bad Scalability
    - ***O(N)*** node state in server
      - Information that must be stored at server grows linearly with number of peers ***N***
    - ***O(N)*** network and system load of server
      - Query and network load of server also grows linearly with number of peers
  - Single point of failure or attack (also for law suites ;-)
- But overall, …
  - Best principle for **small** and **simple** applications



Unstructured P2P

Detour

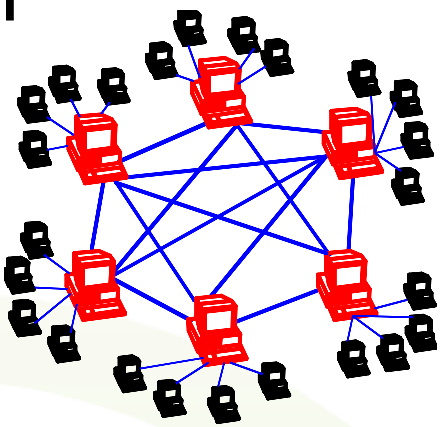
- **Pure P2P networks** counter the problems of centralized P2P
  - **All peers are equal**
  - **Content is not indexed**
    - Queries are **flooded** along the nodes
    - Node state complexity (storage complexity) ***O(1)***
  - **No central point of failure**
  - Theoretically, high **scalability** possible
    - In practice, scalability is limited by possibly degenerated network topologies, high message traffic, and low bandwidth nodes



Unstructured P2P

Detour

- **Hybrid P2P** adds hierarchy layers to P2P
  - High-performance nodes → **super peers**
    - All others are **leaf nodes**
  - **All super peers form a pure P2P**
  - **Leaf nodes connect to a super peer**
    - Super peers index their leaf node's content
      - **Routing tables**; similar to centralized server indexing
    - Node state is also in ***O(1)***
      - Leaf nodes store no index information
      - Maximum load of super peers is capped
        - » More peers → more super peers
    - Queries are flooded within the super peer network
  - Resulting networks usually have a lower diameter and routing bottlenecks are less likely



Unstructured P2P

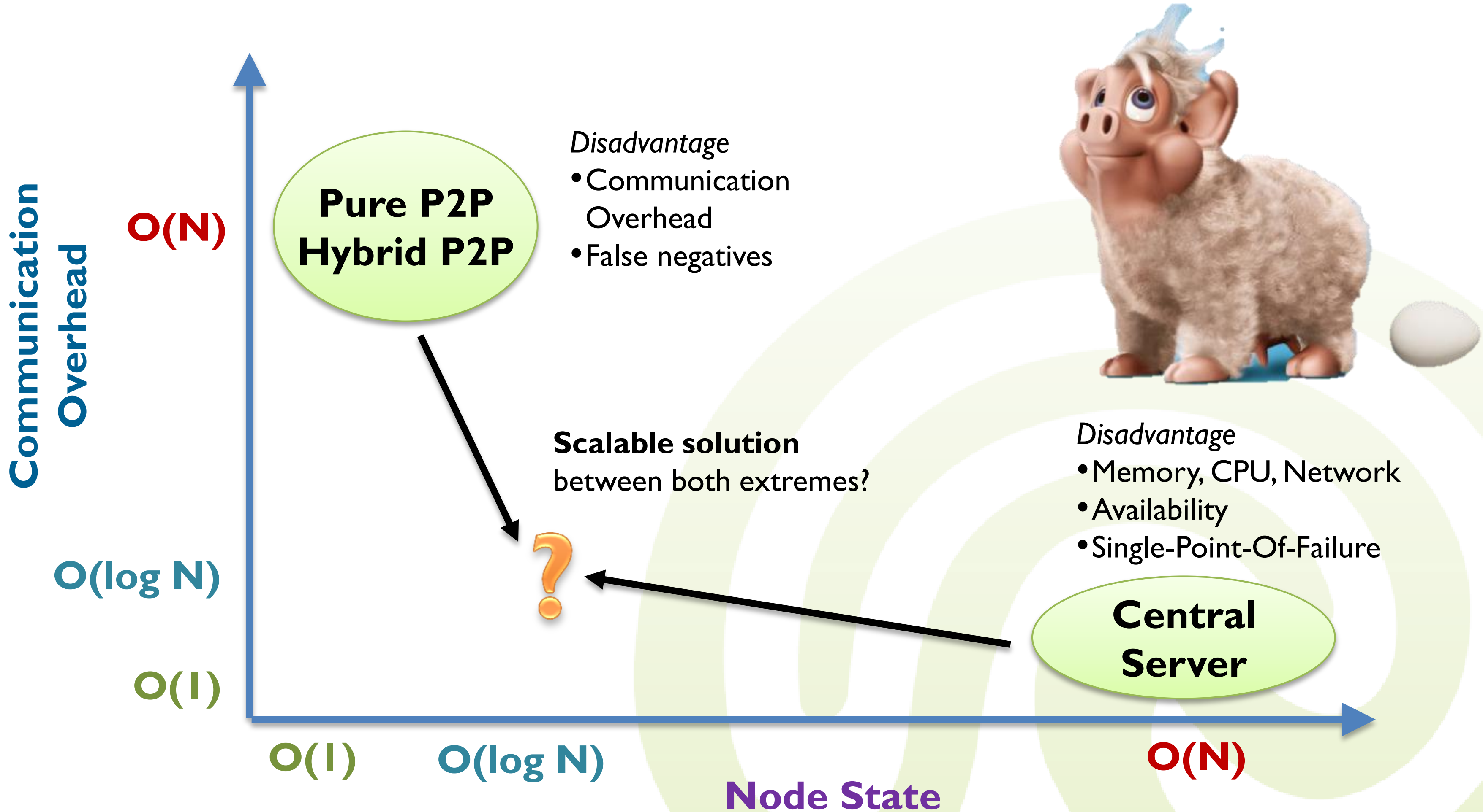
Detour

- Both **pure** and **hybrid** unstructured P2P rely on **query flooding**
  - Query is **forwarded** to all neighbors which also forward the query
    - **TTL** (time-to-life) limits the maximum distance a query can travel
  - Flooding result to
    - **High message and network load**
      - Communication overhead is in ***O(N)***
    - **Possibility of false negatives**
      - Node providing the required data may simply be missed due too short TTL

Unstructured P2P

Detour

- **Communication overhead** vs. **node state**



Distributed Hash Tables

- Idea: use a **Distributed Hash Table (DHT)** to index all data in a P2P network
  - Perform routing and resource discovery in DHT
- **Claims of DHTs**
  - DHT can perform search and routing in ***O(log N)***
  - Required storage per node is low in ***O(log N)***
  - DHT can provide correct query results
    - **No false negatives**
  - P2P systems based on DHTs are resilient to failures, attacks, and weak or short-time users

Hash Tables

Detour

- DHTs are based on **hash tables**
  - Hash tables are **data structures** which may provide an idealized lookup complexity close to ***O(1)***
  - Usually, data consists of key-value pairs
    - Lookup a key, return the according value
- Hash tables consist of two major components
  - **Bucket array**
    - Usually a **fixed-size** array
    - Each array cell is called a **bucket**
  - **Hash function**
    - A hash function maps a key to a bucket of the array

Hash Tables

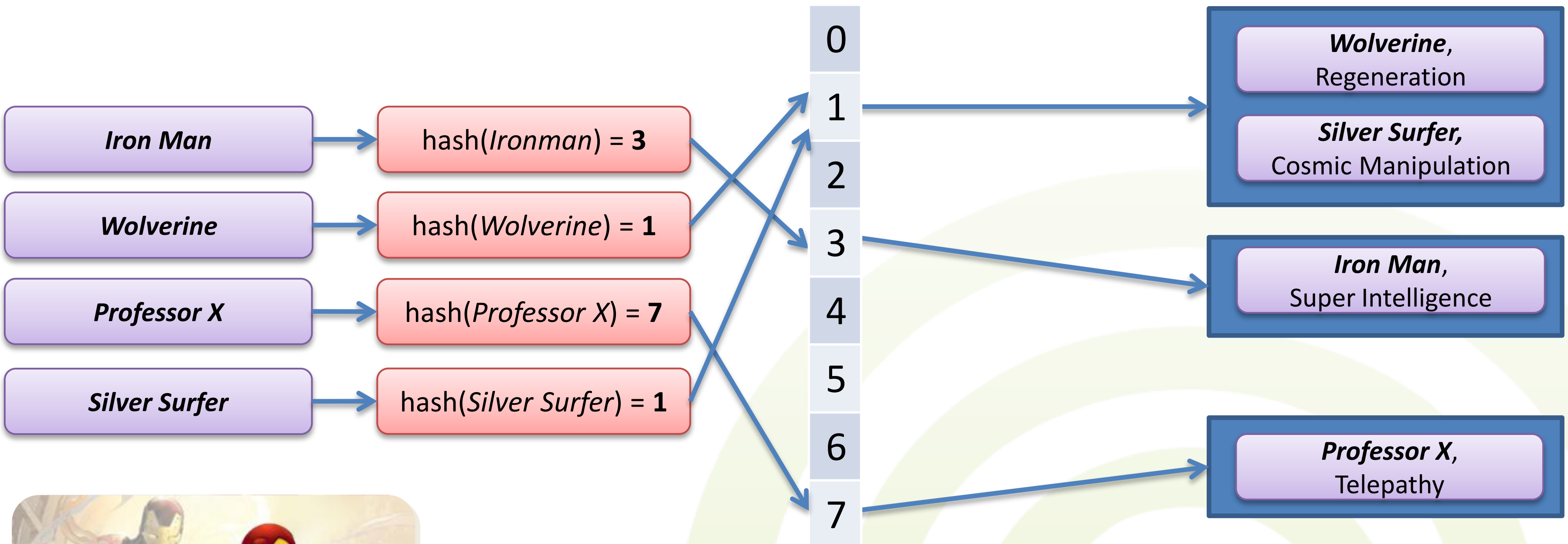
Detour

- Hash functions may **collide**, i.e. two different keys may result in the same hash
  - In many implementations, **buckets** are designed as a pointer to a **list** holding multiple items
  - **Insert**: hash the key and add the data to the respective bucket
  - **Lookup**: hash the key and scan the respective bucket
    - Lookup best case: bucket contains just one item: ***O(1)***
    - Lookup worst case: bucket contains multiple items: ***O(n)***
      - Rare case, even if it happens list should be small such that average complexity is still ~***O(1)***

Hash Tables

Detour

- Example:



Bucket Array (8 buckets)

Hash Functions

Detour

- At the core of hash tables are **hash functions**
  - Hash functions maps any key to a bucket of the array
    - $keyspace \xrightarrow{hash} [0, hashrange - 1]$
    - $hashrange$ is the number of buckets in the array
- Hash funtions should show some important properties
  - **Low Cost**
  - **Determinism**
  - **Uniformity**
  - **Range Variability**
  - Either **Avalanche** or **Continuity** properties

Hash Functions

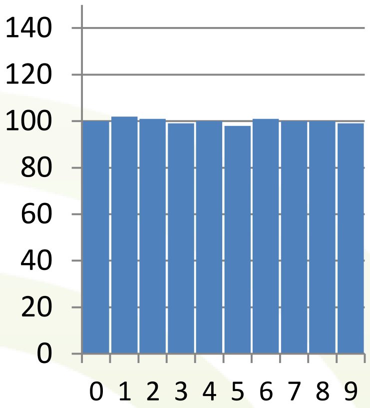
Detour

- **Low Cost**
  - Hashing should have higher average performance than rivaling approaches
    - Hash function thus should have low costs!
- **Determinism**
  - Hashing the same key or object must always result in the same hash
    - If not, no lookups are possible!

Hash Functions

Detour

- **Uniformity**
  - A good hash function should map the keys as evenly as possible over the whole output range
    - i.e. every hash value should be generated with the same probability
  - Hash values thus should be generated following an **uniform distribution**
  - Uniform hash codes will reduce the number of **hash collisions** to a statistical minimum
    - Collisions will severely **degenerate** the **performance** of the hash table



Hash Functions

Detour

- **Continuity** or **Avalanche** property
  - Depending on the actual usage of the hash function, different properties may be needed with respect to **small key changes**
  - **Avalanche property**
    - Changing one bit in the key should change at least 50% of the hash bits
    - Very important property when dealing with **cryptographic** applications or **distributing content** in robust fashion
    - MD5 hash examples
      - P2P is cool! = 788d2e2aaf0e286b37b4e5c1d7a14943
      - P2P is cool" = 8a86f958183b7afa26e15fa83f41de7e

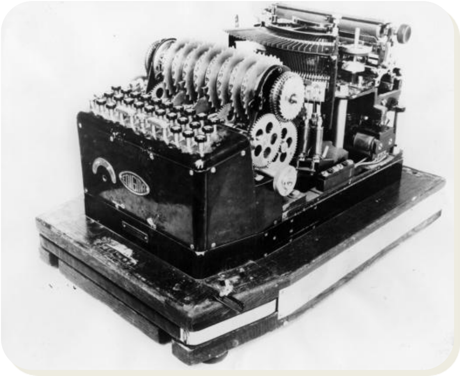
Hash Functions

Detour

- **Continuity property**
  - **Small changes** in **keys** should only result in **small changes** in **hashes**
  - Useful when implementing **similarity searches** with hash functions
    - Simply, hash a search string and inspect surrounding buckets
  - Adler32 hash examples
    - P2P is cool! = 175003bd
    - P2P is cool" = 175103be

Hash Functions

Detour



- Some hash functions
  - **Simple modulo hash**
    - $hash = key \bmod hashrange$
    - Easy and cheap
    - Works only if keys are uniformly distributed!
  - **Cryptographic hash functions**
    - Very expensive hash functions guaranteeing cryptographic properties
      - Variable Input Size
      - Constructing the key from the hash is impossible
      - Extremely low collision probability
      - Avalanche properties
      - **No hash clones constructable**
        - » e.g. given a hash, it is impossible to construct an object which results in the same hash

Hash Functions

- Most popular cryptographic examples
  - **MD-5** (128 Bit)
    - **Practically proven to be prone to clone attacks**
  - **SHA-1** (160 Bit)
    - Fork of MD-4
    - Previous recommendation of NSA
    - **Theoretically proven to be prone to clone attacks**
  - SHA-2 (224, 256, 384, 512 Bit)
    - Fork of SHA-1
    - Current NSA recommendation
    - No weakness known yet (but it is assumed that there should be weaknesses similar to SHA-1)
  - SHA-3
    - Completely new algorithm
    - Currently in competition phase until 2010

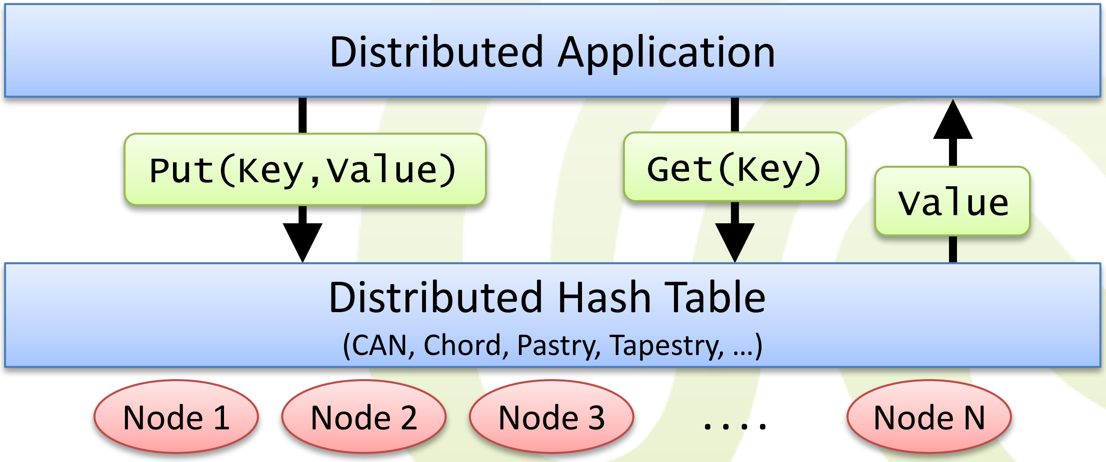
Distributed Hash Tables

- In distributed hash tables (**DHT**), the bucket array is distributed across all participating nodes
- Base idea

  - Use a large **fixed hash range**
  - Each **node** is **responsible** for a **certain section** of the whole hash range
    - Responsible node stores the payload of all data with hash keys in its range
  - Put and get requests are **routed** along the hash range to the responsible nodes

Distributed Hash Tables

- Generic **interface** of distributed hash tables
  - **Provisioning of information**
    - Put(key, value)
  - **Requesting of information** (search for content)
    - Get(key)
  - **Reply**
    - value
- DHT implementations are interchangeable (with respect to interface)



