Proposal Defense
Towards a Framework for DHT Distributed Computing

Andrew Rosen

Georgia State University

July 15th, 2015

Table of Contents

1. Introduction
   - What I am doing
   - Distributed Computing and Challenges
   - What Are Distributed Hash Tables?
   - Why DHTs and Distributed Computing
2. Background
   - The Components and Terminology
   - Example DHT: Chord
3. Completed Work
   - ChordReduce
   - VHash
   - Sybil Attack Analysis
     - Results
4. Proposed Work
   - UrDHT
   - DHT Distributed Computing
   - Autonomous Load-Balancing

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.

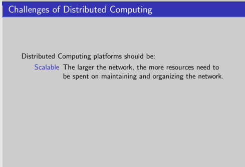
What do I Mean by Distributed Computing?

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 should be:

- Scalable The larger the network, the more resources need to be spent on maintaining and organizing the network.



Challenges of Distributed Computing

Distributed Computing platforms should be:

Scalable The larger the network, the more resources need to be spent on maintaining and organizing the network.

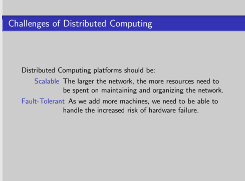
Remember, computers aren't telepathic. There's always an overhead cost. It will grow. The challenge of scalability is designing a protocol that grows this organizational cost at an extremely slow rate. For example, a single node keeping track of all members of the system might be a tenable situation up to a certain point, but eventually, the cost becomes too high for a single node.

Challenges of Distributed Computing

Distributed Computing platforms should be:

Scalable The larger the network, the more resources need to be spent on maintaining and organizing the network.

Fault-Tolerant As we add more machines, we need to be able to handle the increased risk of hardware failure.



Challenges of Distributed Computing

Distributed Computing platforms should be:

- Scalable: The larger the network, the more resources need to be spent on maintaining and organizing the network.
- Fault-Tolerant: As we add more machines, we need to be able to handle the increased risk of hardware failure.

Hardware failure is a thing that can happen. Individually the chances are low, but this becomes high when we're talking about millions of machines. Also, what happens in a P2P environment.

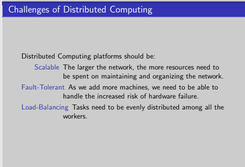
Challenges of Distributed Computing

Distributed Computing platforms should be:

Scalable The larger the network, the more resources need to be spent on maintaining and organizing the network.

Fault-Tolerant As we add more machines, we need to be able to handle the increased risk of hardware failure.

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



Challenges of Distributed Computing

Distributed Computing platforms should be:

Scalable The larger the network, the more resources need to be spent on maintaining and organizing the network.

Fault-Tolerant As we add more machines, we need to be able to handle the increased risk of hardware failure.

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

If we are splitting the task into multiple parts, we need some mechanism to ensure that each worker gets an even (or close enough) amount of work.

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

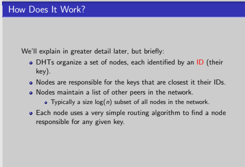
Applications that use or incorporate DHTs:

- P2P File Sharing applications, such as Bittorrent [2] [5].
- Distributed File Storage [3].
- Distributed Machine Learning [4].
- Name resolution in a large distributed database [6].

How Does It Work?

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

- DHTs organize a set of nodes, each identified by an ID (their key).
- Nodes are responsible for the keys that are closest it 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.



How Does It Work?

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

- DHTs organize a set of nodes, each identified by an ID (their key).
- Nodes are responsible for the keys that are closest it 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.

We use ID for nodes and keys for data so we always know our context.

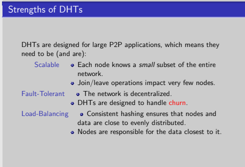
Strengths of DHTs

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

Scalable
- Each node knows a *small* subset of the entire network.
- Join/leave operations impact very few nodes.

Fault-Tolerant
- The network is decentralized.
- DHTs are designed to handle churn.

Load-Balancing
- Consistent hashing ensures that nodes and data are close to evenly distributed.
- Nodes are responsible for the data closest to it.



Strengths of DHTs

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

Scalable
- Each node knows a *small* subset of the entire network.
- Join/leave operations impact very few nodes.

Fault-Tolerant
- The network is decentralized.
- DHTs are designed to handle churn.

