Lesson 3:
consistent hashing, introduction to DHTs

24/1/2021
• A stores I, B wants to find content I, but does not know the location of I
• what mechanisms can be exploited to decide where the information should be stored and how to find it without a centralized server?
• any solution must take into account:
  • system scalability: evaluation of the communication overhead and of the memory needed by any node, as a function of the number of nodes (peers)
  • system adaptability: due to faults and to frequent changes (churn)
RETRIEVING CONTENT IN PURE P2P

• searching: search is guided by the value of a set of attributes of the content
  • “similar to Google search”
  • advantages:
    • “user friendly”: does not require ID computation, auxiliary structures
  • disadvantages
    • poor scalability
    • overhead due to comparing whole objects

• addressing: associate a unique identifier to the content (sometimes called content key) and exploit it to search content
  • “similar to an URL in REST”
  • exploited in DHTs
  • advantages:
    • efficient object detection (theoretical bounds on routing)
  • disadvantages:
    • ID computation (hash)
    • maintaining the addressing structure
DISTRIBUTED HASH TABLES: MOTIVATIONS

- Centralized Approach: a server indexing the data
  - Search: $O(1)$ – “content is stored in a centralized server”
  - Space required: $O(N)$ ($N$ = amount of shared content)
  - Bandwidth (connection server/overlay): $O(N)$
  - complex queries may be easily managed

- Fully Distributed Approach: unstructured network
  - Search: worst case $O(N^2)$ - “each node contact each of its neighbours”
    - possible optimizations (TTL, identifiers to avoid cyclic paths)
    - $O(N)$ with optimizations
  - Space Required: $O(1)$
    - does not depend on the number of nodes in the system
    - no data structure to route queries (flooding)
DISTRIBUTED HASH TABLES: MOTIVATIONS

Flooding

Disadvantages:
- Communication overhead
- False negative

Centralized Server

Disadvantages:
- Memory, CPU, Required Bandwidth richiesta
- Fault Tolerance

Communication Overhead

Flooding

Does it exist a solution
Which is a compromise between the two proposals?
DISTRIBUTED HASH TABLES: MOTIVATIONS

Flooding

- Communication Overhead
- False Negative

Disadvantage: O(N)

Distributed Hash Table

- Scalability: O(log N)
- Avoid False negative
- Self Organization: the system automatically manages
  - Join of new nodes in the system
  - Leave (volunteer/faults)

Disadvantage
- Memory, CPU, Required Bandwidth
- Fault Tolerance

Centralized Server

- O(1)
- O(log N)
- O(N)

Communication Overhead

Memory
• **What is hashing?**
  - by webster vocabulary “*chopped meat mixed with potatoes and browned*”
  - in computer science: an hash function that maps one piece of data to another piece of data, typically an integer
  - more input than output: *collisions*

• **Hash Table (HT): keys-data**
  - key is hashed to directly find a bucket in the hash table
  - each bucket is expected to hold #items/#buckets items

• **Cryptographic Hash Functions**: must fullfill a set of properties, we will see later
SCALING OUT: DISTRIBUTED HASHING

- split the hash tables into several parts and distribute it to several servers
  - for avoiding storage limitations of a single computer
  - hosted on different servers
- **MemCached**: distributed memory-caching system for web caching
  - a pool of caching servers to provide fast access to information
  - access to the database only on cache misses
  - hash of data is the index of the server
    - use hash to map resources (or URLs) to a dynamically changing set of caches
    - each machine (user) can locally compute which cache should contain the required resource, referenced by an URL
      - no inter-cache communication
- The technique is extended to DHT for P2P systems.
THE REHASHING PROBLEM
THE REHASHING PROBLEM

