

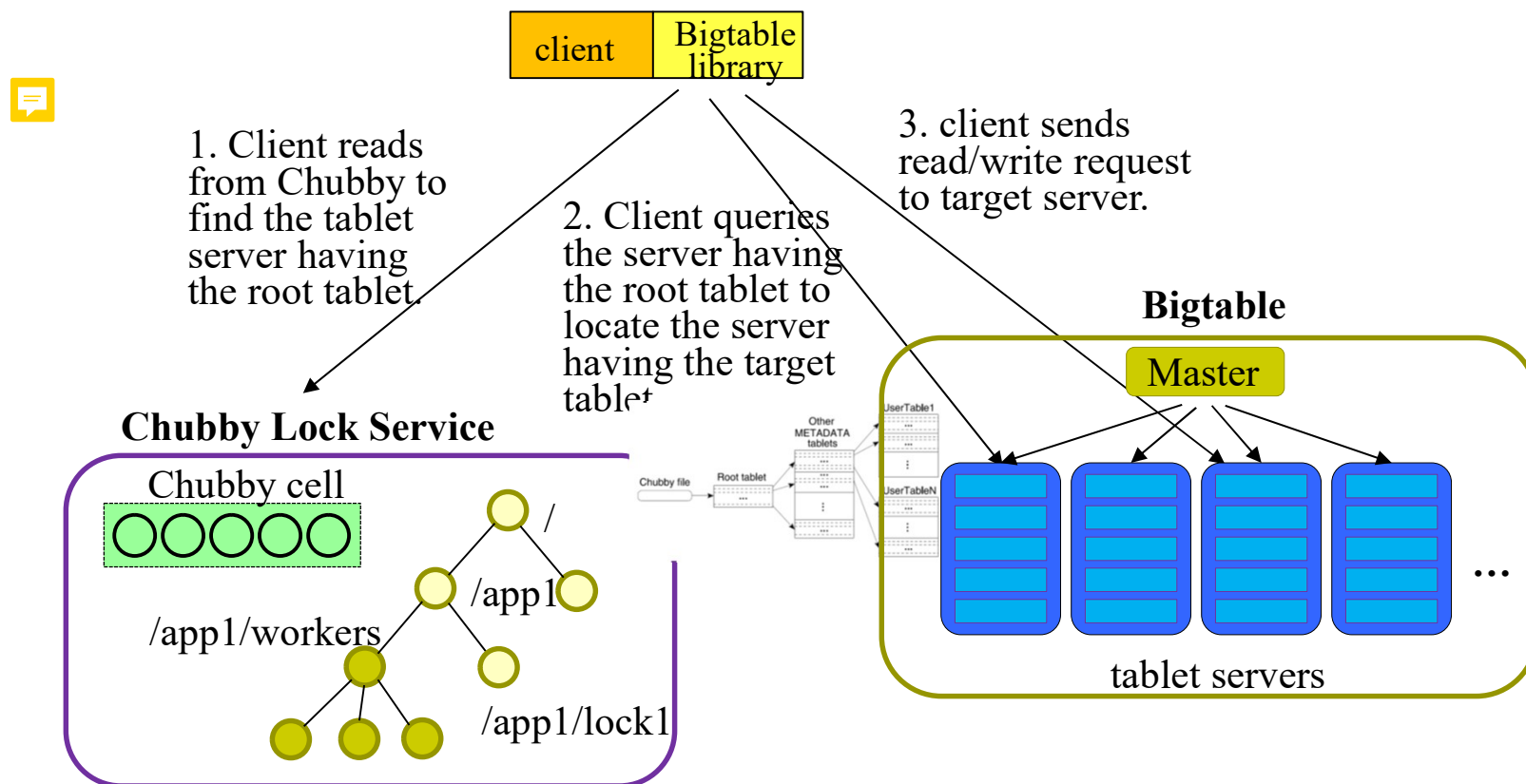
Chubby Lock Service

The Chubby lock service for loosely-coupled distributed systems, M. Burrows, OSDI 2006

Motivation

- ❑ When implementing a distributed application, we often need to coordinate processes, e.g., initialization, synchronization, rendezvous, mutual exclusions, leader election,
- ❑ Why not build a general tool to provide such services?

Recap in Bigtable: Tablet Location



Bigtable uses Chubby to:

- keep track of tablet servers
- locate important information (METADATA)
- ensure there is at most one master.

What is Chubby?

- ❑ A coarse-grained **lock service** (and reliable low-volume storage) for a loosely-coupled distributed system
 - Client interface similar to whole-file advisory locks with notification of various events (e.g., file modifications)
 - Design emphasize on: reliability, availability, easy-to-understand semantics
 - Used in Google: GFS, Bigtable, etc.
 - Open-source counterpart: Apache Zookeeper



An Engineering Effort

“Building Chubby was an engineering effort ... it was not research. We claim no new algorithms or techniques. The purpose of this paper is to describe what we did and why, rather than to advocate it.”

Chubby vs. Zookeeper

❑ Chubby

- provide coarse-grained locking as well as reliable storage for a loosely-coupled distributed system.

❑ Zookeeper

- provide a simple and high performance kernel for building more complex coordination primitives at the client.

Design: Lock service

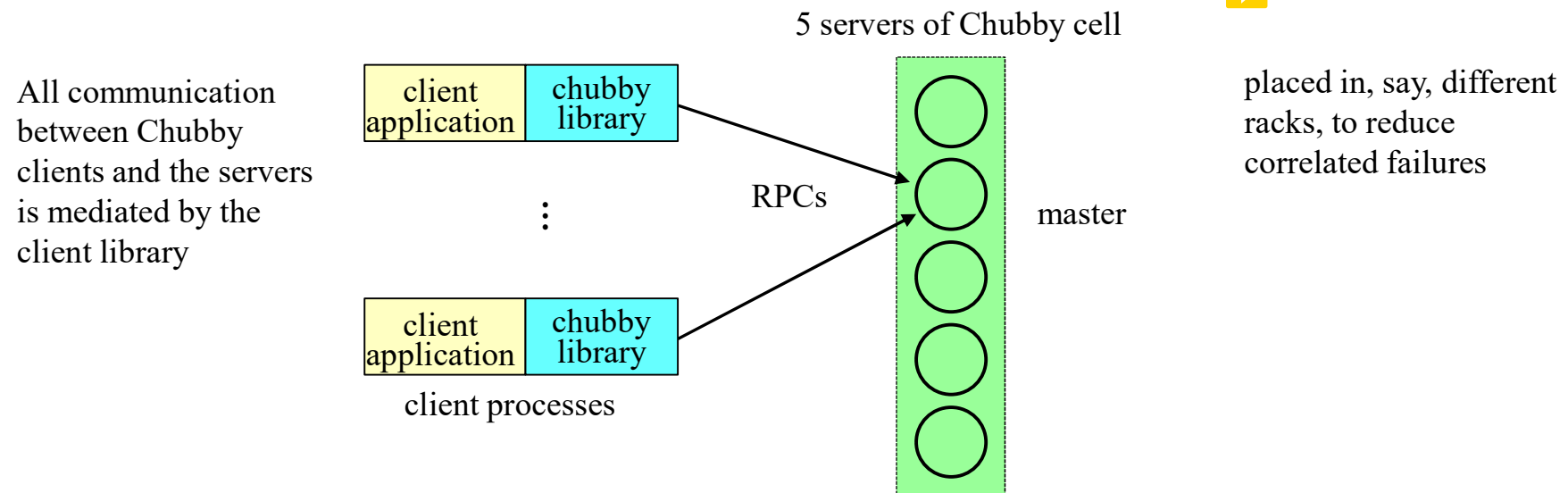
(instead of a library for **Paxos** consensus) 

- ❑ easier to maintain existing program structure and communication pattern
- ❑ familiar to most programmers
- ❑ support small files read/write
 - allow a service to elected primaries (those who acquire locks) and advertise themselves and their parameters
 - must allow thousands of clients to observe this file
 - must also support notification to let clients know file changes
 - need to provide security, including access control

Design: Coarse-grained Locks (as opposed to Fine-grained)

- ❑ Locks are acquired only rarely (e.g., to elect a master)
 - cause less load to lock server
 - temporary lock server unavailability delays clients less
- ❑ clients can implement their own fine-grained locks tailored to their application

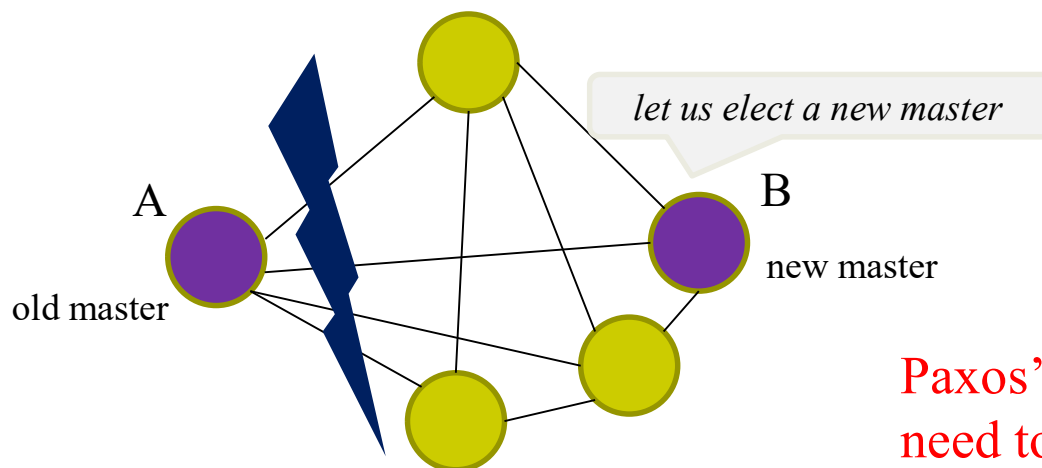
System Structure



- ❑ Chubby cell: a small number of replicas (e.g., 5)
 - a cell might serve 10,000 4-processor machines connected by 1Gbit/s Ethernet.
- ❑ A Master is selected using a consensus protocol (**Paxos**)
 - master gets a **lease** of several seconds
 - lease is periodically renewed
 - if a master fails, a new one is elected *after* the master's lease expires (election takes a few sec.)
- ❑ Client talks to the master via chubby library
 - All replicas are listed in DNS; clients discover master by talking to any replica

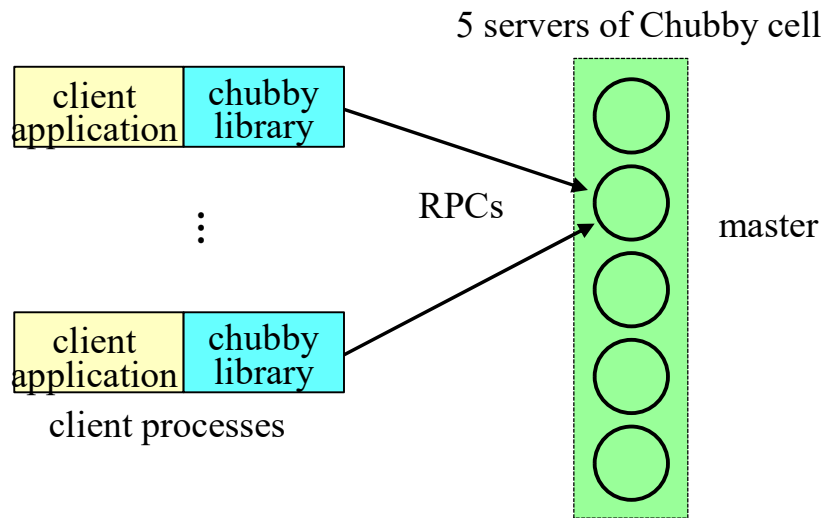
Why use the lease mechanism?