Distributed Hash Tables

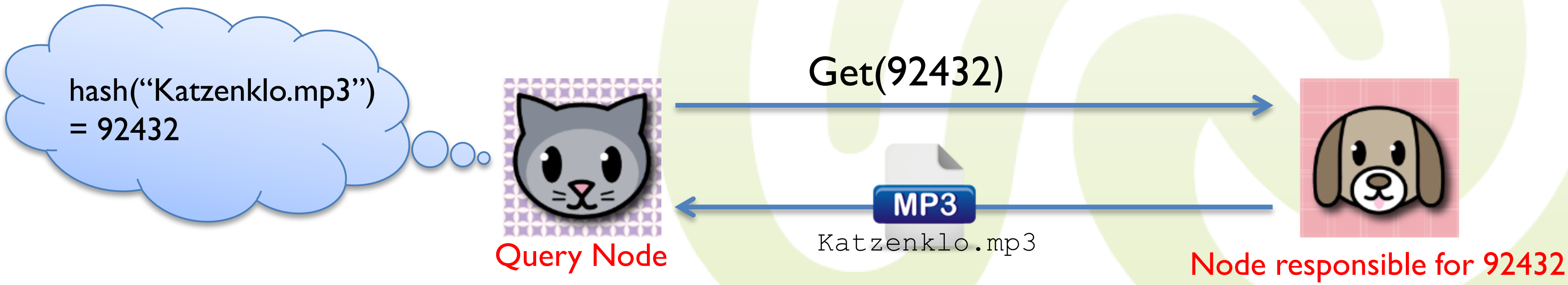
- Important design decisions
  - **How to hash objects?**
    - What to hash? How does hash space look like?
  - **Where to store** objects?
    - Direct? Indirect?
  - **How are responsibilities assigned to nodes**?
    - Random? By also hashing nodes? Evolving responsibilities? Respect load balancing and resilience issues?
  - **How is routing of queries be performed?**
    - Are routing tables needed? What should be stored in routing tables? Which topology to use for the network?
  - **How to deal with failures?**

Distributed Hash Tables

- What are good keys? What to use as values?
  - Answer is very application dependent…
- Commons **keys**
  - **Filenames** or **filepath**
    - Used in early DHT based networks for direct search by filename
  - **Keywords**
    - Hash an object multiple times using its meta data keywords
    - As used in late DHT based Gnutella networks for search
  - **Info Digests**
    - Information on files names, file length, sharing settings, …
    - Used in tracker-less BitTorrent
  - **Peer Identifications**
    - The id of the peer itself can be treated as a key
      - e.g. IP-address, MAC address, unique user ID, etc.
    - Used to hash nodes into the same address space than content
      - The later slides on **node responsibility assignments**

Distributed Hash Tables

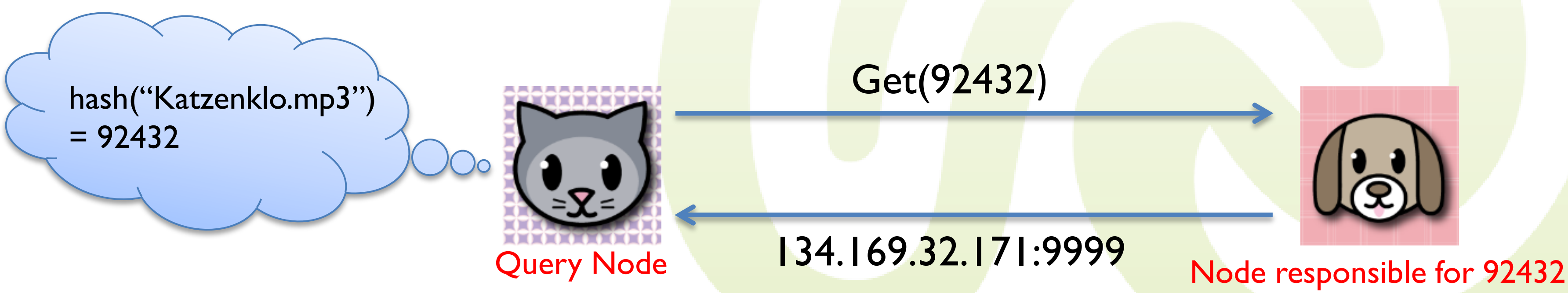
- What to use as values?
  - **Direct Storage**
    - Node stores the content of the object as value
    - When storing an object, hash its key and then **ship the object** to the responsible node and store it there
    - **Inflexible** for larger content objects
      - High network traffic
      - Loss of ownership of content
      - Problems in volatile P2P networks
        - » Join, leave, and repair operations may become expensive
      - OK for small data objects (e.g. <1KB)
    - Can be used for **storage space load balancing** in stable P2P networks



Distributed Hash Tables

– Indirect Storage

- Node stores a **link** to the object
- Content remains with the initial content provider
- DHT is used to announce the availability of a given object
- Value of the hash key-value pair usually contains **physical address** of the content provider
- **More flexible** with large content objects
  - Easy joining and leaving of nodes
  - Minimal communication overhead



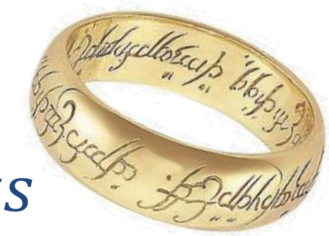
Distributed Hash Tables

- Specific examples of Distributed Hash Tables
  - **Chord** (UC Berkeley, MIT, 2001)
    - We will cover Chord in this lecture as our showcase system
  - **Pastry** (Microsoft Research, Rice University), **CAN (**UC Berkeley, ICSI), **Tapestry** (MIT)
    - With Chord, these are the big 4 academic pioneer systems 2001
    - Foundations of nearly all later DHT implementations
    - We will just briefly summarize these three
  - **Kademlia** (New York University)
    - DHT implementation used in eMule, eDonkey, LimeWire, late Gnutella, and also in some versions of BitTorrent
    - Will be briefly discussed in lecture 8
  - … and many more: P-Grid, Symphony, Viceroy, …

Distributed Hash Tables

- **Properties of DHTs**
  - Use of routing information for **efficient search** for content
  - Keys are **evenly distributed** across nodes of DHT
    - **No bottlenecks**
    - A continuous increase in number of stored keys is admissible
    - **Failure** of nodes can be **tolerated**
    - **Survival of attacks possible**
  - **Self-organizing system**
  - **Simple** and **efficient** realization
  - **Supporting a wide spectrum of applications**
    - Flat (hash) key without semantic meaning
    - Value depends on application

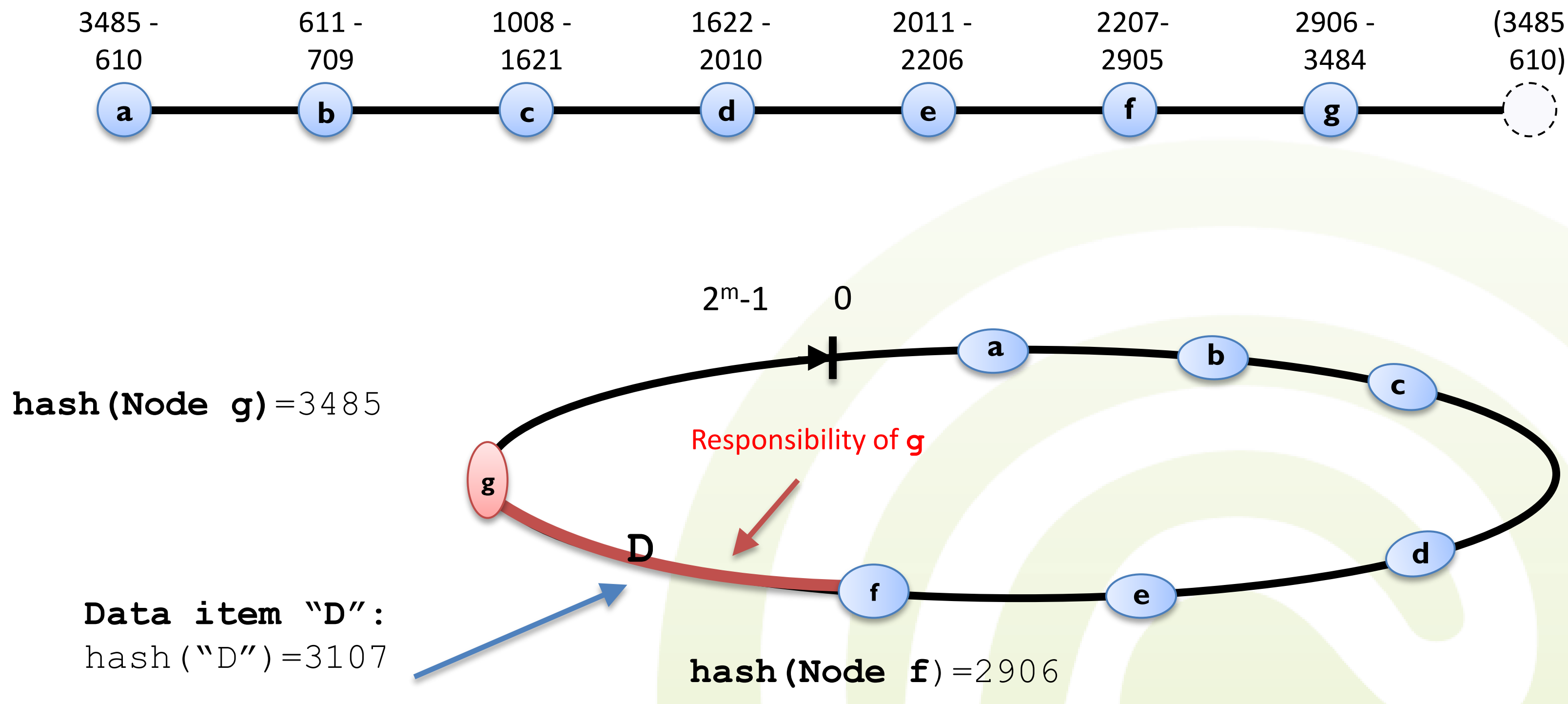
Distributed Hash Tables



- Usual assumptions and **design decisions**
  - Hash range is in $[0, 2^m - 1] \gg \#storedObjects$
  - Hash space is often treated as a **ring** (e.g. Chord)
    - Other architectures are also possible
  - Nodes take responsibility of a specific **arc** of the ring
    - Usually, this is determined by hashing the ID of the node
      - e.g. the IP address, the MAC address, etc.
      - Often, node takes responsibility of the arc ending at the hash code of its ID and beginning at the hash code of the previous node
    - i.e. nodes and data is hashed in the same hash space!
  - Each node knows at least its **predecessor** and **successor**

Distributed Hash Tables

- Example (7 nodes, range 0..4095, m=12)

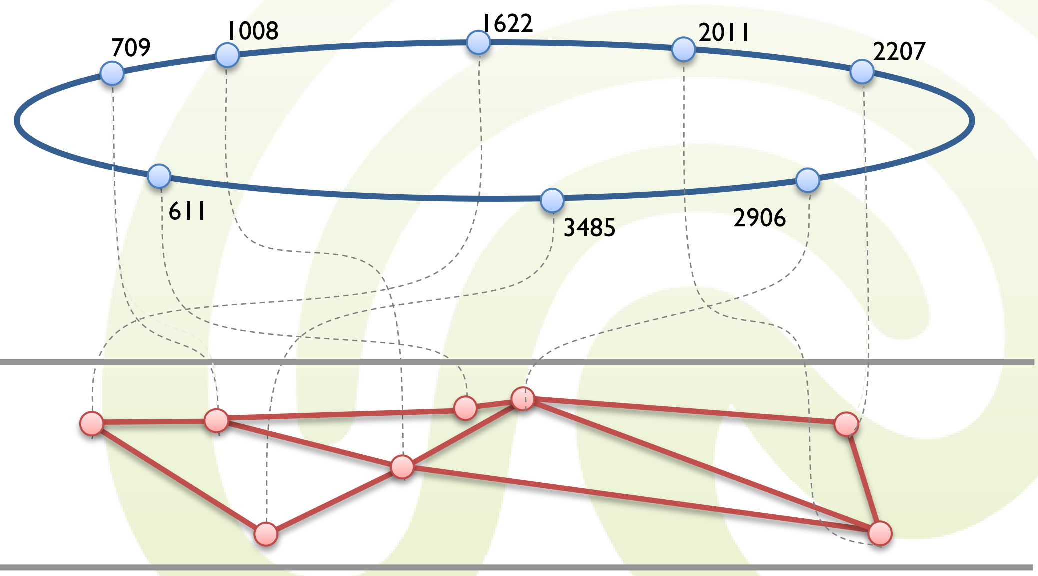


Distributed Hash Tables

- Node responsibilities are usually **agnostic** of the undelaying network topology
  - Additional heuristics can be used during responsibility assignment
    - Redundancy (multi assignments, overlapping arcs, ..)
  - Assignments must be **dynamic**
    - Nodes may join and leave the ring

Logical view of the Distributed Hash Table

Mapping on the real topology

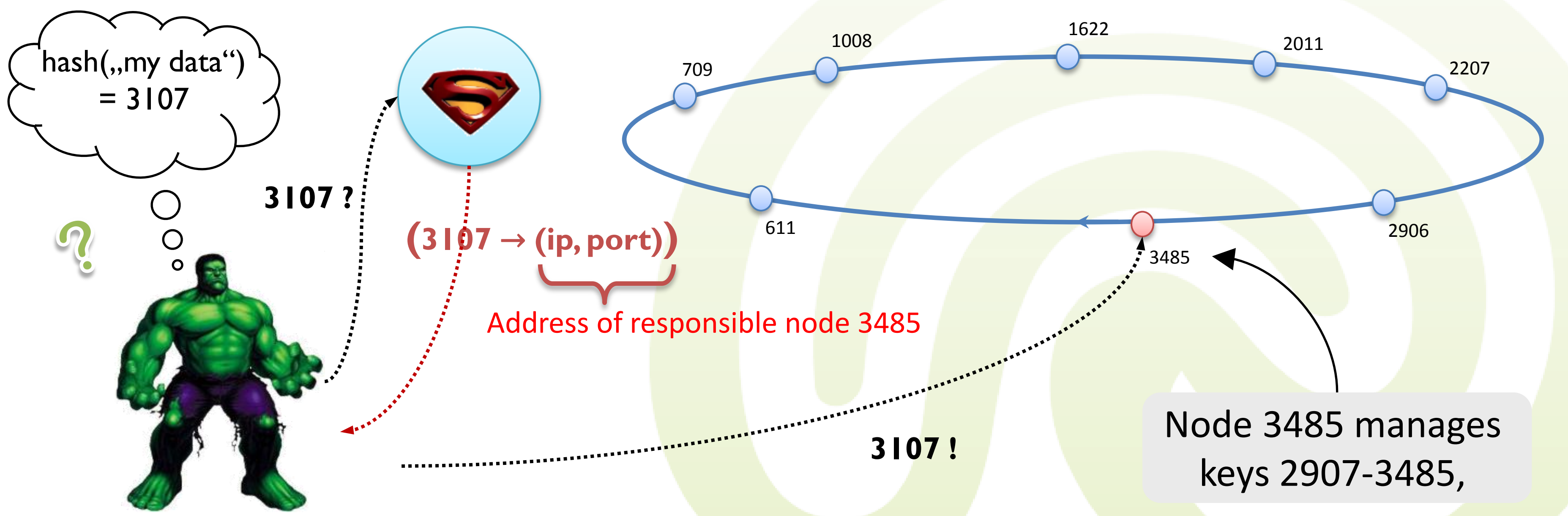


Distributed Hash Tables

- How can data be **accessed** in a DHT?
  - Start the query at any DHT node
  - **Key** of the required data is **hashed**
    - Queries use only keys, no fuzzy queries naively possible
  - **Route** the query to the node responsible for the data key hash
    - So called **key-based routing**
  - Transfer data from responsible peer to query peer

Distributed Hash Tables

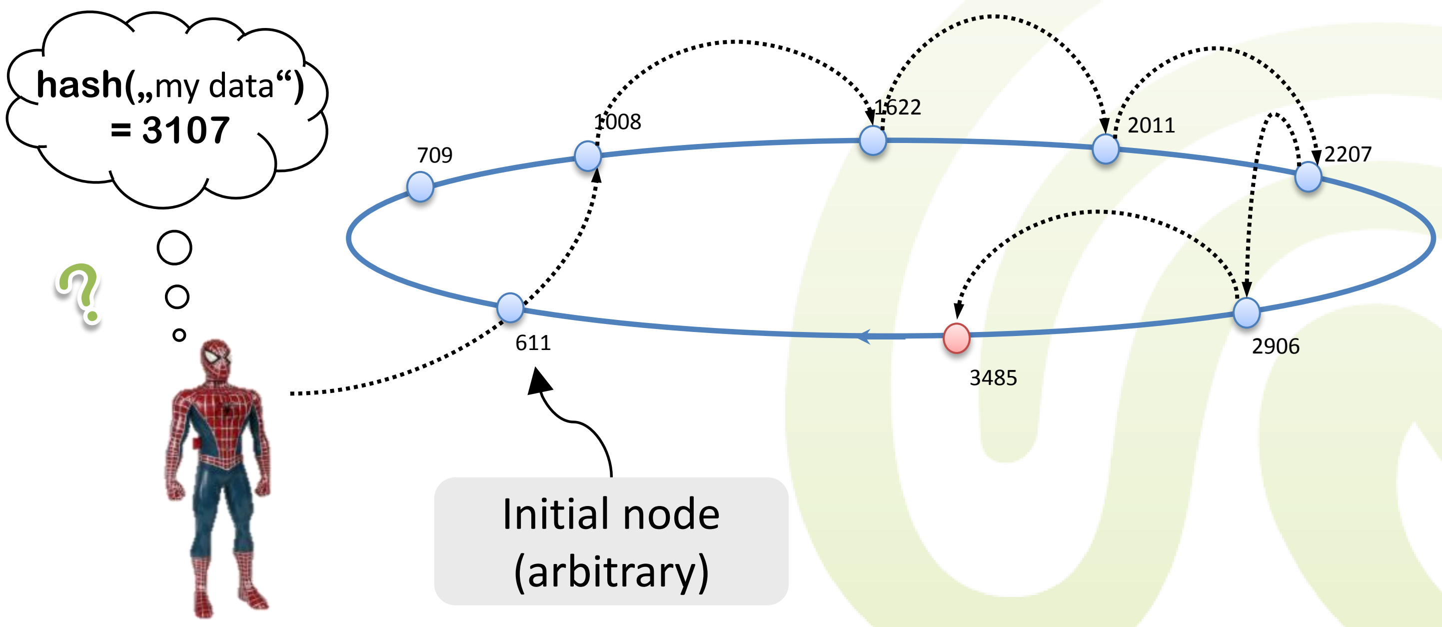
– **Direct Routing**
  - **Central server** knows the responsibility assignments
    - Also: **fully meshed ring** (i.e. each node knows each other node)
  - Shares the common disadvantages of centralized solutions
    - Single point of failure, scalability issues, etc.
    - BAD IDEA!
  - ***O(1)*** routing complexity, ***O(N)*** node state complexity