5 \rightarrow 5 \mod 4 = 1 \rightarrow 1

5

1, 5, 9

2, 6, 10

3, 7, 11

4, 8, 12

1, 2, 3, 4
THE REHASHING PROBLEM

11 →

1,5,9  1
2,6,10  2
3,7,11  3
4,8,12  4
THE REHASHING PROBLEM

11 \rightarrow 11 \% 4 = 3

1, 5, 9
2, 6, 10
3, 7, 11
4, 8, 12

Laura Ricci
Dipartimento di Informatica
Università degli Studi di Pisa

Consistent Hashing, DHT
Laura Ricci
THE REHASHING PROBLEM

What happens when number of server changes?
THE REHASHING PROBLEM

1 1,5,9
2 2,6,10
3 3,7,11
4 4,8,12
THE REHASHING PROBLEM

1. 1, 5, 9
2. 2, 6, 10
3. 3, 7, 11 (marked with an 'X')
4. 4, 8, 12
The Rehashing Problem
THE REHASHING PROBLEM

Keys 3, 4, 5, 6, 7, 8, 9, 10, 11 are remapped

Keys 4, 5, 6, 8, 9, 10 are remapped even if their machines are up
THE REHASHING PROBLEM

- classical hash function: store the Web page with URL x at the cache (with 4 caches):

  \[SHA-1(x) \to \text{ID-160 bit \% 4} \to \text{cache ID}\]

- now, our store gets bigger......suppose we need to add more 2 caches
  - we now have to recalculate where all the URLs are stored
  - the only URLs stored on the same node as before are those where
    \[SHA-1(\text{URL}) \mod 4 = SHA-1(\text{URL}) \mod 6\]
    - with 10 buckets and 1000 keys about 99% of the keys have to be remapped
THE SOLUTION: CONSISTENT HASHING

- we need an hashing scheme scheme that does not depend directly on the number of servers,
  - when adding or removing servers, the number of keys that need to be relocated must be minimized
  - no global reordering of the table
- consistent hashing: a hash techniques guaranteeing that adding more nodes/remove nodes implies moving only a minority of data items
  - distribution scheme does not depend directly on the number of servers
  - each node manages an interval of consecutive hash keys, not a set of sparse keys
  - intervals are joined/splitted when nodes join/leave the network and keys redistributed between adjacent peers
Basic ideas

- map a **contiguous interval of hash values** to the same node, not a set of sparse values (obtained by MOD)
- how to map an interval to a node?
  - in addition to hashing the names of the objects (URLs, IP addresses), hash also the names of all the nodes \( (s_i) \) in the same space.
- define a mapping of objects to nodes
  - suppose \( x \) hashes to \( h(x) \), scan buckets to the right of \( h(x) \) until finding the bucket \( h(s) \) to which the name of some server \( s \) hashes
  - wrap around the array, if necessary.
CONSISTENT HASHING: BASIC IDEA

- imagine to map the hash output range to a circle
- the minimum hash value, 0, corresponds to an angle of 0 with the horizontal line
- the maximum value corresponds to an angle of 360 degree

<table>
<thead>
<tr>
<th>KEY</th>
<th>HASH</th>
<th>ANGLE (DEG)</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;john&quot;</td>
<td>1633428562</td>
<td>58.8</td>
</tr>
<tr>
<td>&quot;bill&quot;</td>
<td>7594634739</td>
<td>273.4</td>
</tr>
<tr>
<td>&quot;jane&quot;</td>
<td>5000799124</td>
<td>180</td>
</tr>
<tr>
<td>&quot;steve&quot;</td>
<td>9787173343</td>
<td>352.3</td>
</tr>
<tr>
<td>&quot;kate&quot;</td>
<td>3421657995</td>
<td>123.2</td>
</tr>
</tbody>
</table>
CONSISTENT HASHING: BASIC IDEA

- hash also the server in the same identifier space
- for instance hash the IP address of the server
- obtain an angle for each server
CONSISTENT HASHING: BASIC IDEA

- keys and servers hashed on the same identifier space
- define a simple rule to map keys to server
  - each key is mapped to the server whose key is the closest, in a counterclockwise direction
CONSISTENT HASHING: REMOTION