- ❑ Suppose A is master and it gets disconnected from B
- ❑ B times out trying to talk to A, thinks A is dead, and proposes that it be the master
- ❑ If other nodes agree and A doesn't hear about the new master, then A will continue to act as master for a while, accepting read requests for what could be stale data.
- ❑ The lease mechanism avoids nodes to elect a new master while the old master thinks that it is still in charge.



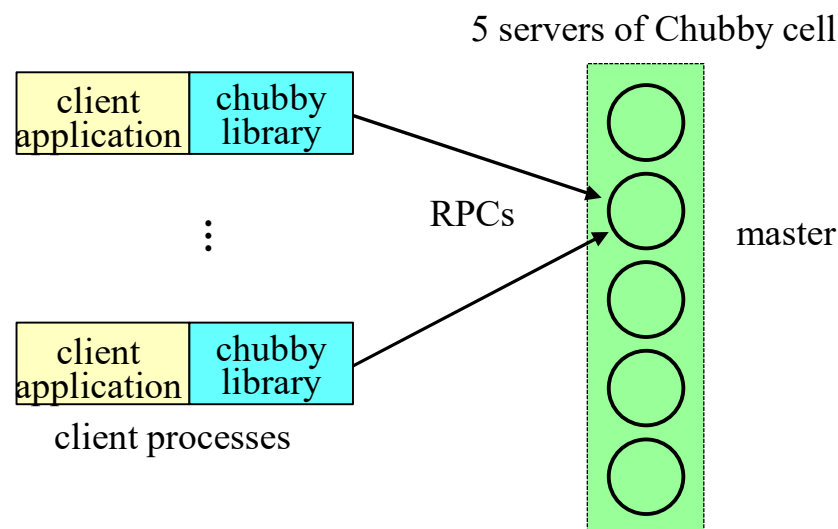
Paxos's algorithm is still
need to ensure a consensus
on who is the new master.

System Structure: Read/Write



- ❑ Replicas maintain copies of a simple database
- ❑ Clients send read/write requests *only* to the master (until it ceases to respond or its lease expires)
- ❑ For a write:
 - the master propagates it to replicas via the **consensus** protocol
 - ack the write until the write reaches a majority of replicas
- ❑ For a read:
 - the master satisfies the read alone

System Structure: Recovery/Replacement

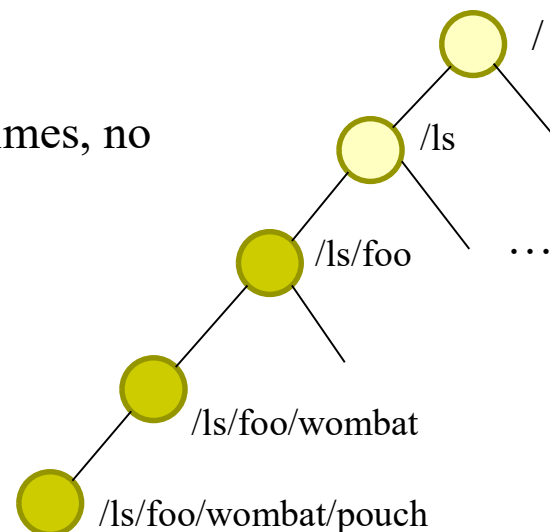


- ❑ If a replica fails and does not recover for a long time (a few hours)
 - a fresh machine is selected to replace the failed one
 - the replacement system updates the DNS
 - the current master polls DNS periodically to discover new replicas, and then updates the cell's members in the cell's database
 - the new replica obtains a recent copy of the database
 - the new replica does not participate in new master election until it has its data up-to-date
 - Integrating the new replica into the group is another Paxos run

UNIX-like File System Interface

❑ Chubby exports a strict tree of files and directories as interface

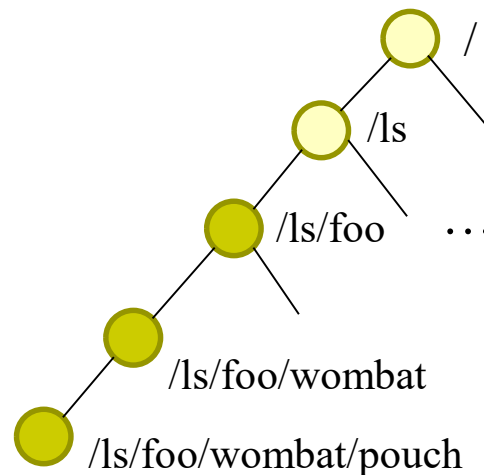
- files and directories are collectively called nodes, e.g.,
- /ls/foo/wombat/pouch
 - 1st component (ls): lock service (common to all names)
 - 2nd component (foo): the chubby cell (used in DNS lookup to find the servers and cell master)
 - A special cell name local to designate to use the client's local Chubby cell
 - The rest (/wombat/pouch): interpreted within the cell
- Support most normal operations (create, delete, open, write, ...)
 - no move, no directory modified times, no symbolic links, no hard links, ...



Files & Directories



- ❑ Nodes (files & directories) can be permanent or ephemeral
 - ephemeral nodes are deleted if no client has them open, or it's an empty directory
 - ephemeral files are useful for indicating others that a client is alive



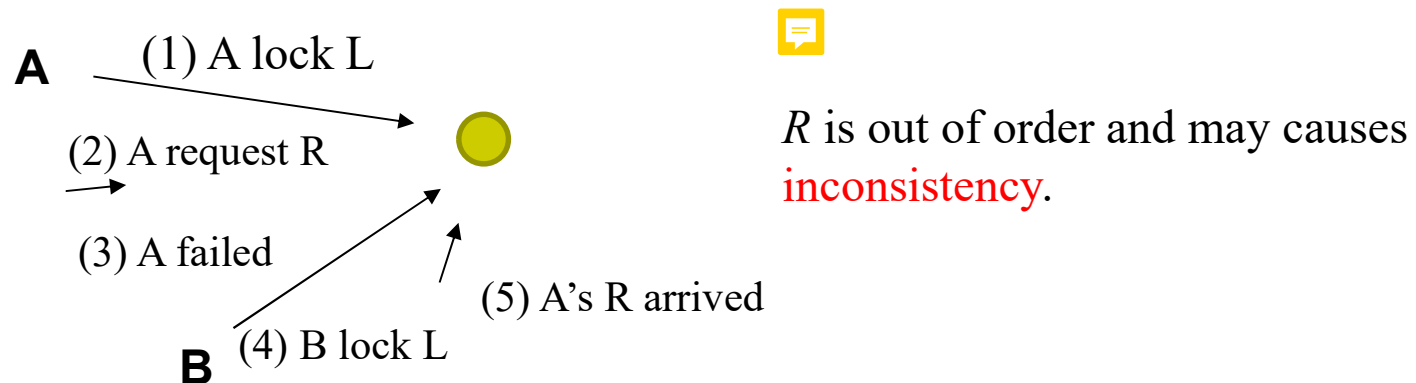
Node Metadata

- ❑ ACLs (Access Control List)
 - reading, writing and changing
- ❑ 4 monotonically increasing 64-bit numbers
 - instance number
 - content generation number (files only): increases when the file's contents are written
 - lock generation number: increases when the node's lock transitions from free to held
 - ACL generation number:
- ❑ 64-bit file-content checksum

Locks

- ❑ Any node can act as an **advisory** reader/writer lock
 - they conflict only with other attempts to acquire the same lock
 - Locks are not **mandatory** as holding a lock called F neither is necessary to access the file F , nor prevents other clients from doing so.
- ❑ Two modes for holding locks:
 - **exclusive mode (write)**: one client may hold the lock
 - **shared mode (read)**: any number of clients may hold the lock

