Dissertation
Towards a Framework for DHT Distributed Computing

Andrew Rosen

Georgia State University

May 13th, 2016

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.

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.

How Does It Work?

- 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 small 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.

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.

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.

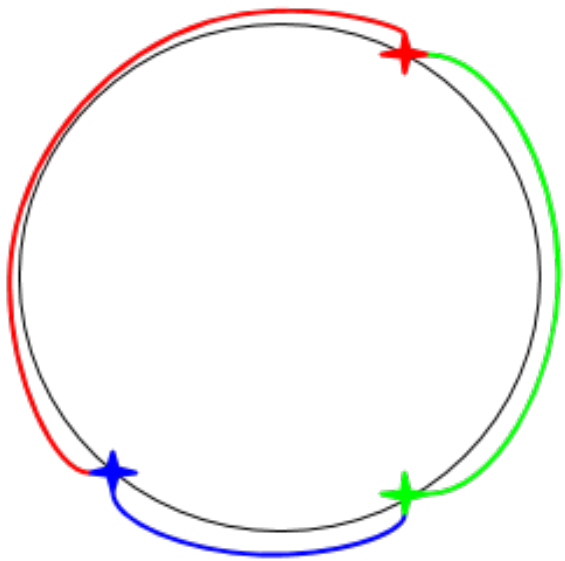
Table of Contents

Required Attributes of DHT

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

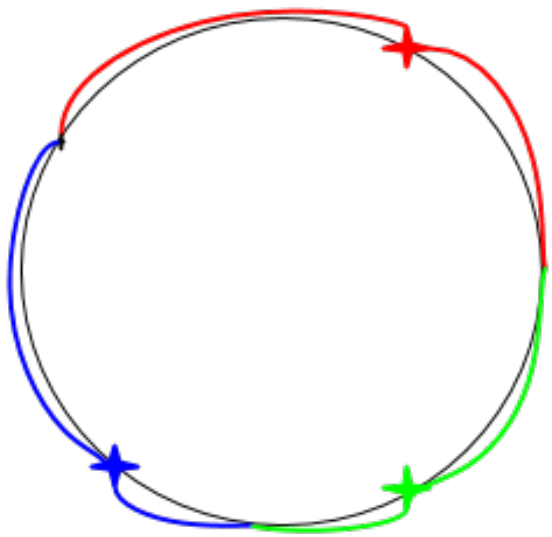
Chord's Closest Metric.

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.

