Advanced Data Management Technologies

Unit 20 — Distributed Hash Tables

J. Gamper

Free University of Bozen-Bolzano
Faculty of Computer Science
IDSE

Acknowledgements: Some slides are from Paul Kryzanowski and Jeff Pang.

Outline

1. **Introduction & Motivation**

2. **Linear Hashing**
   - Centralized Solution
   - Distributed Solution

3. **Consistent Hashing**

Outline

Locating Content in Distributed Systems

- An important issue in P2P applications is content distribution
  - Where to distribute the data and how to locate the data?
- Possible solutions for data/file sharing in P2P systems
  - Central server, e.g., Napster
    - Single point of failure and bottleneck
  - No central server, network flooding, e.g., Gnutella & Kazaa
    - Optimized to flood supernodes ... but it is still flooding

What is 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., Gnutella design), where the replies go through the same chain of machines used in the query, increases latency even more

- Better access structures are needed to make P2P systems scalable!

Direct Access Structures

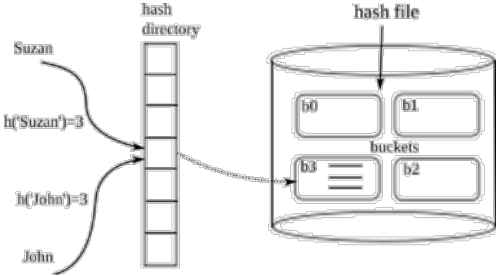
- For point queries, file scan becomes too expensive, and direct access (or index) structures are needed.
- Index on a collection C of data
  - Maps the key of each object in C to its (physical) address
  - A set of pairs (k, a), where k is a key and a the address of an object
  - Object can be raw data, relational tuple, XML document, picture, video, etc.
- Index supports also
  - range queries if keys can be linearly ordered
    - `range(k1,k2)` retrieves all keys (and their addresses) in that range
  - nearest neighbor queries if key space is associated to a metric (a distance function)
- Three main families of access structures:
  - hash tables: constant search complexity – $O(1)$
  - search trees: logarithmic search complexity – $O(\log N)$
  - linear search: linear search complexity – $O(N)$
- We are going to concentrate on hash tables

Distributed Hash-based Solutions

- Aim is to create a peer-to-peer version of a (*key*, *value*) database
  - Distribute data over a large P2P network
  - Quickly find an item in the P2P network
    - a peer queries the database with a key
    - the database finds the peer that has the value
    - that peer returns the (key, value) pair to the querying peer
- Make it efficient!
  - Avoid flooding
- Basic (dictionary) operations
  - insertion: `insert(k,v)`
  - key search: `v = search(k)`
  - deletion: `delete(k)`

Hash-based Index in Centralized DB

- Hash file structure for a data collection C consists of
  - a set of M disk buckets $\{b_0, b_1, \ldots, b_{M-1}\}$ and
  - a memory-resident directory D, where D is an array with M cells, each referring to one of the buckets
- Hash function h determines the placement of objects in the M buckets
  - h maps each item $I \in C$ to the range $[0, M-1]$
  - Item $I \in C$ is stored in bucket b_j if $j = h(I.A)$
    - A is sometimes called the hash field
- Properties
  - Hash function should uniformly assigns objects to buckets
  - M should be of the order $\lceil \frac{|C|}{\text{bucket size}} \rceil$
  - Very efficient for point queries, but does not support range search.



Distributed Hash-based Index – Naive Solution

- Naive solution
  - assign each bucket of the hash file to one of the participating servers and
  - share hash function among all nodes
- Suppose servers $S_0, \ldots, S_{N-1}$ are available
- Hash function $h(key) = \overline{h}(key) \mod N$, where $\overline{h}$ maps the keys to integers.
- Assign each key with hash value i to server S_i
- If a server S_N is added, the hash function is modified to
  $h(key) = \overline{h}(key) \mod (N+1)$