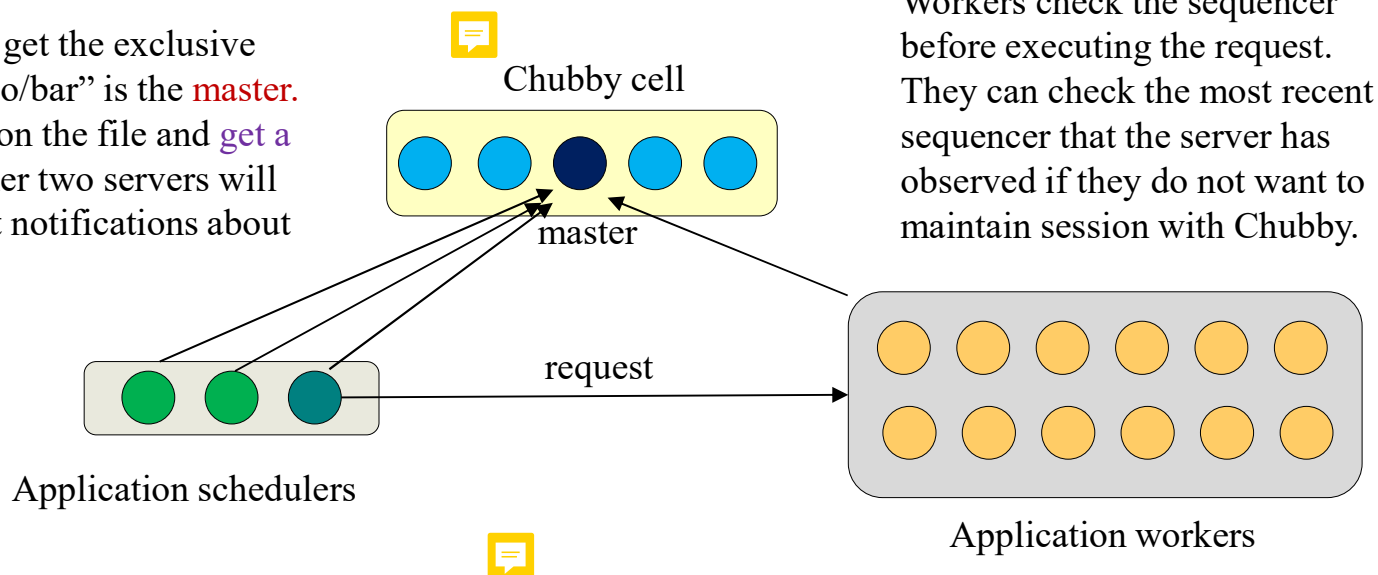
Potential lock problems



- ❑ Solution 1: **lock-delay** (imperfect but it works in most cases)
 - a client may specify a lock-delay (up to 60s) to prevent other clients from claiming the lock during the period in case of failure
- ❑ Solution 2: **sequencer**
 - A lock holder can obtain a sequencer from Chubby
 - Sequencer: name of the lock + lock mode + lock generation number
 - It passes the sequencer to servers (e.g., Bigtable) that wish their operations to be protected by the lock
 - The server can verify the sequencer information to determine whether or not to accept the client's request.

Use of Sequencer

The first server to get the exclusive lock on file “/ls/foo/bar” is the **master**. It writes its name on the file and **get a sequencer**; the other two servers will only receive event notifications about this file



Workers check the sequencer before executing the request. They can check the most recent sequencer that the server has observed if they do not want to maintain session with Chubby.

Chubby Events

- ❑ Clients can subscribe various events when they crease a handle (open phase) :
 - file contents modified: if the file contains the location of a service, this event can be used to monitor the service location
 - child node added, removed, modified
 - master failed over
 - handle becomes invalid (suggesting a communication problem)
 - lock acquired (determining when a primary has been elected, rarely used)
 - conflicting lock request (allowing the cache of locks, rarely used)
- ❑ Events are delivered to clients asynchronously after the corresponding actions have taken place, and via an up-call from Chubby library

Chubby APIs

- ❑ Open/Close node name
 - `Open(), Close()`
 - Handles are created by `open()` and destroyed by `close()`
 - Options: read/write/change ACL; events subscribed; lock-delay; whether to create a new file/directory
- ❑ Poison
 - `Poison()` - causes outstanding and subsequent operations on the handle to fail without closing it
- ❑ Read/Write full content
 - `GetContentsAndStat()` - atomic reading of the entire content and metadata
 - `GetStat()` – reading of the metadata only
 - `ReadDir()` – reading of names and metadata of the directory
 - `SetContents()` - atomic writing of the entire content
- ❑ ACL
 - `SetACL()` – change ACL
- ❑ Delete node
 - `Delete()` – if it has no children

Chubby APIs (2)

❑ Locks

- `Acquire()`, `Release()` – acquire/release a lock.
- `TryAcquire()` – try to acquire a potentially conflicting lock by sending “conflicting lock request” to the holder


❑ Sequencers

- `GetSequencer()` – associate a sequencer with a handle
- `SetSequencer()` – return a sequencer that describes any lock held by this handle
- `CheckSequencer()` – check whether a sequencer is valid

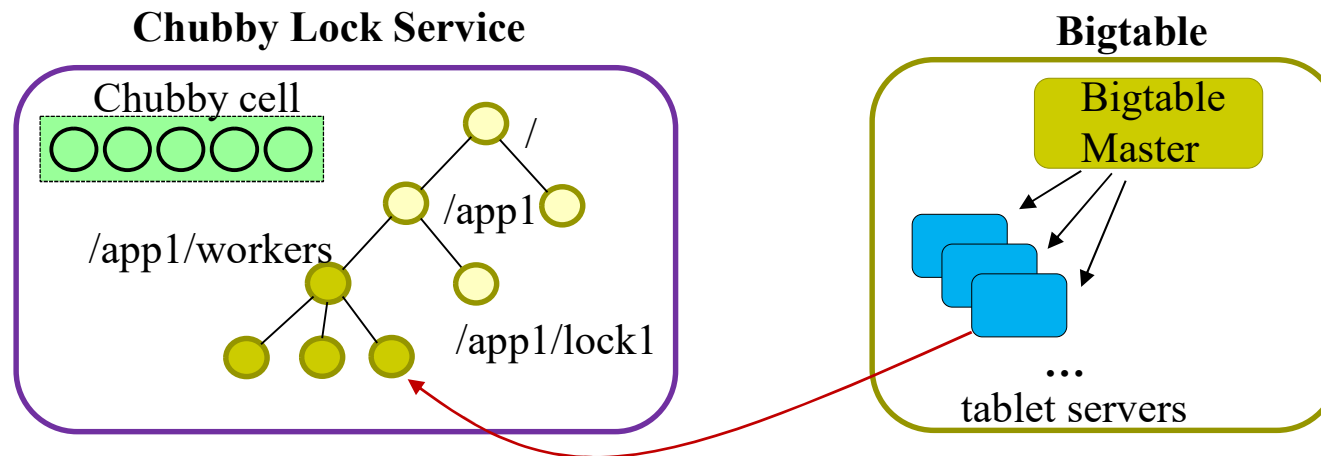
all calls above take an operation parameter to allow the client to

- supply a callback to make the call asynchronous,
- wait for the completion of such a call, and/or
- obtain extended error and diagnostic information.

Example: Primary Election

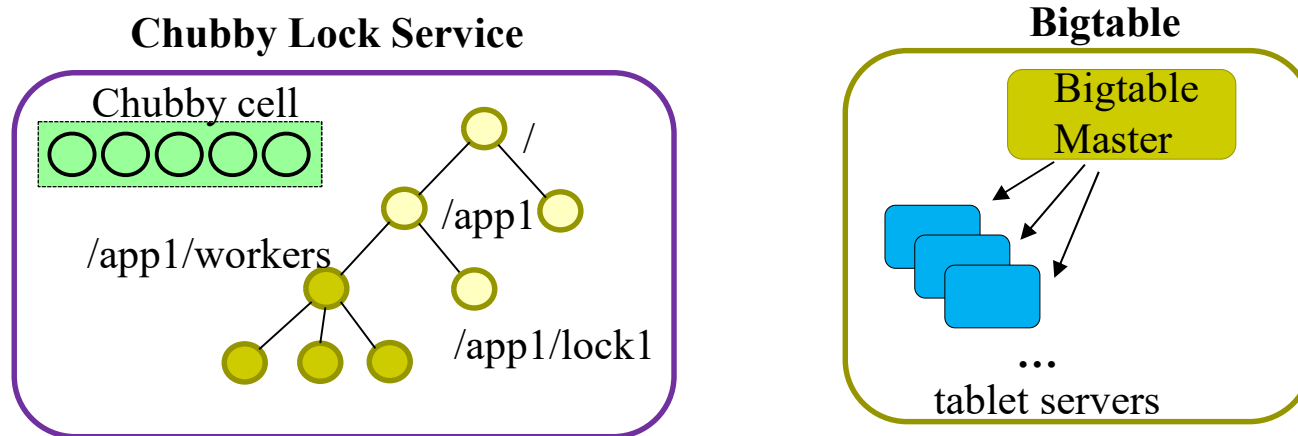
```
Open( "/ls/foo/OurServicePrimary" , "write mode" );  
if (successful) {  
    // primary   
    SetContents(primary_identity);  
} else {  
    // replica  
    Open( "/ls/foo/OurServicePrimary" , "read mode" ,  
        "file-modification event" );  
    when notified of file modification:  
        primary = GetContentsAndStat();  
}
```

Chubby in Bigtable



- On startup, a tablet server creates and acquires an exclusive lock on a uniquely named file in Chubby directory.
 - It stops serving its tablets if it loses its exclusive lock.
- The master monitors the above directory to discover tablet servers.
- The master periodically asks each tablet server for the status of its lock:
 - if the master was unable to reach a server but is able to acquire the server's lock, the master deletes the server's Chubby file and reassigns its tablets.
 - the master kills itself if it is unable to reach Chubby (session expires)
 - new master will be assigned by the cluster management system
 - tablets assignment is not affected by master failure)

Chubby in Bigtable (2)



On startup, a new master executes the following steps:

1. grabs a unique master lock in Chubby to prevent concurrent master instantiations (e.g., existing master is not dead)
2. scans the tablet servers directory in Chubby to find live servers
3. communicates with every live tablet server to discover what tablets are already assigned, and to inform them the new master.
4. scans the METADATA table to learn the set of tablets, and re-assign unassigned tablets.
 - a. For this step, the METADATA tablets must be assigned, which in turn requires the root tablet to be assigned.
 - b. So in Step (3), if the root tablet was not discovered, Step (4) must first assign root tablet in order to obtain the METADATA tablets.