Distributed Hash Tables

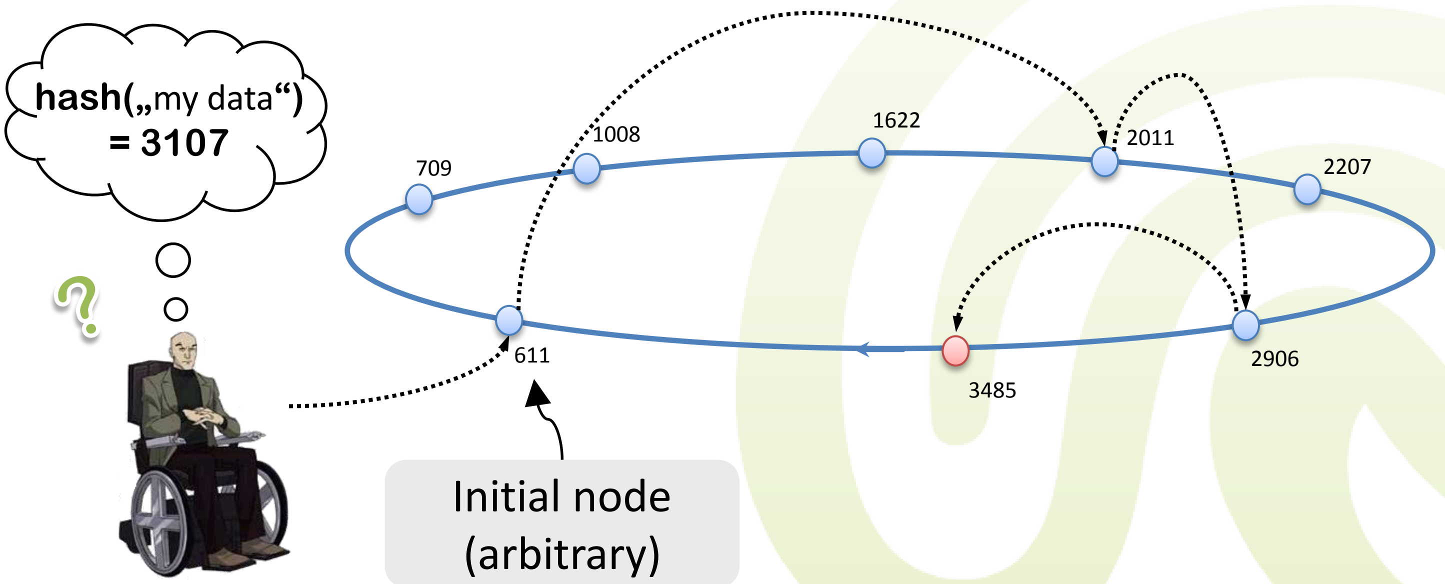
– Linear Routing

- Start query at some node of the DHT
- Route the query along the ring from successor to successor until responsible node is found
- ***O(N)*** Routing complexity, ***O(1)*** node state complexity
  - Also bad idea



Distributed Hash Tables

– Routing using **finger tables**
  - Nodes know additional nodes besides their direct ring neighbors
    – Stored in so called **finger tables** or **routing tables**
  - Routing tables can be used to reach responsible node faster
    – See later: Chord
  - ***O(log n)*** routing complexity, ***O(log n)*** node state complexity



Chord

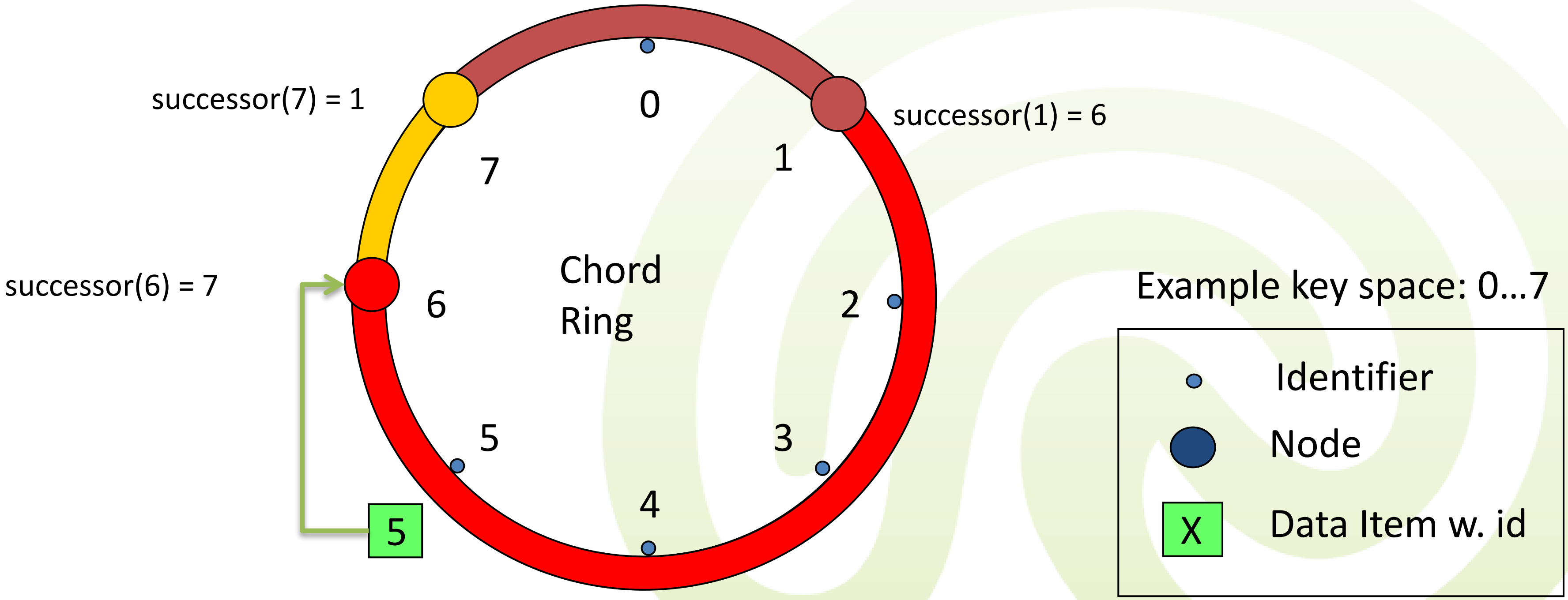
- **Chord** is one of the academic pioneer implementations of **DHTs**
  - I. Stoica, R. Morris, D.Karger, M. F. Kaashoek, H. Balakrishnan. ***Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications***. ACM SIGCOMM, San Diego, USA, 2001.
  - Uses a partially meshed **ring infrastructure**
  - **Main focus**
    - ***O(log n)*** **key-based routing**
      - Flat logical 160-Bit address space hashing both content and peers
    - **Self-organization and basic robustness**
      - Node arrivals and departures, node failures
  - Inspired many later DHT implementations and improvements
    - Better routing, alternative topologies, load balancing, replication, etc.

Chord

- **Generic DHT** interface implementation
  - Put(key, value) to insert data into Chord ring
  - Value = get(key) to retrieve data from Chord
- **Identifier generation**
  - Uses a fixed-size hash space of length $2^m - 1$
    - Limits the maximum number of peers and storable content
    - Most Chord systems use the cryptographic **SHA-1 hash function**
      - SHA 1 has 160 bit; $0 \leq id < 2^{160} \approx 1.46 * 10^{48}$
      - 10^{48} is roughly the estimated number of atoms of the Earth…
    - Data ids are usually **generated from data** itself or by an explicit data identifier
    - e.g. $objectId = sha1(object), objectId = sha1(objectName)$
  - Also, nodes are hashed by their **IP address** and **port** running the Chord application
    - e.g. $nodeId = sha1((IP\ address, port))$

Chord

- Nodes are on a modulo ring representing the full key space
  - Data is managed by clockwise next node wrt. to id
  - Each node stores its sucessor node



Chord Fingers

- **The Chord routing trick**
  - Do not only store just successor link, but also store additional nodes in a **finger table**
    - Each finger table has m entries (keyspace size: $2^m - 1$)
      - i.e. for Chord, using SHA-1, 160 entries per finger table are needed
  - **Distance** to finger nodes increases **exponentially**
    - **Distance** is measured in the **key space**, starting from the ID of the current node
    - Distance ranges from $2^0, 2^1, \ldots, 2^{m-1}$
    - The farthest finger target will cover **half of the key space distance**
  - Each finger table **entry** stores the **distance**, the hash **ID** of the target, and the **node** responsible for that ID
  - Additionally, a **neighborhood table** is needed for ring maintenance

Chord Fingers

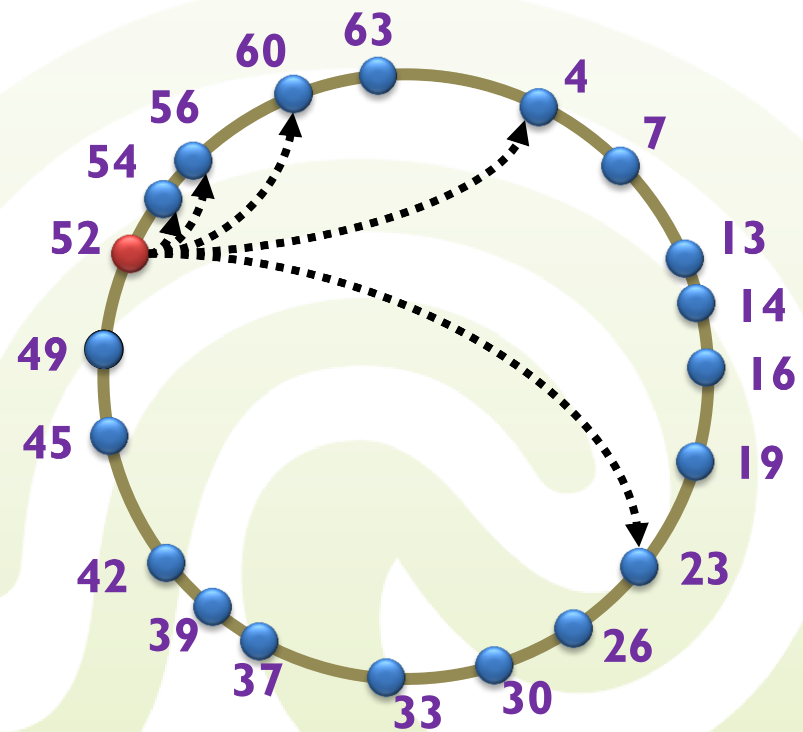
- **Chord finger table example**
  - Assume a key space size of $2^6 = 64$
    - Finger table of each node has 6 entries
    - Finger entries with logarithmic distance $i \in \{0, \dots, 5\}$
  - Build a finger table for node with current ID = 52
    - Compute the finger's target ID
      - $targetId = (currentId + 2^i) \; mod \; 2^m$
      - Find the responsible node later

i log distance	2^i distance	Target ID	Node ID
0	1	53	
1	2	54	
2	4	56	
3	8	60	
4	16	4	
5	32	20	

Chord Fingers

- **Query** the the successor node for the **resposible nodes** of all finger targets
  - Diffent finger targets may have the same responsible node

i log distance	2^i distance	Target ID	Node ID
0	1	53	***54***
1	2	54	***54***
2	4	56	56
3	8	60	60
4	16	4	4
5	32	20	23



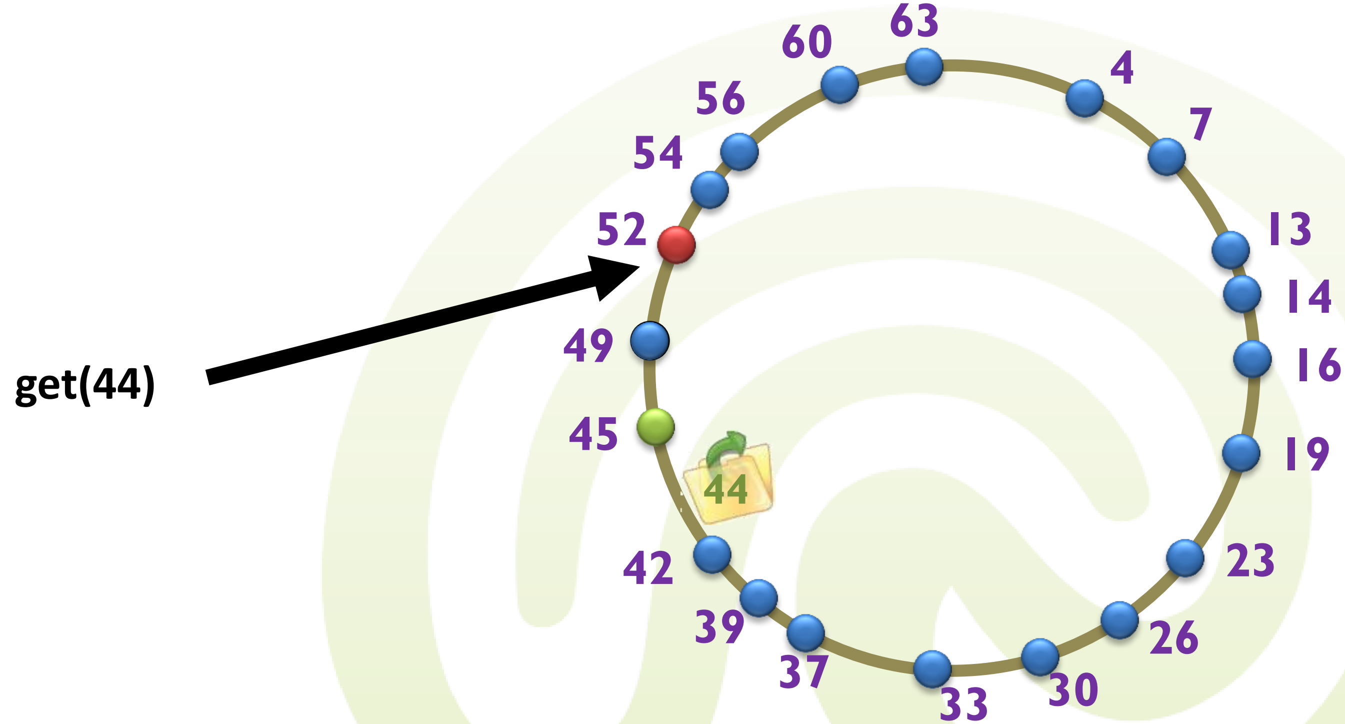
Chord Fingers

- **Querying the DHT**
  - „Which node is responsible for data with hash key ***x***?“
  - **Idea**
    - **Route** query to **finger node** with highest ID which is at most ***x***
    - That node **reroutes** the query in a recursive responsible target node is found

  - Routing complexity is in average ***O(log N)***
    - Compare to binary search!
    - For each routing step, there is a valid finger which covers **at least half the distance** to the target ID!
    - Worst case is ***O(m)***
      - Equals O(log N) for max-sized rings

Chord Routing

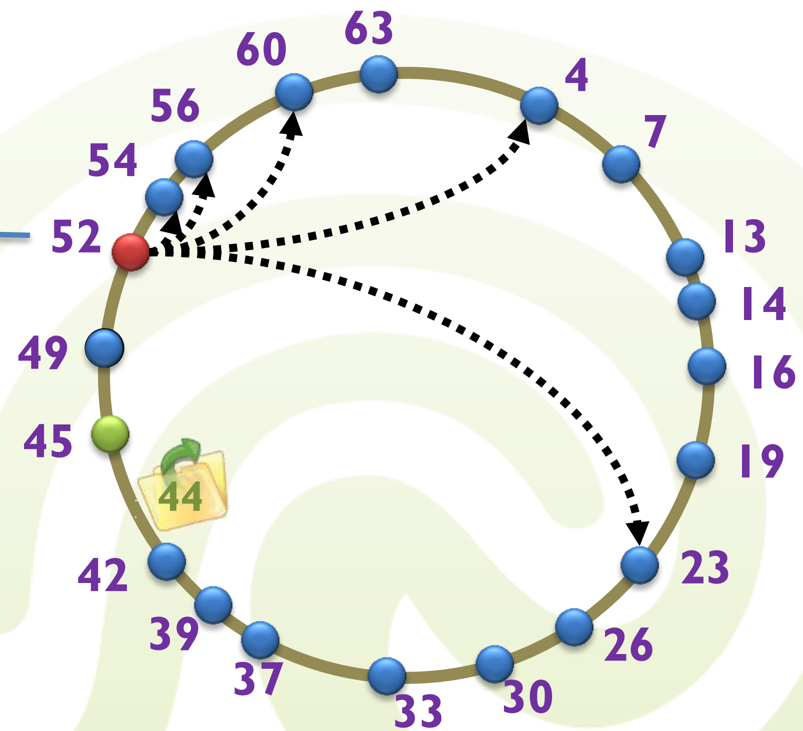
- **Example** (keyspace 2^6, 20 nodes)
  - Query for an object with hash ID 44 from node with ID 52
  - Which node is responsible?
    - Guarantee: find responsible node in at most 5 hops ($\log_2 20 \approx 4.32$)