Problems with Naive Solution

- Distributed systems are (highly) dynamic
  - Data sets evolve over time
  - Nodes are added and deleted
- If the hash function changes, the hash value of most objects changes too
  - Requires essentially a total rebuilding of the hash file
  - New function h has to be transmitted to all participants
  - During these changes, the old hash function is likely to result in an error (difficult to guarantee consistency)
- Hash directory (if stored centrally) provides a bottleneck as it needs to be accessed for each request

Outline

Centralized Linear Hashing (LH)

- Goal
  - An efficient hash structure for a very dynamic collection of data
- Simple solution is to use overflow buckets
  - But problematic if there are many of them (linear scan!)
- Basic idea of (centralized) linear hashing (LH)
  - Dynamic enlargement of hash directory D and hash function h
  - Reorganization of buckets

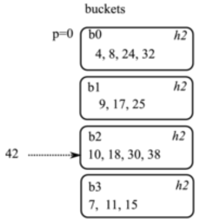
Insert in Centralized LH

- Insert a new data item: `insert(k,v)`
- Buckets $b_0, \ldots, b_{N-1}$
- Split pointer p points to the bucket to be split next
  - Initially $p = 0$
- Two hash functions (h_n, h_{n+1}) are used:
  - h_n applies to the buckets $b_p, \ldots, b_{N-1}$
  - h_{n+1} to all other buckets
- When a bucket b overflows, the following steps are done:
  - an overflow bucket is linked from b to store the new item
  - bucket b_p corresponding to p is split (typically diff. from overflow bucket!)
  - p is incremented by 1
- When (the last) bucket b_{N-1} is split, h_n is no longer used
  - Hash file "switches" to next level, i.e., hash functions (h_{n+1}, h_{n+2}) are used
  - p is reset to $p = 0$
  - (The number of buckets has doubled)

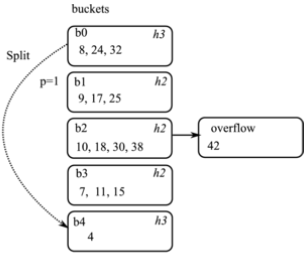
Example of Centralized LH

- Size of hash directory is 4, each bucket holds at most 4 objects
- Actual hash functions (h_2, h_3): $h_2(k) = k \mod 2^2$, $h_3(k) = k \mod 2^3$

  - A new object 42 is inserted into bucket b_2



  - A new bucket is added to b_2; bucket b_0 is split and h_3 applies to b_0; p is set to 1



- When b_3 is split $(p = 3)$, h_3 applies to all buckets, hence the hash file moves to the next level: hash functions (h_3, h_4) and split pointer $p = 0$

Lookup in Centralized LH

- The two hash functions are (h_n, h_{n+1})
- Lookup(k)

  $a = h_n(k)$;
  if $(a < p)$ **then** $a = h_{n+1}(k)$;
  return a

Properties of Centralized LH

- LH provides a linear growth of the file (one bucket at a time)
- Bucket that overflows is not split, but an overflow bucket is added
  - This bucket will eventually be split when the split pointer points to it
  - Delayed management of collision overflows
- A large part of the hash directory remains unchanged when the hash function is modified
  - Not many data need to be reorganized
  - In a distributed environment this avoids to resend the complete directory to the other nodes
- Similar to extendable hashing, where the hash directory growths not so gracefully (i.e., doubles when new hash values are needed)

Distributed Linear Hashing (LH*)/1

- Let
  - n be the hash file level,
  - (h_n, h_{n+1}) be the hash functions, and
  - p be the split pointer
- Assume servers $S_0, S_1, \ldots, S_N$, where $2^n \leq N < 2^{n+1}$.
- Each server holds one bucket
- If server S_i overflows
  - Add an overflow bucket to S_i
  - Split (the bucket on the) server S_p.
    - Allocate a new server S_{N+1} to the hash structure (might be the same physical server hosting several virtual servers)
    - Some objects are transferred from S_p to S_{N+1}.

Distributed Linear Hashing (LH*)/2

- LH* does not require resending entirely the hash directory each time the hash function is modified or nodes are added/deleted
- Only the following localization information needs to be communicated:
  - level n that determines the pair of hash functions (h_n, h_{n+1}) currently in use
  - current split pointer p
  - changes of the hash directory
- If the number of peers grows rapidly, this might still be a lot of overhead.
  - More lightweight maintenance solutions are desirable!