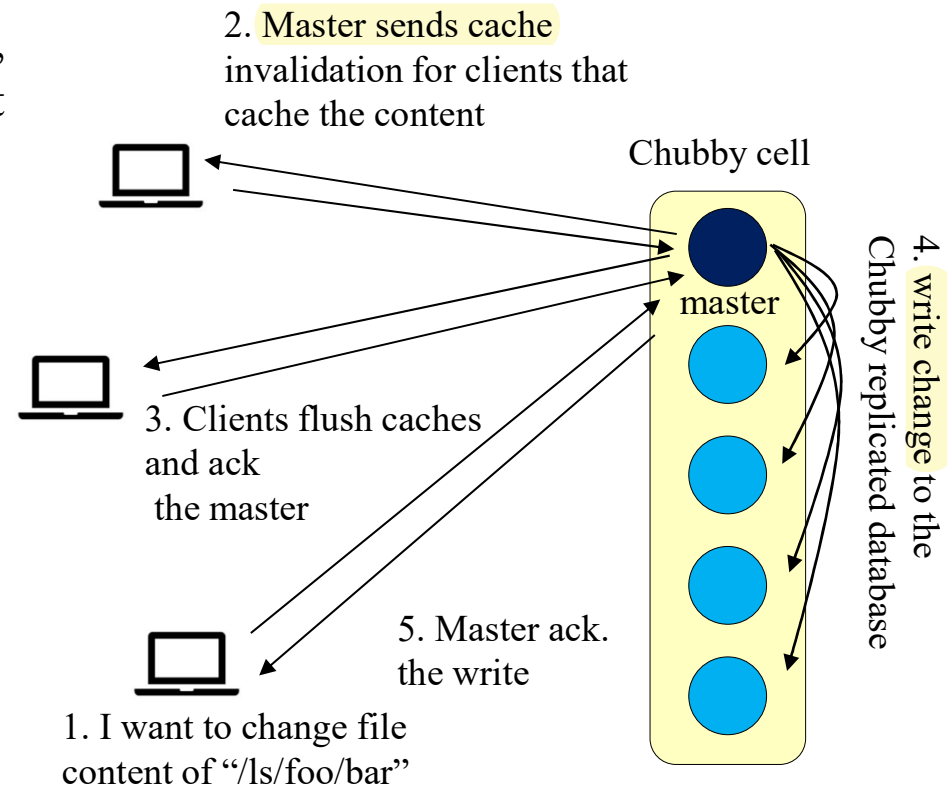
Caching

Clients cache file data and node metadata in a **consistent, write-through** in-memory cache

- cache maintained by a **lease mechanism**, and kept consistent by invalidations sent by the master

When file data or metadata is to be changed

- master block modification while sending invalidations to every client that may cache it
- the receiving client flushes the invalidated state and acknowledges the master by making its next KeepAlive call
- the modification proceeds only after the master knows that each client has invalidated its cache, either by acknowledging the invalidation, or because the client allowed its cache lease to expire



5 nodes?! Is that Enough?

- ❑ Can a cell of 5 nodes supports tens of thousands of clients?
 - It seems so, but Chubby's users need to be very careful!
- most effective scaling technique is to **reduce communication with the master**
- Other approaches
 - create an arbitrary number of Chubby cells
 - clients almost always use a nearby cell (found with DNS)
 - typical deployment uses one Chubby cell for a data center of several thousand machines
 - master increases lease times from the default 12s up to ~60s when it is under heavy load (KeepAlives are by far the dominant type of request)
 - Chubby clients cache file data, meta-data, the absence of files, and open handles to reduce the number of calls they make on the server
 - use protocol-conversion servers that translate the Chubby protocol into less-complex protocols (DNS ...)
 - **proxies**
 - **partitioning**

See [paper](#)

Summary (1)

- ❑ Chubby: a distributed lock service intended for coarse-grained synchronization of activities for Google
 - become Google's primary internal name service
 - common rendezvous mechanism for MapReduce
 - elect a primary server for GFS, Bigtable
 - standard repository for files that require high availability, such as configuration information and access control lists

Summary (2)

- ❑ Design based on well-known ideas that have meshed well
 - distributed consensus among a few replicas for fault tolerance
 - consistent client-side caching to reduce server load while retaining simple semantics
 - timely notification of updates
 - familiar file system interface
 - use caching, protocol-conversion servers, and simple load adaptation to scale

Dynamo: Amazon's Highly Available Key-value Store

Reading:

Dynamo: Amazon's highly available key-value store, Giuseppe DeCandia, et al., ACM SIGOPS Operating Systems Review, October 2007.



Dynamo

□ A distributed storage system that is

- Scalable
- Simple: key-value interface
- Fault-tolerant
- Highly available (always writable)
 - Conflict resolution is executed during read instead of write
- Decentralized
 - Similar to P2P, nodes are symmetric in their roles
- Manageable
- Service Level Agreements (SLA) Guarantee



□ Amazon DynamoDB is built on the principles of Dynamo with some modifications, e.g.,

- Dynamo is based on leaderless replication, DynamoDB uses single-leader replication

Service Level Agreements (SLA)

- ❑ Service Level Agreement (SLA): a formal contract between a client and a service on several system-related characteristics
 - E.g., expected request rate of an API or service latency
 - a performance oriented SLA is to describe it using average, median and expected variance
 - provide a response within 300ms for 99.9% of its requests for a peak client load of 500 requests per second
 - Amazon's engineering and optimization efforts, such as the load balanced selection of write coordinators, are not focused on averages, but on controlling performance at the 99.9th percentile.

Query Model

□ Key-value:

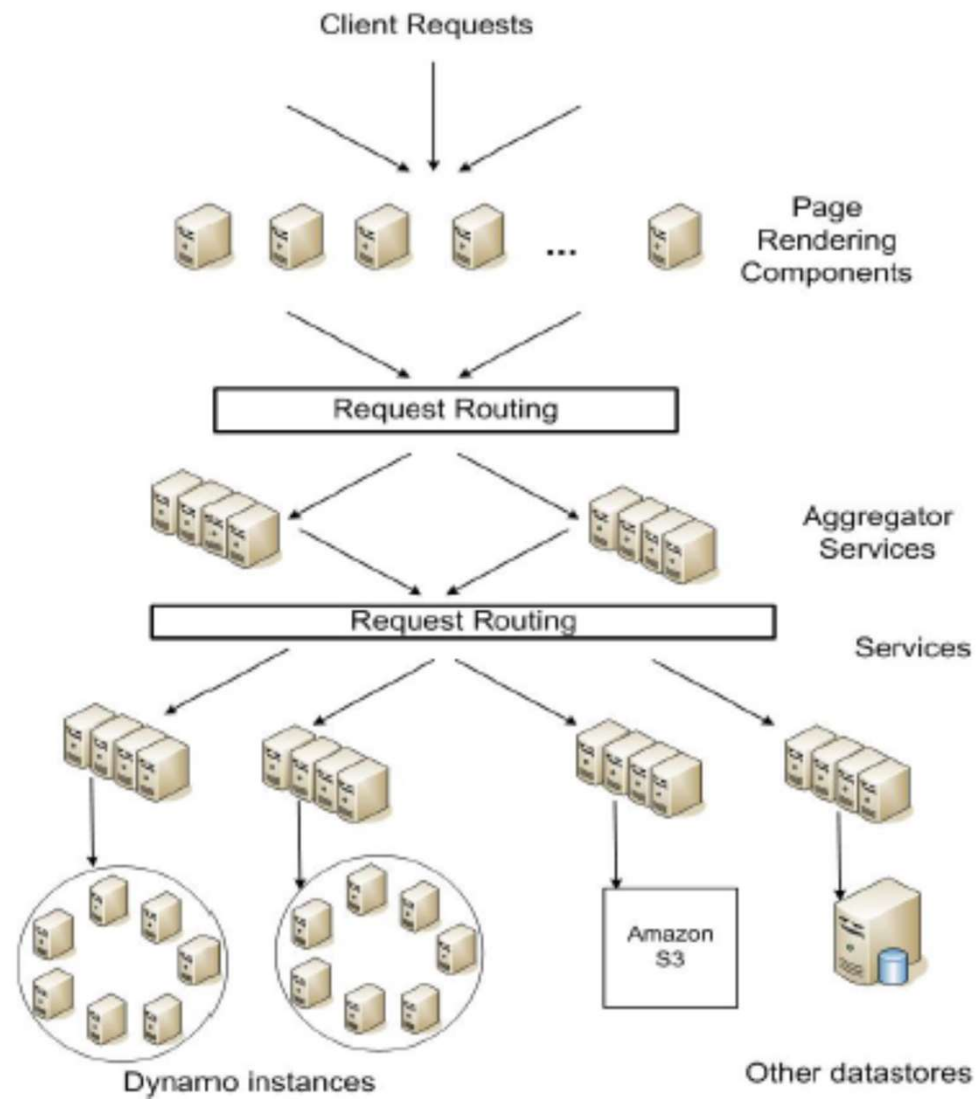
- simple read and write operations to a data item that is uniquely identified by a key
- no need for relational schema
- RDBMS is overkill, expensive hardware requirements
- targets applications that need to store objects that are relatively small (usually less than 1 MB)

Relaxed of ACID



- ❑ ACID Properties: Atomicity, **Consistency**, Isolation, Durability
 - Only weak consistency (eventual consistency)
 - Sacrifice strong consistency for availability
 - Experience at Amazon has shown that data stores that provide ACID guarantees tend to have poor availability.
 - does not provide any isolation guarantees and permits only single key updates

Service-oriented architecture of Amazon's platform



Simple Interface: only two operations

❑ `get(key)`

- locates the object replicas associated with the key in the storage system and returns a single object or a list of objects with conflicting versions along with a context.

