Proposal
Towards a Framework for DHT Distributed Computing

Andrew Rosen

Georgia State University

July 15th, 2015

Table of Contents

Objective

Our objective is to create a generalized framework for distributed computing using Distributed Hash Tables.

Objective

Our objective is to create a generalized framework for distributed computing using Distributed Hash Tables.

Or

Objective

Our objective is to create a generalized framework for distributed computing using Distributed Hash Tables.

Or

We want to build a completely decentralized distributed computing framework.

What do I Mean by Distributed Computing?

A system where we can take a task and break it down into multiple parts, where each part is worked upon individually.

Challenges of Distributed Computing

Distributed Computing platforms experience these challenges:

Scalability As the network grows, more resources are spent on maintaining and organizing the network.

Challenges of Distributed Computing

Distributed Computing platforms experience these challenges:

Scalability As the network grows, more resources are spent on maintaining and organizing the network.

Fault-Tolerance As more machines join the network, there is an increased risk of failure.

Georgia State University

Challenges of Distributed Computing

Distributed Computing platforms experience these challenges:

Scalability As the network grows, more resources are spent on maintaining and organizing the network.

Fault-Tolerance As more machines join the network, there is an increased risk of failure.

Load-Balancing Tasks need to be evenly distributed among all the workers.

Distributed Key/Value Stores

Distributed Hash Tables are mechanisms for storing values associated with certain keys.

- Values, such as filenames, data, or IP/port combinations are associated with keys.
- These keys are generated by taking the hash of the value.
- We can get the value for a certain key by asking any node in the network.

Current Applications

Applications that use or incorporate DHTs:

- P2P File Sharing applications, such as BitTorrent.
- Distributed File Storage.
- Distributed Machine Learning.
- Name resolution in a large distributed database.

How Does It Work?

We'll explain in greater detail later, but briefly:

- DHTs organize a set of nodes, each identified by an **ID**.
- Nodes are responsible for the keys that are closest to their IDs.
- Nodes maintain a list of other peers in the network.
    - Typically a size $\log(n)$ subset of all nodes in the network.
- Each node uses a very simple routing algorithm to find a node responsible for any given key.

Strengths of DHTs

DHTs are designed for large P2P applications, which means they need to be (and are):

- Scalable
- Fault-Tolerant
- Load-Balancing

DHTs Address the Specified Challenges

The big issues in distributed computing can be solved by the mechanisms provided by Distributed Hash Tables.

Uses For DHT Distributed Computing

The generic framework we are proposing would be ideal for:

- Embarrassingly Parallel Computations
  - Any problem that can be framed using Map and Reduce.
  - Brute force cryptography.
  - Genetic algorithms.
  - Markov chain Monte Carlo methods.
- Use in either a P2P context or a more traditional deployment.

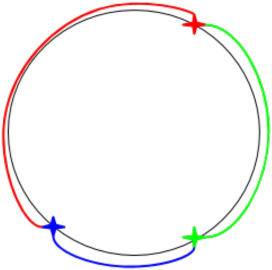
Table of Contents

Required Attributes of DHT

- A distance and midpoint function.
- A closeness or ownership definition.
- A Peer management strategy.

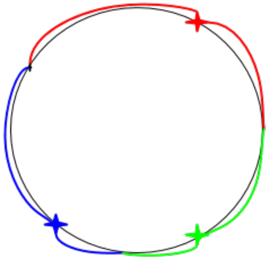
Chord's Closest Metric.

Figure: A Voronoi diagram for a Chord network, using Chord's definition of closest.



Chord Using A Different Closest Metric

Figure: A Voronoi diagram for a Chord network, where closest is defined by the node being the closest in either direction.



Terms and Variables

- Network size is n nodes.
- Keys and IDs are m bit hashes, usually SHA1.
- Peerlists are made up of:

  Short Peers The neighboring nodes that define the network's topology.

  Long Peers Routing shortcuts.
- We'll call the node responsible for a key the *root* of the key.

Functions

lookup(key) Finds the node responsible for a given key.

put(key, $value$) Stores *value* at the node responsible for *key*, where $key = hash(value)$.

get(key) Returns the *value* associated with *key*.

