Distributed Systems

17. Distributed Lookup

Paul Krzyzanowski
Rutgers University
Fall 2016

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
  - – *lookup(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 … but not the data

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

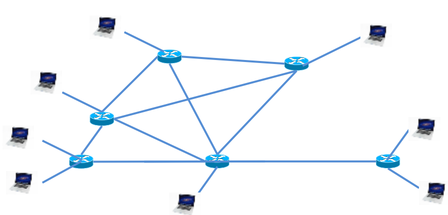
2. Query Flooding

- 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**: series of responses reaches originator

Overlay network

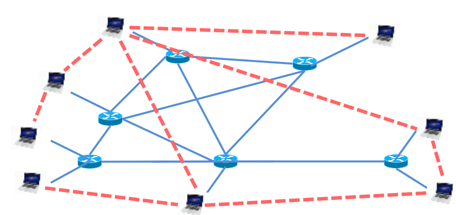
An overlay network is a virtual network formed by peer connections
- Any node might know about a small set of machines
- "Neighbors" may not be physically close to you



Underlying IP Network

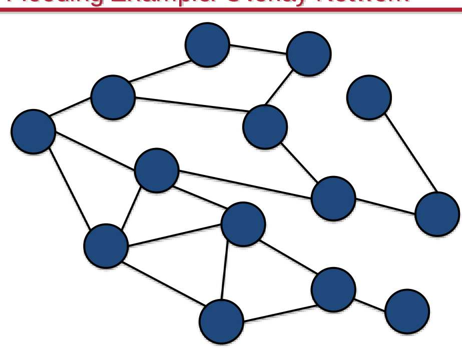
Overlay network

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

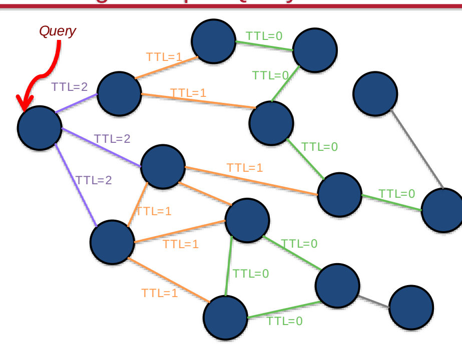


Overlay Network

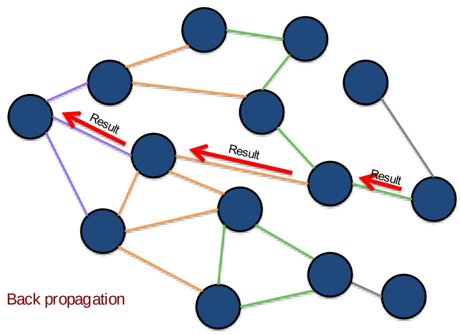
Flooding Example: Overlay Network



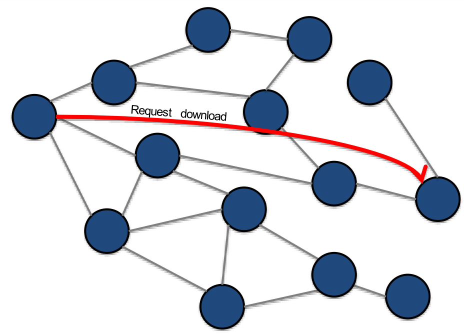
Flooding Example: Query Flood



Flooding Example: Query response



Flooding Example: Download



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

3. Distributed Hash Tables

Locating content

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

Napster	Central server
Gnutella & Kazaa	Network flooding Optimized to flood supernodes … but it's still flooding
BitTorrent	Nothing! It's somebody else's problem

- Can we do better?