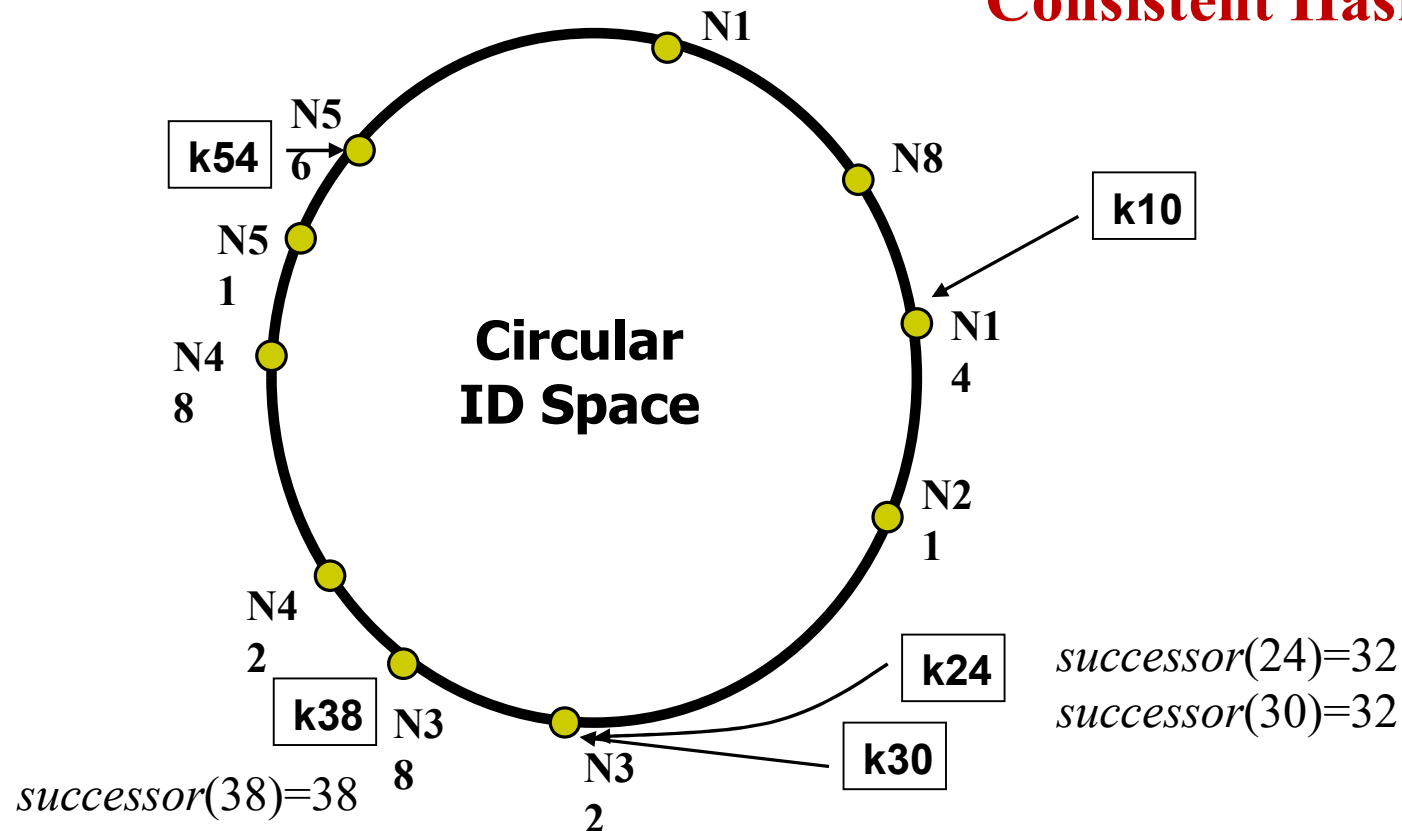
❑ `put(key, context, object)`

- determines where the replicas of the object should be placed based on the associated key, and writes the replicas to disk.
- The context encodes system metadata about the object that is opaque to the caller and includes information such as the version of the object.
- No delete operation
 - “add to cart” and “delete item from cart” operations are translated into put requests to Dynamo

Object Placement in Chord

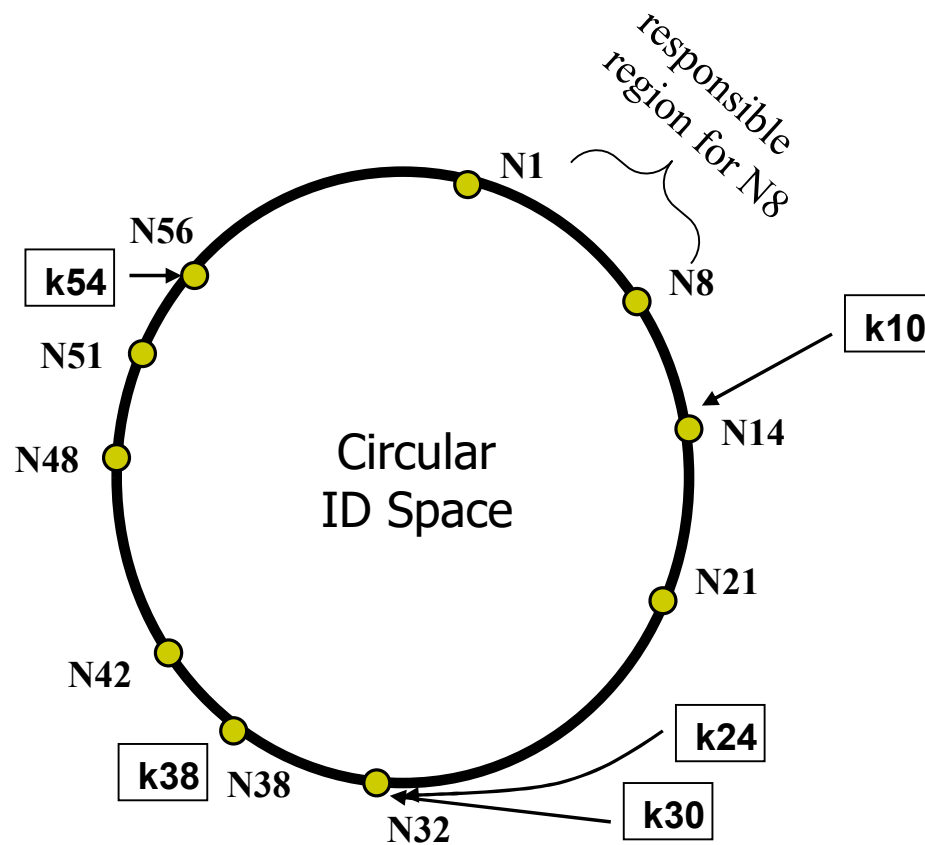
An ID Ring of length 2^6-1

Consistent Hashing



Chord: a scalable peer-to-peer lookup protocol for internet applications, Ion Stoica et al., [SIGCOMM 2001](#), [IEEE/ACM Transactions on Networking](#), 2003

Object Placement in Dynamo Ring

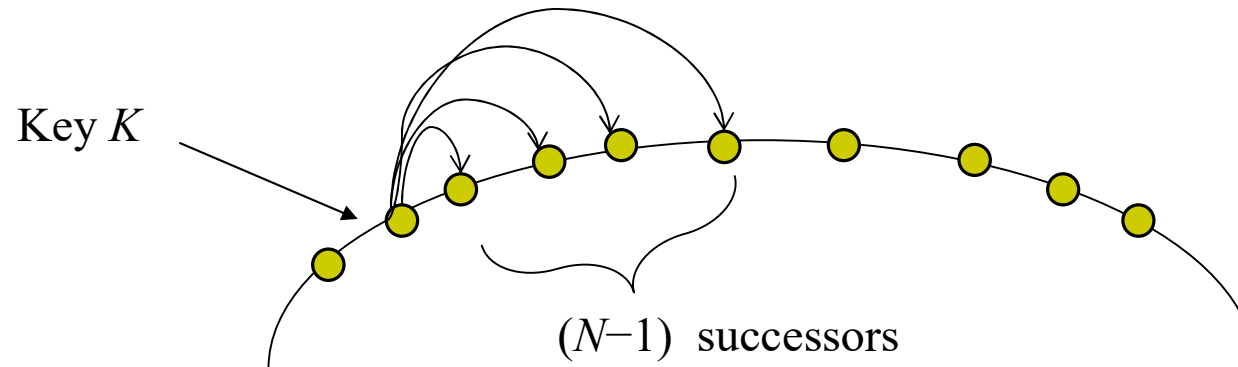


Consistent Hashing

Each (virtual) node in the system is assigned a random value within this space which represents its “position” on the ring, and is responsible for the region in the ring between it and its predecessor node on the ring.

Dynamo uses the concept of “virtual nodes”:
instead of mapping a node to a single point in the circle, each node gets assigned to multiple points in the ring. **Why?**

Replication & Failures Handling



For fault tolerance, the coordinator of a key (the virtual node in charge of the key) replicates the key (and the corresponding data item) at the $(N-1)$ clockwise successor nodes in the ring, where N is a configured parameter set per instance. Ps. this is similar to Chord.

Typically, the first among the top N nodes (called preference list) handles read/write operation for clients

Every node in the system can determine which nodes should be in the list for storing a particular key.

How to achieve this?

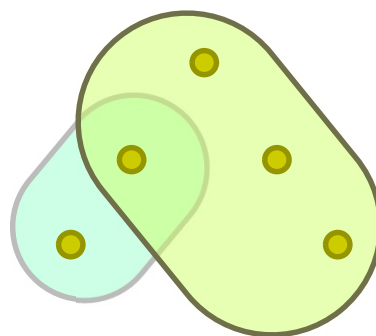
How to avoid the list of nodes all belong to the same physical node?

How to select a node for get() and put()?

- ❑ Route its request through a generic load balancer
 - the client does not have to link any code specific to Dynamo in its application
- ❑ Use a partition-aware client library that routes requests directly to the appropriate coordinator nodes
 - can achieve lower latency

Quorum Systems

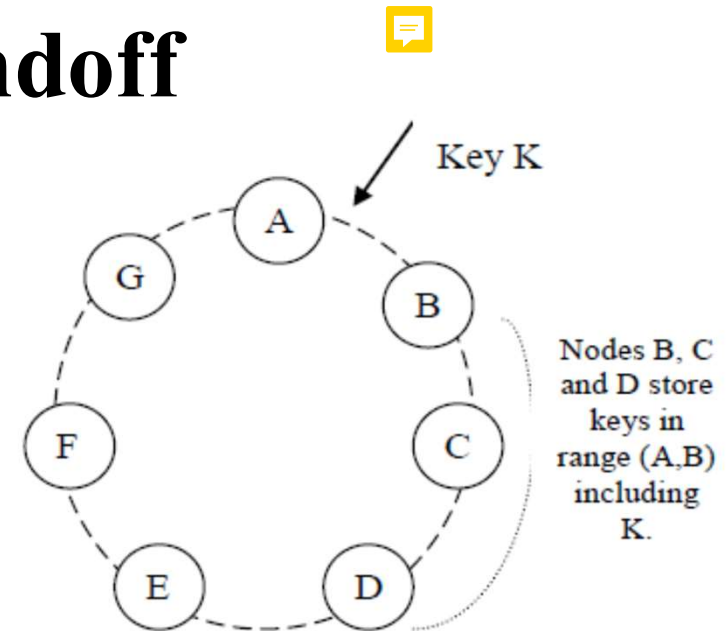
- ❑ Quorums are used to maintain read/write consistency among multiple replicates
 - $R + W > N$, i.e., # of read nodes R and # of write nodes W must be greater than the size N of a preference list
- ❑ But how to achieve “Always writable” in the presence of node failures/network partitions?



Hinted handoff

Hinted handoff: when A is temporarily down or unreachable during a write, send replica to D.

D is hinted that the replica is belong to A and it will deliver to A when A is recovered.



