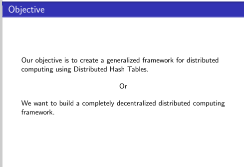
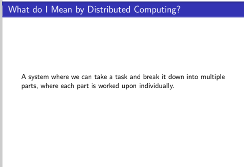
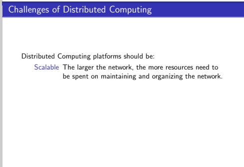


- We want to build a completely decentralized distributed computing framework based on distributed hash tables, or DHTs.
- Doing this will require a generic framework for creating distributed hash tables and distributed applications



A distributed computing framework is a system where we can take a job, break in up into smaller pieces, and send these pieces out to be worked upon by computing agents.



Remember, computers aren't telepathic. There's always an overhead cost. It will grow. The challenge of scalability is designing a protocol that grows in which this cost grows 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 failure.

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 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.

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 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.

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
- Fault-Tolerant
- Load-Balancing

- Remember to mention Napster.
- Distributed Hash Tables were designed to be used for completely decentralized P2P applications involving millions of nodes.
- Scalability
  - Each node knows a *small* subset of the entire network.
  - Join/leave operations impact very few nodes.
  - The subset each node knows is such that we have expected $\lg(n)$ lookup

Strengths of DHTs

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

- Scalable
- Fault-Tolerant
- Load-Balancing

- Fault-Tolerance
  - The network is decentralized.
  - DHTs are designed to handle churn.
  - Because Joins and node failures affect only nodes in the immediate vicinity, very few nodes are impacted by an individual operation.
- Load Balancing
  - Consistent hashing ensures that nodes and data are close to evenly distributed.
  - Nodes are responsible for the data closest to it.
  - The space is large enough to avoid Hash collisions

Uses For DHT Distributed Computing

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

- All MapReduce problems are embarrassingly parallel by definition.
- Individual experiments for genetic algorithms are embarrassingly parallel
- Monte-Carlo Markov Chain: sample a probability distribution in order to build a markov chain with desired distribution.

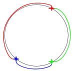
Required Attributes of DHT

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

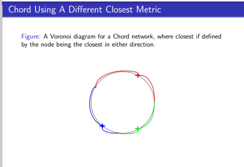
- There needs to be a way to establish how far things are from one another. Once we have a distance metric, we define what we mean when we say a node is responsible for all data *close* to it.
- The closeness metric establishes how a node decides what it is responsible for.
- The peer management strategy encompasses a whole lot: the network topology, the distribution of long links (are they organized and spread out over specified intervals, are they chosen according to a random distribution?), and the network maintenance.

Chord's Closest Metric.

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



Here, the nodes, the stars on this diagram are responsible for the region covered by the arc matching its color. This repesents the space containing all keys greater than the ID of it's predecessor and less than or equal to its own ID.

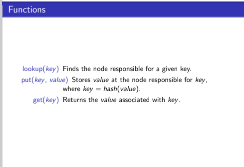


In this figure, the ownership has the more intuitive definition of “all points with to which I am the node with the shortest distance.”

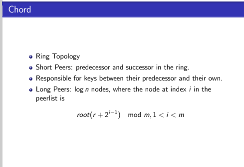
Terms and Variables

- 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, but this is trivial from our perspective.
- Short peers are actively maintained, long peers replaced gradually and are not actively pinged.
- We use root as it's is a topology agnostic term.



- SPEED THRU THIS SLIDE
- These are the common function is every DHT: lookup: find a node, put: store data, get: retrieve data.
- There is usually a delete function as well, but it's not important.
- All work the same way: if I can answer the function, great, otherwise return the node I know that's closest to the key, who will then do the same thing.



- 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

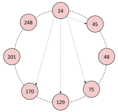


Figure: An 8-node Chord ring where $m = 8$. Node 24's long peers are shown.

- The dashed lines are the short links; each node keeps track of its successor and predecessor.
- The dotted lines are node 24's long links; since $m = 8$ there's 8, but since the network is so small, 4 are duplicates.
- Traffic travels clockwise.

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.

- The notification allows a sucessor to either update its predecessor if incorrect and the predecessor to update its successor if wrong
- Some implementations use predecessor and successor lists.
- The long peers are replaced much slower; maintenance slowly iterates through the long peers and queries to see if there's better node for that particular long peer.

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.

- Short peers need to be actively maintained to keep the topology correct.
-

My research has been focused on:

- Abstracting out DHTs.
- Distributed computation using DHTs.

- I want to get down to what the essence of a DHT is, find out what all DHTs have in common, so that I could create a generic DHT.
- I focused on creating a more abstract framework for MapReduce, so I could move it out of the datacenter and into other contexts.
- We'll first discuss generalizing DHTs.

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.

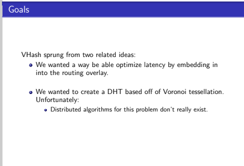
Most DHTs optimize routing for the number of hops, rather than latency.

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:

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



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.

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

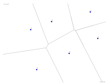


Figure: A set of points and the generated Voronoi regions

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

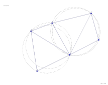


Figure: The same set of nodes with their corresponding Delaunay Triangulation.

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.

It turns out we can look at distributed hash tables in terms of Voronoi tessellation and Delaunay triangulation. So if we have a quick way approximate this, we can build a DHT based directly on Voronoi tessellation and Delaunay triangulation.

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

- Gossip: every cycle, a node chooses a peer and swaps peerlist information, then reruns the approximation.
- d is number of dimensions, but we optimize over latency so that's deceptive
- 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.