Load-Balancing
- Consistent hashing ensures that nodes and data are close to evenly distributed.
- Nodes are responsible for the data closest to it.

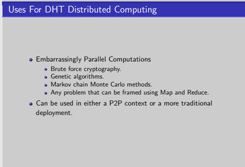
- Remember to mention Napster.
- Distributed Hash Tables were designed to be used for completely decentralized P2P applications involving millions of nodes.
- As a result of the P2P focus, DHTs have the following qualities.
- Scalability
  - The subset each node knows is such that we have expected $\lg(n)$ lookup
- Fault-Tolerance
  - Because Joins and node failures affect only nodes in the immediate vicinity, very few nodes are impacted by an individual operation.
- Load Balancing
  - The space is large enough to avoid Hash collisions

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

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



Uses For DHT Distributed Computing

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

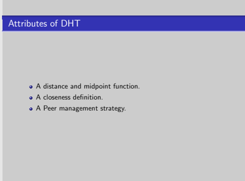
- Need notes here
- Define Monte-Carlo Markov Chain

Table of Contents

1. Introduction
   - What I am doing
   - Distributed Computing and Challenges
   - What Are Distributed Hash Tables?
   - Why DHTs and Distributed Computing
2. Background
   - The Components and Terminology
   - Example DHT: Chord
3. Completed Work
   - ChordReduce
   - VHash
   - Sybil Attack Analysis
     - Results
4. Proposed Work
   - UrDHT
   - DHT Distributed Computing
   - Autonomous Load-Balancing

Attributes of DHT

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



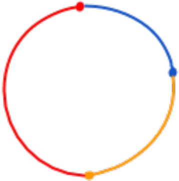
Attributes of DHT

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

- There needs to be a way to establish how far things are from one another. Once we have a distance metric, we define what we mean when we say a node is responsible for all data *close* to it.
- We'll go into more detail about the difference between Distance and Closeness in Chord

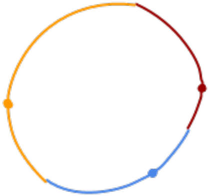
Chord's Closest Metric.

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 if defined by the node being the closest in either direction.

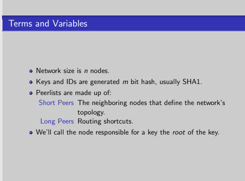


Terms and Variables

- Network size is *n* nodes.
- Keys and IDs are generated *m* bit hash, 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.



Terms and Variables

- Network size is n nodes.
- Keys and IDs are generated m bit hash, 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.

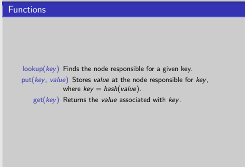
- SHA1 is being depreciated.
- Short peers are actively maintained, long peers replaced gradulally and are not actively pinged.
- We use root as it's is a topology agnostic term.

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*.



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*).

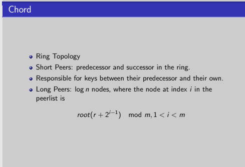
get(*key*) Returns the *value* associated with *key*.

- There is usually a delete function as well, but it's not important.
- All nodes use the same general lookup: Forward the message to the node closest to *key*

Chord

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

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



Chord

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

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

- Chord is a favorite because we can draw it.
- Draw a Chord network on the wall?
- node r is our root node.
- i is the index on the list
- English for the equation, the long peers double in distance from the root node, allowing us to cover at least half the distance to our target in a step
- In this way, we can achieve an expected $\lg n$ hops.

A Chord Network

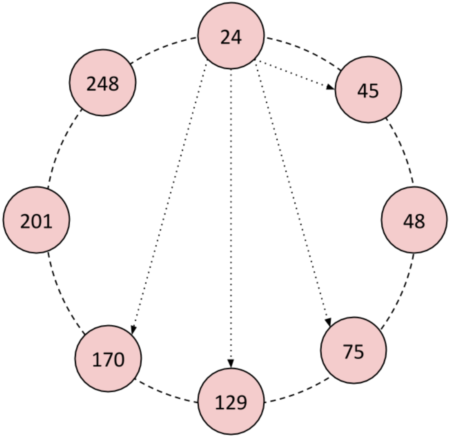


Figure : An 8-node Chord ring where $m = 8$. Node 24's fingers are shown.

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.
- Some implementations use predecessor and successor lists.

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.