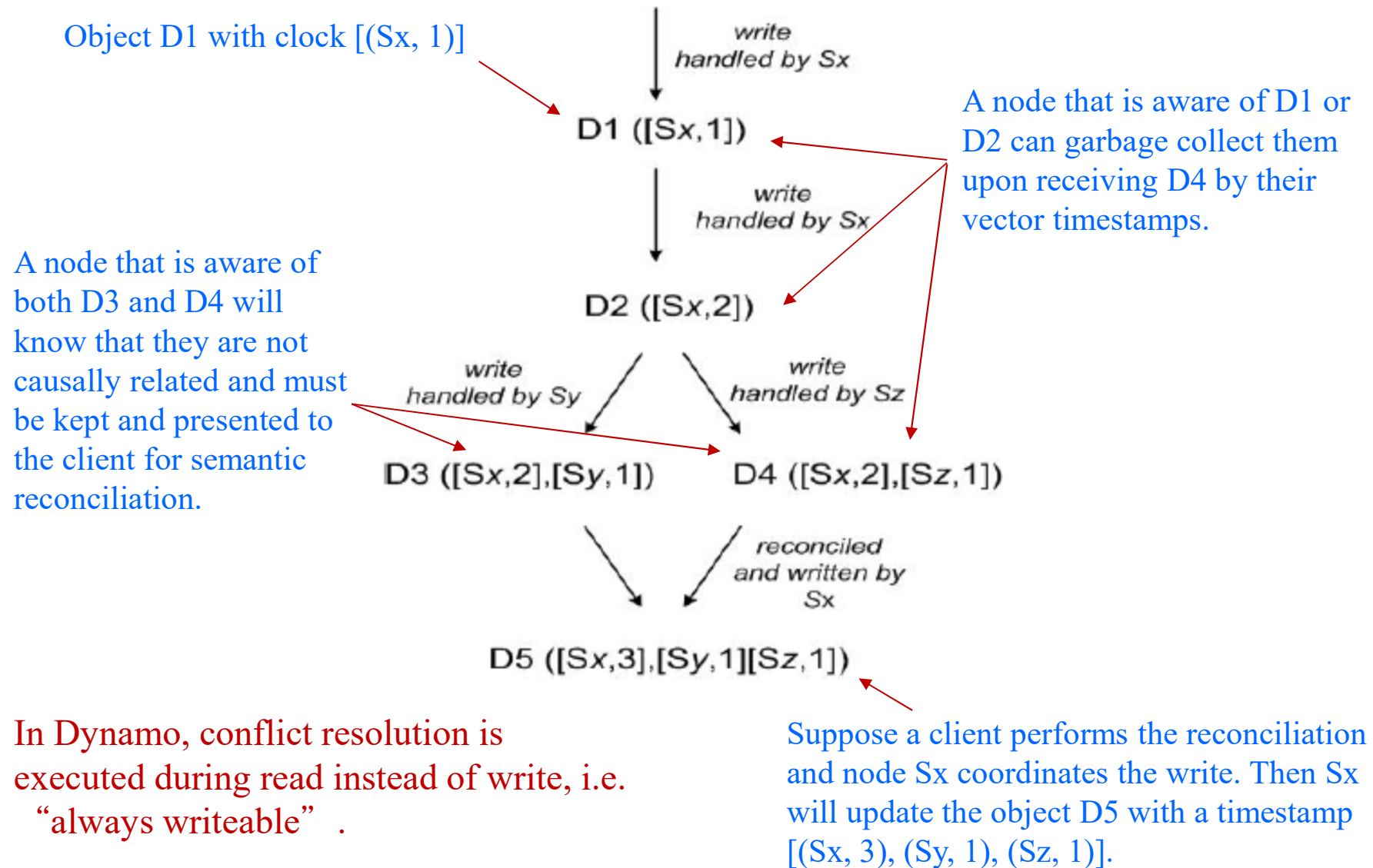
- ❑ With **hinted handoff**, all read and write operations are performed on the first N healthy nodes along the consistent hashing ring (“sloppy quorum”)
 - Setting W to 1 ensures that a write is rejected only if all nodes in the system are unavailable
- ❑ To tolerate the entire data center shutdown, the preference list of a key is constructed such that the storage nodes are spread across multiple data centers; i.e., each object is replicated across multiple data centers.

Concurrent Writes

- ❑ “Always writable” also implies that concurrent writes may occur during network partition.
- ❑ How to detect conflicts?
 - Vector Timestamps
- ❑ How to resolve conflicts?
 - Resolved during read, and by clients (applications)

Lamport, L. Time, clocks, and the ordering of events in a distributed system. *ACM Communications*, 21(7), pp. 558-565, 1978.

Detection of Data Conflicts



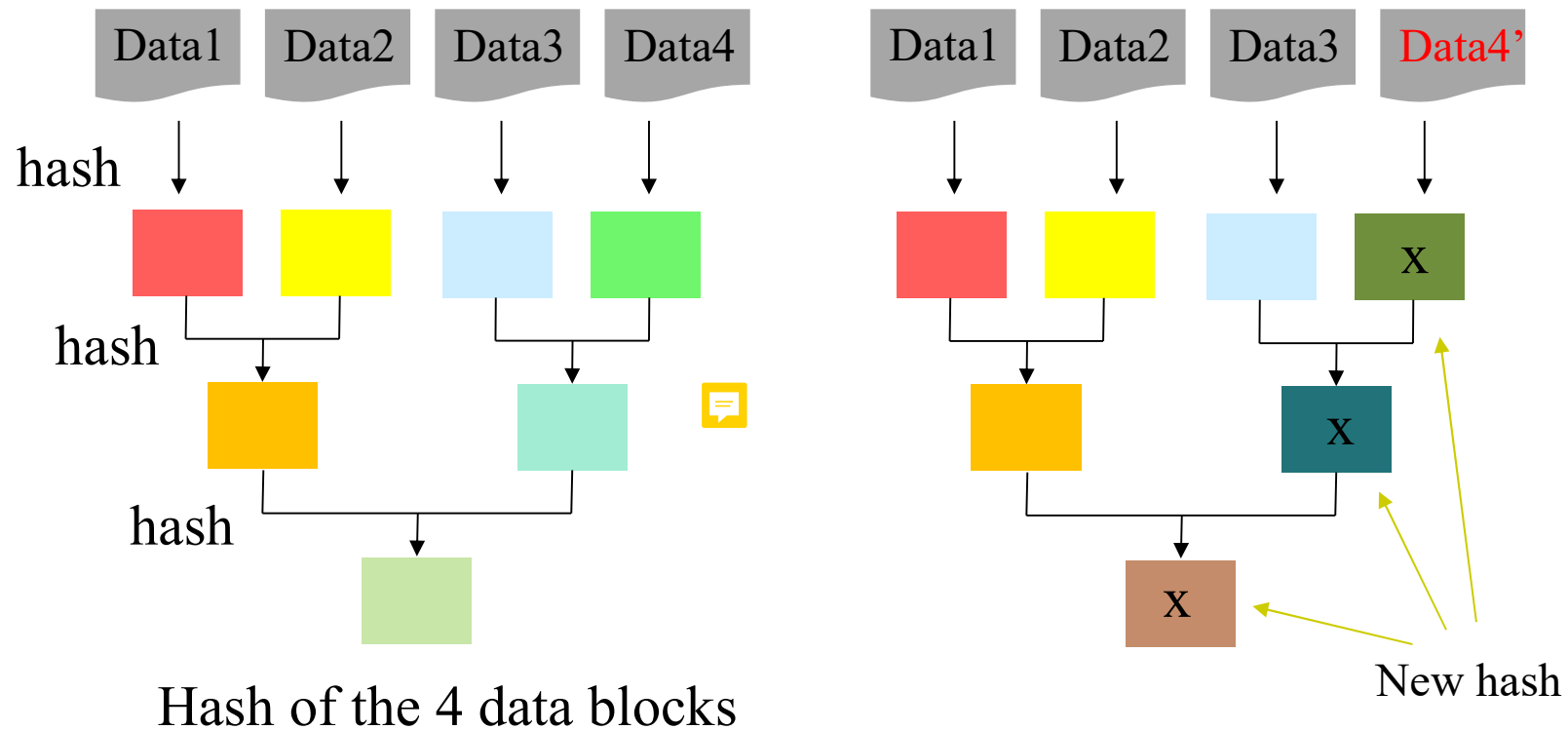
How to avoid the size of vector clocks grow too large?

- ❑ Writes are usually handled by one of the top N nodes in the preference list.
- ❑ The timestamp size grows when network is partitioned or multiple server fails.
- ❑ Set a threshold on the size such that when the number of (node, counter) pairs in the vector clock reaches the threshold (say 10), the oldest pair is removed from the clock.
 - This truncation scheme can lead to inefficiencies in reconciliation as the descendant relationships cannot be derived accurately. However, this problem has not surfaced in production.

Replica Synchronization

- ❑ Hinted handoff works best if the system membership churn is low and node failures are transient
- ❑ Dynamo implements a replica synchronization protocol to keep the replicas synchronized.
 - **Merkle trees** are used to detect the inconsistencies between replicas faster and to minimize the amount of transferred data
 - Each node maintains a separate Merkle tree for each key range (the set of keys covered by a virtual node) it hosts.
 - **Many trees need to be recalculated when a node joins or leaves the system**

Merkle Tree (Hash Tree)



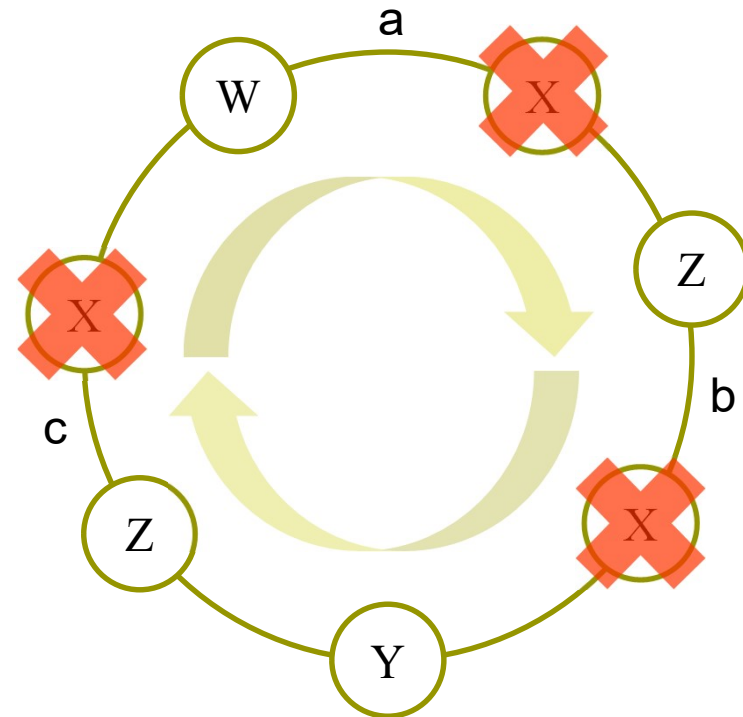
Hash trees allow efficient and secure verification of the contents of large data structures, as well as allow new data blocks to be added without affecting previous hashes (by keeping growing the tree).