Chord Routing

- **Example**
  - Start routing; examine finger table

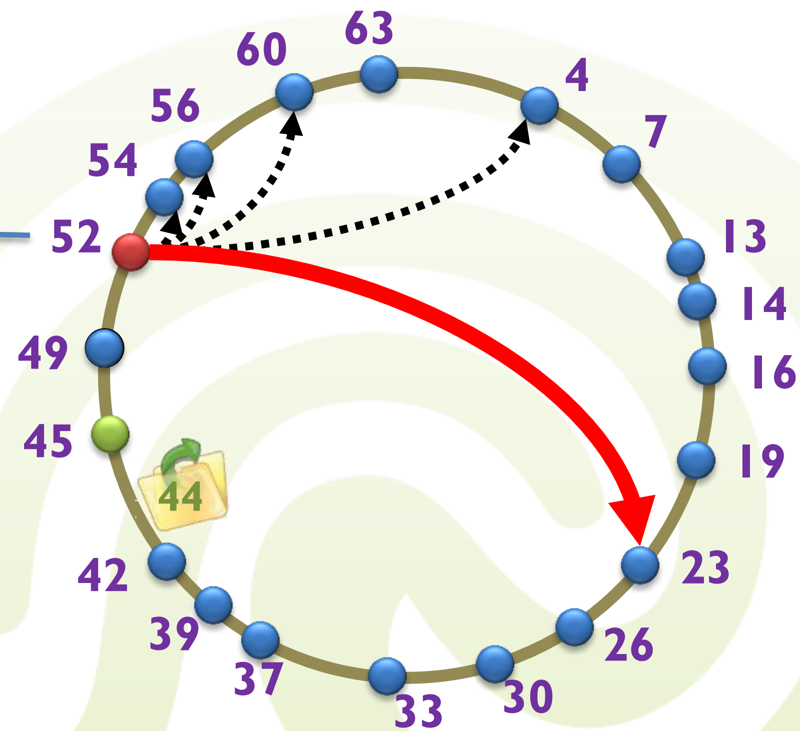
i log distance	2^i distance	Target ID	Node ID
0	1	53	54
1	2	54	54
2	4	56	56
3	8	60	60
4	16	4	4
5	32	20	23



Chord Routing

- **Example**
  - Route to most distant known node which is below lookup ID 44

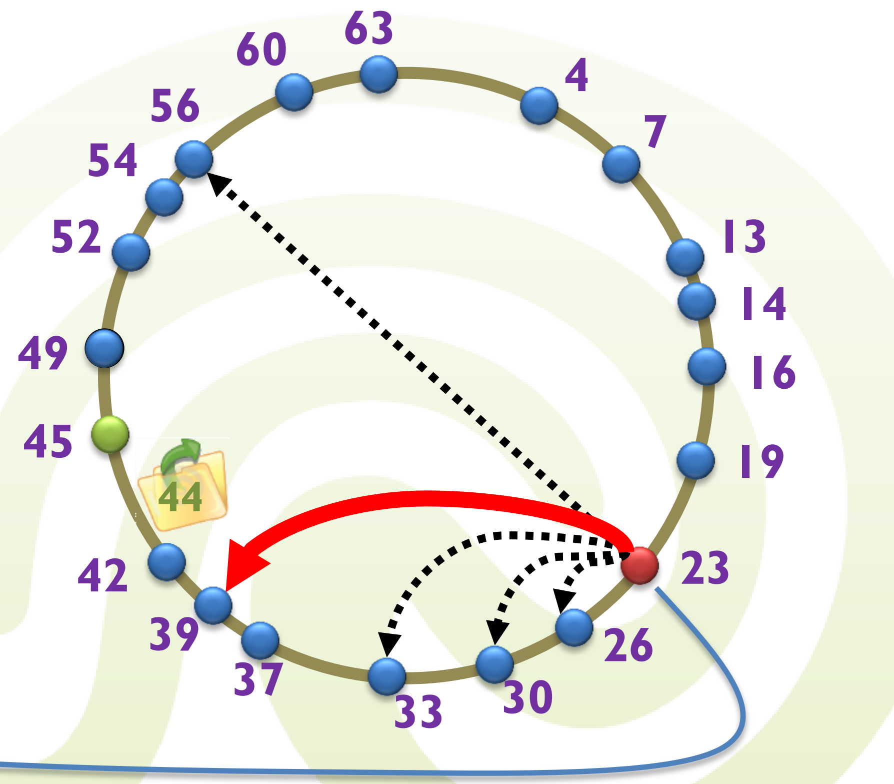
i log distance	2^i distance	Target ID	Node ID
0	1	53	54
1	2	54	54
2	4	56	56
3	8	60	60
4	16	4	4
5	32	20	23



Chord Routing

- **Example**
  - Continue routing, select most distant known node which is below lookup ID 44

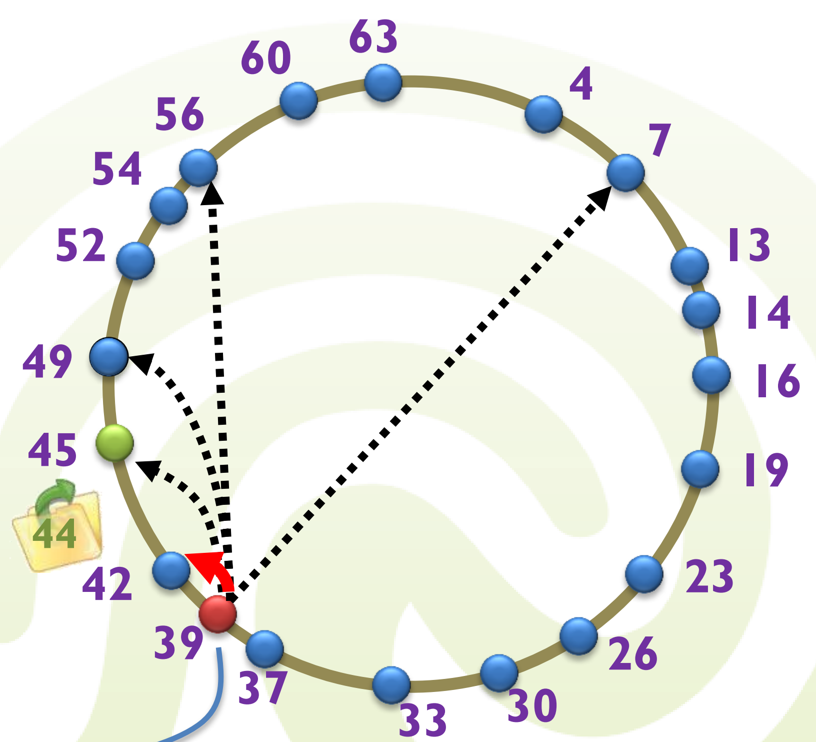
i log distance	2^i distance	Target ID	Node ID
0	1	24	26
1	2	25	26
2	4	27	30
3	8	31	33
4	16	39	39
5	32	55	56



Chord Routing

- **Example**
  - Continue routing, select most distant known node which is below lookup ID 44

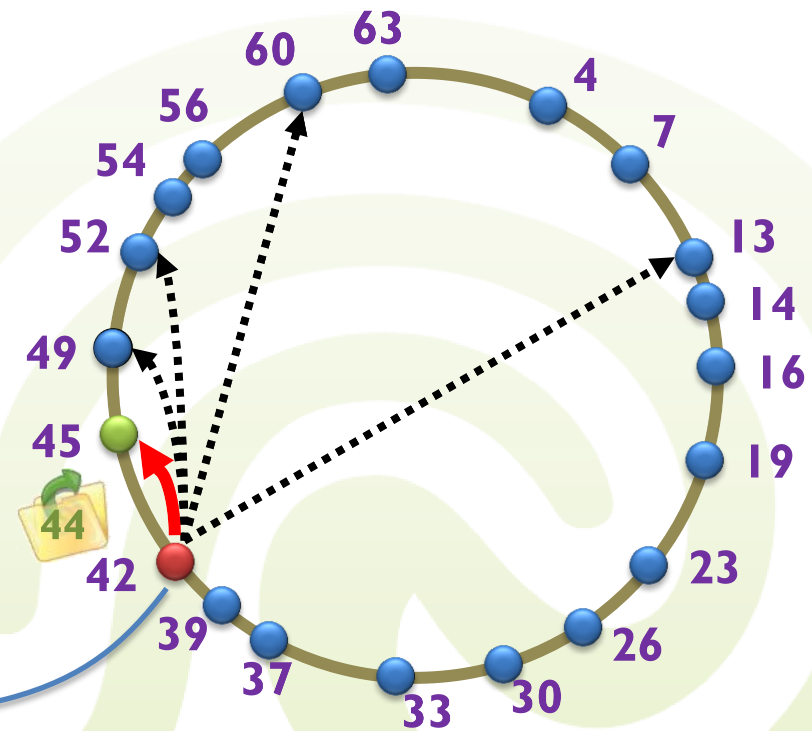
i log distance	2^i distance	Target ID	Node ID
0	1	40	42
1	2	41	42
2	4	43	45
3	8	47	49
4	16	55	56
5	32	7	7



Chord Routing

- **Example**
  - Continue routing to target node
  - Routing finished in 4 hops

i log distance	2^i distance	Target ID	Node ID
0	1	43	45
1	2	44	45
2	4	46	49
3	8	50	52
4	16	58	60
5	32	10	13



Chord Organizing

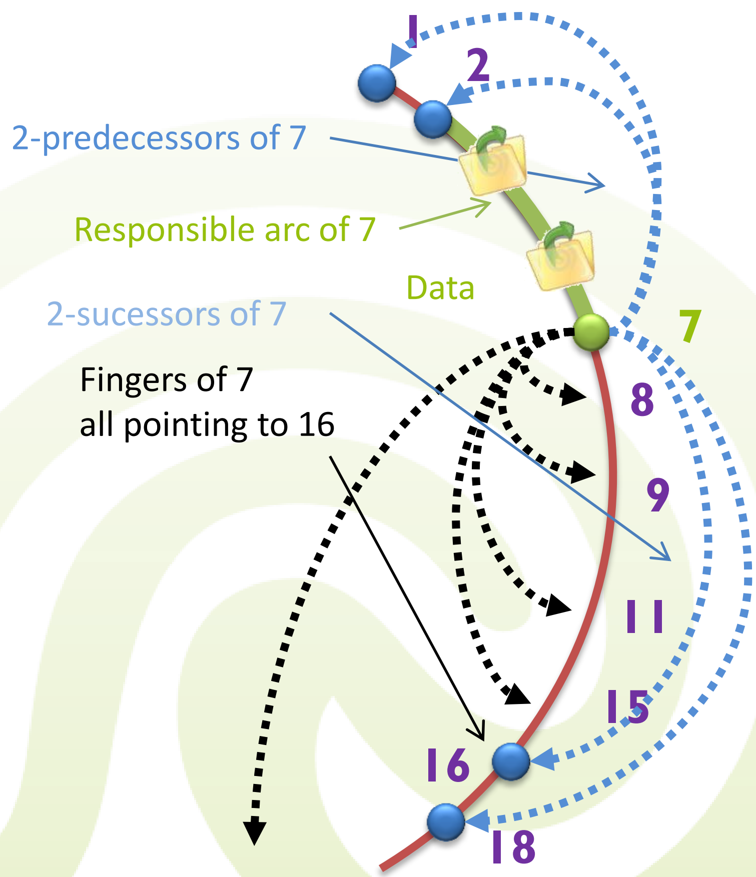
- **Chord is fully self-organized**
  - Management of new node **arrival**
  - Management of node **departure**
  - Management of node or network **failures**
- **Goal:**
  - **Routing abilities must be maintained**
    - If target node is available, it should also be reachable by routing
      - Potential routing problems can occur when nodes stored in **finger tables cannot be reached**
  - **Stored data should be resilient to failure**
    - This properties is usually ensured by the **application** using the Chord DHT and is not a property of the DHT itself
    - Also, additional data properties like **consistency**, **fairness**, **replication**, or **load balancing** is handled by **application**

Chord Organizing

- **Joining in a new node**
  - New node **hashes itself** to obtain new ID
  - Contact any DHT node via **bootstrap discovery**
  - **Contact** node responsible for new node ID
    - Via normal query routing
  - **Split arc responsibility**
    - Move respective key-value pairs from old node to new node
  - New node constructs its **finger table** and **neighborhood table**

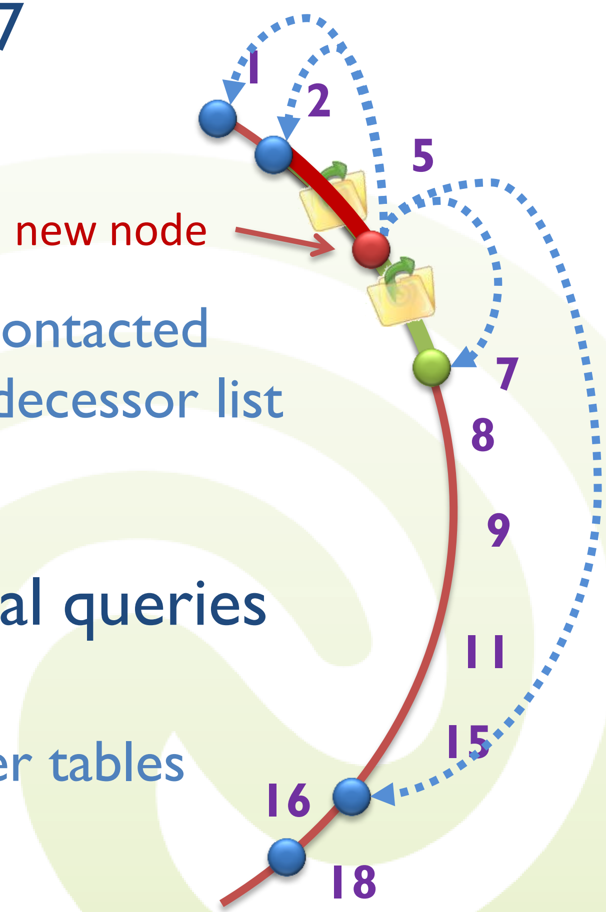
Chord Organizing

- What is the **neighborhood table**?
  - Contains the k-next **successor** and **predecessor** nodes on the ring
  - Different of finger table which is constructed by hash range distances!



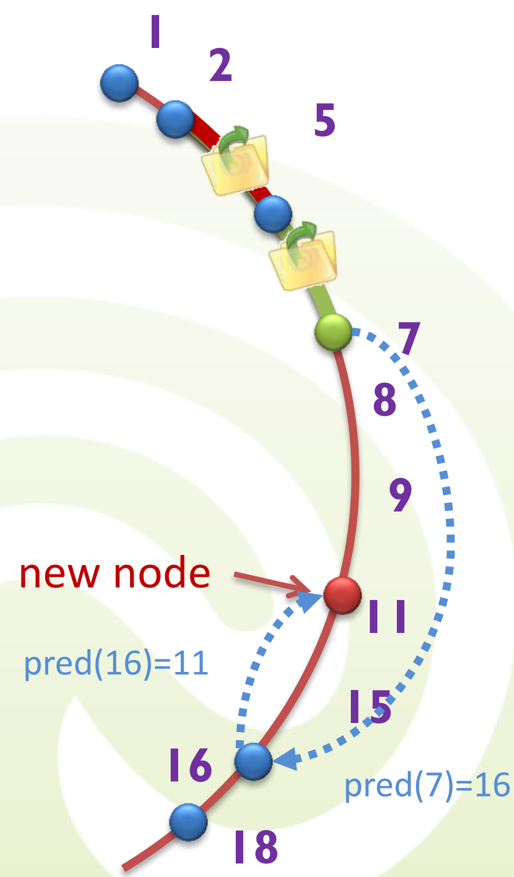
Chord Organizing

- **Joining a node** (Example)
  - New node 5 arrives
  - Takes some **responsibility** of node 7
    - Hash responsibility 3-5
    - **Copy data items** in that range
  - Construct **neighborhood table**
    - Successor is node 7 which was initially contacted
    - Query node 7 for its successor and predecessor list to construct own list
    - Update node 7 predecessor list
  - Construct **finger tables** using normal queries
  - All other nodes do nothing
    - Their respective neighborhood and finger tables are now outdated!