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:
    - *hash("Newark") → 1*
    - *hash("Jersey City") → 6*
    - *hash("Paterson") → 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 = 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 … N-1*
- Collisions
  - Multiple keys may hash to the same value
    - hash("Paterson") → 2
    - hash("Edison") → 2
  - `table[i]` is a bucket (slot) for all such *(key, value)* sets
  - 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)

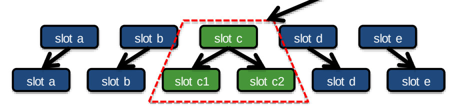
- Create a peer-to-peer version of a (*key, value*) data store
- How we want it to work
  1. A peer (*A*) queries the data store with a key
  2. The data store finds the peer (*B*) that has the value
  3. That peer (*B*) returns the (key, value) pair to the querying peer (*A*)
- Make it efficient!
  - A query should not generate a flood!

Consistent hashing

- Conventional hashing
  - Practically all keys 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



3. Distributed hashing

- Spread the hash table across multiple nodes
- Each node stores a portion of the key space
- *lookup(key) → node ID* that holds (*key*, *value*)
- Questions

  How do we partition the data & do the lookup?

  & keep the system decentralized?

  & make the system scalable (lots of nodes)?

  & fault tolerant (replicated data)?

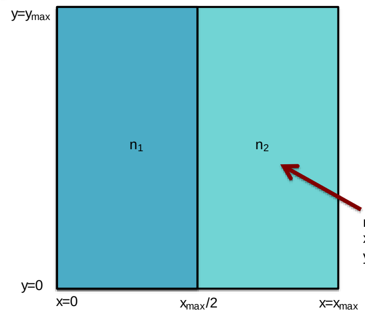
Distributed Hashing Case Study

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: 2 nodes



$x = hash_x(key)$

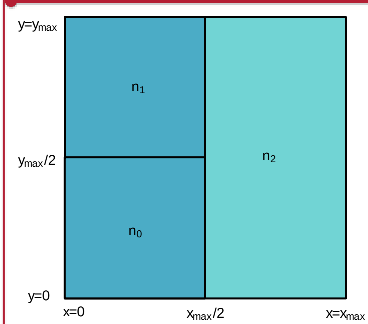
$y = hash_y(key)$

if $x < (x_{max}/2)$
n_1 has (*key, value*)

if $x \geq (x_{max}/2)$
n_2 has (*key, value*)

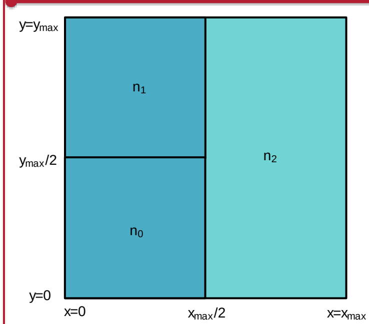
n_2 is responsible for a zone
$x=(x_{max}/2 .. x_{max})$,
$y=(0 .. y_{max})$

CAN partitioning



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

CAN key→node mapping



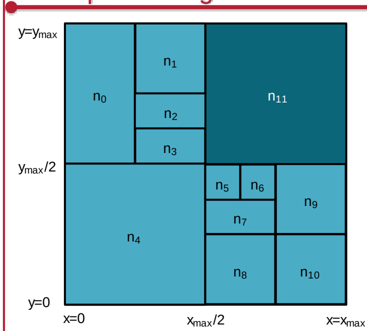
$x = hash_x(key)$

$y = hash_y(key)$

if $x < (x_{max}/2)$ {
  if $y < (y_{max}/2)$
    n_0 has (*key, value*)
  else
    n_1 has (*key, value*)
}

if $x \geq (x_{max}/2)$
n_2 has (*key, value*)

CAN partitioning



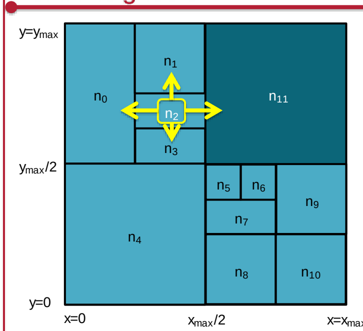
Any node can be split in two – either horizontally or vertically

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

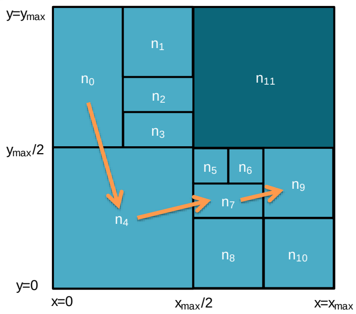
CAN neighbors



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.