Chord

- Ring Topology
- Short Peers: predecessor and successor in the ring.
- Responsible for keys between their predecessor and themselves.
- Long Peers: $\log n$ nodes, where the node at index i in the peerlist is

$$root(r + 2^{i-1} \mod m), 1 < i < m$$

Georgia State University

A Chord Network

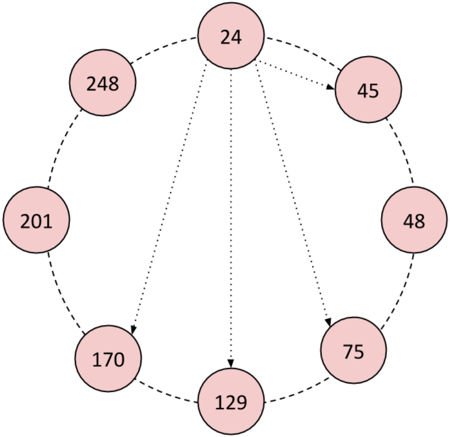


Figure: An 8-node Chord ring where $m = 8$. Node 24's long peers are shown.

Fault Tolerence in Chord

- Local maintenance thread gradually fixes the network topology.
  - Each node “notifies” its successor.
  - The successor replies with a better successor if one exists.
- The long peers are gradually updated by performing a lookup on each entry.

Handling Churn in General

- Short peers, the neighbors, are periodically queried to:
  - See of the node is still alive.
  - See if the neighbor knows about better nodes.
- Long peer failures are replaced by regular maintenance.

Table of Contents

Overarching Goal

My research has been focused on:

- Abstracting out DHTs.
- Distributed computation using DHTs.

Goals

VHash sprung from two related ideas:

- We wanted a way be able optimize latency by embedding it into the routing overlay.

Goals

VHash sprung from two related ideas:

- We wanted a way be able optimize latency by embedding it into the routing overlay.
- We wanted to create a DHT based off of Voronoi tessellations. Unfortunately:

Goals

VHash sprung from two related ideas:

- We wanted a way be able optimize latency by embedding it into the routing overlay.
- We wanted to create a DHT based off of Voronoi tessellations. Unfortunately:
  - Distributed algorithms for this problem don't really exist.

Goals

VHash sprung from two related ideas:

- We wanted a way be able optimize latency by embedding it into the routing overlay.
- We wanted to create a DHT based off of Voronoi tessellations. Unfortunately:
  - Distributed algorithms for this problem don't really exist.
  - Existing approximation algorithms were unsuitable.

Voronoi Tesselation

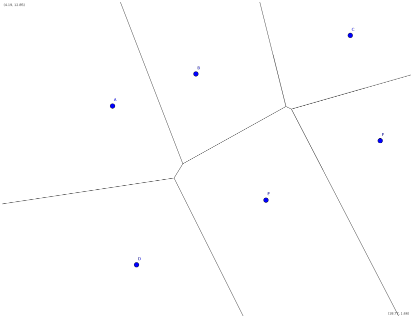


Figure: A set of points and the generated Voronoi regions

Delaunay Triangulation

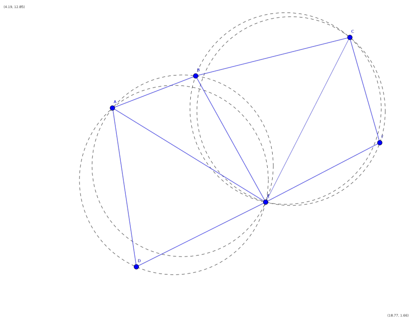


Figure: The same set of nodes with their corresponding Delaunay Triangulation.

DHT and Voronoi Relationship

- We can view DHTs in terms of Voronoi tessellation and Delaunay triangulation.
    - The set of keys the node is responsible for is its Voronoi region.
    - The nodes neighbors are it's Delaunay neighbors.

VHash

- Voronoi-based Distributed Hash Table based on this relationship.
- Uses our approximation to solve for Delaunay neighbors, called DGVH.
- Topology updates occur via gossip-based protocol.
- Routing speed is $O(\sqrt[d]{n})$
- Memory Cost
    - Worst case: $O(n)$
    - Expected maximum size: $\Theta(\frac{\log n}{\log \log n})$

Distributed Greedy Voronoi Heuristic

- Assumption: The majority of Delaunay links cross the corresponding Voronoi edges.
- We can test if the midpoint between two potentially connecting nodes is on the edge of the Voronoi region.
- This intuition fails if the midpoint between two nodes does not fall on their Voronoi edge.