Chord Organizing

- **Stabilize function**
  - Each node regularly contacts its direct successor **stabilize query**
    - "Successor: is your predecessor me?"
      - i.e. ***pred(succ(x)) == x***
  - If not, a **new node** was inserted and the current neighborhood and finger table are **outdated**
    - **Repair tables** with help of direct successor
  - If direct successor cannot be contacted, **it failed**
    - **Repair tables** by contacting 2nd next successor
    - Tell 2nd next successor to take over responsibility for the failed node
      - e.g. take over the hash arc
    - Protocol fails if no successor can be contacted
      - Next time, increase size of neighborhood table



Chord Organizing

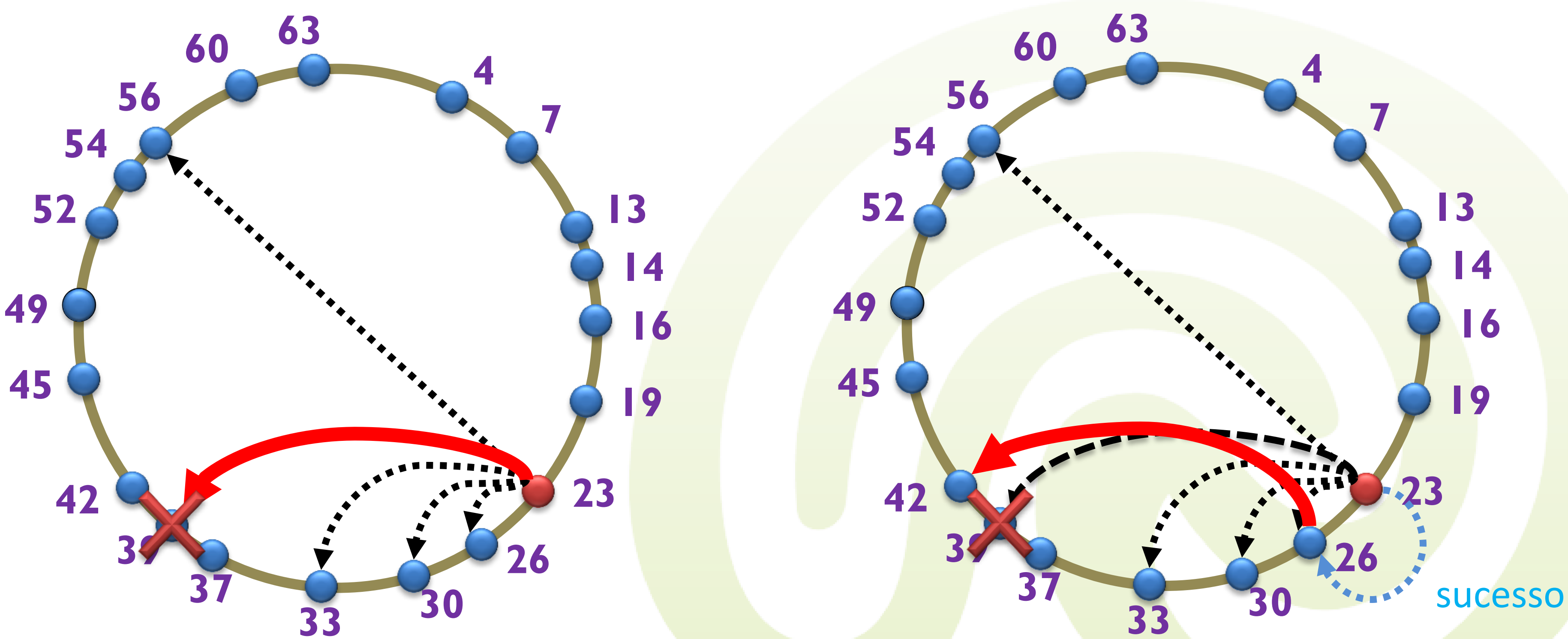
- **Removing nodes**
  - For the sake of simplicity, assume that departing nodes just disappear
    - **Departure == Failure**
  - Any node failures will be detected by **stabilize function**
    - Nodes repair their routing tables during stabilize
    - Send stabilize to next node
      - If next node does not answer, contact 2nd node
      - Use 2nd node as next node if available

Chord Organizing

- Additionally, the **stabilize function** can be used to check and repair the **finger table**
  - Randomly select a finger (less often than normal stabilize)
    - Conatct finger target
  - If target does not answer, contact the sucessor node
    - Successor contacts finger with same distance
    - That finger target has usually already repaired ist neighborhood table and knows the correct target for the broken finger

Chord Organizing

- Stabilizing fingers
  - Contact red finger node → Broken
  - Ask successor to contact same distantance-finger's
    - Either that target or predecessor becomes new finger target



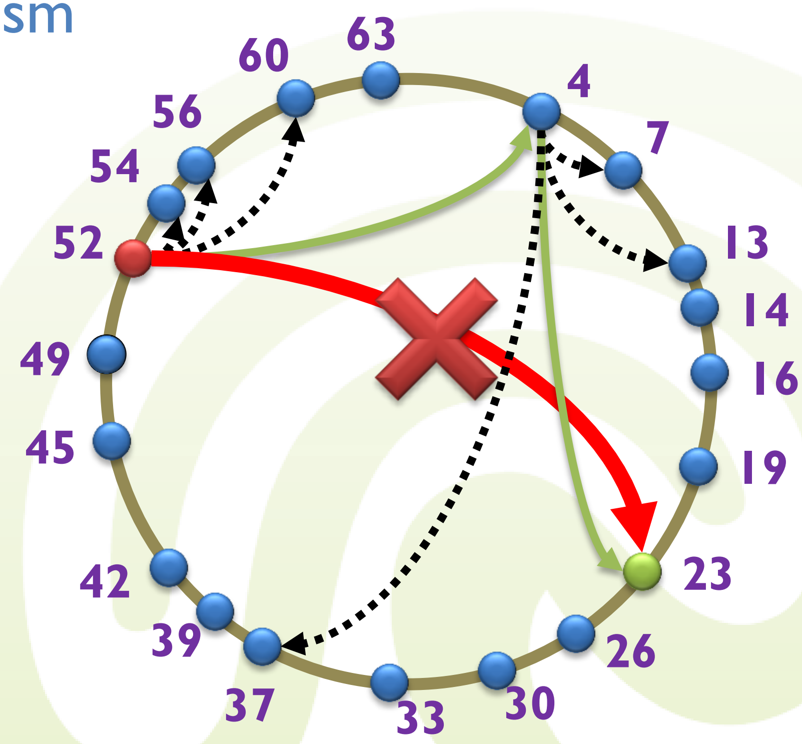
Chord Organizing

- **Maintaining routing capabilities**
  - Routing may break if finger tables are outdated
  - Finger tables can either be maintained **actively** or **passively**
  - **Active maintenance**
    - Periodically contact all finger nodes to check correctness of table information
    - In case of failure, query ring for correct information
    - **Drawback**
      - Maintenance traffic
      - Routing information in finger table may be outdated for short time intervals
    - **Stabilize function**!

Chord Organizing

– Passive maintenance

- A query cannot be forwarded to the finger
- Forward **query** to **previous finger** instead
- Trigger repair mechanism



Chord Organizing

- **Data persistence**
  - Data persistence in case of **node failure** is the responsibility of the **application**
    - Simple Chord implementations use no replication
    - Data in nodes is lost when node disconnects
  - **Scenario**
    - Robust **indirect storage**
    - Goal: as long as the data provider is available, the data should be accessible
      - i.e. query to the DHT should return the correct physical link to the data provider

Chord Organizing

- Fault tolerant data persistency can be archived by using **soft states**
- **Idea**
  - Each key-value pair stored in the DHT has a **decay timer**
  - After the decay timer is up, the key-value pair is **deleted**
    - Content not accessible anymore
  - Content providers (i.e. the application) periodically **re-publish** all their content
    - Re-publishing either **creates new key-value pairs** or **resets the decay timer** of old pairs
  - If a **node managing a key fails**, a new node will be responsible for the key after the next re-publish interval
  - If a **content provider fails**, any links pointing to it will decay soon

Dynamo

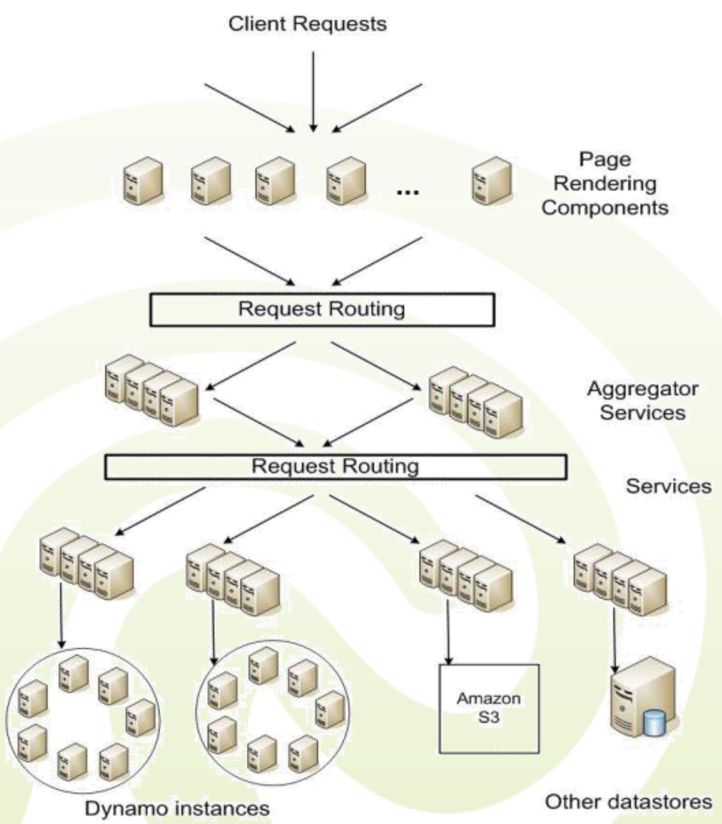
- Example System: **Amazon Dynamo**
    - G. DeCandia, D.Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, W. Vogels **"Dynamo: amazon's highly available key-value store"**, ACM SIGOPS, Stevenson, USA, 2007.
  - Amazon is one of the specialized storage solutions used at Amazon
    - Among S3, SimpleDB, Elastic Block Storage, and others
    - In contrast to the other service, it is only used internally

Dynamo

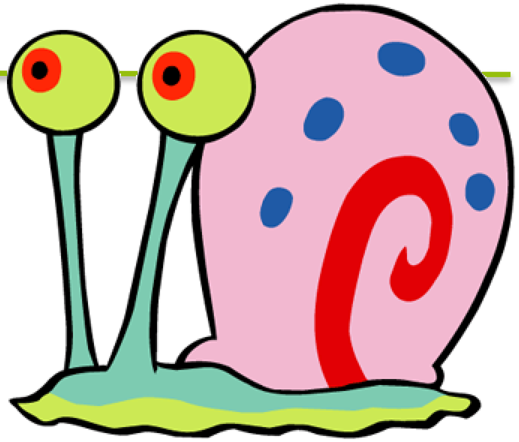
- **Amazon infrastructure**
  - Amazon uses a fully **service oriented architecture**
    - Each function used in any Amazon system is encapsulated in a service
      - i.e. shopping cart service, session management service, render service, catalog service, etc.
    - Each service is described by a service level agreement
      - Describes exactly what the service does
      - Describes what input is needed
      - Gives **quality guarantees**

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



Dynamo



- **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 SLA requirements
    - Final response is generated by aggregating all service responses
      - e.g. often, response times below 1ms for deep core services

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 uses several 10,000s servers
  - **Amazon services must always be available**

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

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

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

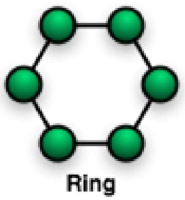
Dynamo - Design

- **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!

Dynamo - Design

- **Further assumptions**
  - All nodes in a Dynamo cluster are **non-malicious**
    - No fraud detection or malicious node removal necessary
  - Each service can set up its **own dynamo cluster**
    - Scalability necessary, but cluster don't need to scale infinitely

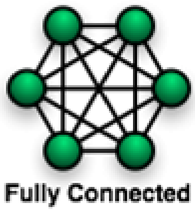
Dynamo - Design



- **Dynamo Implementation Basics**
  - 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

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!



Dynamo - Design

- 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
- 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

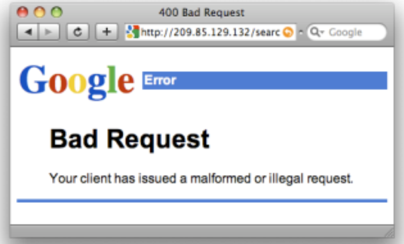
Dynamo - Consistency

- **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**

Dynamo – Requests

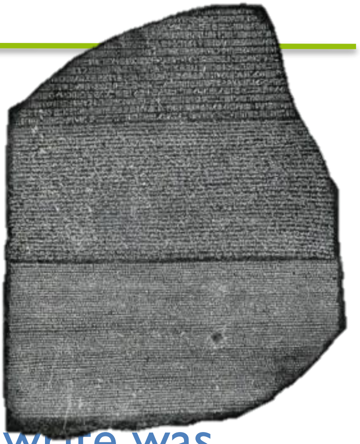
- Clients can send any $put()$ or $get()$ request to any Dynamo node
  - Typically, each client chooses a Dynamo node which is used for the whole user session
  - Responsible node is determined by either
    - Routing requests through a set of **generic load balancers**, which reroute it to a Dynamo 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

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
      - $R + W > N$

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

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

N	R	W	**Application**
3	2	2	Consistent durable, interactive user state (typical)
n	1	n	High performance read engine
1	1	1	Distributed web cache (not durable, not consistent, very high performance)

Dynamo - Consistency

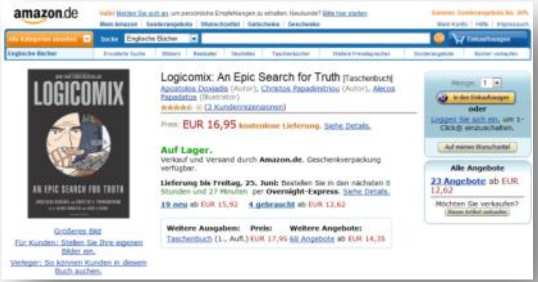
- Theoretically, the same data can reside in **multiple versions** within the system
  - Multiple causes
    - **No failure**, asynchronous update in progress
      - Replicas will be eventual consistent
      - In rare case, branches may evolve
    - **Failure**: ring partitioned or massive node failure
      - 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

Dynamo - Consistency

- **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

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 **CD** to his cart
    - New cart is replicated within the current partition



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
- 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

Dynamo – Vector Clocks

- Version numbers are stored using **vector clocks**
  - 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

Dynamo – Vector Clocks

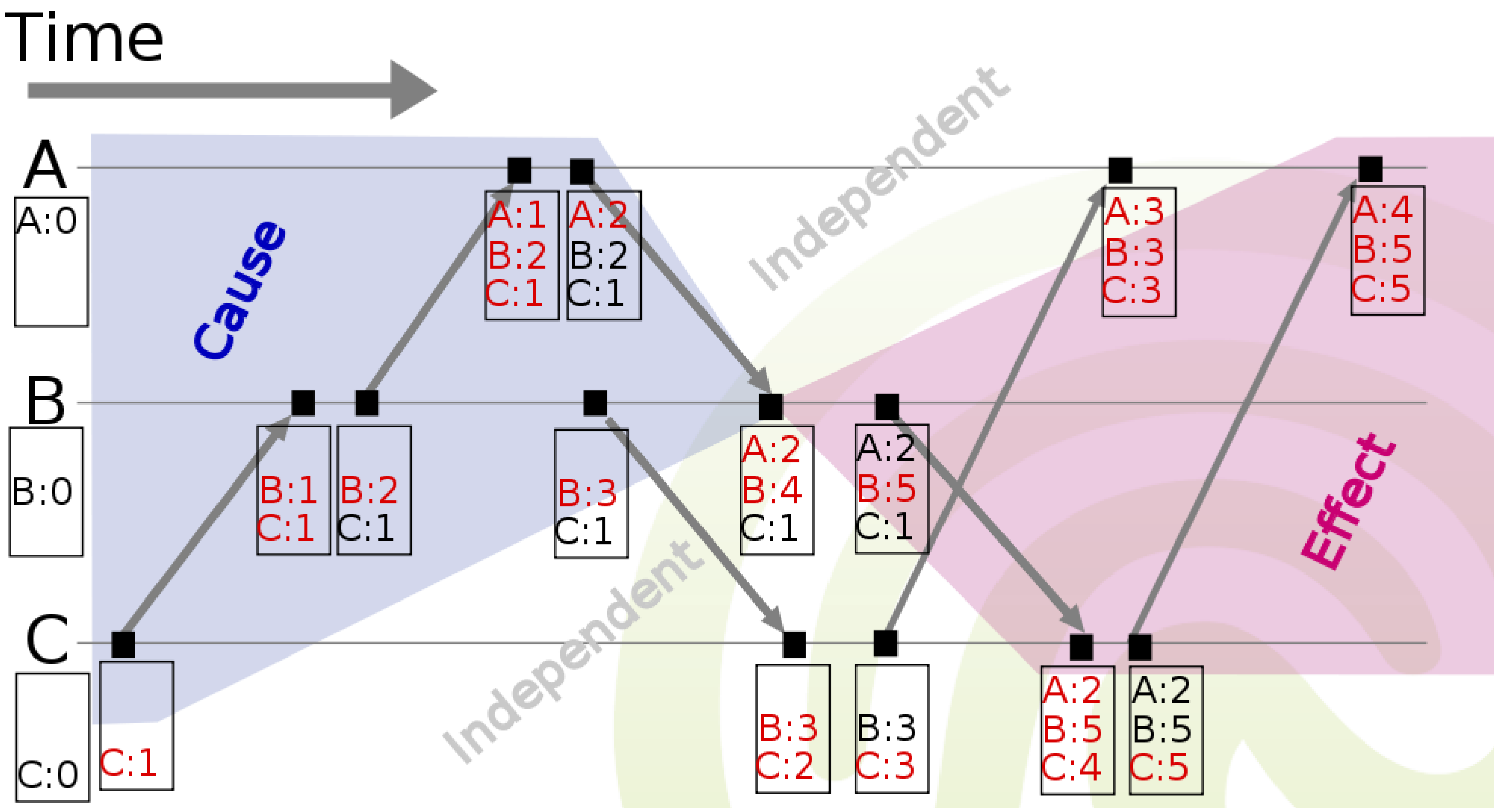
- Base idea vector clocks
  - Each node / process maintains an individual logical clock
    - Initially, all clocks are 0
    - A global clock can be constructed by concatinating all logical clocks in an array
  - Every node stores a local "**smallest possible values" copy** of the global clock
    - Contains the last-known logical clock values of all related other nodes

Dynamo – Vector Clocks

- 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 nodes 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

Dynamo – Vector Clocks

- **Vector clock**



Dynamo – Vector Clocks

- 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

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 = $\{A: 1, C: 1\}$

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

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 ≥ 10

Dynamo – Consistency

- 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.