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

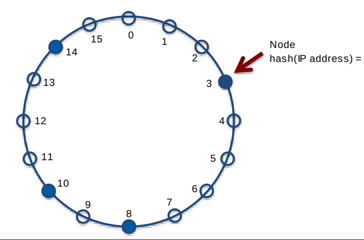
- Performance
  - For *n* nodes in *d* dimensions
  - # neighbors = *2d*
  - Average route for 2 dimensions = $O(\sqrt{n})$ hops
- To handle failures
  - Share knowledge of neighbor's neighbors
  - One of the node's neighbors takes over the failed zone

Distributed Hashing Case Study

Chord

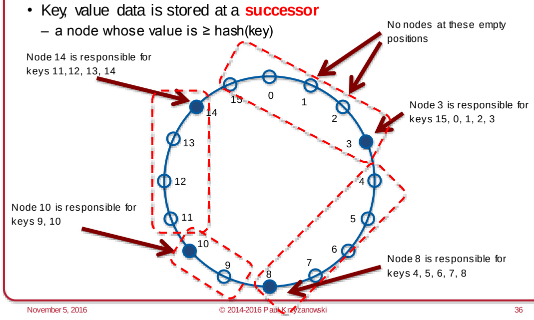
Chord & consistent hashing

- 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)*



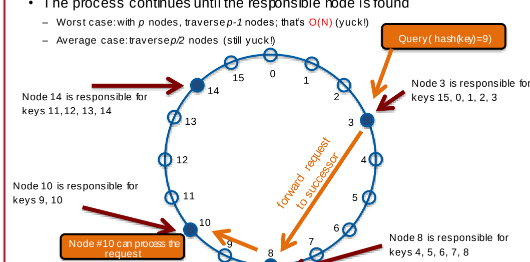
Key assignment

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



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: with *p* nodes, traverse *p-1* nodes; that's O(N) (yuck!)
  - Average case: traverse *p/2* nodes (still yuck!)

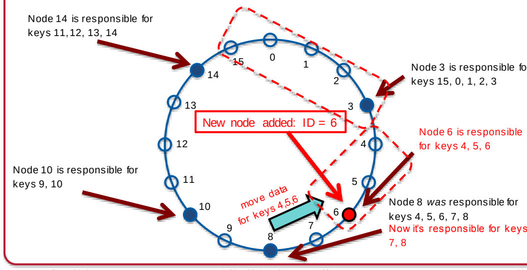


Let's figure out three more things

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

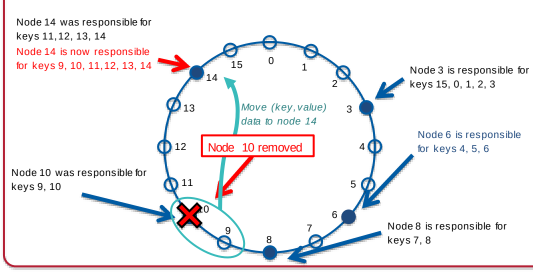
Adding a node

- 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 would need to be replicated
  - Create *R* replicas, storing each one at *R-1* successor nodes in the ring
- Need to know successors
  - 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^{th} entry in finger table identifies node that succeeds it by at least 2^{i-1} in the circle
  - finger_table[0]: immediate (1st) successor
  - finger_table[1]: successor after that (2nd)
  - finger_table[2]: 4th successor
  - 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

The end