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

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

A Chord Network

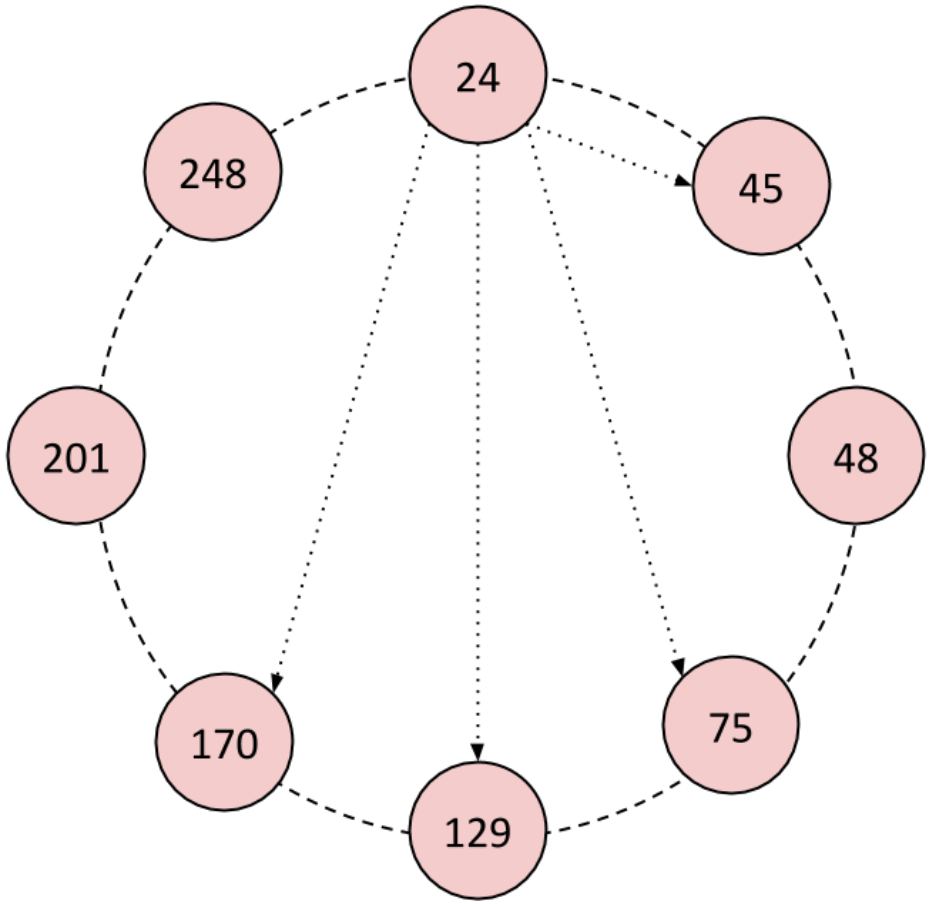


Figure: An 8-node Chord ring where $m = 8$. Node 24's long peers are shown.

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 regularly queried to:
  - See of the node is still alive.
  - See if the neighbor knows about better nodes.
- Long peer failures are replaced by periodic 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:

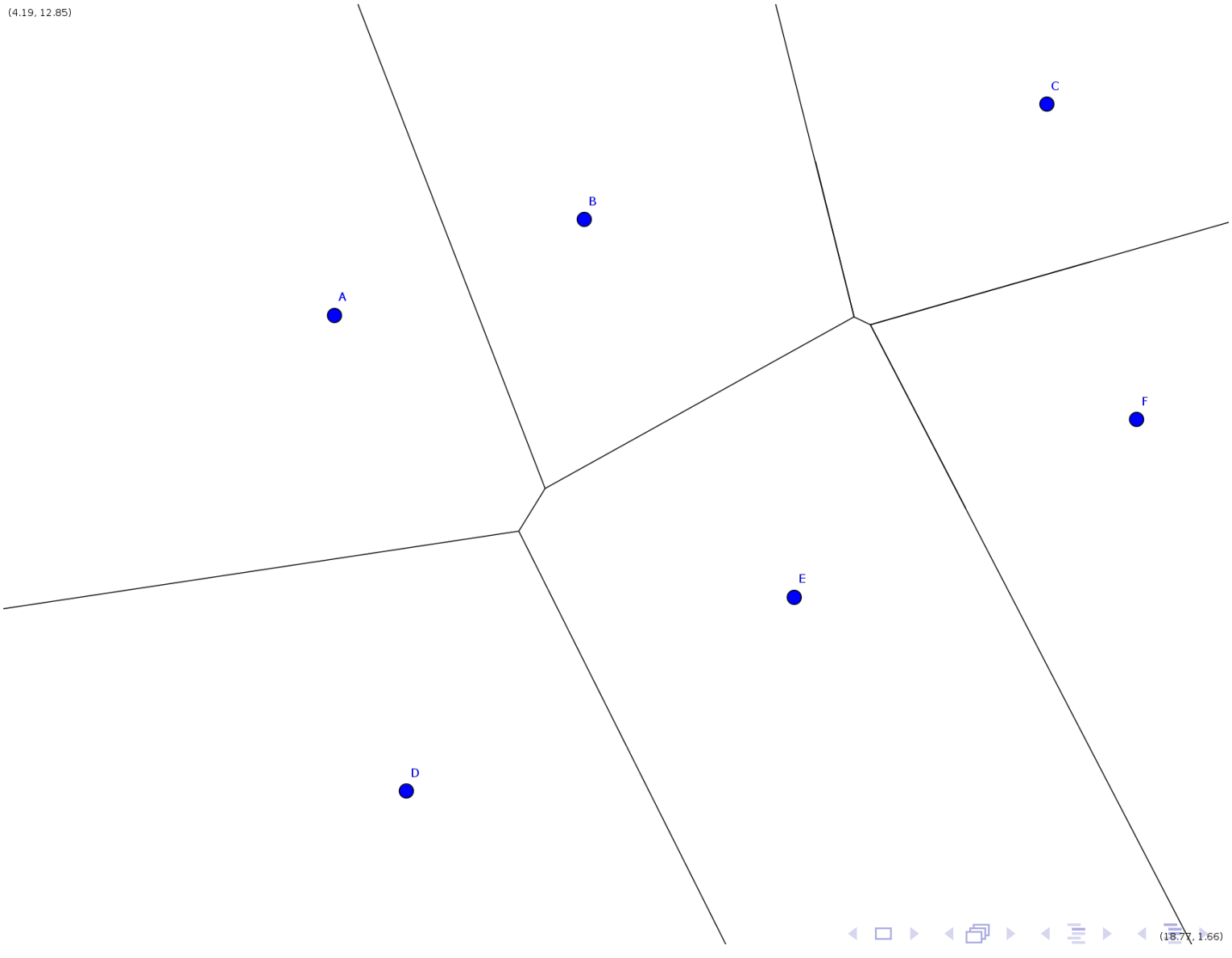
- We wanted a way be able optimize latency by embedding it into the routing overlay.
- We wanted to create a DHT based off of Voronoi tessellations. Unfortunately:
  - Distributed algorithms for this problem don't really exist.

