Distributed Systems
13. Distributed Lookup

Paul Krzyzanowski  
Rutgers University  
Spring 2020

Distributed Lookup

• Look up \((key, value)\)
• Cooperating set of nodes store data
• Ideally:
  – No central coordinator
  – Some nodes can be down

Approaches

1. Central coordinator
   – Napster

2. Flooding
   – Gnutella

3. Distributed hash tables
   – CAN, Chord, Amazon Dynamo, Tapestry, ...

1. Central Coordinator

• Example: Napster
  – Central directory
  – Identifies content (names) and the servers that host it
  – \(\text{lookup(name)} \rightarrow \{\text{list of servers}\}\)
  – Download from any of available servers
    • Pick the best one by pinging and comparing response times

• Another example: GFS
  – Controlled environment compared to Napster
  – Content for a given key is broken into chunks
  – Master handles all queries … but not the data

1. Central Coordinator - Napster

• Pros
  – Super simple
  – Search is handled by a single server (master)
  – The directory server is a single point of control
  • Provides definitive answers to a query

• Cons
  – Master has to maintain state of all peers
  – Server gets all the queries
  – The directory server is a single point of control
  • No directory, no service!

2. Query Flooding

• Example: Gnutella distributed file sharing

• Well-known nodes act as anchors
  – Nodes with files inform an anchor about their existence
  – Nodes select other nodes as peers
2. Query Flooding

- Send a query to peers if a file is not present locally
  - Each request contains:
    - Query key
    - Unique request ID
    - Time to Live (TTL, maximum hop count)

- Peer either responds or routes the query to its neighbors
  - Repeat until TTL = 0 or if the request ID has been processed
  - If found, send response (node address) to the requestor
  - **Back propagation**: response hops back to reach originator

**Overlay network**

An overlay network is a virtual network formed by peer connections
- Any node might know about a small set of machines
- "Neighbors" may not be physically close to you
What's wrong with flooding?

- Some nodes are not always up and some are slower than others
  - Gnutella & Kazaa dealt with this by classifying some nodes as special ("ultrapeers" in Gnutella, "supernodes" in Kazaa.)
- Poor use of network resources
- Potentially high latency
  - Requests get forwarded from one machine to another
  - Back propagation (e.g., in Gnutella’s design): replies go through the same sequence of systems used in the query, increasing latency even more – useful in preserving anonymity

3. Distributed Hash Tables

Hash tables

Remember hash functions & hash tables?

- Linear search: $O(N)$
- Tree: $O(\log N)$
- Hash table: $O(1)$

What’s a hash function? (refresher)

Hash function

- A function that takes a variable length input (e.g., a string) and generates a (usually smaller) fixed length result (e.g., an integer)
- Example: hash strings to a range 0-7:
  - $\text{hash(“Newark”)} \rightarrow 1$
  - $\text{hash(“Jersey City”)} \rightarrow 6$
  - $\text{hash(“Paterson”)} \rightarrow 2$

Hash table

- Table of (key, value) tuples
- Look up a key:
  - Hash function maps keys to a range 0...N-1
  - table of N elements
  - $i = \text{hash(key)}$
  - table[i] contains the item
- No need to search through the table!
Considerations with hash tables (refresher)

- **Picking a good hash function**
  - We want uniform distribution of all values of key over the space $0 \ldots N-1$

- **Collisions**
  - Multiple keys may hash to the same value
    - `hash("Paterson") → 2`
    - `hash("Edison") → 2`
  - Within `table[i]`, use a linked list or another layer of hashing

- **Think about a hash table that grows or shrinks**
  - If we add or remove buckets → need to rehash keys and move items

Distributed Hash Tables (DHT)

Create a peer-to-peer version of a (key, value) data store

How we want it to work
1. A peer (A) queries the data store with a key
2. The data store finds the peer (B) that has the value
3. That peer (B) returns the (key, value) pair to the querying peer (A)

Make it efficient!
- A query should not generate a flood!