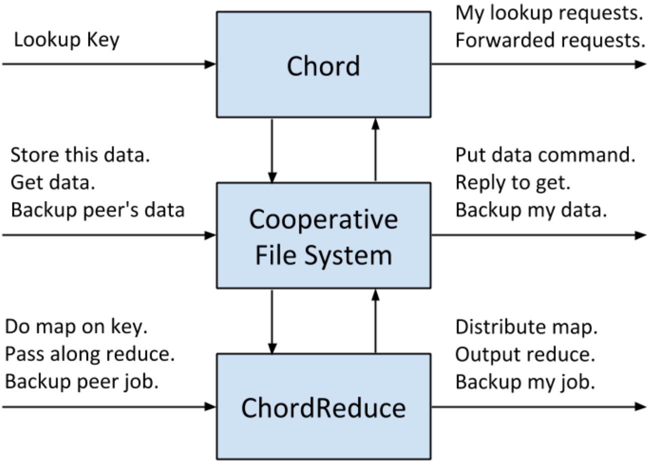
Table of Contents

1. Introduction
   - What I am doing
   - Distributed Computing and Challenges
   - What Are Distributed Hash Tables?
   - Why DHTs and Distributed Computing
2. Background
   - The Components and Terminology
   - Example DHT: Chord
3. Completed Work
   - ChordReduce
   - VHash
   - Sybil Attack Analysis
     - Results
4. Proposed Work
   - UrDHT
   - DHT Distributed Computing
   - Autonomous Load-Balancing

Goals

- We wanted build a more abstract system for MapReduce.
- We remove core assumptions [1]:
  - The system is centralized.
  - Processing occurs in a static network.
- The resulting system must be:
  - Completely decentralized.
  - Scalable.
  - Fault tolerant.
  - Load Balancing.

System Architecture



Lookup Key

Chord

My lookup requests.
Forwarded requests.

Store this data.
Get data.
Backup peer's data

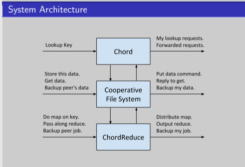
Cooperative File System

Put data command.
Reply to get.
Backup my data.

Do map on key.
Pass along reduce.
Backup peer job.

ChordReduce

Distribute map.
Output reduce.
Backup my job.


System Architecture

- Chord [7], which handles routing and lookup.
- The Cooperative File System (CFS) [3], which handles storage and data replication.
- The MapReduce layer.
- Files are split up, each block given a key based on their contents.
- Each block is stored according to their key.
- The hashing process guarantees that the keys are distributed near evenly among nodes.
- A keyfile is created and stored where the whole file would have been found.
- To retrieve a file, the node gets the keyfile and sends a request for each block listed in the keyfile

Mapping Data

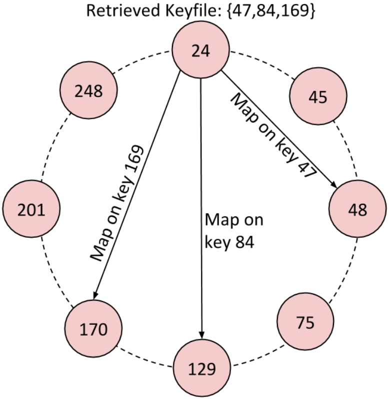
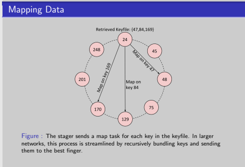


Figure : The stager sends a map task for each key in the keyfile. In larger networks, this process is streamlined by recursively bundling keys and sending them to the best finger.



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.

- this process is recursive

Reducing Results of Data

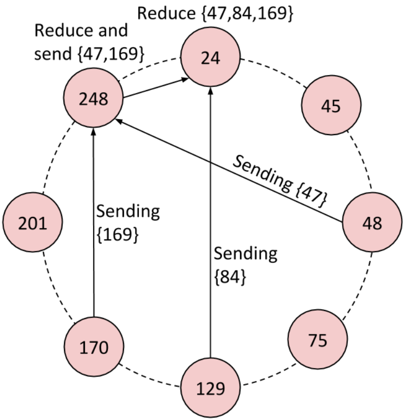
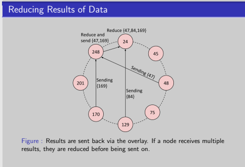


Figure : Results are sent back via the overlay. If a node receives multiple results, they are reduced before being sent on.