Lazy Adjustment to Reduce LH* Maintenance Cost

- Each peer maintains a local image that records partial information about the distributed hash structure, i.e.,
  - n, p, and a partial replication of the hash directory D
- Local image might be outdated for several reasons:
  - Peer is temporarily disconnected
  - An asynchronous replication protocol is used
  - Update is complex and expensive if clients are frequently connected/disconnected
- A "reasonably outdated" image represents a good trade-off, provided that the client knows how to cope with lookup errors and outdated information.

Lookup in LH* with the Forward Algorithm

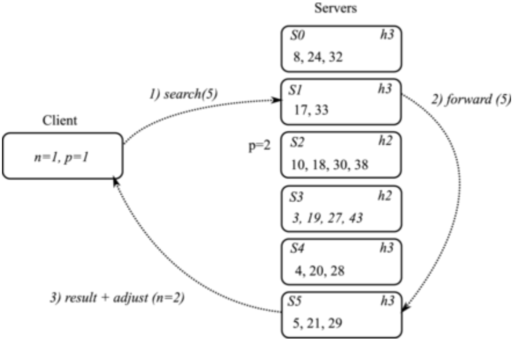
- Let k be the search key
- Client
  - Compute the bucket address a of k using the Lookup algorithm of LH
  - Send the request to server S_a
- Server
  - LH* server S_a checks whether it is indeed the right recipient by applying the forward algorithm
    - Attempts to find the correct hash value a' for k, using the local image
  - If a' is not the server address, the client made an addressing error due to an outdated local image
  - The request is then forwarded to server a'

Algorithm: Forward(a)

```
// j denotes the server level
a' := h_j(k);
if (a' = a) then
    k is in S_a;
else
    // a' ≠ a
    a'' := h_{j-1}(k);
    if (a'' > a and a'' < a') then
        a' := a'';
    Forward request to server S_a';
```

LH* Lookup Example

- Client issues a request `search(5)`
  - Level is $n = 1$
  - Lookup computes the bucket address $a = h_1(5) = 5 \mod 2^1 = 1$
  - The request is sent to server S_1
- Server S_1 receives Client request
  - S_1 is the last server that split, and its level is 3.
  - Hence, $a' = h_3(5) = = 5 \mod 2^3 = 5$
  - Since $a' \neq a$, the client made an addressing error
  - Compute $a'' = h_2(5) = = 5 \mod 2^2 = 1$
  - Since $a'' \not> a$, the request is forwarded to S_5, where key 5 is found
  - Data and new value p is returned to the client



LH* Properties

- The number of messages to reach the correct server is 3 in the worst case.
- This makes the structure fully decentralized with one exception:
  - When a Server overflows, the exact value of p must be accurately determined, i.e., the server that splits (in order to split that server)
- This can be achieved by assigning a special role (Master) to one of the servers:
  - Keeps the value of p and informs the other nodes when necessary.
- Since this only happens during a split, the structure remains scalable.

LH* Lessons Learned

- A relative inaccuracy of the information maintained by a component is acceptable, if associated to a stabilization protocol that guarantees that the structure eventually converges to a stable and accurate state.
- In order to limit the number of messages, the "metadata" information related to the structure maintenance (local image) can be piggybacked with messages that answer Client requests.

Outline

1. Introduction & Motivation
2. Linear Hashing
   - Centralized Solution
   - Distributed Solution
3. Consistent Hashing

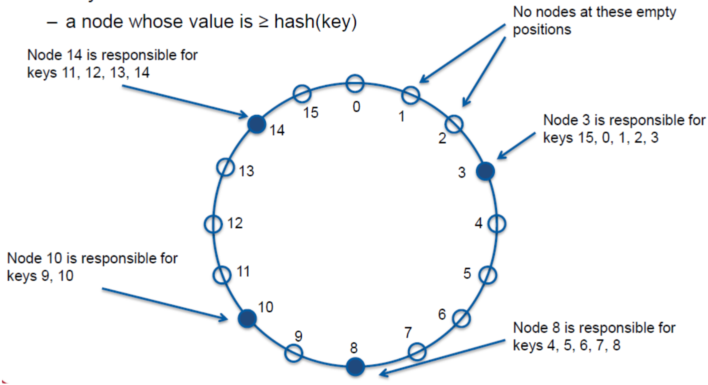
Consistent Hashing (CH)