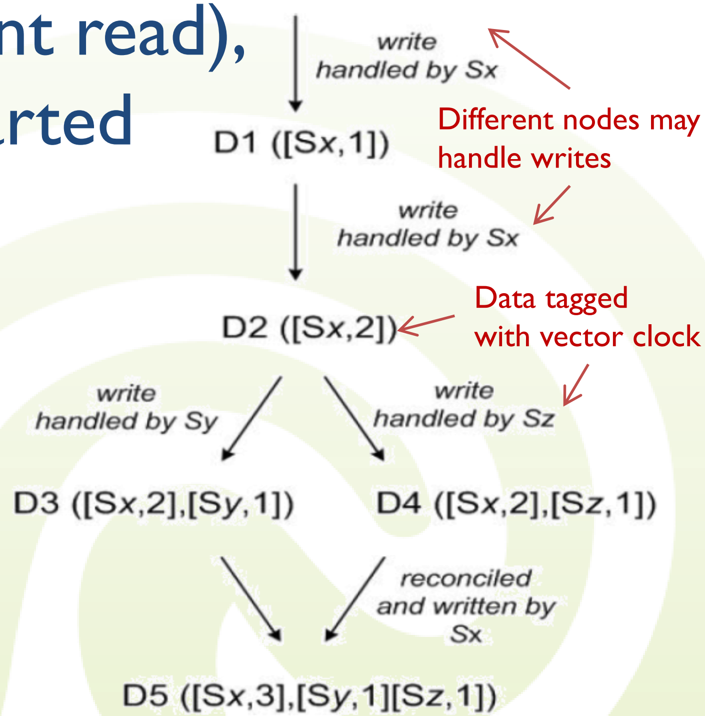
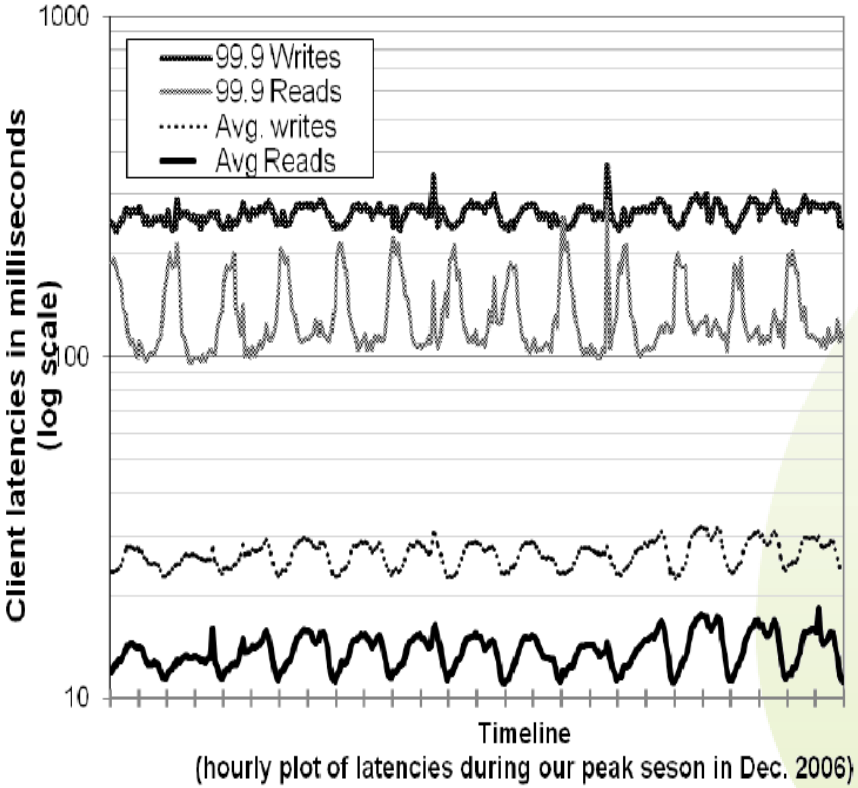
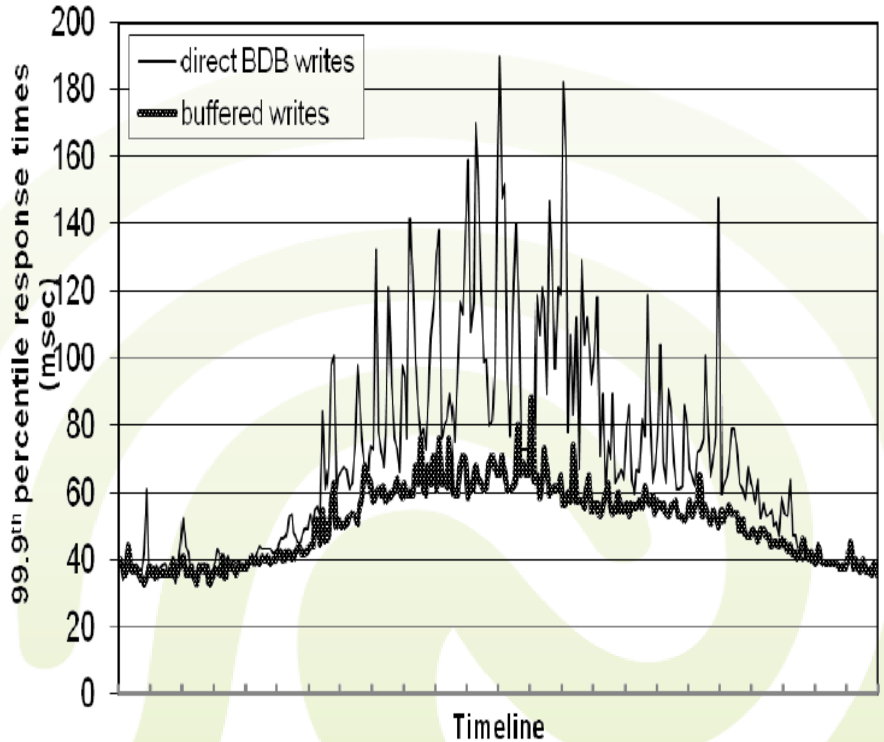


Figure 3: Version evolution of an object over time.

Dynamo - Evaluation

- Test results for response requirement is 300ms for any request (read or write)





Dynamo - Evaluation

- **Load distribution**

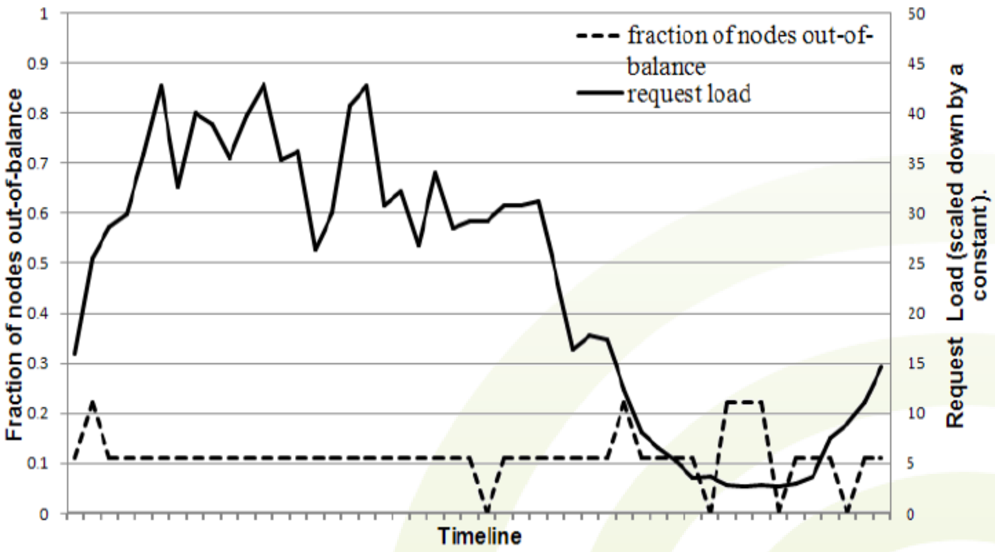


Figure 6: Fraction of nodes that are out-of-balance (i.e., nodes whose request load is above a certain threshold from the average system load) and their corresponding request load. The interval between ticks in x-axis corresponds to a time period of 30 minutes.

Dynamo - Evaluation

- **Consistency** vs. **Availability**
  - 99.94% one version
  - 0.00057% two versions
  - 0.00047% three versions
  - 0.00009% 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 the 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)

Dynamo - Summary

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**

Google

Detour

- 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

Google

Detour

- Privately held until 2004
  - Now NASDAQ: GOOG
  - Market capitalization of over 140 billion USD
  - 2009 revenue of 23.7 billion USD (6.5 billion profit)

Google

Detour

- 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

Google

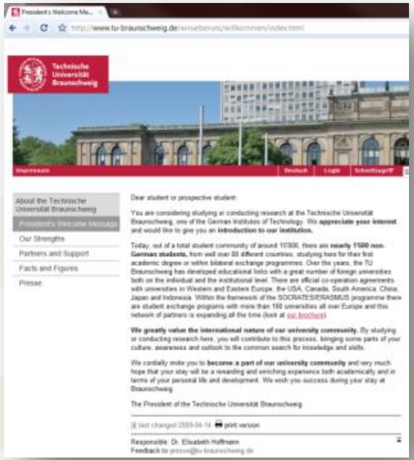
Detour

- Currently, Google offers a multitude of services
  - Google Search
  - Google Mail
  - Google Maps
  - Google Earth
  - Google Documents
  - Picasa Web Album
  - etc.
- Thus, Google hosts and actively uses several Petabytes of data!

Google Challenges

Detour

- Google needs to **store** and **access** lots of (semi-)structured data
  - **URLs** and their contents
    - Content, meta data, links, anchors, pageranks, etc.



  - **User data**
    - User preferences, query history, search results
  - **Geographic information**
    - Physical entities (shops, restaurants, etc), roads, annotations, POIs, satellite images, etc.

Bigtable

- **Bigtable**
  - F. Chang et al, "***Bigtable: A Distributed Storage System for Structured Data***", ACM Transactions on Computer Systems (TOCS), Vol 26, Iss 2, June **2008**
  – Bigtable is a high-performance proprietary **database system** used by multiple Google services
    - e.g. used in Google Maps, Google Books, Google Earth, Gmail, Google Code, etc.
    - Uses an abstracted and very flexibly row and column storage model
    - Is based on versioning for updates

Bigtable Requirements

- Originally designed for storing Google's **Web index**
- Special requirements
  - Processes **continuously** and **asynchronously update** different pieces of data
    - i.e. continuous Web crawling
    - Store version, usually access just newest one
    - Multiple version can be used to examine change of data in time
  - Very **high read / write rates** necessary
    - Millions per seconds
  - Support efficient **scanning** of interesting data subsets

Bigtable Requirements

- Additional requirements as usual for web-scale applications
  - Fault tolerant, persistent
  - Use cheap hardware
  - Scale to huge sized infrastructures
    - Support **incremental scaling**
    - Thousands of servers
      - Terabytes of in-memory data
      - Petabytes of disk-based data
  - Self-managing
    - Servers auto-load balance
    - Servers can be dynamically added and removed

Bigtable Cells

- Each distributed Bigtable cluster is responsible for the data of one or multiple applications
  - Called a “cell”
    - Several hundred cells are deployed
    - Cell size range from 10-20 up to thousands machines
    - In 2006, the largest cell was 0.5 PB
      - Now it is probably much larger…

Bigtable Environment

- Bigtable heavily relies on additional systems and concepts
  - **Google File System (GFS)**
    - A distributed and fail-safe file system
    - Physically stores Bigtable data on disks
      - S. Ghemawat, H. Gobioff, S.T. Leung. **"The Google File System",** ACM Symp. Operating Systems Principles, Lake George, USA, 2003
  - **Google Chubby**
    - A distributed lock manager, also responsible for bootstrapping
      - M. Burrows. **"The Chubby Lock Service for Loosely-Coupled Distributed Systems",** Symp. Operating System Design and Implementation, Seattle, USA, 2006
  - **Google MapReduce**
    - Programming model for distributing computation jobs on parallel machines
      - J. Dean, S. Ghemawat. **"MapReduce: Simplified Data Processing on Large Clusters",** Symp. Operating System Design and Implementation, San Francisco, USA, 2004

Bigtable Implementation

- Bigtable is a "database" especially designed to run ontop of GFS
  - Bigtable data model also focuses on appends
    - Assumption: rows are frequently added, but rarely updated
    - Row "updates" will just result in new rows with a different timestamp
  - GFS takes care of replication and load-balancing issues
- To accommodate for Google's applications, Bigtable uses a very flexible data model

Bigtable: Data Model

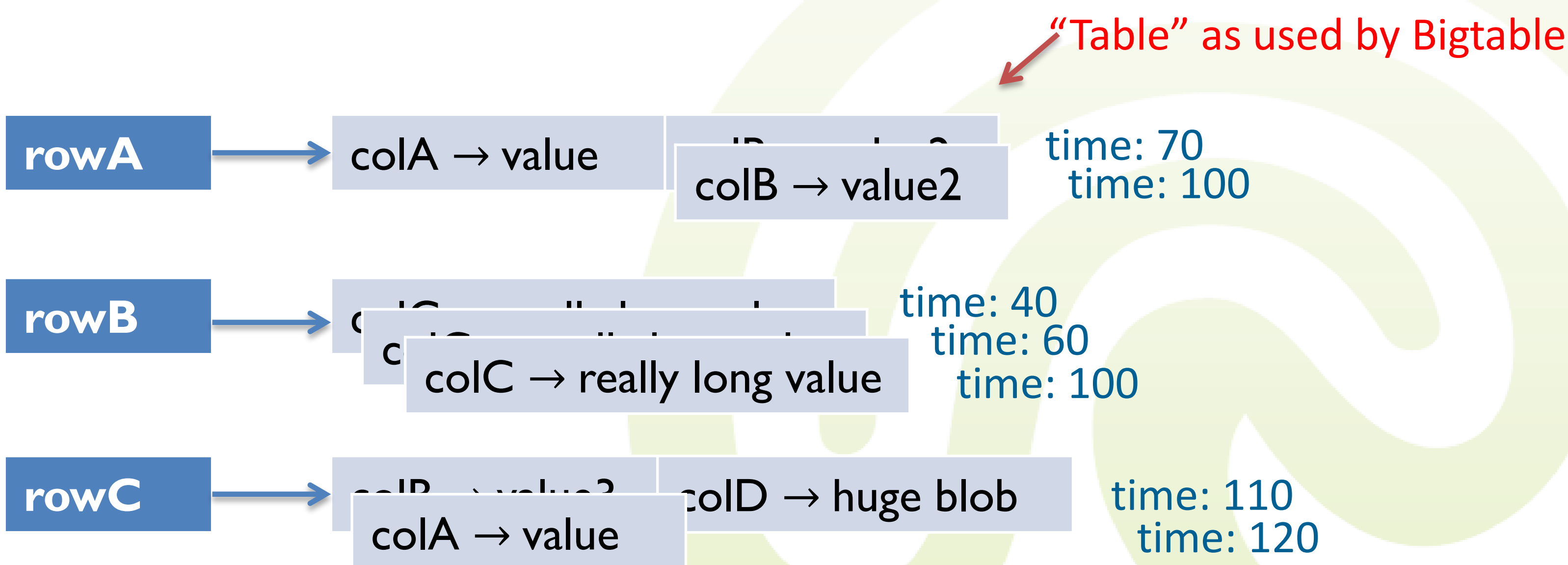
- Don't think of Bigtables as spreadsheet or traditional DB table
  - Unfitting name….
  - e.g. not each row has a fixed size of attributes
    - Not: Each column has a data type
    - Not: Missing values denoted as null

Table as NOT used by Bigtable

	colA	colB	colC	colD
rowA				NULL?
rowB	NULL?			
rowC			NULL?	
rowD				

Bigtable: Data Model

- Instead, Bigtable implements a **multi-dimensional sparse map**
  - Think of columns just as available tags
    - "Cells" are referenced by $(row\_name, col\_name, timestamp)$
  - Each row can use just some columns and story any value
    - Columns are just roughly typed, i.e. binary, string, numeric, …



Bigtable: Data Model

- **Rows**
  - Each row has a **unique name**
    - Name is just an arbitrary **string**
      - e.g. “www.ifis.cs.tu-bs.de”
  - Each access to a **row** is **atomic**
    - Load and store whole rows
  - Rows are **ordered lexicographically**
    - Idea: after partitioning the table, lexicographically similar rows are within the same or a nearby fragment
      - e.g. “www.ifis.cs.tu-bs.de” is close to “www.ifis.cs.tu-bs.de/staff”
  - **Rows will never be split** during partitioning

Bigtable: Data Model

- **Columns**
  - Each column has a two-level name structure
    - **Family** name and **qualifier** name
      - e.g. <family:qualifier>
  - All **column families** must be created explicitly as part of schema creation
    - Columns within a family have usually a similar type
    - Data of a row within a family are often stored and compressed together
  - **Individual columns** can be used by application freely and **flexibly**
    - Individual columns are not part of schema creation
      - Flexible data model
  - **Aims**
    - Have a few (max. 100 (!)) column families which rarely change
    - Let application create columns as needed

Bigtable: Data Model

- **Timestamps**
  - Of each cell, different **versions** are maintained with their respective timestamps
    - 64 Bit integers
  - **Updates** to a cell usually create a new version with the current system time as timestamp
    - But timestamp can also be set explicitly by application
  - During **column family** creation, **versioning options** are provided
    - Either "keep *n* copies" or "keep versions up to the age of *n* seconds"
  - Typical queries ask for timestamp ranges