Consistent hashing

- **Conventional hashing**
  - Practically all keys must be remapped if the table size changes

- **Consistent hashing**
  - Most keys will hash to the same value as before
  - On average, $K/n$ keys will need to be remapped
  - $K = \#$ keys, $n = \#$ of buckets

Example: splitting a bucket

Only the keys in slot c get remapped

3. Distributed hashing

- Spread the hash table across multiple nodes
- Each node stores a portion of the key space
  
  $\text{lookup(key)} \rightarrow \text{node ID that holds (key, value)}$
  
  $\text{lookup(node ID, key)} \rightarrow \text{value}$

Questions
- How do we partition the data & do the lookup?
- & keep the system decentralized?
- & make the system scalable (lots of nodes with dynamic changes)?
- & fault tolerant (replicated data)?

CAN design

- **Create a logical grid**
  - $x$-$y$ in 2-D (but not limited to two dimensions)

- **Separate hash function per dimension**
  - $h_x(key)$, $h_y(key)$

- **A node**
  - Is responsible for a range of values in each dimension
  - Knows its neighboring nodes
CAN key→node mapping: 2 nodes

\[ x = \text{hash}(\text{key}) \]
\[ y = \text{hash}(\text{key}) \]
\[ \text{if } x < \left( \frac{x_{\text{max}}}{2} \right) \]
\[ n_1 \text{ has (key, value)} \]
\[ \text{else} \]
\[ n_2 \text{ has (key, value)} \]
\[ n_0 \text{ is responsible for a zone } x^0 \cdots x^\left( \frac{x_{\text{max}}}{2} \right), \]
\[ y^0 \cdots y^{y_{\text{max}}} \]

CAN partitioning

Any node can be split in two – either horizontally or vertically

\[ x = \text{hash}(\text{key}) \]
\[ y = \text{hash}(\text{key}) \]
\[ \text{if } x < \left( \frac{x_{\text{max}}}{2} \right) \}
\[ \text{if } y < \left( \frac{y_{\text{max}}}{2} \right) \]
\[ n_0 \text{ has (key, value)} \]
\[ \text{else} \]
\[ n_1 \text{ has (key, value)} \]
\[ \text{if } x \geq \left( \frac{x_{\text{max}}}{2} \right) \]
\[ n_2 \text{ has (key, value)} \]

Any node can be split in two – either horizontally or vertically

Associated data has to be moved to the new node based on \( \text{hash}(\text{key}) \)

Neighbors need to be made aware of the new node

A node knows only of its neighbors

CAN routing

lookup(key) on a node that does not own the value

Compute \( \text{hash}(\text{key}), \text{hashy}(\text{key}) \) and route request to a neighboring node

Ideally: route to minimize distance to destination

Neighbors refer to nodes that share adjacent zones in the overlay network

\( n_4 \) only needs to keep track of \( n_5, n_7, \) or \( n_8 \) as its right neighbor.
**Performance**
- For \( n \) nodes in \( d \) dimensions
- \# neighbors = \( 2d \)
- Average route for 2 dimensions = \( O(\sqrt{n}) \) hops

**To handle failures**
- Share knowledge of neighbor’s neighbors
- One of the node’s neighbors takes over the failed zone

**Chord & consistent hashing**
- A key is hashed to an \( m \)-bit value: 0 … \( 2^m-1 \)
- A logical ring is constructed for the values 0 … \( 2^m-1 \)
- Nodes are placed on the ring at \( \text{hash}(\text{IP address}) \)

**Key assignment**
- Example: \( n=16 \); system with 4 nodes (so far)
- Key, value data is stored at a successor
  - a node whose value is \( \geq \text{hash(key)} \)

**Handling query requests**
- Any peer can get a request (insert or query). If the \( \text{hash(key)} \) is not for its ranges of keys, it forwards the request to a successor.
- The process continues until the responsible node is found
  - Worst case: with \( p \) nodes, traverse \( p-1 \) nodes; that’s \( O(N) \) (yuck!)
  - Average case: traverse \( p/2 \) nodes (still yuck!)