- Each node (peer) is identified by an integer in the range $[0, 2^n - 1]$
- Each key is hashed into the same range $[0, 2^n - 1]$
- Arrange the peers in a logical ring (clockwise, incrementing IDs)
  - 0 is the successor of $2^n - 1$
- Each peer will be responsible for specific keys
  - A key is stored at the closest successor node
  - This is the first node whose ID $\geq$ hash(key)
- Very simple – a peer needs to know only of its successor and predecessor!
- Chord is one of the first DHT based on consistent hashing
  - Proposed as an index in P2P networks

Key Assignment

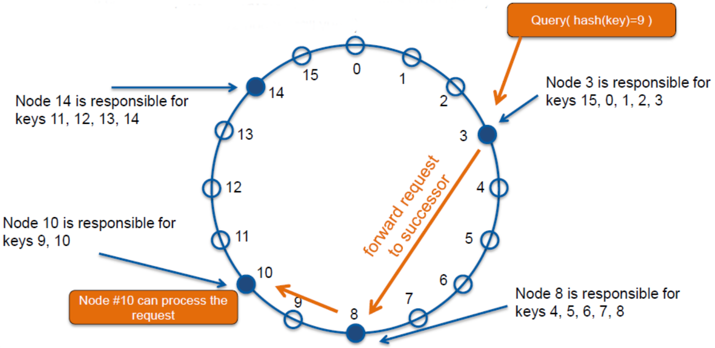
Example: $n = 16$, and four nodes are added so far.

- A key is stored at a successor
  - a node whose value is ≥ hash(key)



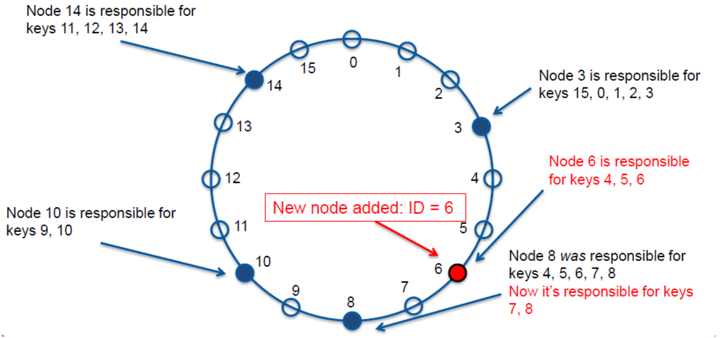
Handling Requests

- Any peer can get a request (insert or query).
- If the hash key is not in the peer's range of keys, the request is forwarded to the successor
- The process continues until the responsible node is found
  - Worst case: with p nodes, traverse $p-1$ nodes – that's $O(N)$
  - Average case: traverse $p/2$ nodes (still not exciting!)



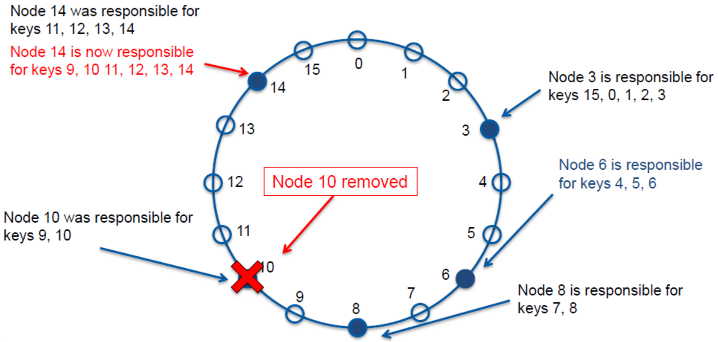
Adding/Joining a New Node

- 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.