Bigtable: Data Model

- The base unit of load balancing and partitioning are called **tablets**
  - i.e. **tables** are **split** in multiple **tablets**
  - Tablets hold a **contiguous** range of rows
    - Hopefully, row ordering will result in locality
  - Tablets are **disjoint**
    - No overlapping value ranges
  - **Tablets** are rather large (1GB by default) and are later stored in **GFS**
    - i.e. tablets will usually have multiple GFS chunks
    - Tablets need to contain full rows
      - A single row should not exceed several hundred MB such that it will fit into a tablet…

Bigtable - API

- Bigtable provides only very simple **native API interfaces** to applications
    - e.g. in C++ or Python
  - No complex query language like SQL
  - API can
    - Create and delete tables and column families
    - Modify cluster, table, and column family metadata such as access control rights,
    - Write or delete directly addressed values in Bigtable
      - Supports just single row transactions (i.e. read-modify-write)
      - No multi-row transactions
    - Look up values from individual rows
    - Iterate over a subset of the data in a table,
      - Can be restricted to certain column families or timestamps
      - Relies on regular expressions on row and columns names

Bigtable - API

Summary

- **Recap**
  - Semi-flexible schemas are supported
  - A table consist of named **rows** and **columns**
    - All data cells are **versioned** with **timestamps**
    - Columns are grouped in **column families** which are defined in the schema
      - Families are usually stable during application life
    - Columns can be dynamically used and added by applications as they seem fit
    - As a result, table is **very sparse**
      - i.e. it resembles a **multi-dimensional map**
  - Tables are broken down into **tablets**
    - Tables hold a **continuous** and **ordered non-overlapping row name** range
    - **Horizontal fragmentation**

Bigtable

Detour

- Application 1: **Google Analytics**
  - Enables webmasters to analyze traffic pattern at their web sites.
  - Provides statistics such as:
    - Number of unique visitors per day and the page views per URL per day
    - Percentage of users that made a purchase given that they earlier viewed a specific page
  - How is it done?
    - A small JavaScript program that the webmaster embeds in their web pages
    - Every time the page is visited, the program is executed
    - Program records the following information about each request
      - User identifier
      - The page being fetched

Bigtable

Detour

- Application 2: **Google Earth & Maps**
  - Functionality: Storage and display of satellite imagery at different resolution levels
  - One Bigtable stores raw imagery (~ 70 TB):
    - Row name is a geographic segments
      - Names are chosen to ensure adjacent geographic segments are clustered together
    - Column family maintains sources of data for each segment.
  - There are different sets of tables for serving client data, e.g., index table

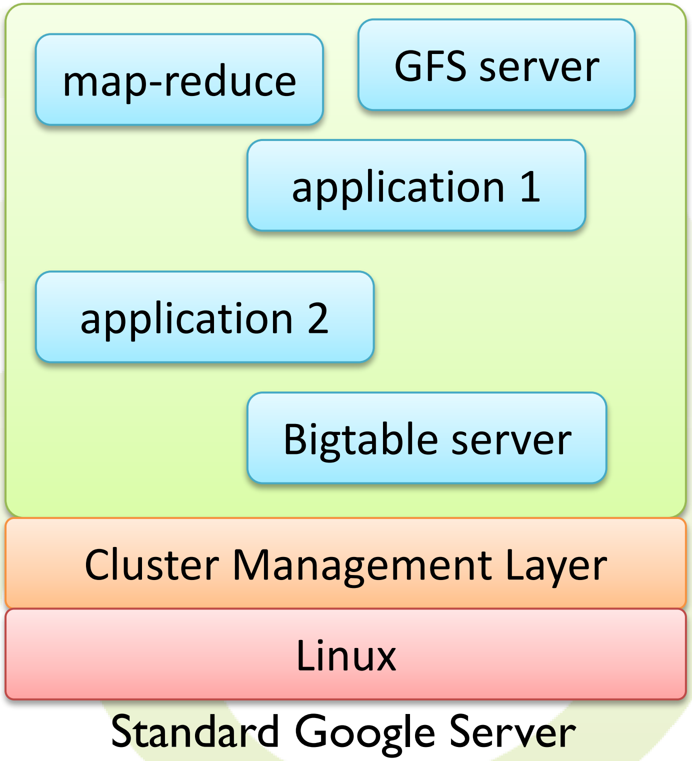
Bigtable

Detour

- Application 3: **Personalized Search**
  - Records user queries and clicks across Google properties
  - Users browse their search histories and request for personalized search results based on their historical usage patterns
  - One Bigtable
    - Row name is userid
    - A column family is reserved for each action type, e.g., web queries, clicks
    - User profiles are generated using MapReduce.
      - These profiles personalize live search results
    - Replicated geographically to reduce latency and increase availability

Bigtable: Implementation

- Implementing Bigtable
  - Bigtable runs on standard Google server **nodes**
  - Each server node usually runs multiple **services**
    - Some application server instances
      - e.g. a web renderer, a crawler, etc.
    - A map-reduce worker
      - Can accept any map-reduce request by a scheduler when idling
    - A GFS chunk server instance
    - A Bigtable server



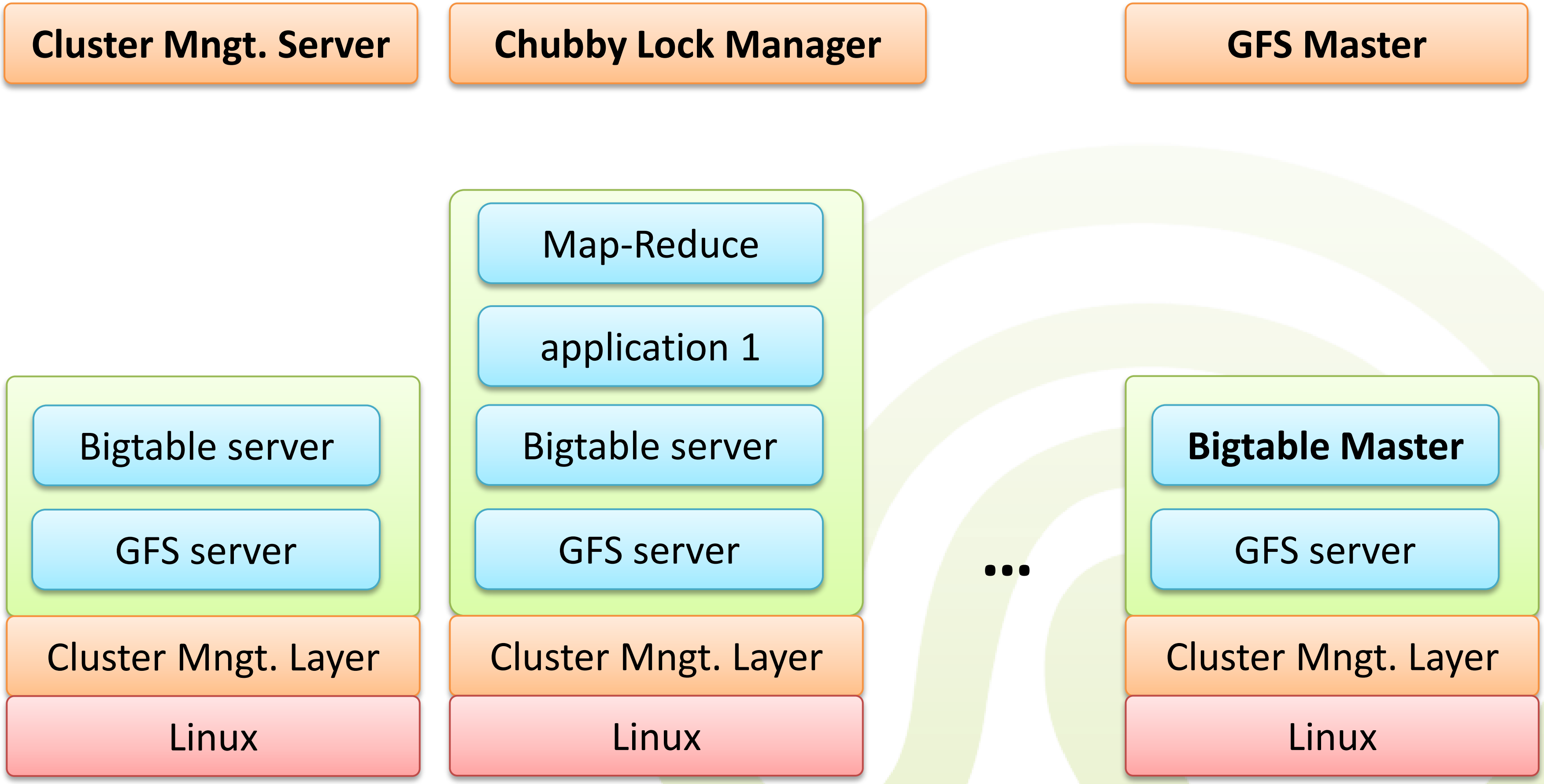
Bigtable: Implementation

- Usually, a Bigtable cluster consists of multiple **tablet servers** and a **single master server**
  - **Master** controls and maintains tablet servers
    - Assigns and migrates tablets
    - Controls garbage collection and load balancing
    - Maintains schema
    - Clients usually never contact master

  - **Tablet servers** are responsible for tablets
    - Can be dynamically added and removed
    - Master controls tablet migrations
    - Clients know the tablet server responsible for their data

Bigtable: Implementation

- **Typical Bigtable cell**

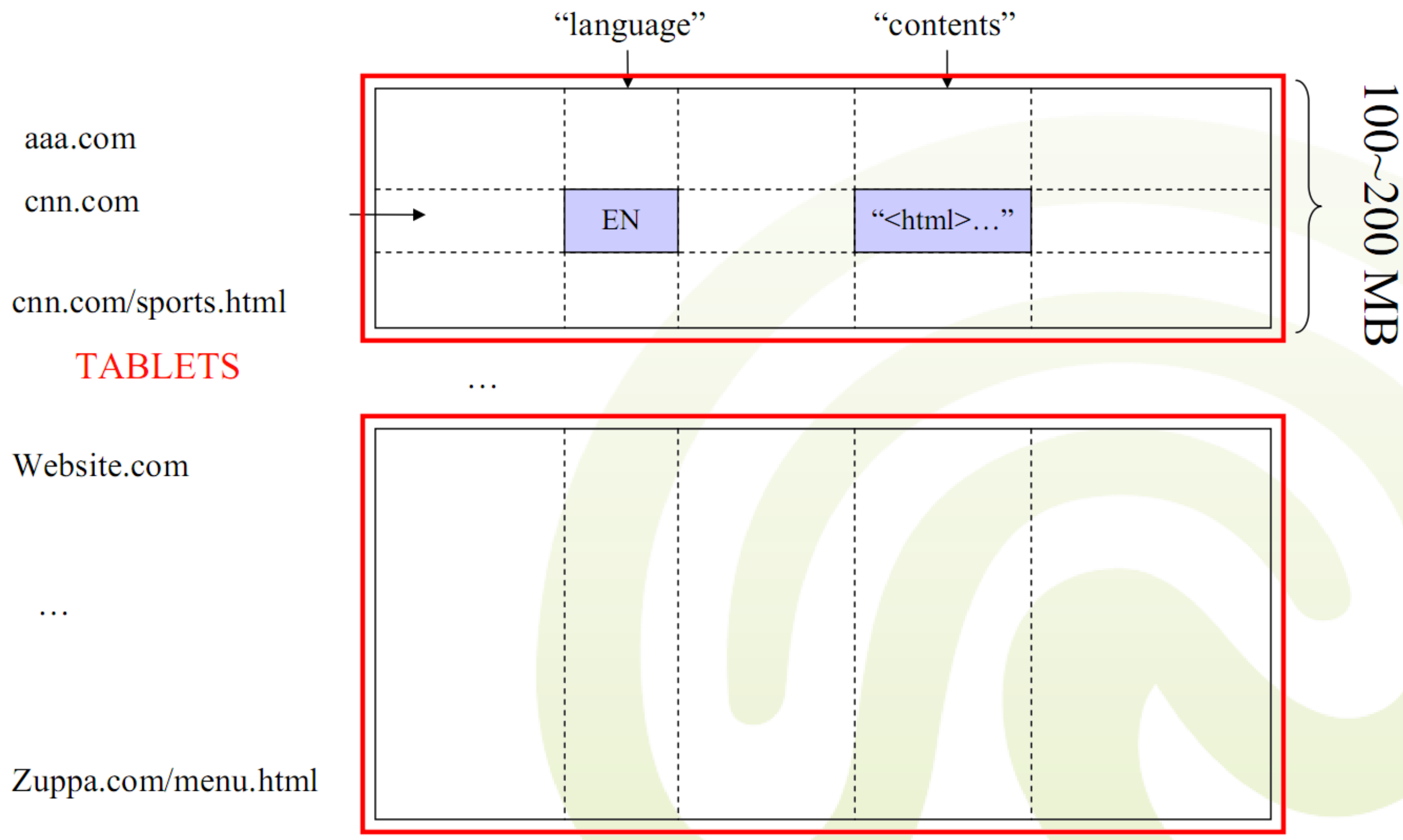


Bigtable: Managing Tablets

- Each tablet server node is **responsible** for around 10 to 1000 randomly scattered tables
  - Much more tablets than nodes!
    - **Each tablet is assigned to just one node**
  - **Easy recovery**
    - After a Bigtable node fails, 10 to 1000 machines need to pick up just one tablet
  - **Good initial load balancing**
    - Remember: rows within tablets are continuous for locality
    - Node holds very different tablets
      - Some may be hot and some may be cold
  - **Very easy runtime load balancing**
    - Overloaded node simply migrates a tablet to a under-utilized node
    - Bigtable master decides on load-balancing migration

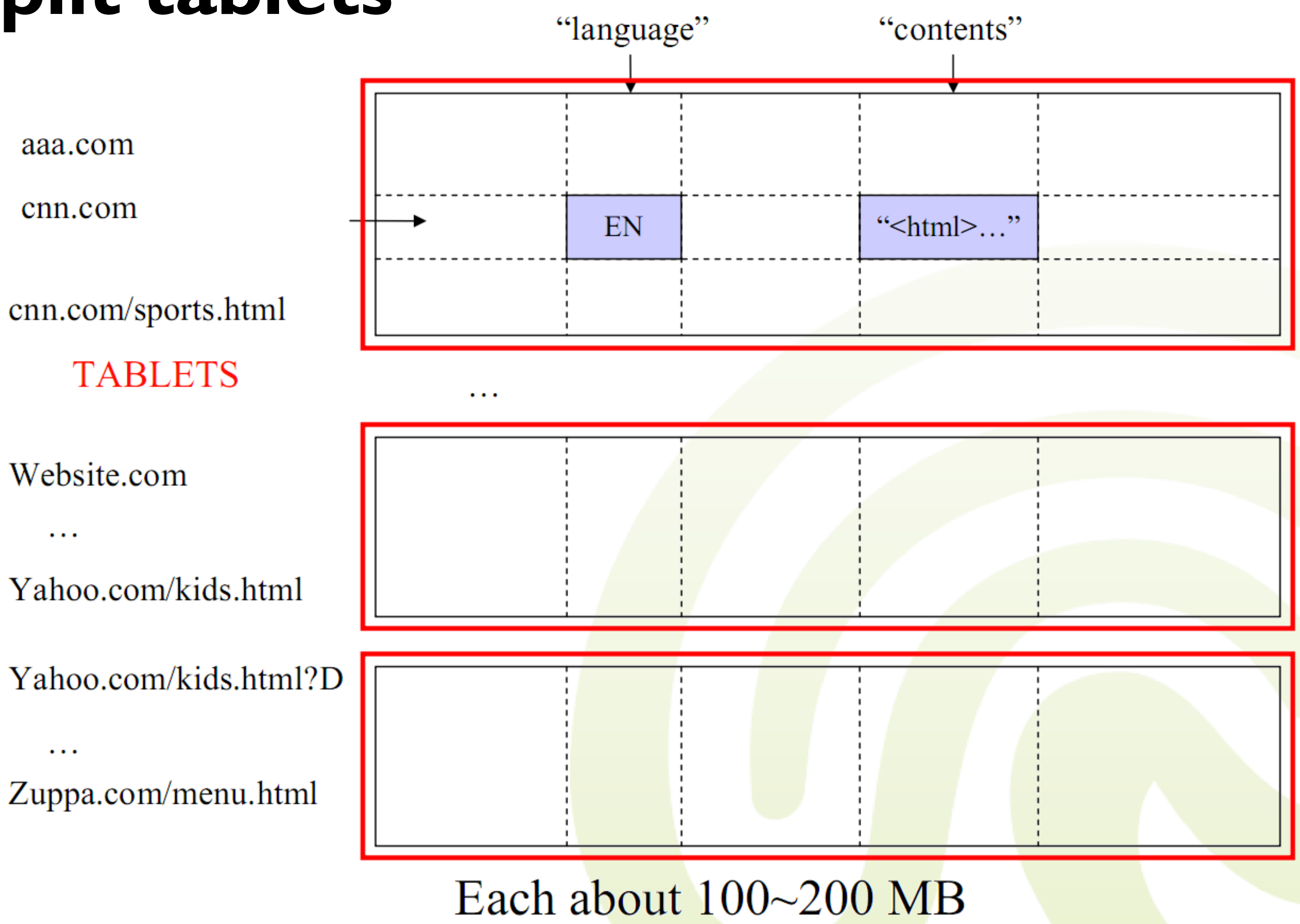
Bigtable: Managing Tablets

- Tablets can be **split** and **migrated** if they grow to big



Bigtable: Managing Tablets

- **Split tablets**



Bigtable: Managing Tablets

- **Clients** which try to work on certain data must first **locate the responsible tablet**
  – Tablets may freely move across the servers

- Two options
  A) Just ask master server which must then keep a directory
  B) Store tablet location in a index within Bigtable itself
- Option B is implemented
  – Tablets are organized in a **3-tier hierarchy** which serves as a distributed index
    - Think of a B-Tree…