DGVH Heuristic

```
Given node n and its list of candidates.
peers ← 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 ← 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
```

DVGH is very efficient in terms of both space and time. Suppose a node n is creating its short peer list from k candidates in an overlay network of N nodes. The candidates must be sorted, which takes $O(k \cdot \lg(k))$ operations. Node n must then compute the midpoint between itself and each of the k candidates. Node n then compares distances to the midpoints between itself and all the candidates. This results in a cost of

$$k \cdot \lg(k) + k \text{ midpoints} + k^2 \text{ distances}$$

Since k is bounded by $\Theta(\frac{\log N}{\log \log N})$ (the expected maximum degree of a node), we can translate the above to

$$O(\frac{\log^2 N}{\log^2 \log N})$$

In the vast majority of cases, the number of peers is equal to the minimum size of *Short Peers*. This yields $k = (3d+1)^2 + 3d + 1$ in the expected case, where the lower bound and expected complexities are

Results

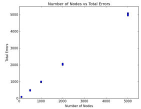


Figure: As the size of the graph increases, we see approximately 1 error per node.

- The error is approximately 1 missed delaunay neighbor per node, so the entire generated mesh is a subset of the true delaunay trinagulation.
- This error is acceptable since we miss an edge when it is occluded by another node in the way.

Results

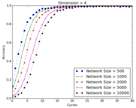


Figure: These figures show, starting from a randomized network, VHash forms a stable and consistent network topology.

- Our first experiment was to confirm that VHash creates a stable mesh.

Results

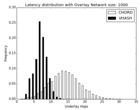
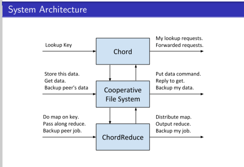


Figure: Comparing the routing effectiveness of Chord and VHash.

- For this experiment, we constructed a scale free network with 10000 nodes placed at random (which has an approximate diameter of 3 hops) as an underlay network .
- scale free networks model the Internet's topology.
- We then chose a random subset of nodes to be members of the overlay network.
- We then measured the distance in underlay hops between 10000 random source-destination pairs in the overlay.
- VHash generated an embedding of the latency graph utilizing a distributed force directed model, with the latency function defined as the number of underlay hops between it and its peers.
- 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

- 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.

- DGVH is of similar complexity to picking k-nearest node or nodes in distance k.
- other methods don't guarantee full connectivity
- It caps out at $O(n^2)$ complexity, no matter how many dimensions or complexities of the metric space (unless calculating distance or midpoint is worse than $O(1)$)
- for example This means you can use in it an 100-dimensional euclidean space in $O(n^2)$ time rather than $O(n^{50})$ time (maybe we should have opened with this...)



- Chord, which handles routing and lookup.
- The Cooperative File System (CFS), 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

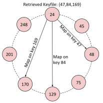


Figure: The stager sends a map task for each key in the keyfile. In larger networks, this process is streamlined by recursively bundling keys and sending them to the best finger.

- this process is recursive

Reducing Results of Data

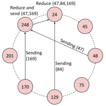


Figure: Results are sent back via the overlay. If a node receives multiple results, they are reduced before being sent on.

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

Experimental Results

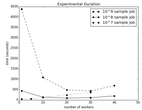


Figure: For a sufficiently large job, it was almost always preferable to distribute it.

- 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.
- Diminishing returns.

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.

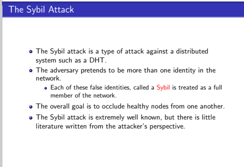
Our experiments established:

- 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.

One of the goals coming out of this is that I want to be able to harness this speedup due to churn. I've come up with a number of potential strategies I've listed in the proposal, but a number of them involve nodes being able to create virtual nodes, in other words, be in multiples places at once and have multiple identities.

It turns out the security world has something analogous to that.



- DHTs don't have a centralized point of failure, but this opens up different strategies for attack
- How the identities are obtained is a question for later, but we assume they don't have to be real hardware.
- By occlude, we me that that traffic must travel thru the adversary.
- What distinguishes the Sybil from the Eclipse attack is the fact that the Sybil attack relies only on false identity

2015-07-13

Analysis

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

$$P \approx \frac{1}{n} \cdot num\_ips \cdot num\_ports \qquad (1)$$

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

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

- Have proofs!

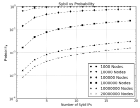
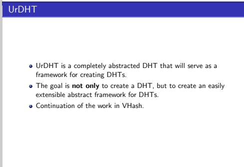


Figure: Our simulation results.
The dotted line traces the line corresponding to the Equation 2:
$P_{bad\_neighbor} = \frac{num\_ips \cdot 16383}{num\_ips \cdot 16383 + n - 1}$

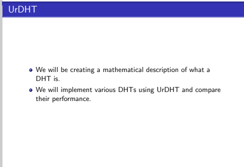
The x-axis corresponds to the number of IP addresses the adversary can bring to bear. The y-axis is the probability that a random region between two adjacent normal members of the network can be mashed. Each line maps to a different network size of n.

- 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.

- By cost, that is the Amazon EC2 cost and assumed half the links of 20,000,000 nodes we occluded.



- *Ur* as in the germanic prefix for proto or first
- UrDHT is a project that presents a minimal and extensible implementation for all the essential components for a DHT: the different aspects for a protocol definition, the storage of values, and the networking components. Every DHT has the same components, but there has yet to be an all-encompassing framework that clearly demonstrates this.



- The purpose of doing this is that
- it's cool
- We want a framework for creating DHTs and DHT based applications to be easily available
- I need it to do the rest of my proposal and Brendan needs it for the same reason.

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.

- Additionally this will serve as an example of how to implement our framework.