Lesson 4
DISTRIBUTED HASH TABLES
CONSISTENT HASHING

Laura Ricci
6/3/2017
OUTLINE OF NEXT LESSONS

- Searching versus Addressing

- Consistent hashing / traditional hashing

- An example of a generic DHT
  - data addressing
  - routing
  - join/leave

- DHT instances:
  - Chord, Kademlia/KAD

- Formal tools
  - modelling Chord routing through Markov chains

- DHT applications
A stores I, B wants to find content I, but does not know the location of I

What mechanisms exploited to decide where the information should be stored and how to find it?

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)
P2P CONTENT RETRIEVAL: TWO STRATEGIES

- 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 address 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
DHT: MOTIVATIONS

- **Centralized Approach:** a server indexing the data
  - Search: $O(1)$ - "content is stored in a centralized server"
  - Space required: $O(N)$ ($N = \text{amount of shared content}$)
  - 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)
DHT: MOTIVATIONS

Flooding

Disadvantages:
• Communication overhead
• False negative

Centralized Server

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

Does it exist a solution Which is a compromise between the two proposals?

Communication Overhead

$O(N)$
$O(\log N)$
$O(1)$

Memory

$O(1)$
$O(\log N)$
$O(N)$
DHT: MOTIVATIONS

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

Disadvantage:
- Communication Overhead
- False Negative

- Memory, CPU, Required Bandwidth
- Fault Tolerance

- $O(1)$
- $O(\log N)$
- $O(N)$

Flooding

Distributed Hash Table

Centralized Server

Communication Overhead
FROM HASH TABLES TO DHT

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

- Distributed Hash Table (DHT)
  - distribute set of buckets to peers
  - hash key to find the responsible peer
  - balance load across nodes
FROM HASH TABLES TO DHT

- DHT is an ordinary hash table which is.....
DHT is an ordinary hash table which is distributed!

but, be careful! ....can we apply the same mechanisms of classical hash tables?

- suppose each peer is paired with a bucket
- number of buckets do not change in classical scenarios, while the number of peers is very dynamic:
- we do not want to restructure all the system for each peer join/leave
- need of particular hashing techniques: consistent hashing
CONSISTENT HASHING: MOTIVATION

- an idea originally developed for Web Caching (1997)

- a browser requests a URL, like amazon.com to a Web server.
  - use a Web cache, which stores a local copy of recently visited pages
  - reduce access time

- the same web cache exploited by many users, for instance all the users of the University of Pisa
  - accessed pages shared by all the users of the Pisa University
  - a single cache does not fit a single node.
  - cache has to be spread n multiple nodes.

- we could use one of the nodes as a central directory server
  - but... with the hash of the URL, we don't need a central server
CONSISTENT HASHING: MOTIVATION

• classical hash function: store the Web page with URL x at the cache (with 4 caches):
  \[ \text{sha1}(x) \to 160 \text{ bit ID} \% 4 \to \text{cache ID} \]

• now, our store gets bigger......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
  \[ \text{SHA-1(URL)} \mod 4 = \text{SHA-1(URL)} \mod 6 \]

• if has are in the range 1..k, and the number of nodes is n
  \[ \frac{(k-n)}{k} = 1 - \frac{n}{k} \]
  keys have to be remapped

• with 10 buckets and 1000 keys about 99% of the keys have to be remapped
The same problem in the scenario of P2P systems:

\[ \text{sha1}(x) \rightarrow 160 \text{ bit ID} \mod \text{Number of Peers} \rightarrow \text{Peer ID} \]

- traditional hashing depends on the number of peer
- join and leave of peers require to completely repartition the hash table
- problem is even more serious because high peer churn
CONSISTENT HASHING: DEFINITION

Consistent hashing: a hash technique guaranteeing that adding more nodes/remove nodes implies moving only a minority of data items.

Basic ideas (we refer to web caching scenario):
- Map a contiguous interval of hash values to the same node, not a set of sparse values (obtained by MOD).
- And now... how to map an interval to a node? Basic idea:
  - In addition to hashing the names of the objects (URLs), hash also the names of all the nodes (s) in the same space.
  - Mapping of an object x:
    - 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: DEFINITION

- hash of object = closest clockwise bucket ("successor")
- N nodes partition the circle into N segments, with each node responsible for all objects in one of these segments.
...but how do we actually implement the standard operations of an hash table, Lookup and Insert?

- cannot simply exploit the hash function to find the node

- **Insert**: given a content $x$
  - compute $h(x)$
  - define an algorithm **FindSuccessor implementing** the rightward/clockwise scan to find the server $s$ that minimizes:
    $$ h(s) \text{ subject to } h(s) > h(x) $$

- **lookup**: similar, starting with the key to search
SOME HISTORY: 1997-2015

• the idea of consistent hashing first appeared in a research paper in 1997 (STOC)

• in 1999, the trailer “Star Wars: The Phantom Menace” release put apple.com servers offline, while akamai.com, implementing consistent hashing, was able to serve a unauthorised copy

• consistent hashing is re-purposed in 2001 to address technical challenges that arise in peer-to-peer (P2P) networks
  • first Chord
  • then BitTorrent

• in 2006 Amazon implements its internal Dynamo system using consistent hashing.
**CONSISTENT HASHING AND P2P SYSTEMS**

- **DHT:** distributed table + consistent hashing
  - stores (key-value) pairs
  - given a key, returns the value

- hash peer to a very large hash space
  - key to hash may be IP address

- hash also content to the same space
  - key to hash may be the file name or a part of the content

- peers are given a set of consecutive bucket in this space

- each peer manages a segment of the logical space

![Assign particular nodes to hold particular content (or know where it is)](image)

![When a node wants this content, go to the node that is supposed to hold it (or know where it is)](image)

![peer](image)

![peer](image)
CONSISTENT HASHING FLAVOURS IN DHT

- **Hypercube**
  - Plaxton, Chord[2], Kademlia[7], Pastry[4], CAN[5]
- **Butterfly**
  - Viceroy[7], Mariposa
- **De Bruijn Graph**
  - Koorde[9]
- **Skip List**
  - Skip Graph, SkipNet

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)
DISTRIBUTED HASH TABLE

1. Decide on a common key space for nodes and values

2. Connect nodes using a small, bounded number of links s.t. max hop count is minimized

3. Define a strategy for assigning items to nodes
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
Simplest strategy: the same proposed in the original proposal of consistent hashing

map nodes and content into a linear address space 0, ..., $2^m - 1$ totally ordered and whose size is $>>$ with respect to the number of objects to store (for instance $m=160$)

define hash function

$$\text{Hash(String)} \mod 2^m, \quad \text{Hash("mydata")}=2313$$
LINEAR ORDERED SPACE

- each node is responsible of an consecutive interval of identifiers
- interval overlapping may introduce a level of redundancy
- continuous adaptation
- underlay topology and logical overlay are not correlated
LINEAR ORDERED SPACE: FINDPEER

- How to find the peer $P$ which manages a bucket?
- Key-based Routing: routing guided by the knowledge of the key to find $P$
- Each node maintains a routing table storing a partial view of the network
  - to implement efficient 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

$H(\text{"my data"}) = 3107$
LINEAR ORDERED SPACE: FINDSUCCESSOR

- How to find the peer P which manages a bucket?
- each node maintains a routing table storing a partial view of the network
- Key-based Routing: routing guided by the knowledge of the key to find P

Key = H("my data")

Value = pointer to location of data

Node 3485 manages keys 2907-3485,

H(\text{"my data"})\ = \ 3107
STORING DATA IN THE DHT

- 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
- 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)
STORING DATA IN THE DHT

Direct Storage

Indirect Storage
Value = may be a reference to the data (ex: the physical address of the node storing the content)

A flexible solution, but it needs a further step to access the data
• send IP address and port to the requesting peer
• if indirect addressing is exploited, the requesting peer download the content from a third peer.
• connection reversal in presence of NATs
LINEAR SPACE: NODE JOIN

The black peer enters the DHT:

- hash the node ID
- contact an arbitrary node of the DHT (bootstrap node)
- detect the exact point of the DHT where to join (predecessor and successor node)
- assign a portion of the logical address space to the new peer
- copy the assigned Key/value pairs (with redundancy)
LINEAR SPACE: NODE LEAVE

The green peer leaves the DHT

- the neighbours inherit the responsibilities for the data of the leaving peers
- which neighbours depend from the DHT
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 among the nodes 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 (we will see in next lessons)
### 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>x</td>
</tr>
<tr>
<td>Pure P2P (flooding)</td>
<td>$O(1)$</td>
<td>$O(N^2)$</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
</tr>
<tr>
<td>DHT</td>
<td>$O(\log N)$</td>
<td>$O(\log N)$</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
DHT API

- API to access a DHT
  - content insertion:
    - PUT(key, value)
  - content search
    - GET(key)
  - replies
    - Value
- The interface is common to several DHT systems

Distributed Application

Put(Key, Value) Get(Key) Value

Distributed Hash Table
(CAN, Chord, Pastry, Tapestry, …)

Node 1 Node 2 Node 3 . . . Node N
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
  - DNS implementation
    - key: host name, value: list of corresponding IP addresses
  - P2P storage systems: example Freenet, PAST
  - 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
MAKE SHA1 WORK IN JAVA!

- generate **identifiers** through a **cryptographic hash functions**.

- **Chord** exploits 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());

            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));

input = "abcdefghijklmnopqrstuvwxyz";
md.update(input.getBytes());
output = md.digest();
System.out.println();
System.out.println("SHA1("+input+") ="+bytesToHex(output));

} 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();
}

The output produced by the algorithm:

Message digest object info:
Algorithm = SHA1
Provider = SUN version 1.8
SHA1("") = DA39A3EE5E6B4B0D3255BFEF95601890AFD80709
SHA1("abc") = A9993E364706816ABA3E25717850C26C9CD0D89D
SHA1("abcdefghijklmnopqrstuvwxyz") = 32D10C7B8CF96570CA04CE37F2A19D84240D3A89
What do you observe from the JAVA output?

• variable length input, fixed length output (digest)

• 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

• SHA mathematical properties: presented in another lesson

• Use SHA function to implement consistent hashing
STRUCTURED/UNSTRUCTURED: RECAP

Unstructured Overlay
- no addressing: TTL enhanced flooding
- no mapping rule to map content to peers
- no rule to define the neighbours of a peer: each peer may arbitrarily choose its neighbours
- this does not imply a complete lack of structure...the network may assume a structure
  - scale free, power-law, small world,...

Structured Overlay
- define an addressing mechanism
- overlay topology follows some rule
- define a mapping rule to map content to peers
- deterministic routing
- Distributed Hash Tables (DHT)