Membership and Failure Detection

- ❑ Dynamo uses an explicit mechanism to initiate the addition and removal of nodes from a Dynamo ring
 - In Amazon's environment node outages (due to failures and maintenance tasks) are often transient and rarely signifies a permanent departure
 - So should not result in rebalancing of the partition assignment or repair of the unreachable replicas.
- The node that is informed of new members updates its membership and the change is propagated by a gossip-based protocol
 - If informed nodes are randomly chosen, temporary partition may occur
 - Some nodes are configured to play the role of **seeds** to avoid partition
- New nodes choose their virtual nodes in the ring and start to obtain their data by propagation.
- Earlier design used decentralized failure detection to detect temporary node failures

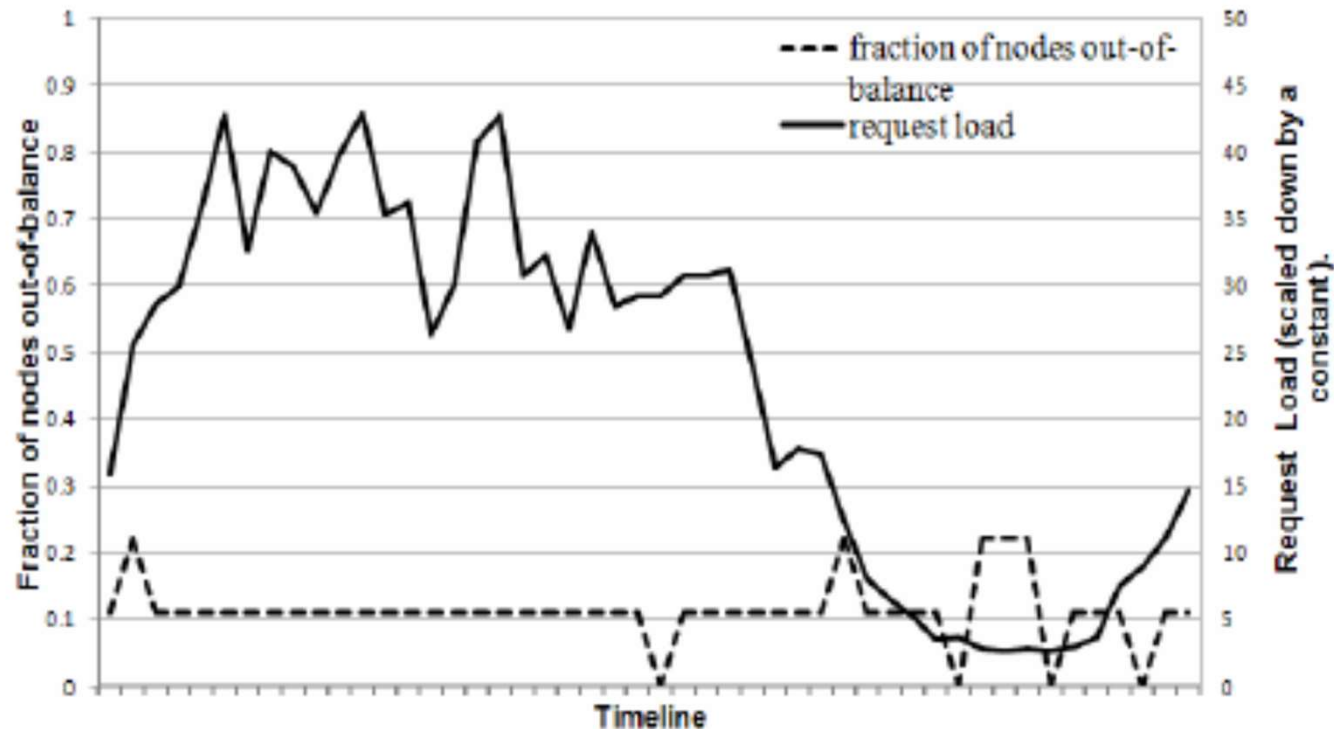
Adding/Removing Storage Nodes

- ❑ Node X has 3 virtual nodes
- ❑ When X fails, its load is evenly distributed to other nodes.
 - Key a → node Z
 - Key b → node Y
 - Key c → node W



Load Balance

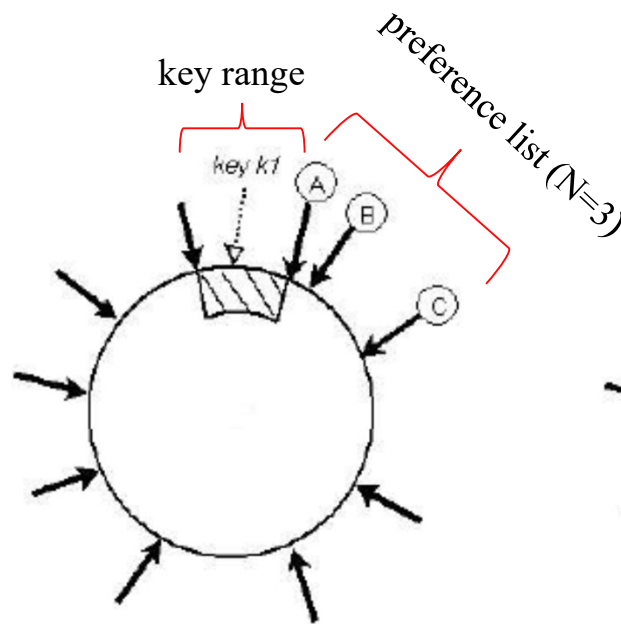
Imbalance ratio decreases with increasing load



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.

Partitioning and placement of keys



Strategy 1

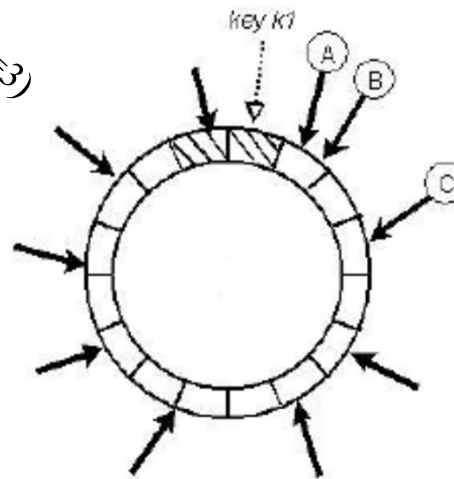
T random tokens per node
and partition by token value

Problems:

Imbalanced load

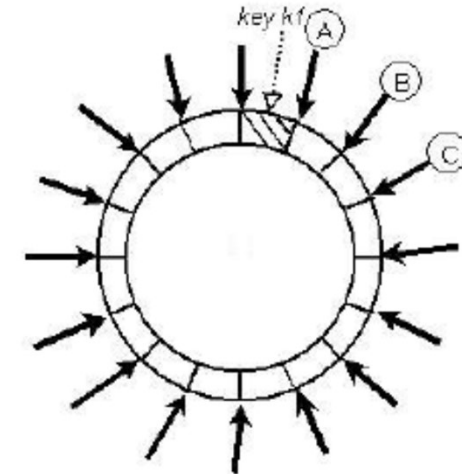
Long bootstrapping

Difficult to take a snapshot of keys



Strategy 2

T random tokens per node
and equal sized partitions



Strategy 3

Equal-sized partitions,
Q/S tokens per nodes,
where

Q: no. of partitions in hash
space

S: no. of nodes in system

Summary of techniques used in Dynamo

Problem	Technique	Advantage
Partitioning	Consistent Hashing	Incremental Scalability
High Availability for writes	Vector clocks with reconciliation during reads	Version size is decoupled from update rates.
Handling temporary failures	Sloppy Quorum and hinted handoff	Provides high availability and durability guarantee when some of the replicas are not available.
Recovering from permanent failures	Anti-entropy using Merkle trees	Synchronizes divergent replicas in the background.
Membership and failure detection	Gossip-based membership protocol and failure detection.	Preserves symmetry and avoids having a centralized registry for storing membership and node liveness information.

InterPlanetary File System (IPFS)

Reading:

- IPFS - Content Addressed, Versioned, P2P File System (DRAFT 3), Juan Benet, 2014.
- IPFS Concepts & Documents, IPFS.



Motivation and Goal

- ❑ Observation on current Web:
 - inefficient as lots of data duplication
 - expensive for distributing large files
 - can't preserve files
 - average lifespan of a web page is 100 days before it's gone forever (source: [IPFS](#))
 - centralized, limiting opportunity
 - addicted to the backbone
- ❑ Goal: To offer a new decentralized Internet infrastructure for building many different kinds of applications.
- ❑ Vision: **a permanent Web.**