Goals

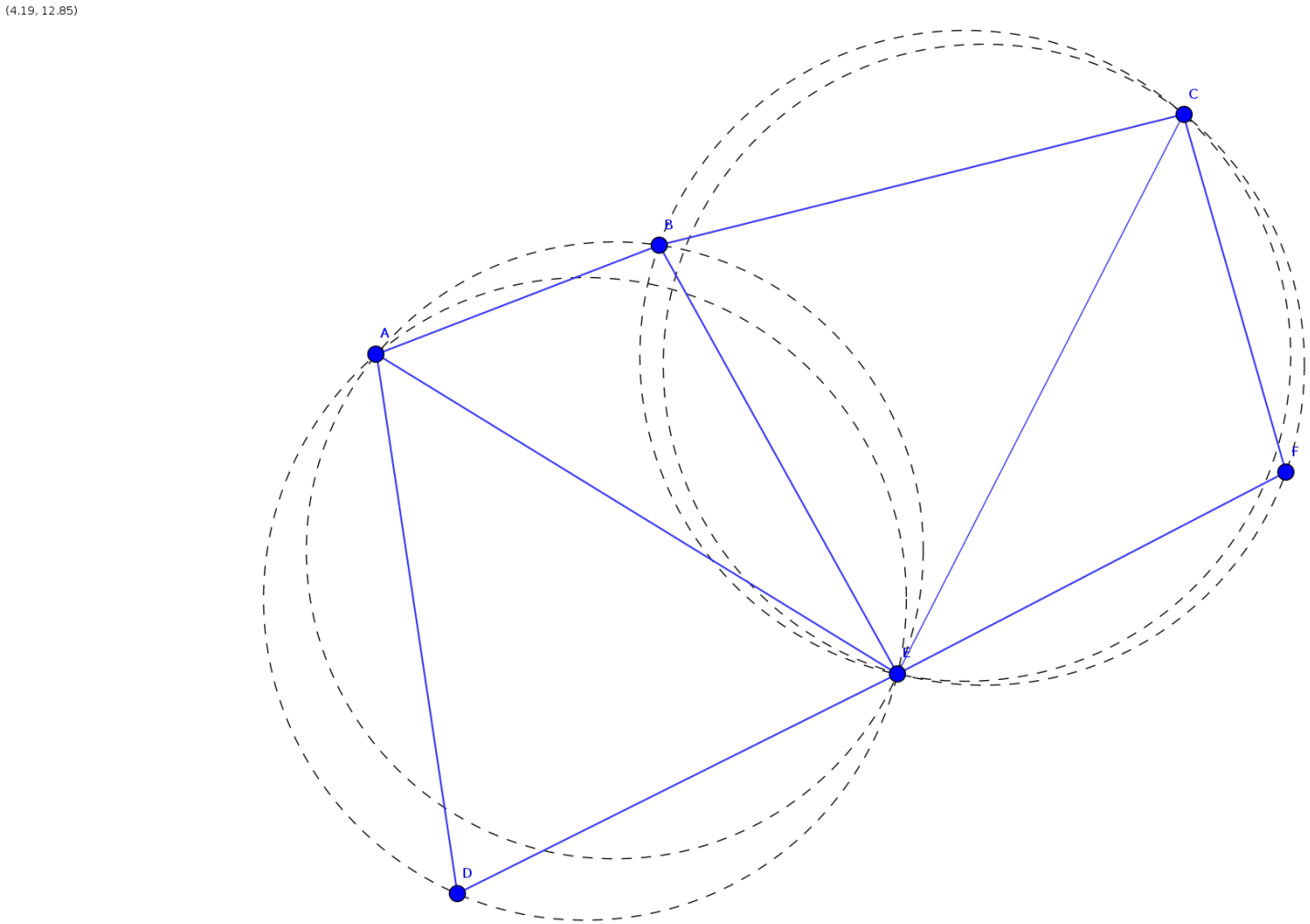
VHash sprung from two related ideas:

- 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



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 its 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

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

DGVH Time Complexity

For k candidates, the cost is:

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

However, the expected maximum for k is $\Theta(\frac{\log n}{\log \log n})$, which gives an expected maximum cost of

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

Results

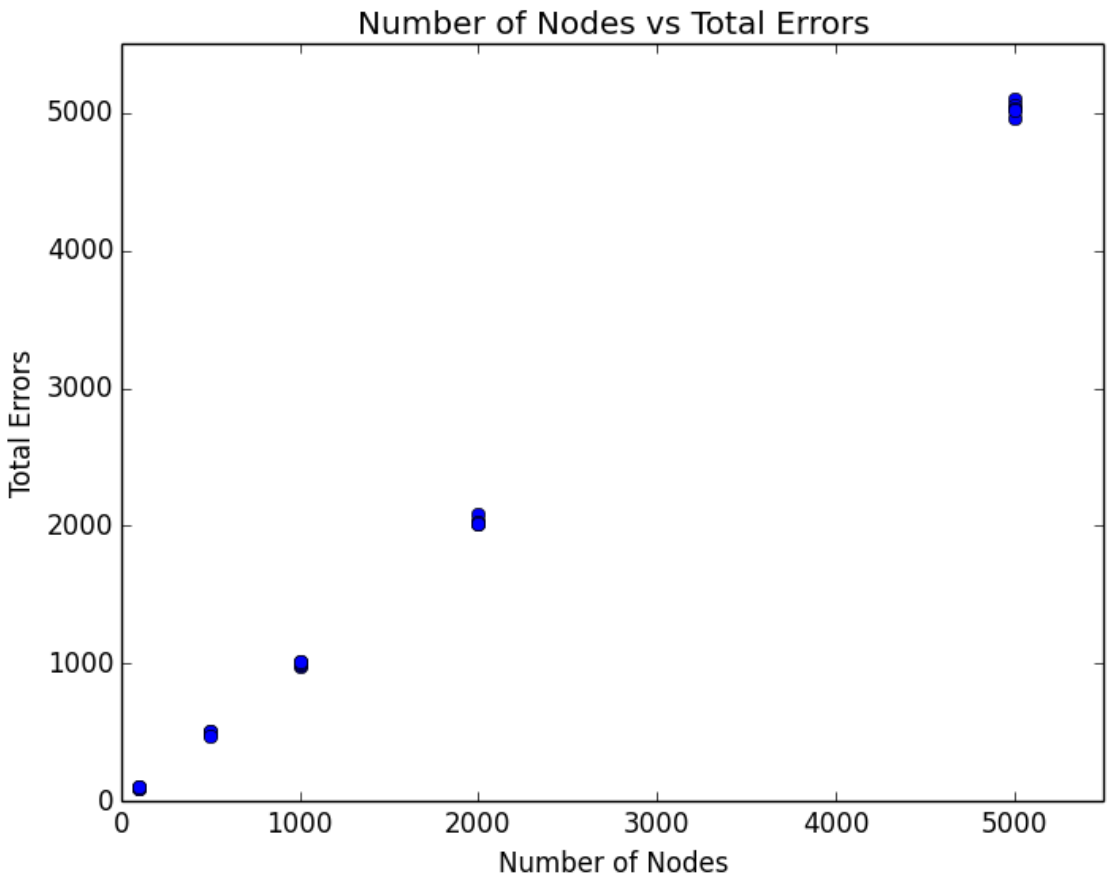


Figure: As the size of the graph increases, we see approximately 1 error per node.