DGVH Heuristic

Given node n and its list of *candidates*.
peers $\leftarrow$ empty set that will contain n's one-hop peers
Sort *candidates* in ascending order by each node's distance to n
Remove the first member of *candidates* and add it to *peers*
for all c in *candidates* **do**
    m is the midpoint between n and c
    if Any node in *peers* is closer to m than n **then**
        Reject c as a peer
    else
        Remove c from *candidates*
        Add c to *peers*
    end if
end for

Results

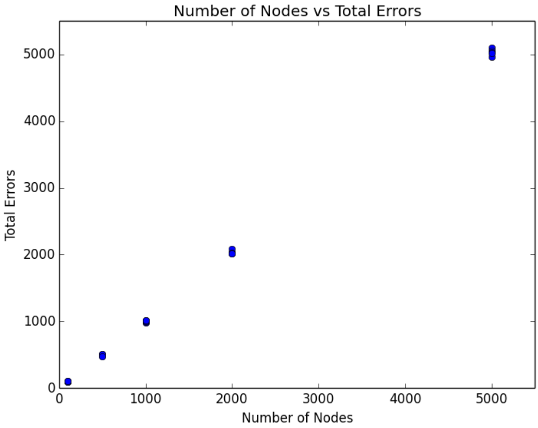


Figure: As the size of the graph increases, we see approximately 1 error per node.

Results

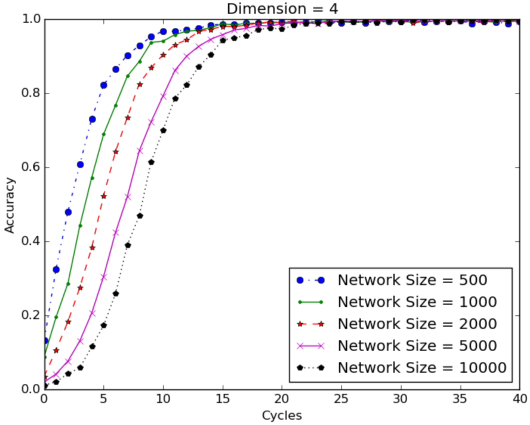


Figure: These figures show, starting from a randomized network, VHash forms a stable and consistent network topology.

Results

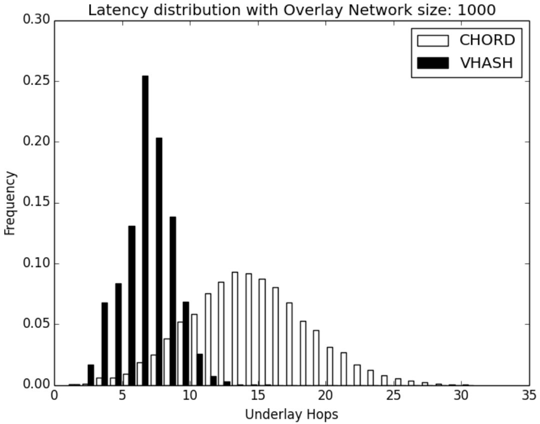


Figure: Comparing the routing effectiveness of Chord and VHash.

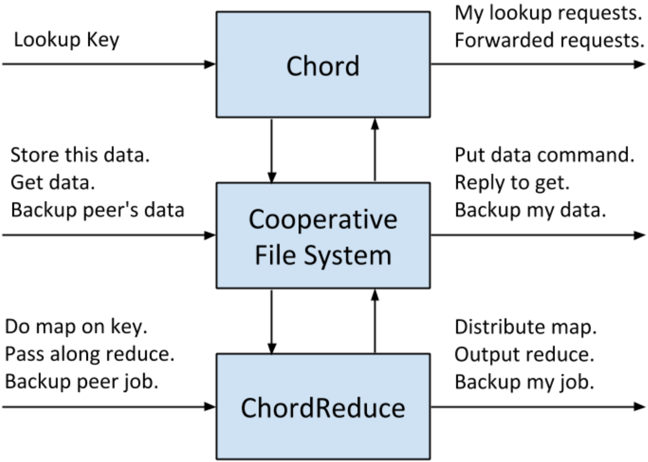
Conclusions

- DGVH is simple approximation for Delaunay Triangulation that guarantees a fully connected graph.
- VHash can optimize over a metric such as latency and achieve superior routing speeds as a result.

Goals

- We wanted build a more abstract system for MapReduce.
- We remove core assumptions:
  - The system is centralized.
  - Processing occurs in a static network.
- The resulting system must be:
  - Completely decentralized.
  - Scalable.
  - Fault tolerant.
  - Load Balancing.

System Architecture



Georgia State University

Mapping Data

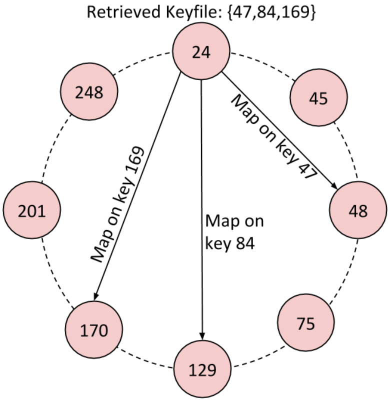


Figure: The stager sends a map task for each key in the keyfile. In larger networks, this process is streamlined by recursively bundling keys and sending them to the best finger.