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.

The paths here are arbitrary edges that I came up with for the example.

Experiment Details

Our test was a Monte Carlo approximation of π.

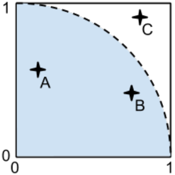
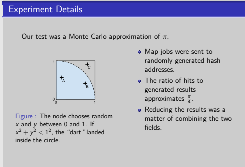


Figure : The node chooses random x and y between 0 and 1. If $x^2 + y^2 < 1^2$, the “dart ”landed inside the circle.

- 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.



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.

- Experiment had these goals
    1. ChordReduce provided significant speedup during a distributed job.
    2. ChordReduce scaled.
    3. ChordReduce handled churn during execution.

Experimental Results

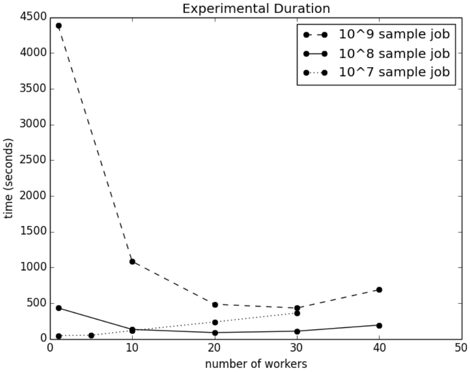
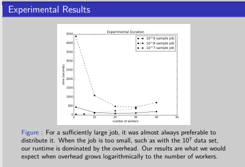


Figure : For a sufficiently large job, it was almost always preferable to distribute it. When the job is too small, such as with the 10^7 data set, our runtime is dominated by the overhead. Our results are what we would expect when overhead grows logarithmically to the number of workers.



Experimental Results

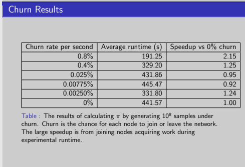
Figure : For a sufficiently large job, it was almost always preferable to distribute it. When the job is too small, such as with the 10^7 data set, our runtime is dominated by the overhead. Our results are what we would expect when overhead grows logarithmically to the number of workers.

- Diminishing returns.
- The 10^7 job was dominated by the overhead communication costs.

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.



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.

- We tested at rates from 0.0025% to 0.8% **per second**, 120 times the fastest rate used to test P2P-MapReduce.
- ChordReduce finished twice as fast under the unrealistic levels churn (0.8% per second) than no churn (Table 1).
- Churn is a disruptive force; how can it be aiding the network? We have two hypotheses
- Deleting nodes motivates other nodes to work harder to avoid deletion (a "beatings will continue until morale improves" situation).
- Our high rate of churn was dynamically load-balancing the network. How?
- Nodes that die and rejoin are more likely to join a region owned by a node with larger region and therefore more work.
- It appears even the smallest effort of trying to dynamically load balance, such as rebooting random nodes to new locations, has benefits for runtime. Our method is a poor approximation of dynamic load-balancing, and it still shows improvement.

Conclusions

Our experiments established:

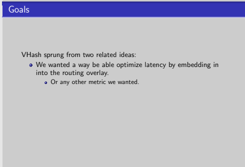
- ChordReduce can operate under high rates of churn.
- Execution follows the desired speedup.
- Speedup occurs on sufficiently large problem sizes.

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

Goals

VHash sprung from two related ideas:

- We wanted a way be able optimize latency by embedding in into the routing overlay.
    - Or any other metric we wanted.



Goals

VHash sprung from two related ideas:

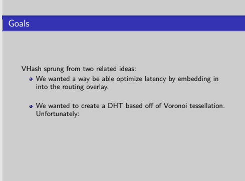
- We wanted a way be able optimize latency by embedding in into the routing overlay.
  - Or any other metric we wanted.

Most DHTs optimize routing for the number of hops, rather than latency.

Goals

VHash sprung from two related ideas:

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



Goals

VHash sprung from two related ideas:

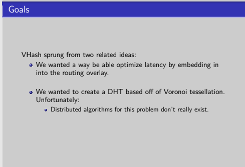
- We wanted a way be able optimize latency by embedding in into the routing overlay.
- We wanted to create a DHT based off of Voronoi tessellation. Unfortunately:

We discovered a mapping between Distributed Hash Tables and Voronoi/Delaunay Triangulations.