Location-Based Addressing

https://cool.ntu.edu.tw/IM5007/Large_Distributed_Storage_Systems.pdf



resolve to location

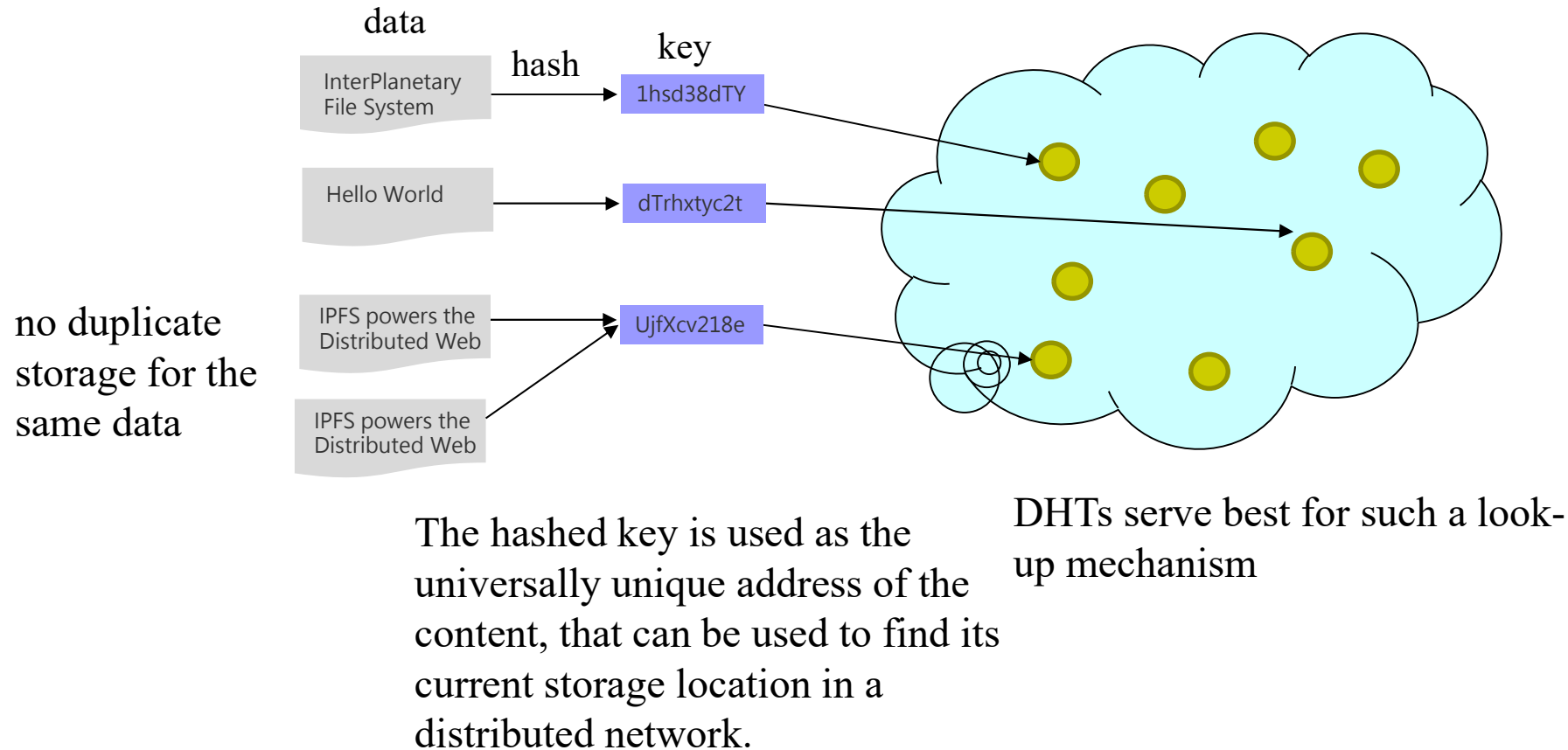
https://140.112.145.153/IM5007/Large_Distributed_Storage_Systems.pdf

HTTP Error 404 Not Found



Content Addressable & DHT

Distributed Hash Table



IPFS (InterPlanetary File System)

- ❑ IPFS: a “Content Addressed, Versioned, P2P File System “
- ❑ developed by [Juan Benet](#), who later founded [Protocol Labs](#) in 2014.05
 - [Filecoin](#) is also created by [Protocol Labs](#) (2014.07)
 - Protocol Labs [funding](#)
 - white paper: [IPFS - Content Addressed, Versioned, P2P File System](#), Juan Benet, 2014.
- ❑ Usage:
 - distributed file system
 - (encrypted) Content Delivery Network (CDN)
 - data storage of blockchains
 - permanent Web
 - ...



Key Ideas

- ❑ **Distributed Hash Table (DHT)** for locating data blocks based on their contents (content-addressable)
 - use Kademlia DHT, but with modifications from Coral DSHT and S/Kademlia for locality & security.
- ❑ **Block exchanges** based on **BitTorrent**, but with a modified incentive mechanism called Bitswap.
- ❑ **Merkle DAG** (Directed Acyclic Graph) version-based organization of files, similar to Git Version Control Systems
- ❑ **Self-Certified Filesystems (SFS)**
 - addressing remote filesystems:
 - /sfs/<Location>:<HostID>where Location is the server network address, and
 - HostID = hash(public_key || Location)

IPFS Protocol

- ❑ **Identities:** manage node identity generation and verification.
- ❑ **Network:** manages connections to other peers, uses various underlying network protocols.
- ❑ **Routing:** maintains information to locate specific peers and objects. Responds to both local and remote queries (defaults to a DHT, but swappable).
- ❑ **Block exchange (BitSwap):** governs efficient block distribution. Modelled as a market, weakly incentivizes data replication. Trade Strategies swappable.
- ❑ **Objects:** a Merkle DAG of content-addressed immutable objects with links. Used to represent arbitrary data structures, e.g. file hierarchies and communication systems.
- ❑ **Files:** versioned file system hierarchy inspired by Git.
- ❑ **Naming:** a self-certifying mutable name system.

Node & Object Identities

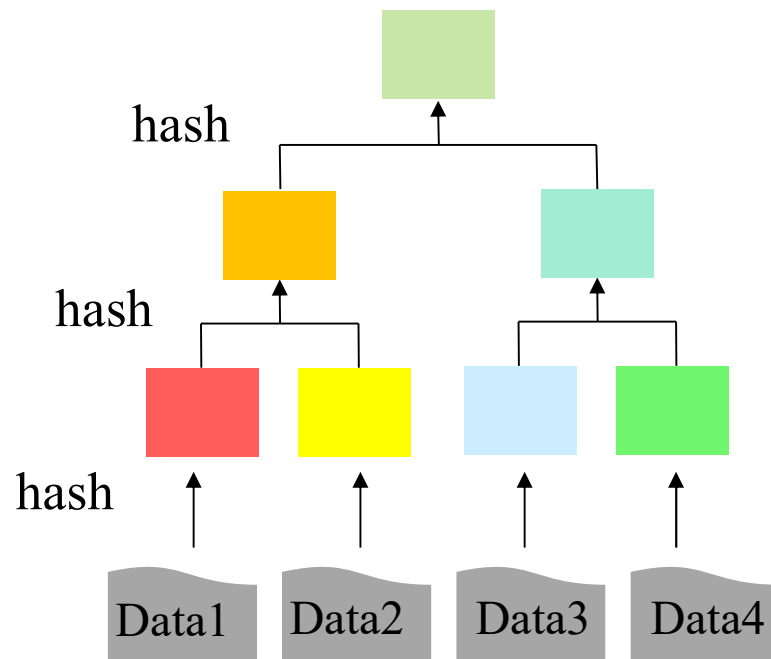
- ❑ Users are free to instantiate a “new” node identity on every launch, but are incentivized to remain the same to accrue network benefits.
- ❑ S/Kademlia-based node ID generation:

```
difficulty = <integer parameter>
n = Node{}
do {
    n.PubKey, n.PrivKey = PKI.genKeyPair()
    n.NodeId = hash(n.PubKey)
    p = count_preceding_zero_bits(hash (n.NodeId))
} while (p < difficulty)
```

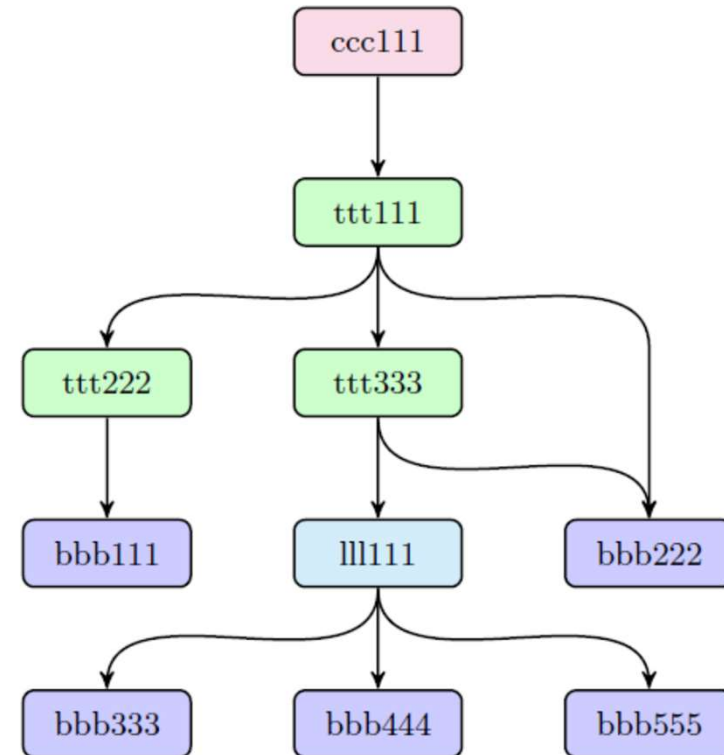
- ❑ Hash digests are stored in **multihash** format:
<function code><digest length><digest bytes>

Merkle DAGs

Hash of the 4 data blocks



Merkle Tree



Merkle DAG (source: [IPFS](#))

BitSwap

- ❑ core module of IPFS for exchanging blocks of data.
- ❑ Peers have a `want_list` (blocks looking to acquire) and `have_list` (blocks hold).
- ❑ Like BT's tit-for-tat, but with some modification to better resist free riding:
 - Peers track their balance (in bytes verified) with other nodes, e.g.,

$$\text{debt ratio } r = \frac{\text{byte_sent}}{\text{byte_recv} + 1}$$

- Peers send blocks to debtor peers probabilistically, according to a function that falls as debt increases, e.g.,

$$P(\text{send}|r) = 1 - \frac{1}{1 + e^{6-3r}}$$

BitSwap Ledger

- ❑ Peers keep their ledgers (bytes transfer) with other peers.
 - When activating a connection, peers exchange their ledger information. If it does not match exactly, the ledger is reinitialized from scratch, losing the accrued credit or debt.
 - Malicious nodes may purposefully “lose” the Ledger, hoping to erase debts.
 - however the partner node is free to count it as misconduct, and refuse to trade.
- ❑ BitSwap’s “**barter system**” implies a virtual currency could be created, this would require a global ledger to track ownership and transfer of the currency (see [Filecoin](#))

IPNS: InterPlanetary Name System

- ❑ IPFS serves to publish and retrieve immutable objects.
- ❑ IPNS serves to create an address whose content can be updated
 - Each user is assigned a mutable namespace at:
/ipns/<NodeId>
where
NodeId = hash(node.PubKey)

See <https://docs.ipfs.io/concepts/ipns/> for more details.