Results

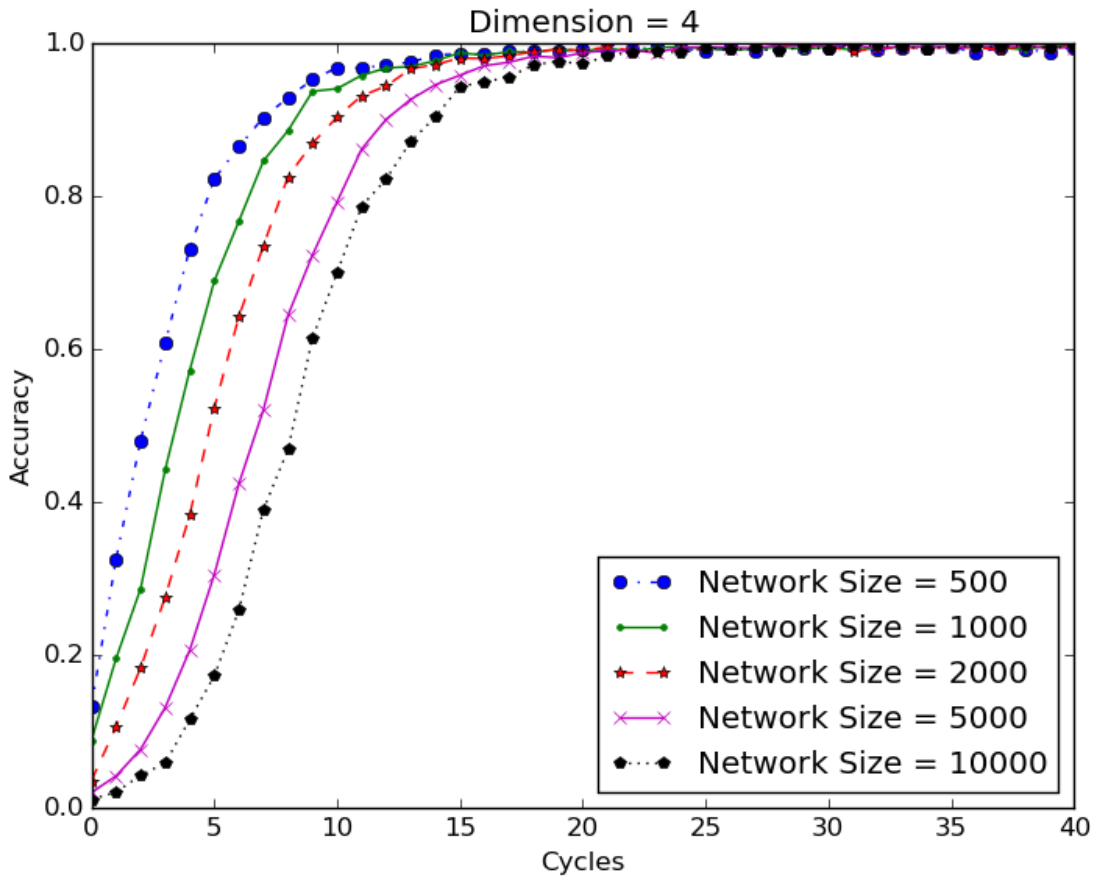


Figure: These figures show, starting from a randomized network, VHash forms a stable and consistent network topology.