- what happen when server B is removed?
- we need to remap only the keys that were mapped to the removed server
  - only Bill will be remapped
- these keys are now mapped on C
- the absence of B does not affect keys belonging to other servers
CONSISTENT HASHING: INSERTION

- what happen when server D is inserted?
- we need to remap **only the keys mapped on server C**
  - Steve is now mapped to D
- the insertion of D does not affect keys belonging to other servers
CONSISTENT HASHING PROPERTIES

• when the hash table is resized, on the average,
  • only $k/n$ keys need to be remapped on average, where $k$ is the number of keys and $n$ is the number of servers.
• when a node is removed from the hash ring
  • only the keys associated with that node are rehashed and remapped rather than remapping all the keys.
• when a new node is added
  • we map the keys between the new node and the previous node in the hash ring to point to the new node and the keys will no longer be associated with their old node.
• proper consistent hashing functions distribute keys evenly on the ring
• but nodes may be different, some more powerful, some less powerful
  • assign more positions to more powerful servers
consistent hashing works also for P2P systems

- content addressing: map content to peer and use the key to retrieve content
- but...
  - the size of the system is different
  - the level of churn is greater
- because of this, peers cannot take maintain trace of all the keys of all the peers in the system
• location addressing
  • you use http:// or https:// link to point to locate a webpage, image, spreadsheet, dataset, tweet, etc,
  • the link is an identifier that points to a particular location on the web corresponding to a particular server, or set of servers, somewhere on the web.
  • whoever controls that location controls the content.
  • even if a thousand people have downloaded copies of a content, so that the content exists in a thousand locations, HTTP points to a single location.
    • the location-addressed approach forces us all to pretend that the data are in only one location.
    • whoever controls that location decides what content to return when people use that link.
  • this is how HTTP and WWW worked in the last 30 years...
CONTENT ADDRESSING VS. LOCATION ADDRESSING

• content addressing
  • identify content by its "fingerprint" rather than by its location.
    • use a cryptographic hash
  • having the fingerprint of a content allows you to get the content from anyone who has a copy
  • the cryptographic hash do not change: content addressing guarantees that the links will always return the same content, regardless
    • of where the content is retrieved from
    • of who added the content to the network
    • of when the content was added.

• This approach is adopted by IPFS (Internet Planetary File System), to develop a distributed Web.
• identifiers of peers/data are generated through a cryptographic hash functions

• exploit the Secure Hash Algorithm SHA (Secure Hash Standard)
  • cryptographic hash function: produces a message digest of the input
  • different families of SHA: SHA-1, SHA-224, SHA-256, SHA-384 and SHA-512
    • last four variant are referred as SHA-2.
    • postfix depends from the length of the message digest which is produced
  • not a simple hash, it must satisfy a set of properties: we will see the properties in more detail when we introduce cryptocurrencies, for the moment being
    • make SHA1 work through JAVA and look at the program output!
import java.security.*;

class SHA {
    public static void main(String[] a) {
        try {
            MessageDigest md = MessageDigest.getInstance("SHA1");
            System.out.println(" Message digest object info: ");
            System.out.println(" Algorithm = " + md.getAlgorithm());
            System.out.println(" Provider = " + md.getProvider());
            System.out.println("SHA1(" + input + ") = " + bytesToHex(output));
        }
    }
}

String input = "";
md.update(input.getBytes());
byte[] output = md.digest();
System.out.println();
System.out.println("SHA1(" + input + ") = " + bytesToHex(output));
input = "abc";
md.update(input.getBytes());
output = md.digest();
System.out.println();
System.out.println("SHA1(" + input + ") =" + bytesToHex(output));
//will print SHA1("abc") = A9993E364706816ABA3E25717850C26C9CD0D89D
input = "abd";
md.update(input.getBytes());
output = md.digest();
System.out.println();
System.out.println("SHA1(" + input + ") =" + bytesToHex(output));
//will print SHA1("abc") = CB4CC28DF0FDBE0ECF9D9662E294B118092A5735
} catch (Exception e) {
    System.out.println("Exception: "+e); }


public static String bytesToHex(byte[] b) {
    char hexDigit[] = {'0', '1', '2', '3', '4', '5', '6', '7',
                      '8', '9', 'A', 'B', 'C', 'D', 'E', 'F'};