Reducing Results of Data

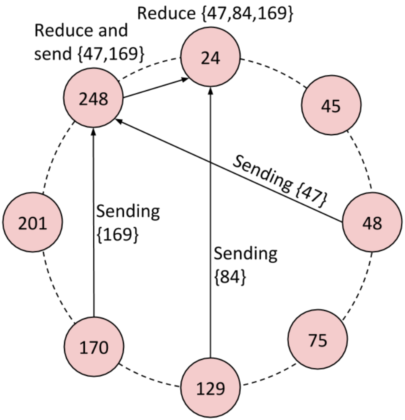


Figure: Results are sent back via the overlay. If a node receives multiple results, they are reduced before being sent on.

Experiment Details

Our test was a Monte Carlo approximation of π.

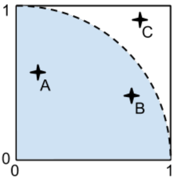


Figure: The node chooses random x and y between 0 and 1. If $x^2 + y^2 < 1^2$, the "dart" landed inside the circle.

- Map jobs were sent to randomly generated hash addresses.
- The ratio of hits to generated results approximates $\frac{\pi}{4}$.
- Reducing the results was a matter of combining the two fields.

Experimental Results

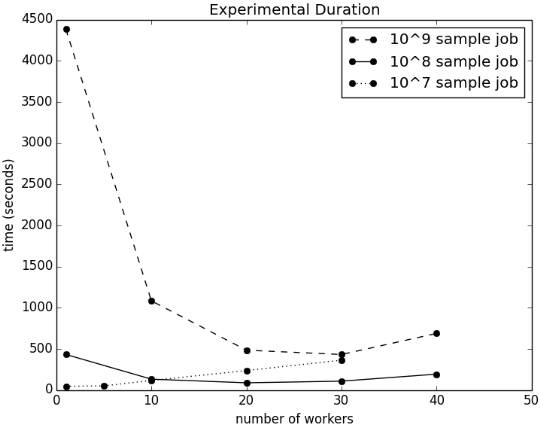


Figure: For a sufficiently large job, it was almost always preferable to distribute it.

Churn Results

Churn rate per second	Average runtime (s)	Speedup vs 0% churn
0.8%	191.25	2.15
0.4%	329.20	1.25
0.025%	431.86	0.95
0.00775%	445.47	0.92
0.00250%	331.80	1.24
0%	441.57	1.00

Table: The results of calculating π by generating 10^8 samples under churn. Churn is the chance for each node to join or leave the network. The large speedup is from joining nodes acquiring work during experimental runtime.

Conclusions

Our experiments established:

- ChordReduce can operate under high rates of churn.
- Execution follows the desired speedup.
- Speedup occurs on sufficiently large problem sizes.

This makes ChordReduce an excellent platform for distributed and concurrent programming in cloud and loosely coupled environments.

The Sybil Attack

- The Sybil attack is a type of attack against a distributed system such as a DHT.
- The adversary pretends to be more than one identity in the network.
    - Each of these false identities, called a Sybil is treated as a full member of the network.
- The overall goal is to occlude healthy nodes from one another.
- The Sybil attack is extremely well known, but there is little literature written from the attacker's perspective.

The Sybil Attack in A P2P network

See Whiteboard

- We want to inject a Sybil into as many of the regions between nodes as we can.
- The question we wanted to answer is what is the probability that a region can have a Sybil injected into it, given:
    - The network size n
    - The number of IDs available to the attacker (the number of identities they can fake).

Assumptions

- The attacker is limited in the number of identities they can fake.
    - To fake an identity, the attacker must be able to generate a valid IP/port combo he owns.
    - The attacker therefore has $num\_IP \cdot num\_ports$ IDs.
    - We'll set $num\_ports = 16383$, the number of ephemeral ports.
    - Storage cost is 320 KiB.
- We call the act of finding an ID by modulating your IP and port so you can inject a node *mashing*.
- In Mainline DHT, used by BitTorrent, you can choose your own ID at "random." The implications should be apparent.