Bigtable: Managing Tablets

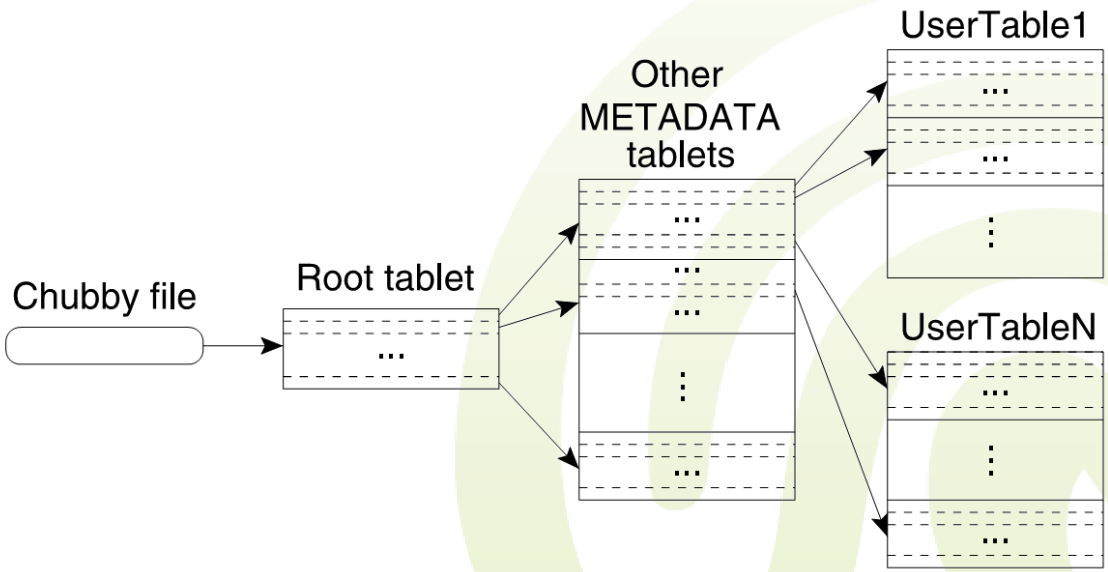
- **Entry point** is always a Chubby file
  - Chubby: **distributed lock manager**
    - In short: can store a tiny file in a distributed, persistent and indestructible fashion
    - May hand out exclusive locks on the files
- **Root tablet** serves as entry point and is never split
  - Just points forward to metadata tablets
- **Metadata tablets** represent an index table
  - For each **actual data tablet**, the row name range (start and end) and the responsible tablet server are stored
  - **Root tablet** stores row name range (start and end) of the responsible metadata tablet

Bigtable: Managing Tablets

– Chubby file points to the tablet server holding the **root tablet**

– **Root tablet** links to meta-data tablets

– **Meta-data tablets** link to actual data tablets



Bigtable: Managing Tablets

- Each tablet is assigned to one **tablet server**
- Each tablet is stored as a **GFS file**
  - Thus, tablets are durable and distributed
  - Usually, the GFS **primary replica** and the GFS **lease** of a tablet file are held by the same machine as the tablet server
    - Remember: each Bigtable server also runs a GFS server
    - Read and writes are thus performed on **local disk**
      - If a tablet server is assigned a new tablet, it is usually a good idea to request the background transfer of all GFS chunks related to that tablet to the new server

Bigtable: Managing Tablets

- Master **keeps** track of available tablet servers and all tablets not assigned to any server
  - Master can use metadata tables for this
    - Metadata list all tablets
    - Orphaned tablets can be assigned by Master
  - A tablet server **opens** all tablets it is assigned to
    - e.g. load indexes into main memory

Bigtable: Managing Tablets

- **A new tablet server joins**
  - Tablet server **registers** itself with the lock-manager (Chubby) by creating an **ID file** in a special directory and obtaining a time-decaying **lock** for it
    - Tablet server periodically **re-acquires lock**
  - **Bigtable master monitors directory** and contacts new servers
- **A tablet server leaves or fails**
  - **Server lock expires**
    - Bigtable master notices when a lock is lost

Bigtable: Managing Tablets

- **Detecting lost tablet servers**
  - Master server periodically tries to obtain locks on the ID files of all known tablet servers
    - If everything is OK, request is denied
    - If lock is granted, the respective server is dead
      - All its tablets are reassigned (tablets themselves are stored on GFS and are not affected by tablet server loss)
      - Delete the servers ID file

Bigtable: Managing Tablets

- If Chubby session holding the **server ID file** expires or has a time out, **masters kills itself**
- **A new master starts**
  - A unique Chubby lock is acquired to ensure that there is just one master
    - Lock also identifies master
    - Lock may decay and must be renewed
      - If lock is lost, the master failed and a new master must be elected
  - Load current tablet assignments from **root tablets**
    - Root tablet location is also in Chubby
    - Contact all tablets servers to check if they are OK

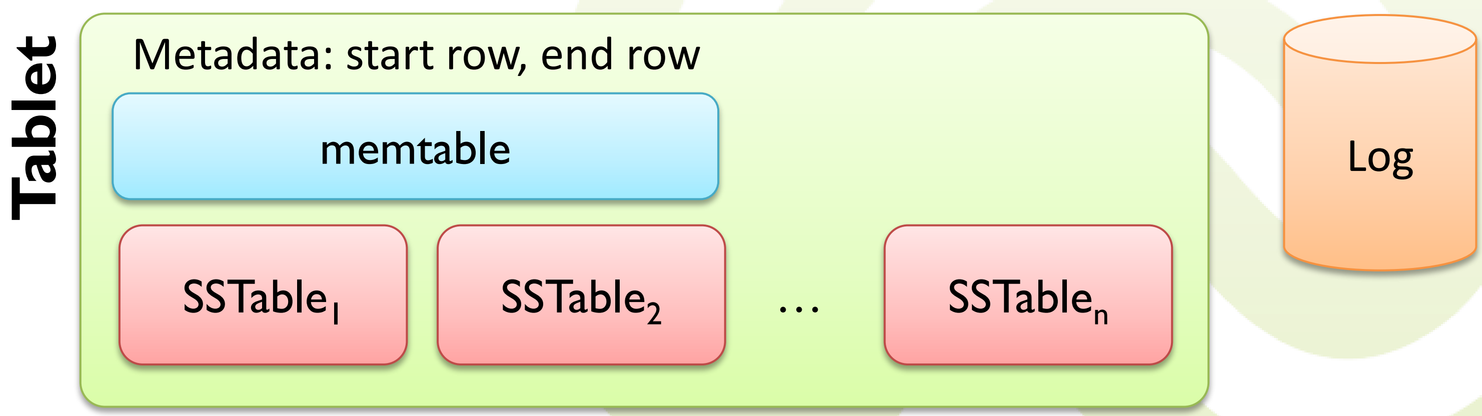
Bigtable: Managing Tablets

Summary

- Recap
  - A big table cell consist of multiple **tablet servers** and a single **master server**
    - Distributed lock services is used to check for node failures
    - Bigtable server also run a GFS server
  - **Master server** distributed tablets to tablet servers
    - Responsible for maintenance
    - Load balancing, failure recovery, etc.
  - Specialized **root tablets** and **metadata tablets** are used as an index to look up responsible tablet servers for a given data range
    - Clients don't communicate with master server
    - Usually, they work only with one or very few tablet servers on small data ranges
      - Bigtable can become very complicated to use if clients don't work on limited ranges!

Bigtable: Implementation

- Each **tablet** directly interacts with several components
  - Tablet data is stored in several **immutable SSTables**
    - SSTable are stored in GFS
  - An additional **memtable** holds data not yet stored in a SSTable
    - Stored in main memory
    - All writes are preformed on memtable first
  - A persistent append-only **log** for all write operations
    - Log is shared with all tablets of the tablet server in is also stored in GFS

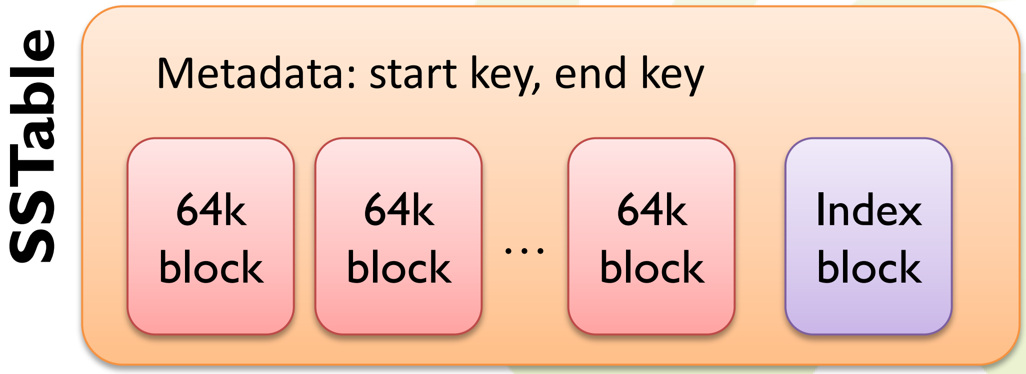


Bigtable: Implementation

- SSTables are **immutable ordered maps** holding **key-value pairs**
  - Each entry represents a cells
    - Key are triples of **<row, column, timestamp>**
    - Value is the actual cell value
  - SSTables can very easily be **traversed** as they are **ordered**
    - Each SSTable has a clearly defined **start key** and **end key**
      - However, ranges of SSTables may overlap!
  - **Immutability** eliminates consistency problems
    - A SSTable can never be changed (only completely deleted compaction)
    - **No locks** necessary for reads and writes
      - Parallel read are always possible without danger of interference

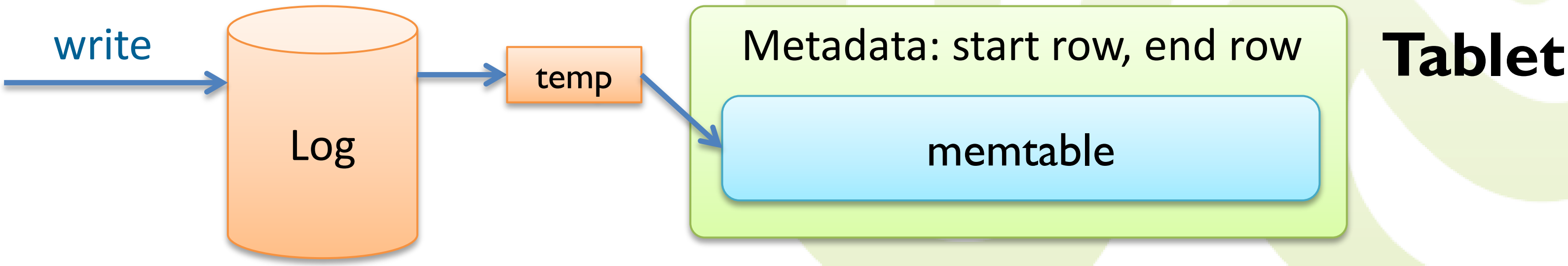
Bigtable: Implementation

- Internally, SSTables consist of multiple 64KB **blocks** of data
  - Again, each block is an **ordered map**
  - Each SSTable has a special **index block** mapping key ranges to their responsible block number
  - Every time a tablet is opened, all SSTable **index** blocks are loaded to the tablet server **main memory**



Bigtable: Write and Read

- Write operations must ensure **atomicity** and also store the data within the SSTables
- **Write operation** arrives at a tablet server
  - Server checks if the client has sufficient **privileges** for the write operation (Chubby)
  - A **log record** is generated to the commit log file
  - Once the write commits, its contents are inserted into the **memtable**
    - **Copy-on-write** on row basis to maintain row consistency
      - e.g. a write request is completed at a temporary location and then atomically copied into the memtable
    - Memtable is also **sorted by keys** similar to SSTables
    - Nothing stored in SSTables yet!

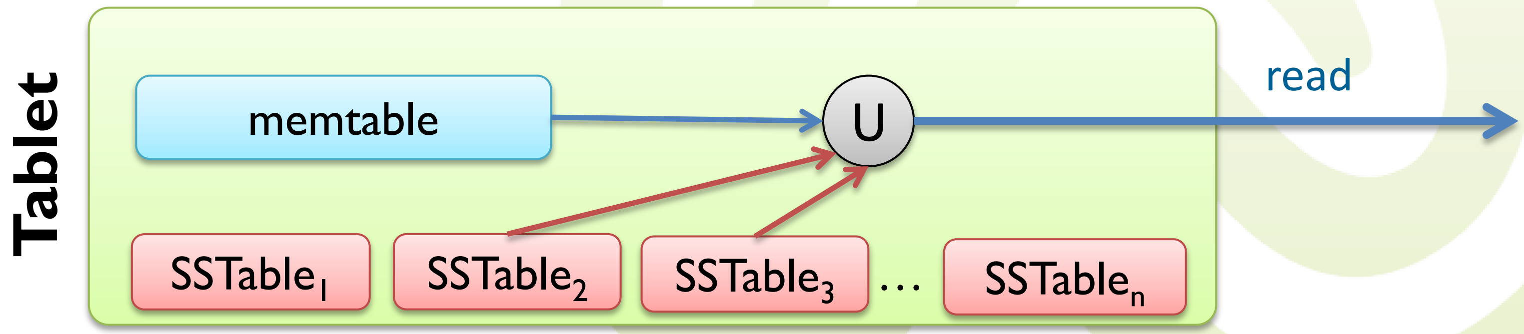


Bigtable: Write and Read

- Memtable **size increases** with number of write operations
  - After a threshold is reached, the current memtable is **frozen** and a **new** one is created
  - Frozen memtable is **serialized** to disk
    - Called **minor compaction**
    - Note: with a quite high probability, SSTables will now have overlapping ranges!
    - Also committed to log after operation was successful
      - Data is now persistent and does probably not need recovery from log files

Bigtable: Write and Read

- **Read operation** for a certain range / key arrives at a tablet server
  - Server ensures client has **sufficient privileges** for the read operation (Chubby)
  - Tablet server uses **index blocks** of all SSTables and the memtable to find all blocks with matching range
    - All related **blocks and the memtable are merged** into a sorted, unified view
      - Merge can be performed very efficiently as all components are pre-sorted (e.g. like merge-sort)
    - **Binary search** is possible on the merged view



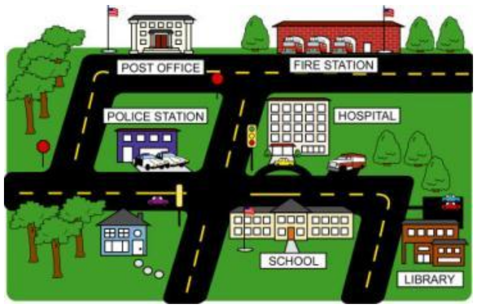
Bigtable: Write and Read

- If keys are to be **deleted**, they are written with a special **delete flag** as value
- In periodic intervals, **major compactions** are performed
  - Background maintenance operation, normal read and writes can still continue
  - Several overlapping SSTables and/or the memtable are **compacted** into a set of **non-overlapping** SSTables
    - Increases read performance (less overlapping SSTable → less merging)
    - **Deleted** records may now be removed
      - Possibly, also all its old versions (sensible data must be guaranteed to be deleted)

Bigtable: Write and Read

- If a **tablet server crashes**, tablets are reassigned by the Bigtable master to a new tablet server
  - All SSTable files are persistently stored in GFS and are not affected by the server failure
  - **Memtable is lost**
    - Memtable can be reconstructed by **replaying** the crashed servers **log files** starting from last minor compaction checkpoint
    - Server log file was also stored in GFS!

Bigtable: Write and Read



- Further Bigtable optimizations
- **Locality Groups**
  - Group columns **frequently accessed** together such that their values will be in the same or a close SSTable
    - Creates semantic locality
    - Locality group provided manually by developers
    - Access to SSTables minimized for certain applications
  - e.g. webcraweler: keywords, name, pagerank in one locality group, content in another

Bigtable: Write and Read

- **Compression**
  - Most data in Google can be easily compresses (HTML files, keywords, etc.)
  - SSTable blocks are compressed individually
    - Takes advantage of locality groups: data within a block should be similar
      - E.g. two pages of the same website sharing most navigation components
    - Simple two-pass frequent term compression
      - Due to locality very good reduction rates of 10-to-1

Bigtable: Write and Read

Summary

- **Recap**
  - Tablets are persistently stored in multiple SSTables in GFS
  - **SSTable** are immutable ordered key-value maps
    - Contains table cells
    - No locking problems for SSTable access
  - All write operations are performed in RAM memtable
    - After memtable is big enough, it is serialized into a new, full and immutable SSTable
  - Read operations dynamically merge all responsible SSTables (from index) and the memtable
  - SSTable need to be compacted from time to time
    - If not, too many SSTable are responsible for the same ranges

Bigtable

Summary

- Google Bigtable is a NoSQL database
  - **No complex query language supported**
    - Mainly based on scans and direct key accesses
  - **Single table data model**
    - **No joins**
    - **No foreign keys**
    - No integrity constraints
  - **Flexible schemas**
    - Column may be added dynamically
  - Usually, Bigtable is not a direct replacement for a distributed database

HBase

Detour

- **Hbase** is an open-source **clone** of Bigtable
  - http://hbase.apache.org/
  - Created originally at Powerset in 2007
- Hbase is a **Apache Hadoop** subproject
  - Hadoop is strongly supported by Microsoft and Yahoo
  - http://hadoop.apache.org/
  - Hadoop reimplements multiple Google-inspired infrastructure services
    - MapReduce ←Google Map And Reduce
    - Hbase ← Bigtable
    - HDFS ← GFS
    - ZooKeeper ← Chubby