    StringBuffer buf = new StringBuffer();
    for (int j=0; j<b.length; j++) {
        buf.append(hexDigit[(b[j] >> 4) & 0x0f]);
        buf.append(hexDigit[b[j] & 0x0f]);
    }
    return buf.toString();
}

- 0x0F is a hexadecimal number which equals 15 in decimal.
- represents the lower four bits and translates the the bit-pattern 0000 1111
- hexDigit[(b[j] >> 4) & 0x0f] retains only the rightmost 4 bits, clearing
  the left 4 bits.
- hexDigit[b[j] & 0x0f] retains the leftmost 4 bits
what do you observe from the JAVA output?

properties of the SHA:

- variable length input, fixed length output (digest)
- a small change in the input function produces very different outputs
- the function is deterministic
- output length: 40 hexadecimal digit, $40 \times 4 = 160$ bits (SHA-1=SHA-160). $2^{160}$ different values

- currently further SHA algorithms characterized by a larger length of the digest are available. Kekka for Ethereum

- use this function or other functions as basic brick to implement
  - consistent hashing
  - complex cryptographic puzzles (PoW of Bitcoin)
CONSISTENT HASHING: CONTENT ADDRESSING?

- ...but how do we actually implement the content addressing?
  - the hash function is not enough

- lookup given a content \( x \)
  - compute \( h(x) \)
  - scan to find the peer \( p \) that minimizes:
    \[ h(p): h(p) > h(x) \]

- how much does it cost?
  - complete mesh of peers: each peer store a reference to each other peer:
    search is completed in \( O(1) \)
    - huge routing tables
  - reduce the size of the routing tables:
    - a routing algorithm is required. Complexity?

- insertion of content is similar
CONSISTENT HASHING: CONTENT LOOKUP?

- Content-based Routing: routing guided by the knowledge of the content to find P
- Each node maintains a routing table storing a partial view of the network
  - Should be efficient (logarithmic) routing

Routing requires $O(\log(N))$ steps to reach the node storing the information

$O(\log(N))$ size of the routing table of each node

$H(\text{"my data"}) = 3107$
DHT DATA STORAGE

- data is stored, when it is inserted in the DHT, onto the responsible node
  - such a node is not, in general, the node which has inserted the data into the DHT
- distributed storage network
- An example:
  - key = H(“Data”) = 3107.
  - data is inserted by the node with IP address 134.2.11.68
    - data is stored onto the node which manages the address portion including the address 3107. (different from 134.2.11.68)
NODE LEAVE

• Voluntary leave of a node
  • partitioning of its address space to the neighbour nodes
  • copy key/value pairs to the corresponding nodes
  • deletion of the node from the routing tables of the other nodes

• Node failure
  • if a node suddenly disconnect from the network, all data stored on it are lost if they are not stored on other nodes
    • introduce some redundancy (data replication)
    • information loss: periodical information refresh
  • exploit alternative/redundant routing paths
    • periodical probing of the neighbour nodes to detect their activity. When a fault is detected, update routing tables.
DHT: LOAD BALANCING

- main reasons for load unbalance:
  - a node manages a bigger portion of the logical address space
    - can be resolved by exploiting an uniform hash function
  - the address space is uniformly distributed, but the addresses managed by a node correspond to lot of data
  - the address space is uniformly distributed, but a node manages a lot of queries, because the data paired with the addresses assigned to it are very popular

- Load unbalance implies:
  - less system robustness
  - less scalability
  - \( O(\log N) \) bounds are not guaranteed