Goals

VHash sprung from two related ideas:

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



Goals

VHash sprung from two related ideas:

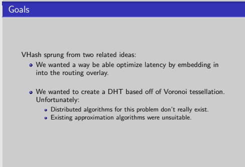
- We wanted a way be able optimize latency by embedding in into the routing overlay.
- We wanted to create a DHT based off of Voronoi tessellation. Unfortunately:
  - Distributed algorithms for this problem don't really exist.

I lie, they do exist, but they all are “run the global algorithm on your local subset. And if we move out of or above 2D Euclidean space, as Brendan wanted to, no fast algorithms exist at all. We quickly determined that solving was never really a feasible option. So that leaves approximation. A distributed algorithm would be helpful for some WSNs solving the boundary coverage problem.

Goals

VHash sprung from two related ideas:

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



Goals

VHash sprung from two related ideas:

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

Simple approximations have no guarantees of connectivity, which is very bad for a routing topology. Better algorithms that existed for this problem technically ran in constant time, but had a prohibitively high sampling. So to understand what I'm talking about here, let's briefly define what a Voronoi tessellation is.

Voronoi Tesselation

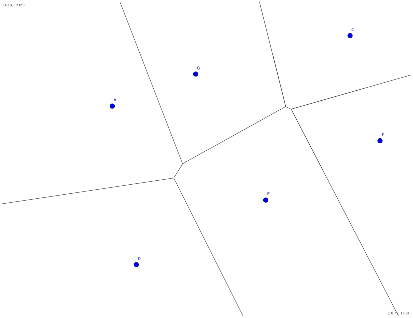
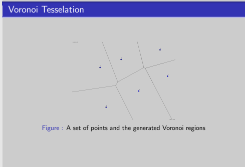


Figure : A set of points and the generated Voronoi regions



Voronoi Tesselation

Figure : A set of points and the generated Voronoi regions

Define

- A Voronoi tessellation or Voronoi diagram divides a space into regions, where each region encompasses all the points closest to Voronoi generators (point).
- Voronoi generators
- Voronoi Region
- Voronoi Tessellation/ Diagram

Delaunay Triangulation

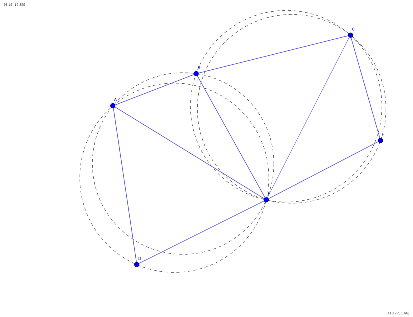
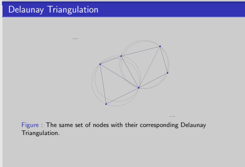


Figure : The same set of nodes with their corresponding Delaunay Triangulation.



Delaunay Triangulation

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

Define

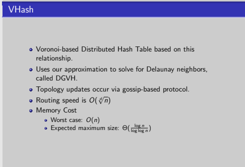
- Delaunay Triangulation is a triangulation of a set of points with the following rule:
- No point falls within any of the circumcirles for every triangle in the triangulation,
- The Voronoi tessellation and Delaunay Triangulation are dual problems
  - Solving one yields the other.
  - We can get the Voronoi diagram by connecting all the centers of circumcircles.
  -

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: $\Theta(\frac{\log n}{\log \log n})$



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: $\Theta(\frac{\log n}{\log \log n})$

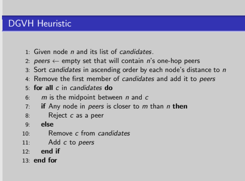
- d is number of dimensions, but we optimize over latency so that's deceptive
- Gossip: every cycle, a node chooses a peer and swaps peerlist information, then reruns the approximation.
- This would only happen in a contrived case and would give $O(1)$ routing
- We found a paper that proved $\Theta(\frac{\log n}{\log \log n})$ is the expected maximum degree of a vertex in a Delaunay Triangulation.