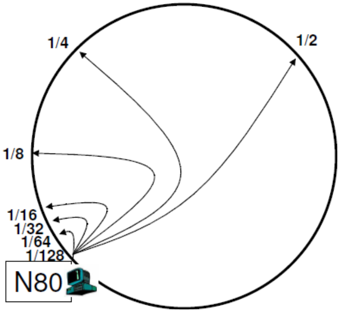


Performance

- We are not excited about an $O(N)$ lookup!
- A simple approach to get great performance would be:
  - All nodes know about each other (index/node table).
  - When a peer gets a query, it searches its index for the node that owns those values.
  - Gives us $O(1)$ performance
  - Add/remove node operations must inform everyone.
- Not a good solution if we have millions of peers (huge tables)!
  - Finger tables are a better solution

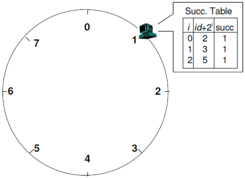
Finger Tables

- Each node stores a so-called finger table – compromise to avoid huge per-node tables
- Finger table is a partial list of successor nodes
  - The i-th entry in the finger table of a node n identifies the first node that succeeds or is equal $n + 2^i$.
    - finger_table[0]: 1st (immediate) successor
    - finger_table[1]: 2nd successor
    - finger_table[2]: 4th successor
    - finger_table[3]: 8th successor
  - In other words, the i-th finger points $1/2^{n-i}$ way around the ring

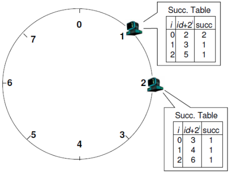


Join Example in Chord with Finger Table

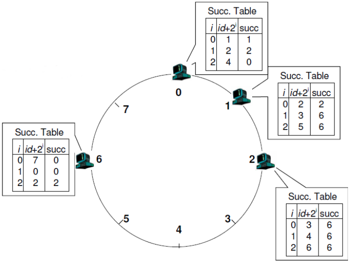
Node n_1 joins the network



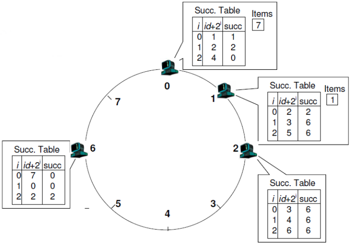
Node n_2 joins the network



Nodes n_0 and n_6 join the network



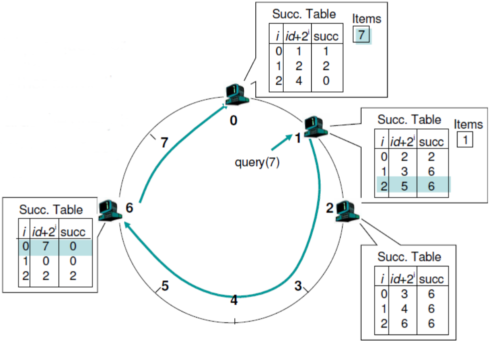
Item f_7 and f_1 are added



Lookup with Finger Table

Algorithm: Lookup(k)

Let n' be the ID of the local node;
Look in finger table for the highest node n s.t. $n' < n < k$;
if *n exists* **then**
    Call Lookup(k) on node n;
else
    return successor node;



Lookup Performance in Chord with Finger Table

- Finger table size: $\log N$ entries
- Lookup: $O(\log N)$ nodes need to be contacted to find the node that stores a key
  - With each hop you go $1/2$ the way towards the destination.
  - Not as cool as $O(1)$ but way better than $O(N)$!

Summary

- Content location and fast access to single content items are two important issues in P2P networks.
- Hash tables are well known for a constant search complexity in centralized databases.
- Aim is to used hash-based solution to distribute content in P2P systems – aka peer-to-peer version of a $(key, value)$ database.
- Linear hashing and consistent hashing are two efficient solutions for P2P systems, which are characterized by dynamicity
  - peers are entering and exiting the network;
  - data is growing quickly.
- Chord is one of the first solutions based on consistent hashing for content distribution and indexing in P2P systems.