Results

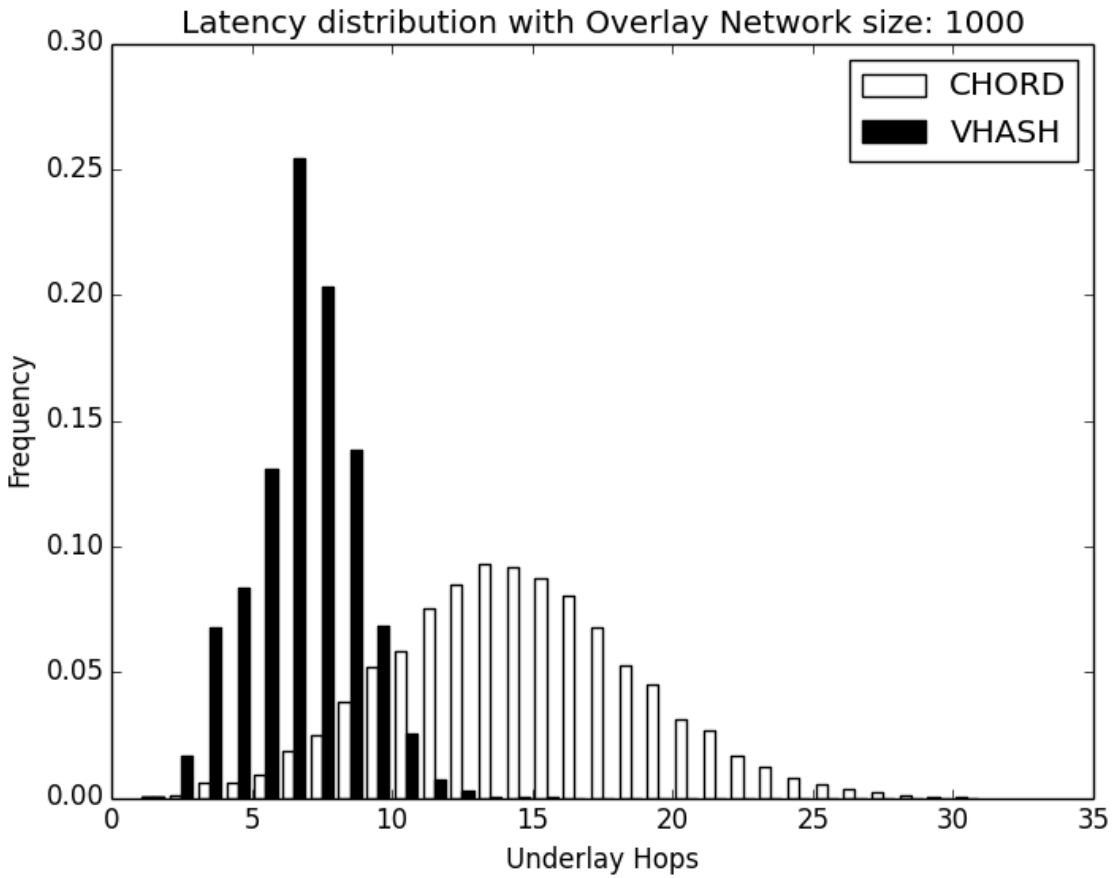


Figure: Comparing the routing effectiveness of Chord and VHash.