Distributed Greedy Voronoi Heuristic

- 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

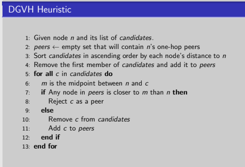
```
Given node $n$ and its list of $candidates$.
$peers \leftarrow$ empty set that will contain $n$'s one-hop peers
Sort $candidates$ in ascending order by each node's distance to $n$
Remove the first member of $candidates$ and add it to $peers$
for all $c$ in $candidates$ do
    $m$ is the midpoint 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
```



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
```

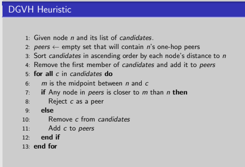
- The majority of Delaunay links cross the corresponding Voronoi edges.
- We can test if the midpoint between two potentially connecting nodes is on the edge of the voronoi region.
- This intuition fails if the midpoint between two nodes does not fall on their voronoi edge.



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
```

1. 'n' is the "myself" node, and the location we are seeking to find the peers of.
2. peers is a set that will build the peerlist in
3. We sort the candidates from closest to farthest.
4. The closest candidate is always guaranteed to be a peer.
5. Iterate through the sorted list of candidates and either add them to the peers set or discard them.
6. We calculate the midpoint between the candidate and the center 'n'.
7. If this midpoint is closer to a peer than 'n', then it does not fall on the interface between the location's Voronoi regions.
8. in this case discard it
9. otherwise add it the current peerlist