- Solutions:
  - virtual servers: more segment for each peer
DHT CHALLENGES

- avoid hotspots
- evenly distribute responsibilities
- red peer is more loaded than green

- handle churn
- redistribute responsibilities to/of joining/leaving nodes

- tradeoff
- amount of routing state
- traffic in the overlay
- stretch with respect to underlay
Different proposals “consistent-hashing compliant”, which differ in the way

- data is paired with peers
- peer paired with a bucket is retrieved (FindPeer or FindSuccessor operator) (related to previous choice)
Most DHT provide a very simple interface:

- content insertion:
  - **PUT** (key, value)
- content search
  - **GET** (key)
- replies
  - **Value**

Generally, no function in the API for moving keys
## COMPARING DIFFERENT APPROACHES

<table>
<thead>
<tr>
<th>Approach</th>
<th>Memory for each node</th>
<th>Communication Overhead</th>
<th>Complex Queries</th>
<th>False Negatives</th>
<th>Robustness</th>
</tr>
</thead>
<tbody>
<tr>
<td>Central Server</td>
<td>$O(N)$</td>
<td>$O(1)$</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>Pure P2P (flooding)</td>
<td>$O(1)$</td>
<td>$O(N^2)$</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>DHT</td>
<td>$O(\log N)$</td>
<td>$O(\log N)$</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
DHT: APPLICATIONS

- DHT offer a generic distributed service for information storing and indexing

- The value paired with a key may be
  - a file
  - an IP address
  - or every further data

- applications exploiting a DHT
  - Internet Planetary File System
  - storing references to the peers in a swarm: Bittorrent
  - storing references to peers in Ethereum
  - define a support for higher level services

......
CONCLUSIONS

- DHT Properties
  - routing is based on key (unique identifier)
  - key are uniformly distributed to the DHT nodes
    - bottleneck avoidance
    - incremental insertion of the keys
    - fault tolerance
  - auto organizing system
  - simplex and efficient organization
  - the terms “Structured Peer-to-Peer“ and “DHT“ are often used as synonyms
  - support several applications
    - the values paired with the keys depend on the application
## CONCLUSIONS

<table>
<thead>
<tr>
<th>Client-Server</th>
<th>Peer-to-Peer</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Server is the central entity and only provider of service and content.</td>
<td>1. Resources are shared between the peers</td>
</tr>
<tr>
<td>2. Network managed by the Server</td>
<td>2. Resources can be accessed directly from other peers</td>
</tr>
<tr>
<td>3. Peer is provider and requestor (Servent concept)</td>
<td>3. Peer as the higher performance system</td>
</tr>
<tr>
<td>Clients as the lower performance system</td>
<td>Example: WWW</td>
</tr>
</tbody>
</table>

### Unstructured P2P

<table>
<thead>
<tr>
<th>Centralized P2P</th>
<th>Hybrid P2P</th>
<th>Pure P2P</th>
<th>DHT-Based</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. All features of Peer-to-Peer included</td>
<td>1. All features of Peer-to-Peer included</td>
<td>1. All features of Peer-to-Peer included</td>
<td>1. All features of Peer-to-Peer included</td>
</tr>
<tr>
<td>2. Central entity is necessary to provide the service</td>
<td>2. Any terminal entity can be removed without loss of functionality</td>
<td>2. Any terminal entity can be removed without loss of functionality</td>
<td>2. Any terminal entity can be removed without loss of functionality</td>
</tr>
<tr>
<td>3. Central entity is some kind of index/group database</td>
<td>3. Dynamic central entities</td>
<td>3. No central entities</td>
<td>3. No central entities</td>
</tr>
<tr>
<td>Example: Napster</td>
<td>Example: Gnutella 0.6, JXTA</td>
<td>Examples: Gnutella 0.4, Freenet</td>
<td>Examples: Chord, CAN</td>
</tr>
</tbody>
</table>

### Structured P2P

Source: