Overlay and P2P Networks

Structured Networks and DHTs

Prof. Sasu Tarkoma

3.10.2011
Contents

• Structured networks
• Foundations
• Cluster-based structures
Structured Overlays

Structured overlays are typically based on the notion of a semantic free index.

They utilize hashing extensively to map data to servers.

The cluster-based techniques typically can guarantee a very small number of hops to reach a given destination.

The decentralized DHTs balance hop count with the size of the routing tables, network diameter, and the ability to cope with changes.
Consistent hashing

Consistent hashing was first introduced in 1997 as a solution for distributing requests to a dynamic set of web servers.

In this solution, incoming messages with keys were mapped to web servers that can handle the request.

Consistent hashing has had dramatic impact on overlay algorithms.

DHTs utilize consistent hashing to partition an identifier space over a distributed set of nodes. The key goal is to keep the number of elements that need to be moved at minimum.
Consistent hashing continued

In most traditional hash tables a change in the number of array elements causes nearly all keys to be remapped. They are therefore useful for balancing load to a fixed collection of servers, but not suitable for dynamic server collections.

Consistent hashing is a technique that provides hash table functionality in such a way that the addition or removal of an element does not significantly change the mapping of keys to elements. The technique requires only $K/n$ keys to be remapped on average, where $K$ is the number of keys, and $n$ is the number of nodes.
Ranged hash functions

Hashing applied to the distributed case

Ranged hash functions are hash functions that depend on the set of available buckets

A typical ranged hash function hashes items to positions in some space

Then assigns each item to the nearest available bucket

As the set of buckets changes, an item may move to a new nearest available bucket
Another view

A ranged hash function changes minimally as the range of the function changes

Range changes when a server is added or removed
Ranged hash with a ring

Items and buckets are mapped to a uniformly random place on continuous unit ring $[0,1)$.

Each item is assigned to the closest possible bucket.

Bucket order determines placement on the ring.

Optimality proven for growth-restricted metric spaces
Given point $q$ and distance $d$, the number of points within distance $2d$ is at most constant factor larger than within distance $d$.

Example of Consistent Hashing

- Creating the structure
  - Assign each of C hash buckets to random points on mod $2^n$ circle, where, hash key size = $n$
  - Map object to random position on circle
  - Hash of object = closest clockwise bucket

![Diagram of consistent hashing with buckets numbered 0 to 14 on a circle, with an arrow pointing to a red dot at bucket 8, indicating to go to the left bucket.](image-url)
Problem

Having only one location for a bucket is not good

Does not ensure good spread

Solution: have multiple virtual locations for a bucket

Implication: when removing / adding a bucket, have to move data from several servers
Replication with virtual buckets

One point is not sufficient to characterize a bucket due to the required properties.

A bucket is replicated $\kappa \log(C)$ times, where $C$ is the number of distinct buckets, and $\kappa$ is a constant. The $\log(C)$ term comes from the theory, basically it is needed to get the good fraction $O(1/|V|)$ of buckets to servers.

When a new bucket is added, only those items are moved which are closest to one of its points. Similarly for the removal of a bucket.
Properties of ranged hash functions

Monotone
Each item has its own preference list and hashes to the first available bucket

This minimizes rearrangement cost

Multiple virtual locations for the item are possible

Can be described with a preference matrix
Items and the buckets
Properties of Consistent Hashing I

A **view** is a subset of the buckets (cache servers available from certain part of the network).

Consistent hashing uses a **ranged hash function** to specify an assignment of items to buckets for every possible view.

A ranged hash family is said to be **balanced** if given a particular view, a set of elements, and a randomly chosen function from the hash family, with high probability the fraction of items mapped to each bucket is $O(1/|V|)$, where $V$ is the view.

In other words, items are uniformly distributed over the buckets of the view.
Properties of consistent hashing II

Load: A balanced ranged hash function distributes load evenly across the buckets

Monotonicity is another important property for the hash function. This property says that some items can be moved to a new bucket from old buckets, but not between old buckets. The aim is to preserve an even distribution

Spread is about ensuring that at least a constant fraction of the buckets are visible to clients
Example of a ranged hash function (RHF)

Let I be the items, C the caches, and V the views. \( V_i \) is a subset of C.

RHF is a map that takes a view (all possible views \( 2^C \)) and hashes it to a cache in which the item can be found:

\[
h : 2^C \times I \to C
\]

For an item: pick a point \( r \) uniformly and independently at random
For the buckets: pick a set of \( \kappa \log C \) points uniformly and independently at random.
For an item \((V,i)\) map it to the first bucket \( b \) in \( V \) that is encountered clockwise starting from \( r \).
Bad examples

Pick $b$ in $V$ at random: bad spread properties (needs to be the preference list of many buckets)

Take mod of the number of caches in a view: good balance but not smooth (e.g. problems when adding or removing a server)
public class ConsistentHash<T> {
    private final HashFunction hashFunction;
    private final int numberOfReplicas;
    private final SortedMap<Integer, T> circle =
        new TreeMap<Integer, T>();

    public ConsistentHash(HashFunction hashFunction,
                          int numberOfReplicas, Collection<T> nodes) {
        this.hashFunction = hashFunction;
        this.numberOfReplicas = numberOfReplicas;

        for (T node : nodes) {
            add(node);
        }
    }

    public void add(T node) {
        for (int i = 0; i < numberOfReplicas; i++) {
            circle.put(hashFunction.hash(node.toString() + ":" + i),
                       node);
        }
    }

    public void remove(T node) {
        for (int i = 0; i < numberOfReplicas; i++) {
            circle.remove(hashFunction.hash(node.toString() + ":" + i));
        }
    }

    public T get(Object key) {
        if (circle.isEmpty()) {
            return null;
        }

        int hash = hashFunction.hash(key);
        if (!circle.containsKey(hash)) {
            SortedMap<Integer, T> tailMap =
                circle.tailMap(hash);  
            hash = tailMap.isEmpty() ?
                circle.firstKey() : tailMap.firstKey();
        }
        return circle.get(hash);
    }
}
Main point in consistent hashing

The technique requires only $K/n$ keys to be remapped on average, where $K$ is the number of keys, and $n$ is the number of nodes.

Used in most DHT algorithms

Developed by Karger et al. at MIT

Somewhat involved for example in Chord

Used by CDNs and caches
  Akamai
Semantic free indexing I

With semantic free indexing in structured overlays, data objects are given unique identifiers called keys that are chosen from the same identifier space.

Keys are mapped by the overlay network protocol to a node in the overlay network.

The overlay network needs to then support scalable storage and retrieval (key, value) pairs.
In order to realize the insertion, lookup, and removal of (key, value) pairs, each peer maintains a routing table that consists of its neighbouring peers (their node identifiers and IP addresses).

Lookup queries are then routed across the overlay network using the information contained in the routing tables.

Typically each routing step takes the query or message closer to the destination.
DHT interfaces

- DHTs offer typically two functions
  - `put(key, value)`
  - `get(key) \rightarrow value`
  - `delete(key)`
- Supports wide range of applications
  - Similar interface to UDP/IP
    - `Send(IP address, data)`
    - `Receive(IP address) \rightarrow data`
- No restrictions are imposed on the semantics of values and keys
- An arbitrary data blob can be hashed to a key
- Key/value pairs are persistent and global
Distributed applications

DHT balances keys and data across nodes

Node Node Node Node

put(key, value) get(key) value
Foundations of Structured Networks

We distinguish between a routing algorithm and the routing geometry. The algorithm pertains to the exact details of routing table construction and message forwarding.

Geometry pertains to the way in which neighbours and routes are chosen. Geometry is the foundation for routing algorithms.

The key observation is that the geometry plays a fundamental part in the construction of decentralized overlays.
Geometries

The five frequently used overlay topologies are:

- trees
- tori (k-ary n-cubes)
- butterflies (k-ary n-flies)
- de Bruijn graphs
- rings
- XOR geometry

The differences between some of the geometries are subtle

For example, it can be seen that the static DHT topology emulated by the DHT algorithms of Pastry and Tapestry are Plaxton trees; however, the dynamic algorithms can be seen as approximation of hypercubes.
Trees

The tree’s hierarchical organization makes it a suitable choice for efficient routing.

In a tree geometry, node identifiers represent the leaf nodes in a binary tree of depth $\log n$.

The distance between any two nodes is the height of their smallest common subtree.

One of the first DHT algorithms, the Plaxton’s algorithm, is based on this geometry (object rooted at a node).

For scalable networking, each node maintains a routing table with $\log n$ neighbours. In this table, the $i$th neighbour is at distance $i$ from the current node. Greedy routing can then be used to forward a message to its destination on the network given the target identifier.
Observations on Plaxton

Global ordering of nodes (only one root node possible)

Static configuration

Forest of trees where each server is a root

Populate routing table to reflect possible distances
   One suffix digit at a time
Plaxton’s algorithm: routing table of node 3642

<table>
<thead>
<tr>
<th>Levels</th>
<th>Entries</th>
<th>1 Primary neighbour</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0642</td>
<td>X042</td>
<td>XX02</td>
<td>XXX0</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1642</td>
<td>X142</td>
<td>XX12</td>
<td>XXX1</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2642</td>
<td>X242</td>
<td>XX22</td>
<td>XXX2</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>3642</td>
<td>X342</td>
<td>XX32</td>
<td>XXX3</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>4642</td>
<td>X442</td>
<td>XX42</td>
<td>XXX4</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>5642</td>
<td>X542</td>
<td>XX52</td>
<td>XXX5</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>6642</td>
<td>X642</td>
<td>XX62</td>
<td>XXX6</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>7642</td>
<td>X742</td>
<td>XX72</td>
<td>XXX7</td>
<td></td>
</tr>
</tbody>
</table>

Wildcards are marked with X
Primary neighbour is one digit away

Example lookup

Node 3642 receives message for 2342
- The common string is XX42
- Two shared digits, consult second column and choose the correct digit
- Send to node with one digit closer
- Fourth line with X342

Table size: base * address length
In this example octal base (8) and 4 digit addresses

Each routing table is organized in routing levels and each entry points to a set of nodes closest in network distance to a node which matches the given suffix.
Comparison to IP routing

IP routing is based on the longest matching prefix
Keep a prefix data structure (ternary tree, TCAM)
Find next hop based on the list (or the destination)

IP addresses are obtained through a local configuration process and/or BGP tables, default routes as well

For the Plaxton / DHT case we do not have the IP address semantics and mapping to the IP topology

The Plaxton/DHT topology is flat!

Hence the table structure with suffixes/prefixes.
Rings

Rings are a popular geometry for DHTs due to their simplicity. In a ring geometry, nodes are placed on a one-dimensional cyclic identifier space. The distance from an identifier A to B is defined as the clockwise numeric distance from A to B on the circle.

Rings are related with tori and hypercubes, and the 1-dimensional torus is a ring. Moreover, a k-ary 1-cube is a k-node ring.

The Chord DHT is a classic example of an overlay based on this geometry.

Each node has a predecessor and a successor on the ring, and an additional routing table for pointers to increasingly far away nodes on the ring.
Finger | Maps to | Real node
--- | --- | ---
1,2,3 | x+1,x+2,x+4 | N14
4 | x+8 | N21
5 | x+16 | N32
6 | x+32 | N42

for j=1,...,m the fingers of p+2^{j-1}
The distance between two nodes in the hypercube geometry is the number of bits by which their identifier differ. At each step a greedy forwarding mechanism corrects (or fixes) one bit to reduce the distance between the current message address and the destination.

Hypercubes are related to tori. In one dimension a line bends into a circle (a ring) resulting in a 1-torus. In two dimensions, a rectangle wraps into the two-dimensional torus, 2-torus. An n-dimensional hypercube can be transformed into an n-torus by connecting the opposite faces together.

The Content Addressable Network (CAN) is an example of a DHT based on a d-dimensional torus.
Differences

The main difference between hypercube routing and tree routing is that the former allows bits to be fixed in any order.

Tree routing requires that the bits are corrected in a strict order (digit by digit, still can be redundancy in the table).

Thus hypercube is more restricted in selecting neighbours in the routing table but offers more possibilities for route selection!
Hypercubes

\[ d = 0 \]
\[ N = 1 \]

\[ d = 1 \]
\[ N = 2 \]

\[ d = 2 \]
\[ N = 4 \]

\[ d = 3 \]
\[ N = 8 \]

\[ d = 4 \]
\[ N = 16 \]
Butterfly Geometry

A *k-ary n-fly* network consists of $k^n$ source nodes, $n$ stages of $k^{n-1}$ switches, and $k^n$ destination nodes.

The network is unidirectional and the degree of each switching node is $2k$.

The diameter of the network is logarithmic to the number of source nodes. At each level $l$, a switching node is connected to the identically numbered element at level $l + 1$ and to a switching node whose number differs from the current node only at the $l$th most significant bit.

The main drawback of this structure is that there is only one path from a source to a destination, in other words, there is no path diversity. In addition, butterfly networks do not have as good locality properties as tori.
Butterfly network (with a tree)
De Bruijn Graph

An $n$-dimensional de Bruijn graph of $k$ symbols is a directed graph representing overlaps between sequences of symbols. It has $k^n$ vertices that represent all possible sequences of length $n$ of the given symbols.

In a $n$-dimensional de Bruijn graph with 2 symbols, there are $2^n$ nodes, each of which has a unique $n$-bit identifier.
Creating a de Bruijn graph

The node with identifier $i$ is connected to nodes $2i \mod 2^n$ and $2i + 1 \mod 2^n$.

A routing algorithm can route to any destination in $n$ hops by successively shifting in the bits of the destination identifier.

Routing a message from node $m$ to node $k$ is accomplished by taking the number $m$ and shifting in the bits of $k$ one at a time until the number has been replaced by $k$. 
De Bruijn Graph

Consider a node $n$ with identifier $b_1 b_2 \ldots b_k$, $b_i \in \{0, 1\}$

$n$ has an out-edge to the nodes with identifier $b_2 \ldots b_k 0$ and $b_2 \ldots b_k 1$.

Node 00: out edge to 00 and 01
Node 01: out edge to 10 and 11
Node 10: out edge to 00 and 01
Node 11: out edge to 10 and 11

This adjacency scheme, based on shifting the identifier strings associated with a node yields a simple prefix based routing policy.
Constructing de Bruijn Graphs

De Bruijn graph for $2^m$ node network can be constructed in a recursive fashion from a $2^{m-1}$ node network.

Take the edge of the $2^{m-1}$ node network

Add a node in the middle

Details:
Example: Adding a digit

Example: Adding a digit

The XOR Geometry

The Kademlia P2P system defines a routing metric in which the distance between two nodes is the numeric value of the exclusive OR (XOR) of their identifiers.

The idea is to take messages closer to the destination by using the XOR distance $d(x,y) = \text{XOR}(x,y)$ (taken as an integer).

The routing therefore "fixes" high order bits in the current address to take it closer to the destination.

Satisfies triangle property, symmetric, unidirectional.
XOR Metric and Triangle Property

Triangle inequality property
\[ d(x,z) \leq d(x,y) + d(y,z) \]

Easy to see that XOR satisfies this

Useful for determining distances between nodes

Unidirectional:
For any given point x and a distance D > 0, there is exactly one point y such that \( d(x,y) = D \). This means that lookups converge.
Comparing geometries

Gummadi et al. compared the different geometries, including the tree, hypercube, butterfly, ring, and XOR geometries.

Loguinov et al. complemented this list with de Bruijn graphs.

The conclusions of these comparisons include that the ring, XOR, and de Bruijn geometries are more flexible than the others and permit the choice of neighbours and alternative routes.

The ring and XOR geometries were also found to be the most flexible in terms of choosing neighbours and routes.

Only de Bruijn graphs allow alternate paths that are independent of each other.
Comparison

Can you choose neighbours?

Can you choose routes?

Are there alternative routes?

Are there alternative routes without overlap?
## Comparison

<table>
<thead>
<tr>
<th></th>
<th>Tree</th>
<th>Hypercube</th>
<th>Ring</th>
<th>Butterfly</th>
<th>XOR</th>
<th>De Bruijn</th>
</tr>
</thead>
<tbody>
<tr>
<td>Neighbour selection</td>
<td>Yes</td>
<td>1</td>
<td>Yes</td>
<td>1</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Route selection</td>
<td>1</td>
<td>Yes</td>
<td>Yes</td>
<td>1</td>
<td>Some</td>
<td>Yes</td>
</tr>
<tr>
<td>Sequential neighbours</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Independent paths</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Discussion

Based on previous table the ring looks pretty good

But this is partly due to the sequential neighbours property (predecessor and successor on the ring)

If sequential neighbours is added to other geometries, XOR and de Bruijn are also good
Distributed Data Structures (DDS)

- DHTs are an example of DDS
- DHT algorithms are available for clusters and wide-area environments
  - They are different!
- Cluster-based solutions
  - Ninja
  - LH* and variants
- Wide-area solutions
  - Chord, Tapestry, ..
  - Flat DHTs, peers are equal
  - Maintain a subset of peers in a routing table
Distributed Data Structures (DDS)

• Ninja project (UCB)
  – New storage layer for cluster services
  – Partition conventional data structure across nodes in a cluster
  – Replicate partitions with replica groups in cluster
    • Availability
      – Sync replicas to disk (durability)

• Other DDS for data / clusters
  – LH* Linear Hashing for Distributed Files
  – Redundant versions for high-availability
Taxonomy

Data Structures
- SDDS (1993)
  - Classic data structures
  - Hash-based
    - 1-dimensional
      - LH*, DDH, DHTs (Chord, ...)
    - d-dimensional
      - DHTs (CAN, ...)
  - Tree-based
    - 1-d Tree
      - RP*, ...
    - m-d Tree
      - k-RP*, ...
    - Security
      - LH\textsuperscript{*}_m, LH\textsuperscript{*}_g
    - k-Availability
      - LH\textsuperscript{*}_{RS}
      - LH\textsuperscript{*}_{sa}
      - LH\textsuperscript{*}_{RS}^{p2p}
      - LH\textsuperscript{*}_s
Linear Hashing

Use a family of hash functions $h_0, h_1, h_2, \ldots$
Each function’s range is twice that of its predecessor

When all the pages at one level (the current hash function) have been split, a new level is applied

Splitting occurs gradually
Current hash function, then you know if a bucket has been split from a split counter

Pages are split when overflows occur – but not necessarily the page with the overflow

Splitting a round robin fashion
Linear Hashing II

Use a family of hash functions $h_0, h_1, h_2, ...$

$h_i(key) = h(key) \mod (2^iN)$

$N = \text{initial number of buckets}$

$h$ is some hash function

$h_{i+1}$ doubles the range of $h_i$

Keep track of the next bucket to split and the current level: half of a split bucket is moved to the new bucket
Algorithm proceeds in rounds. Current round number is Level, Next = 0

There are $N_{level} \cdot (N \cdot 2^{Level})$ buckets at round start

Buckets 0 to Next-1 have been split
Next to $N_{Level}$ have not been split yet
Round ends when all initial buckets have been split (when Next = $N_{Level}$).

To start next round:
Level=Level+1
Next = 0
Linear Hashing III

Algorithm proceeds in rounds. Current round number is Level, Next = 0

There are $N_{\text{level}} \cdot (N \cdot 2^{\text{Level}})$ buckets at round start

- Buckets 0 to Next-1 have been split
- Next to $N_{\text{Level}}$ have not been split yet
- Round ends when all initial buckets have been split (when Next = $N_{\text{Level}}$).

To start next round:
- Level = Level + 1
- Next = 0
Example

Start: $i = 0$, $N = 4$, next = 0
Overflow of 3: $i = 0$, $N = 4$, next = 1
Overflow of 1: $i = 0$, $N = 4$, next = 2
Overflow of 4: $i = 0$, $N = 4$, next = 3
Overflow of 2: $i = 0$, $N = 4$, next = 0
Next level: $i = 1$, $N = 4$, next = 0

When splitting, half of the content is moved to the new bucket, just take this into account when looking up (old and new hash function)

Now we have moved to the new hash function altogether, splitting starts again!
Read operation

Use $h(\text{level}, \text{key})$ if it is greater than or equal to the next counter.

Otherwise use $h(\text{level}+1, \text{key})$, because they have been rehashed with the new level.
Overflow of a bucket

What happens if there is no space, bucket overflows and it is not the next bucket to split?

Use overflow buckets, normal bucket has a pointer to the overflow bucket

Overflow bucket taken into account when the bucket in question is split (round robin)
Linear hashing

Spreads the cost of the expansion across insertion operations

Buckets split one at a time
LH* Linear Hashing for Distributed Files

LH* generalizes linear hashing to decentralized distributed operation.

The system supports constant time insertion and lookup of data objects in a cluster.

Data items are hashed into buckets with each bucket residing on a server. New servers are incorporated into the system when a bucket overflows using a split operation.

A split controller manages the split operation. When a split is performed, a new server is added to the system from a supply of servers and the hashing parameters are adjusted accordingly.

In a distributed environment, the clients have a view to these system parameters which in some cases maybe out of date. This requires auto-correction and synchronization mechanisms.
LH* Example

Client 1
n’=5
i’=6

Client 2
n’=0
i’=2

Client m
n’=31
i’=9

srvr 0
10

srvr 1
10

srvr 80
9

srvr 512
10

srvr 583
10

srvr 591
10

n=80
LH* Bucket Split

1. Overflow
2. Split
3. Init
4. Splitdone

Insert

Bucket c

Bucket n

Bucket n+2^l

Tuples
Cluster-based Distributed Hash Tables (DHT)

- The NINJA project
- Directory for non-hierarchical data
- Several different ways to implement
- A distributed hash table
  - Each “brick” maintains a partial map
    - “local” keys and values
  - **Overlay addresses** used to direct to the right “brick”
    - “remote” key to the brick using hashing
- Resilience through **parallel**, unrelated mappings
The API provides services with `put()`, `get()`, `remove()`, `destroy()` operations on hash tables.

Behind the API the DDS needs to implement the mechanisms to access, partition, replicate, scale, and recover data.

A distributed hash table was implemented as an example of the DDS concept in Ninja. All operations inside the distributed hash table are atomic meaning that a given operation is either performed fully or not at all. In order to ensure reliability,

Elements are replicated within the DDS across multiple nodes called *bricks*. A **two-phase commit algorithm** is used to keep the replicas coherent. A brick consists of a buffer cache, a lock manager, a persistent chained hash table implementation, and an RPC communications system.
clients interact with any service “front-end”

Service interacts with DSS lib
Hash table API

SAN

Brick = single-node, durable hash table, replicated

Redundant, low latency, high throughput network

client

service

DSS lib

storage “brick”

storage “brick”

storage “brick”
Summary

Geometries form the basis of the structured overlay algorithms.

A Distributed Data Structure (DDS) is a self-managing storage layer that runs on a cluster. The aim of the DDS is to support high throughput, high concurrency, availability, incremental scalability, offer strict consistency guarantees for the data.

The LH* family of algorithms are scalable DDSes intended for clusters.

**Consistent hashing** allows buckets to be added in any order, whereas Litwin’s Linear Hashing (LH*) scheme requires buckets to be added one at a time in sequence.

The Ninja system was designed to support robust distributed Internet services. One key component of the system was a cluster of servers for scalable service.