Distributed Systems

20. Distributed Lookup

Paul Krzyzanowski
Rutgers University
Fall 2014

Distributed Lookup

• Look up \((key, value)\)
• Cooperating set of nodes
• 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
  – \(\text{lookup}(\text{name})\) → (list of servers)
  – Download from any of available servers
    • Pick the best one by pinging and comparing response times

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!

1. Central Coordinator

• Another example: GFS
  – Controlled environment compared to Napster
  – Content for a given key is broken into chunks
  – Master handles all queries

• How about Bigtable?
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

- 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: series of responses reaches originator

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

Overlay network

Underlying IP Network

Overlay Network

Flooding Example: Overlay Network

Flooding Example: Query Flood
Flooding Example: Query response

Flooding Example: Download

Flooding

- Problems
  - For gnutella
    - Nodes not always up and some are much slower than others
    - All treated as peers
    - Flooding is not an efficient use of network resources
    - Back propagation may require a high hop count

3. Distributed Hash Tables

Locating content

- How do we locate distributed content?
  - A central server is the easiest

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Napster</td>
<td>Central server</td>
</tr>
<tr>
<td>Gnutella &amp; Kazaa</td>
<td>Network flooding (optimized to flood supernodes) but it’s still flooding</td>
</tr>
<tr>
<td>BitTorrent</td>
<td>Nothing! It’s somebody else’s problem</td>
</tr>
</tbody>
</table>

- Can we do better?

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 “supernodes” (called “ultrapeers” in Gnutella)

- Poor use of network resources

- Potentially high latency
  - Requests get forwarded from one machine to another
  - Back propagation (e.g., in Gnutella’s design), where the replies go through the same chain of machines used in the query, increases latency even more
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 $0 \ldots N-1$
    • Table of $N$ elements
    • $i = \text{hash(key)}$
    • $\text{table}[i]$ 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
    • $\text{hash(“Paterson”)} \rightarrow 2$
    • $\text{hash(“Edison”)} \rightarrow 2$
    • $\text{table}[i]$ is a bucket (slot) for all such (key, value) sets
      • Within $\text{table}[i]$, 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) database

• How we want it to work
  1. A peer (A) queries the database with a key
  2. The database 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 have to 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
Distributed Hashing Case Study

CAN: Content Addressable Network

**CAN design**

- Create a logical grid
  - x-y in 2-D but not limited to 2-D
- 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:**

- $x = h_x(key)$
- $y = h_y(key)$
- If $x < (x_{max}/2)$
  - $n_1$ has (key, value)
- If $x \geq (x_{max}/2)$
  - $n_2$ has (key, value)

$n_i$ is responsible for a zone $x(x_{max}/2 - x_{max}) / y(y_{max} - y_{max})$

**CAN partitioning**

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

Associated data has to be moved to the new node based on $hash(key)$
Neighbors need to be made aware of the new node
A node knows only of its neighbors
Distributed Systems

CAN neighbors

Neighbors refer to nodes that share adjacent zones in the overlay network. $n_i$ only needs to keep track of $n_i$, $n_{i+1}$ or $n_{i-1}$ as its right neighbor.

CAN routing

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

Compute $hash_x(key)$, $hash_y(key)$ and route request to a neighboring node

Ideally: route to minimize distance to destination

CAN

• 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

Distributed Hashing Case Study

Chord

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 $hash(IP address)$

Key assignment

• Example: $n=16$; system with 4 nodes (so far)
• Key, value data is stored at a successor — a node whose value is $\geq hash(key)$. No nodes at these empty positions

Node 14 is responsible for keys 11, 12, 13, 14
Node 3 is responsible for keys 5, 6, 7, 8
Node 10 is responsible for keys 9, 10
Node 11 is responsible for keys 1, 2, 3
Node 8 is responsible for keys 4, 5, 6, 7, 8
Handling query requests

• Any peer can get a request (insert or query). If the 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: traverse p-1 nodes; that’s O(N) (yuck!)
  – Average case: traverse p/2 nodes (still yuck!)

Node 3 is responsible for keys 15, 0, 1, 2, 3
Node 8 is responsible for keys 4, 5, 6, 7, 8
Node 10 is responsible for keys 9, 10
Node 14 is responsible for keys 11, 12, 13, 14

Let’s figure out three more things

1. Adding/removing nodes
2. Improving lookup time
3. 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

Node 3 is responsible for keys 15, 0, 1, 2, 3
Node 8 is responsible for keys 4, 5, 6, 7, 8
Node 10 is responsible for keys 9, 10
Node 14 is responsible for keys 11, 12, 13, 14

Removing a node

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

Node 3 is responsible for keys 15, 0, 1, 2, 3
Node 8 is responsible for keys 4, 5, 6, 7, 8
Node 10 is responsible for keys 9, 10
Node 14 is responsible for keys 11, 12, 13, 14

Fault tolerance

• Nodes might die
  – (key, value) data would need to be replicated
  – Create R replicas, storing each one at R-1 successor nodes in the ring
• It gets a bit complex
  – A node needs to know how to find its successor’s successor (or more)
  – Easy if it knows all nodes!
  – 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 millions 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
- At each node, i\textsuperscript{th} entry in finger table identifies node that succeeds it by at least 2\textsuperscript{i-1} in the circle
  - finger_table[0]: immediate (1\textsuperscript{st}) successor
  - finger_table[1]: successor after that (2\textsuperscript{nd})
  - finger_table[2]: 4\textsuperscript{th} successor
  - finger_table[3]: 8\textsuperscript{th} 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

Amazon Dynamo

- Not exposed as a web service
  - Used to power parts of Amazon Web Services (such as S3)
  - 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 easy 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
Compared to Google Bigtable

- Dynamo targets apps that only need key/value access with a primary focus on high availability
  - *key-value store versus column-store* (column families and columns within them)
  - Bigtable: distributed DB built on GFS
  - Dynamo: distributed hash table
  - Updates are not rejected even during network partitions or server failures

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
  - Resolve during reads: *Dynamo approach*
    - Design for an "always writable" data store - highly available
    - Read/Write operations can 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
  - The nodes that hold replicas are based on the key.
    - 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

- Break up database into chunks distributed over all nodes
  - Key to scalability
  - Example: Bigtable’s tablets, Map-Reduce partitioning

- Relies on *consistent hashing*
  - Regular hashing: change in # slots requires all keys to be remapped
  - Consistent hashing:
    - All keys need to be remapped, $K = \# keys, n = \# slots$

- Logical ring of nodes: just like Chord
  - Each node assigned 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

Partitioning: virtual nodes

• Advantages: 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

Replication

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

Dynamo Replication

• Coordinator replicates keys at the $N-1$ clockwise successor nodes in the ring

Versioning

• Not all updates may arrive at all replicas
• 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 hints to remember original destination
  – If a node was unreachable, the replica is sent to another node in the ring
  – Metadata sent with the data contains a hint stating 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 service (powered by Dynamo)**

- 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/bucket/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**