Let’s figure out three more things
1. Adding/removing nodes
2. Improving lookup time
3. Providing fault tolerance
Adding a node

- Some keys that were assigned to a node’s successor now get assigned to the new node.
- Data for those (key, value) pairs must be moved to the new node.

Removing a node

- Keys are reassigned to the node’s successor.
- Data for those (key, value) pairs must be moved to the successor.

Fault tolerance

- Nodes might die.
  - (key, value) data should be replicated.
  - Create $R$ replicas, storing each one at $R-1$ successor nodes in the ring.
- Need to know multiple successors.
  - A node needs to know how to find its successor’s successor (or more).
  - When a node is back up, it needs to check with successors for updates.
  - Any changes need to be propagated to all replicas.

Performance

- We’re not thrilled about $O(N)$ lookup.
  - Simple approach for great performance:
    - Have all nodes know about each other.
    - When a peer gets a node, it searches its table of nodes for the node that owns those values.
    - Gives us $O(1)$ performance.
    - Add/remove node operations must inform everyone.
    - Maybe not a good solution if we have lots of peers (huge tables).

Finger tables

- Compromise to avoid large tables at each node.
  - Use finger tables to place an upper bound on the table size.
- Finger table = partial list of nodes, progressively more distant.
- At each node, $i$th entry in finger table identifies node that succeeds it by at least $2^{i-1}$ in the circle.
  - $\text{finger_table}[0]$: immediate (1st) successor.
  - $\text{finger_table}[1]$: successor after that (2nd).
  - $\text{finger_table}[3]$: 8th successor.
  - ...
- $O(\log N)$ nodes need to be contacted to find the node that owns a key … not as cool as $O(1)$ but way better than $O(N)$.

Improving performance even more

- Let’s revisit $O(1)$ lookup.
- Each node keeps track of all current nodes in the group.
  - Is that really so bad?
  - We might have thousands of nodes … so what?
- Any node will now know which node holds a (key, value).
- Add or remove a node: send updates to all other nodes.
Distributed Hashing Case Study

Amazon Dynamo

- Not exposed as a web service
  - Used to power parts of Amazon Web Services and internal services
  - Highly available, key-value storage system
- In an infrastructure with millions of components, something is always failing!
  - Failure is the normal case
- A lot of services within Amazon only need primary-key access to data
  - Best seller lists, shopping carts, preferences, session management, sales rank, product catalog
  - No need for complex querying or management offered by an RDBMS
  - Full relational database is overkill: limits scale and availability
  - Still not efficient to scale or load balance RDBMS on a large scale

Core Assumptions & Design Decisions

- Two operations: get(key) and put(key, data)
  - Binary objects (data) identified by a unique key
  - Objects tend to be small (< 1MB)
- ACID gives poor availability
  - Use weaker consistency (C) for higher availability.
- Apps should be able to configure Dynamo for desired latency & throughput
  - Balance performance, cost, availability, durability guarantees.
- At least 99.9% of read/write operations must be performed within a few hundred milliseconds:
  - Avoid routing requests through multiple nodes
  - Dynamo can be thought of as a zero-hop DHT

Core Assumptions & Design Decisions

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

Consistency & Availability

- Strong consistency & high availability cannot be achieved simultaneously
- Optimistic replication techniques – eventually consistent model
  - Propagate changes to replicas in the background
  - Can lead to conflicting changes that have to be detected & resolved
- When do you resolve conflicts?
  - During writes: traditional approach – reject write if cannot reach all (or majority) of replicas – but don't deal with conflicts
  - Resolve conflicts during reads: Dynamo approach
  - Design for an “always virtua” data store – highly available
  - Read/write operations continue even during network partitions
  - Rejecting customer updates won’t be a good experience
  - A customer should always be able to add or remove items in a shopping cart

