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.

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.

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?

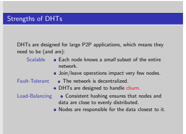
We'll explain in greater detail later, but briefly:

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

2015-07-10

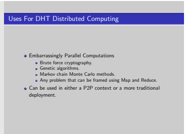
DHT Distributed Computing
└─Introduction
  └─What Are Distributed Hash Tables?
    └─Strengths of DHTs



- 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

2015-07-10

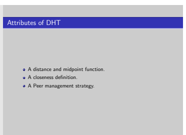
DHT Distributed Computing
└─Introduction
  └─Why DHTs and Distributed Computing
    └─Uses For DHT Distributed Computing



- Need notes here
- Define Monte-Carlo Markov Chain

2015-07-10

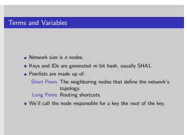
DHT Distributed Computing
└─Background
  └─The Components and Terminology
    └─Attributes of DHT



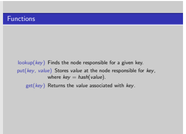
- There needs to be a way to establish how far things are from one another. Once we have a distance metric, we define what we mean when we say a node is responsible for all data *close* to it.
- We'll go into more detail about the difference between Distance and Closeness in Chord

2015-07-10

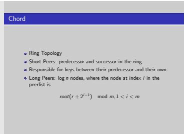
DHT Distributed Computing
└─Background
  └─The Components and Terminology
    └─Terms and Variables



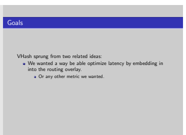
- SHA1 is being depreciated.
- Short peers are actively maintained, long peers replaced gradulally and are not actively pinged.
- We use root as it's is a topology agnostic term.



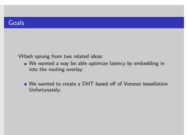
- There is usually a delete function as well, but it's not important.
- All nodes use the same general lookup: Forward the message to the node closest to *key*



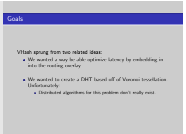
- Chord is a favorite because we can draw it.
- Draw a Chord network on the wall?
- node r is our root node.
- i is the index on the list
- English for the equation, the long peers double in distance from the root node, allowing us to cover at least half the distance to our target in a step
- In this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



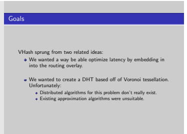
Most DHTs optimize routing for the number of hops, rather than latency.



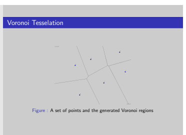
We discovered a mapping between Distributed Hash Tables and Voronoi/Delaunay Triangulations.



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.

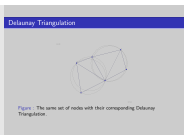


Simple approximations have no guarantees of connectivity, which is very bad for a routing topology. Better algorithms that existed for this problem technically ran in constant time, but had a prohibitively high sampling. So to understand what I'm talking about here, let's briefly define what a Voronoi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



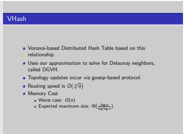
Define

- 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

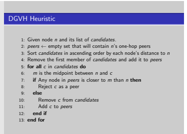


Define

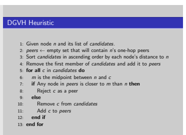
- Delaunay Triangulation is a triangulation of a set of points with the following rule:
- No point falls within any of the circumcirles for every triangle in the triangulation,
- The Voronoi tessellation and Delaunay Triangulation are dual problems
  - Solving one yields the other.
  - We can get the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



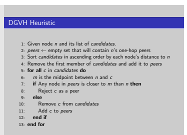
- d is number of dimensions, but we optimize over latency so that's deceptive
- Gossip: every cycle, a node chooses a peer and swaps peerlist information, then reruns the approximation.
- This would only happen in a contrived case and would give $O(1)$ routing
- We found a paper that proved $\Theta(\frac{\log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



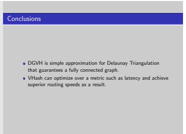
- The majority of Delaunay links cross the corresponding Voronoi edges.
- We can test if the midpoint between two potentially connecting nodes is on the edge of the voronoi region.
- This intuition fails if the midpoint between two nodes does not fall on their voronoi edge.



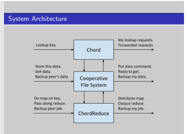
1. 'n' is the "myself" node, and the location we are seeking to find the peers of.
2. peers is a set that will build the peerlist in
3. We sort the candidates from closest to farthest.
4. The closest candidate is always guaranteed to be a peer.
5. Iterate through the sorted list of candidates and either 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



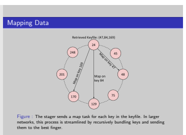
Theoretically, this is worst case $O(n^2)$.
However in practice, this is $O(nlog(n)(sorting) + kn)$ where$k$ is the number of Delaunay peers.
We are well aware that 2d-euclidean algorithms exist in O(nlog(n)) time.
While we use that use case to communicate the algorithm, it is intended to be used in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



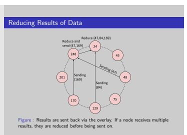
- DGVH is of similar complexity to picking k-nearest node or nodes in distance k.
- other methods don't guarantee full connectivity
- It caps out at $O(n^2)$ complexity, no matter how many dimensions or complexities of the metric space (unless calculating distance or midpoint is worse than $O(1)$)
- 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...)



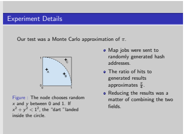
- Chord [7], which handles routing and lookup.
- The Cooperative File System (CFS) [3], which handles storage and data replication.
- The MapReduce layer.
- Files are split up, each block given a key based on their contents.
- Each block is stored according to their key.
- The hashing process guarantees 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



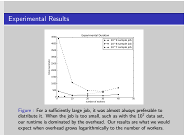
- this process is recursive



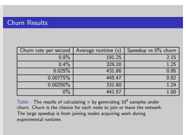
The paths here are arbitrary edges that I came up with for the example.



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



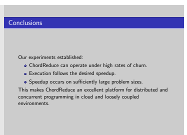
- Diminishing returns.
- The 10^7 job was dominated by the overhead communication costs.



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

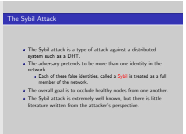
Table : The results of calculating π by generating 10^8 samples under churn. Churn is the chance for each node to join or leave the network. The large speedup is from joining nodes acquiring work during experimental runtime.

- 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.

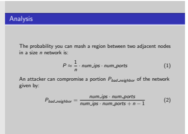


One of the goals coming out of this is that I want to be able to harness this speedup due to churn. I've come up with a number of potential strategies I've listed in the proposal, but a number of them involve nodes being able to create virtual nodes, in other words, be in multiples places at once and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

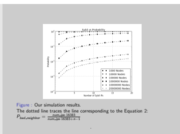
It turns out the security world has something analogous to that.



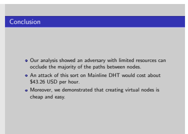
- DHTs don't have a centralized point of failure, but this opens up different strategies for attack
- How the identities are obtained is a question for later, but we assume they don't have to be real hardware.
- By occlude, we me that that traffic must travel thru the adversary.
- What distinguishes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



- Have proofs!



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.



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

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.

- *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.

UrDHT

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

- 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.

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.