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
```

Theoretically, this is worst case $O(n^2)$.
However in practice, this is $O(nlog(n)(sorting) + kn)$ where$k$ is the number of Delaunay peers.
We are well aware that 2d-euclidean algorithms exist in O(nlog(n)) time. While we use that use case to communicate the algorithm, it is intended to be used in more exotic spaces.
realistically k is the function of the metric space and is O(1) for euclidean spaces.

Results

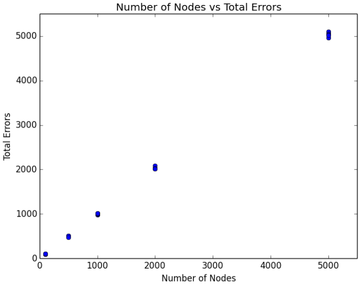


Figure : As the size of the graph increases, we see approximately 1 error per node. We can also see that the error rate and number of nodes has a linear relationship.

Results

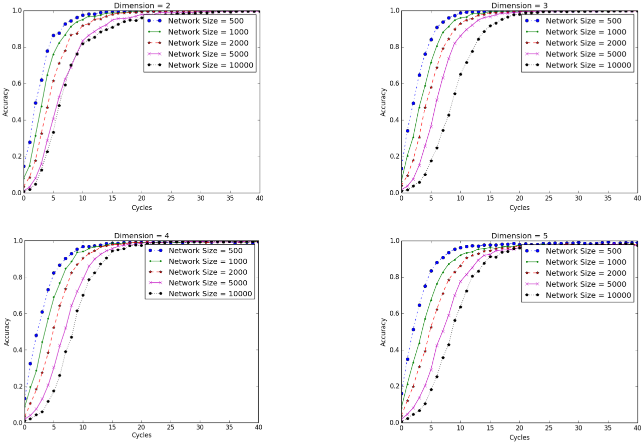


Figure : These figures show, starting from a randomized network, VHash forms a stable and consistent network topology.

Results

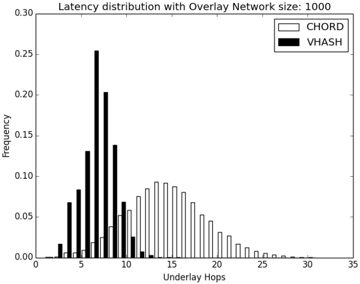


Figure : This how the difference in the performance of Chord and VHash for 10,000 routing samples on a 10,000 node underlay network for differently sized overlays. The Y axis shows the observed frequencies and the X axis shows the number of hops traversed on the underlay network. VHash consistently requires fewer hops for routing than Chord.

The Sybil Attack

- The

The goal of 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 I 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 keys (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.

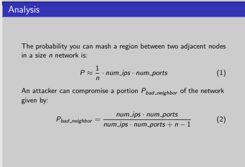
Analysis

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:

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



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:

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

- Have proofs!

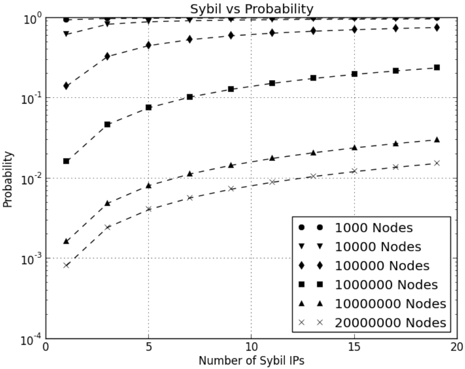


Figure : Our simulation results. The x-axis corresponds to the number of IP addresses the adversary can bring to bear. The y-axis is the probability that a random region between two adjacent normal members of the network can be mashed. Each line maps to a different network size of n. The dotted line traces the line corresponding to the Equation 2: $P_{bad\_neighbor} = \frac{num\_ips \cdot 16383}{num\_ips \cdot 16383 + n - 1}$

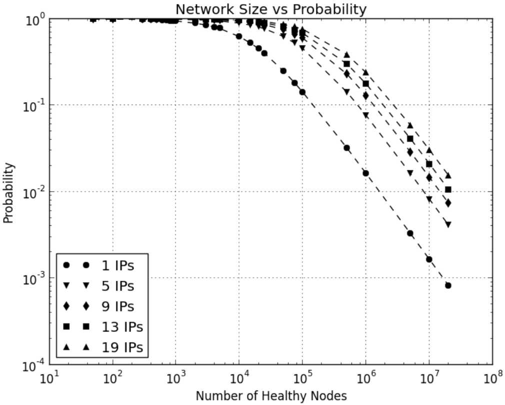


Figure : These are the same as results shown in Figure 14, but our *x*-axis is the network size *n* in this case. Here, each line corresponds to a different number of unique IP addresses the adversary has at their disposal.

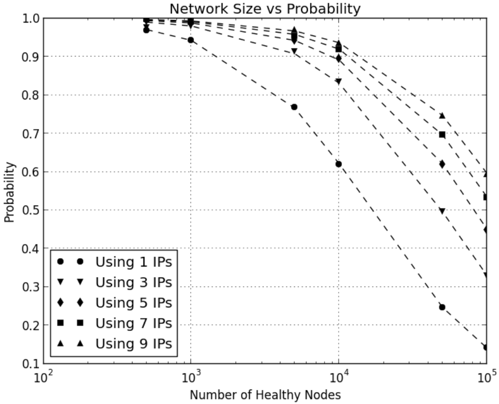


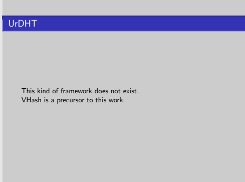
Figure : This graph shows the relationship between the network size and the probability a particular link, adjacent or not, can be mashed.

Table of Contents

1. Introduction
   - What I am doing
   - Distributed Computing and Challenges
   - What Are Distributed Hash Tables?
   - Why DHTs and Distributed Computing
2. Background
   - The Components and Terminology
   - Example DHT: Chord
3. Completed Work
   - ChordReduce
   - VHash
   - Sybil Attack Analysis
     - Results
4. Proposed Work
   - UrDHT
   - DHT Distributed Computing
   - Autonomous Load-Balancing

UrDHT

This kind of framework does not exist.
VHash is a precursor to this work.



UrDHT

This kind of framework does not exist.
VHash is a precursor to this work.

At Brendan's suggestion We plan on extending the functionality of DHTs, post poll

DHT Distributed Computing

content

Autonomous Load-Balancing

content

Virtual hadoop.
http://wiki.apache.org/hadoop/Virtual

Bram Cohen.
Incentives build robustness in bittorrent.
In *Workshop on Economics of Peer-to-Peer systems*, volume 6, pages 68–72, 2003.

Frank Dabek, M Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica.
Wide-Area Cooperative Storage with CFS.
ACM SIGOPS Operating Systems Review, 35(5):202–215, 2001.

Mu Li, Li Zhou, Zichao Yang, Aaron Li, Fei Xia, David G Andersen, and Alexander Smola.
Parameter server for distributed machine learning.

Andrew Loewenstern and Arvid Norberg.
BEP 5: DHT Protocol.

http://www.bittorrent.org/beps/bep_0005.html, March 2013.

- Gabriel Mateescu, Wolfgang Gentzsch, and Calvin J. Ribbens. Hybrid computing—where {HPC} meets grid and cloud computing. *Future Generation Computer Systems*, 27(5):440 – 453, 2011.

- Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. *SIGCOMM Comput. Commun. Rev.*, 31:149–160, August 2001.