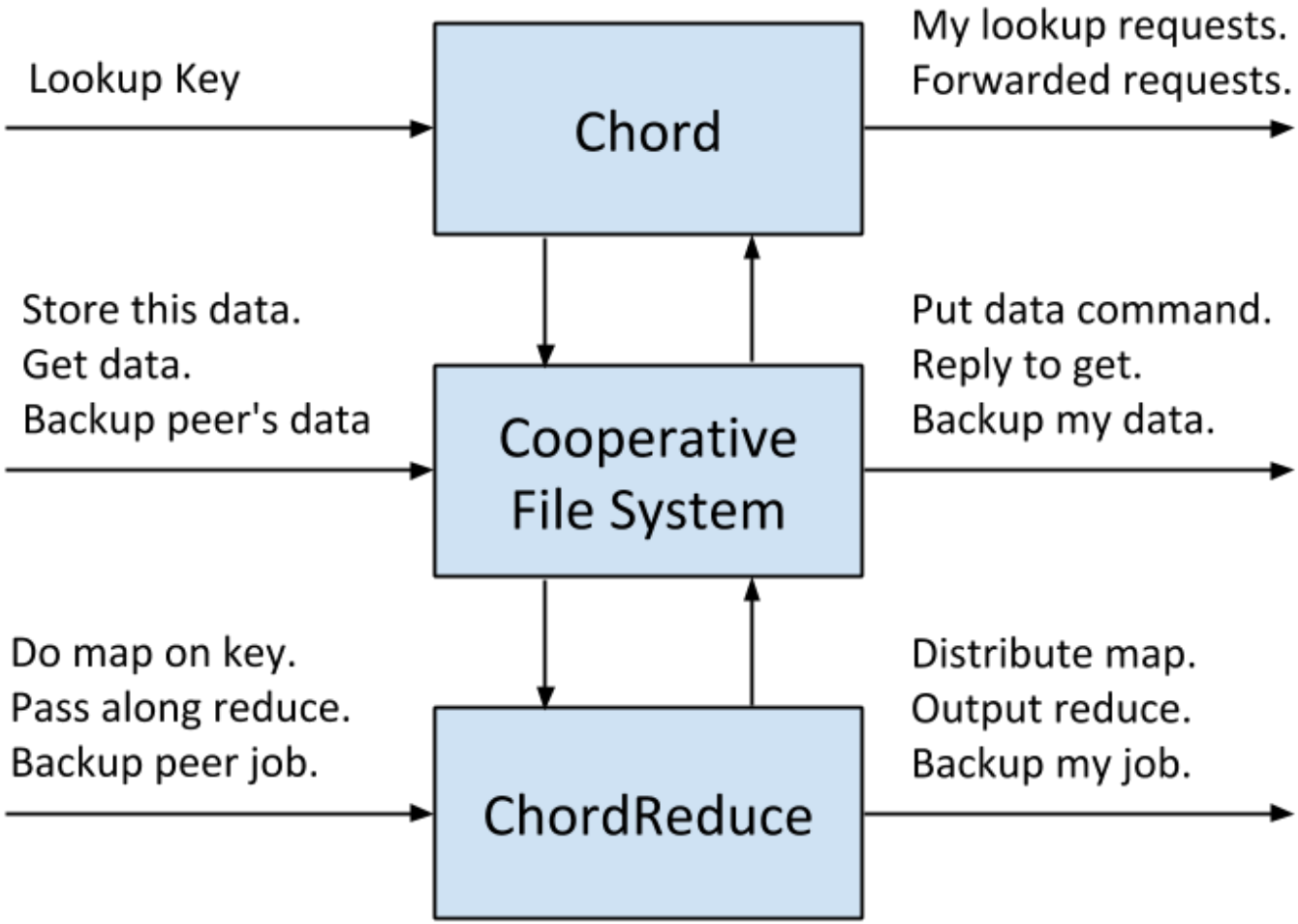
Conclusions

- 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



Mapping Data

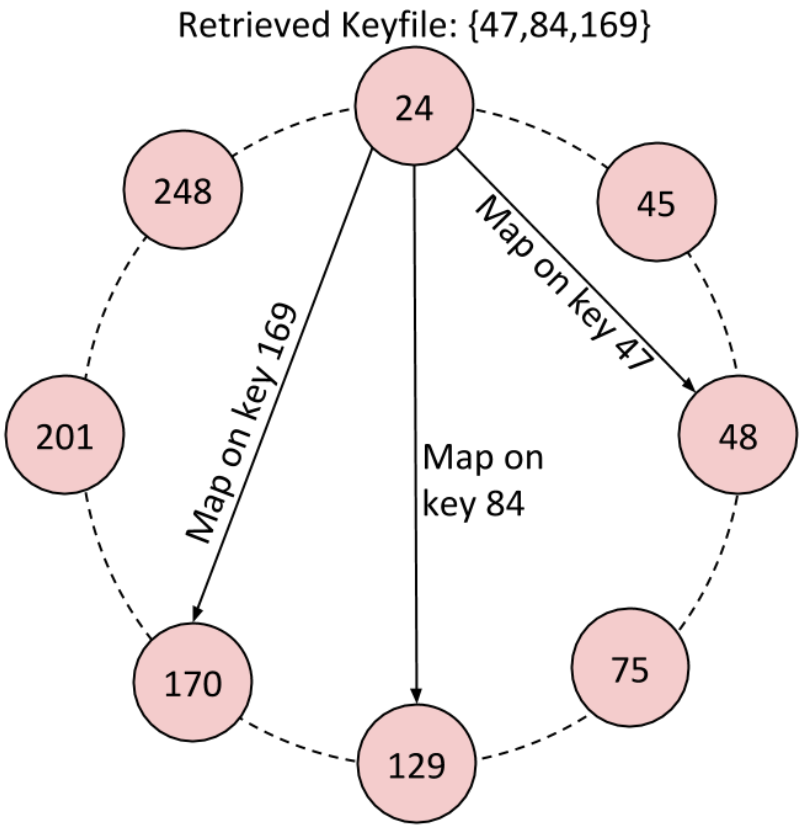


Figure: The stager sends a map task for each key in the keyfile. In larger networks, this process is streamlined by recursively bundling keys and sending them to the best finger.