Consistency & Availability

- Who resolves conflicts?
  - Choices: the data store system or the application?
- Data store
  - Application-unaware, so choices limited
  - Simple policy, such as “last write wins”
- Application
  - App is aware of the meaning of the data
  - Can do application-aware conflict resolution
  - E.g., merge shopping cart versions to get a unified shopping cart.
- Fall back on “last write wins” if app doesn’t want to bother
Reads & Writes

Two operations:

- `get(key)` returns
  1. object or list of objects with conflicting versions
  2. context (resultant version per object)

- `put(key, context, value)`
  - stores replicas
  - context: ignored by the application but includes version of object
  - key is hashed with MD5 to create a 128-bit identifier that is used to determine the storage nodes that serve the key
    hash(key) identifies node

Partitioning the data

- Break up database into chunks distributed over all nodes
  - Key to scalability

- Relies on consistent hashing
  - K/n keys need to be remapped, K = # keys, n = # slots

- Logical ring of nodes: just like Chord
  - Each node assigned a random value in the hash space: position in ring
  - Responsible for all hash values between its value and predecessor’s value
  - Hash(key), then walk ring clockwise to find first node with position>hash
  - Adding/removing nodes affects only immediate neighbors

Partitioning: virtual nodes

- A node is assigned to multiple points in the ring
- Each point is a “virtual node”

Dynamo virtual nodes

- A physical node holds contents of multiple virtual nodes
- In this example: 2 physical nodes, 5 virtual nodes

Replication

- Data replicated on N hosts (N is configurable)
  - Key is assigned a coordinator node (via hashing) = main node
  - Coordinator is in charge of replication
  - Coordinator replicates keys at the N-1 clockwise successor nodes in the ring

Partitioning: virtual nodes

**Advantage: balanced load distribution**

- If a node becomes unavailable, load is evenly dispersed among available nodes
- If a node is added, it accepts an equivalent amount of load from other available nodes
- # of virtual nodes per system can be based on the capacity of that node
- Makes it easy to support changing technology and addition of new, faster systems
Dynamo Replication
 Coordinator replicates keys at the N-1 clockwise successor nodes in the ring

Example: N=3

Node 14 holds replicas for Nodes 1 and 3
Node 10 holds replicas for Node 14 and 1
Node 8 holds replicas for Nodes 10 and 14

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

Availability
• Configurable values
  – R: minimum # of nodes that must participate in a successful read operation
  – W: minimum # of nodes that must participate in a successful write operation
• Metadata to remember original destination
  – If a node was unreachable, the replica is sent to another node in the ring
  – Metadata sent with the data states the original desired destination
  – Periodically, a node checks if the originally targeted node is alive
  – If so, it will transfer the object and may delete it locally to keep # of replicas in the system consistent
• Data center failure
  – System must handle the failure of a data center
  – Each object is replicated across multiple data centers

Storage Nodes
Each node has three components
1. Request coordination
  – Coordinator executes read/write requests on behalf of requesting clients
  – State machine contains all logic for identifying nodes responsible for a key, sending requests, waiting for responses, retries, processing retries, packaging response
  – Each state machine instance handles one request
2. Membership and failure detection
3. Local persistent storage
  – Different storage engines may be used depending on application needs
    • Berkeley Database (BDB) Transactional Data Store (most popular)
    • BDB Java Edition
    • MySQL (for large objects)
    • In-memory buffer with persistent backing store

Amazon S3 (Simple Storage Service)
Commercial service that implements many of Dynamo’s features
• Storage via web services interfaces (REST, SOAP, BitTorrent)
  – Stores more than 449 billion objects
  – 99.9% uptime guarantee (43 minutes downtime per month)
  – Proprietary design
  – Stores arbitrary objects up to 5 TB in size
• Objects organized into buckets and within a bucket identified by a unique user-assigned key
• Buckets & objects can be created, listed, and retrieved via REST or SOAP
  – http://s3.amazonaws.com/key
• Objects can be downloaded via HTTP GET or BitTorrent protocol
  – S3 acts as a seed host and any BitTorrent client can retrieve the file
  – reduces bandwidth costs
• S3 can also host static websites

The end