Georgia State University

Analysis

The probability you can mash a region between two adjacent nodes in a size n network is:

$$P \approx \frac{1}{n} \cdot num\_ips \cdot num\_ports \tag{1}$$

An attacker can compromise a portion $P_{bad\_neighbor}$ of the network given by:

$$P_{bad\_neighbor} = \frac{num\_ips \cdot num\_ports}{num\_ips \cdot num\_ports + n - 1} \tag{2}$$

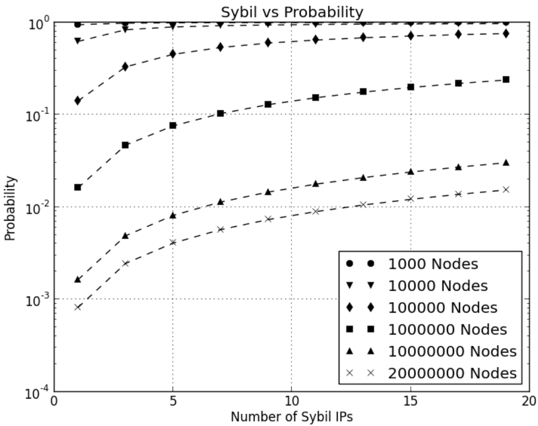


Figure: Our simulation results.

The dotted line traces the line corresponding to the Equation 2:

$$P_{bad\_neighbor} = \frac{num\_ips \cdot 16383}{num\_ips \cdot 16383 + n - 1}$$

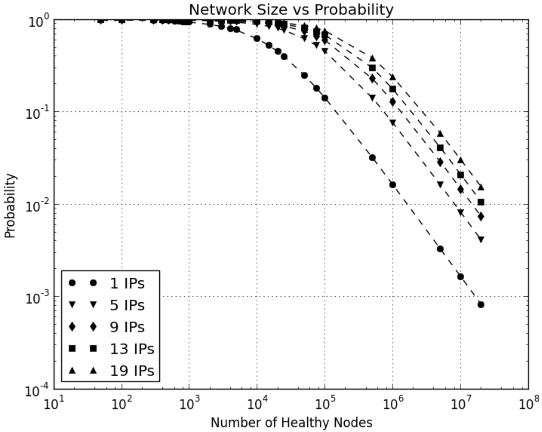


Figure: These are the same as results shown in Figure 13, but our *x*-axis is the network size *n* in this case. Here, each line corresponds to a different number of unique IP addresses the adversary has at their disposal.

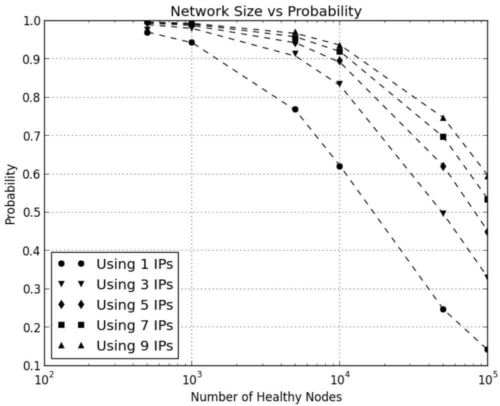


Figure: This graph shows the relationship between the network size and the probability a particular link, adjacent or not, can be mashed.

Conclusion

- Our analysis showed an adversary with limited resources can occlude the majority of the paths between nodes.
- An attack of this sort on Mainline DHT would cost about $43.26 USD per hour.
- Moreover, we demonstrated that creating virtual nodes is cheap and easy.

Table of Contents

UrDHT

- UrDHT is a completely abstracted DHT that will serve as a framework for creating DHTs.
- The goal is **not only** to create a DHT, but to create an easily extensible abstract framework for DHTs.
- Continuation of the work in VHash.

UrDHT

- We will be creating a mathematical description of what a DHT is.
- We will implement various DHTs using UrDHT and compare their performance.

DHT Distributed Computing

- We will use UrDHT to implement a few of the more popular DHTs.
    - See if there is a difference for distributed computing.
    - Using UrDHT for all the implementations will minimize the differences between each DHT.

DHT Distributed Computing

- Implement distributed computing on each of the implemented DHTs.
  - The emphasis is robustness and fault-tolerance.
- Test each framework using a variety of embarrassingly parallel problems, such as:
  - Brute-force cryptanalysis.
  - MapReduce problems.
  - Monte-Carlo computations.

Autonomous Load-Balancing

- We will confirm that the effect from the high rate of Churn exists.
- We must create a scoring mechanism for nodes.
- The last step is to implement load-balancing strategies.

Autonomous Load-Balancing Strategies

A few strategies we've thought up.

- Passive load-balancing: Nodes create virtual nodes based on their score.
- Traffic analysis: Create replicas where there is a high level of traffic.
- Invitation: Nodes with large areas of responsibility can invite other nodes to help.