Reducing Results of Data

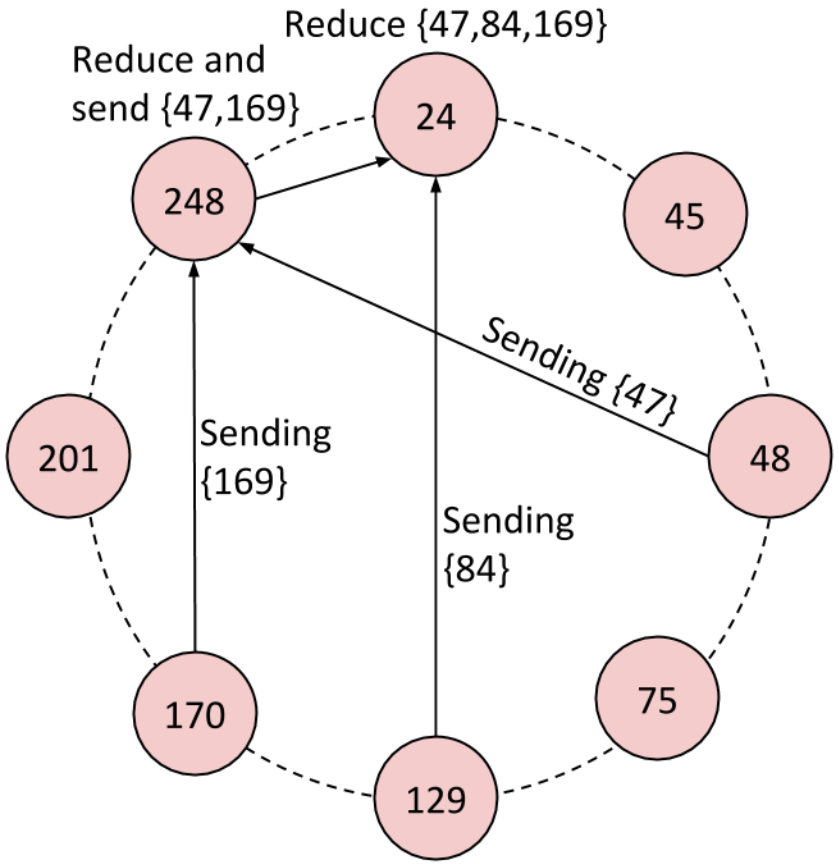


Figure: Results are sent back via the overlay. If a node receives multiple results, they are reduced before being sent on.

Experiment Details

Our test was a Monte Carlo approximation of π.

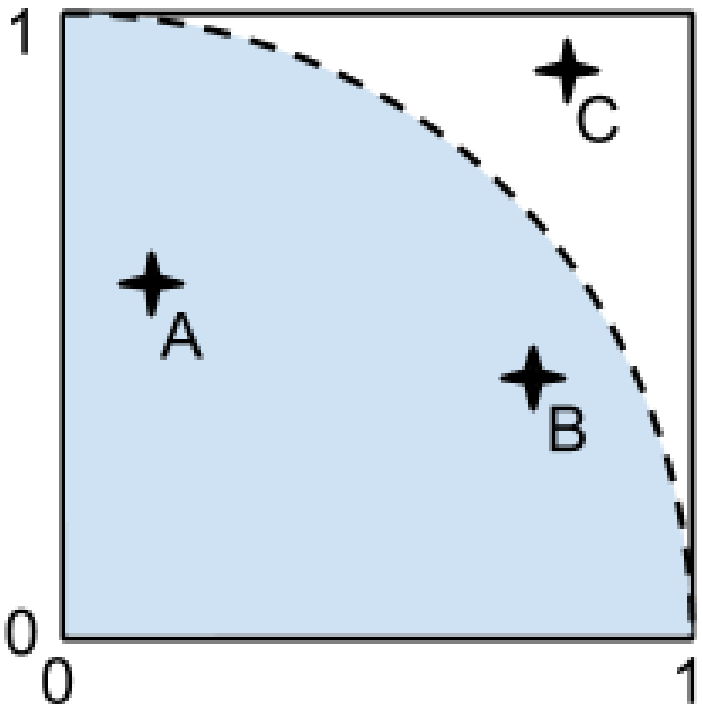


Figure: The node chooses random x and y between 0 and 1. If

- 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

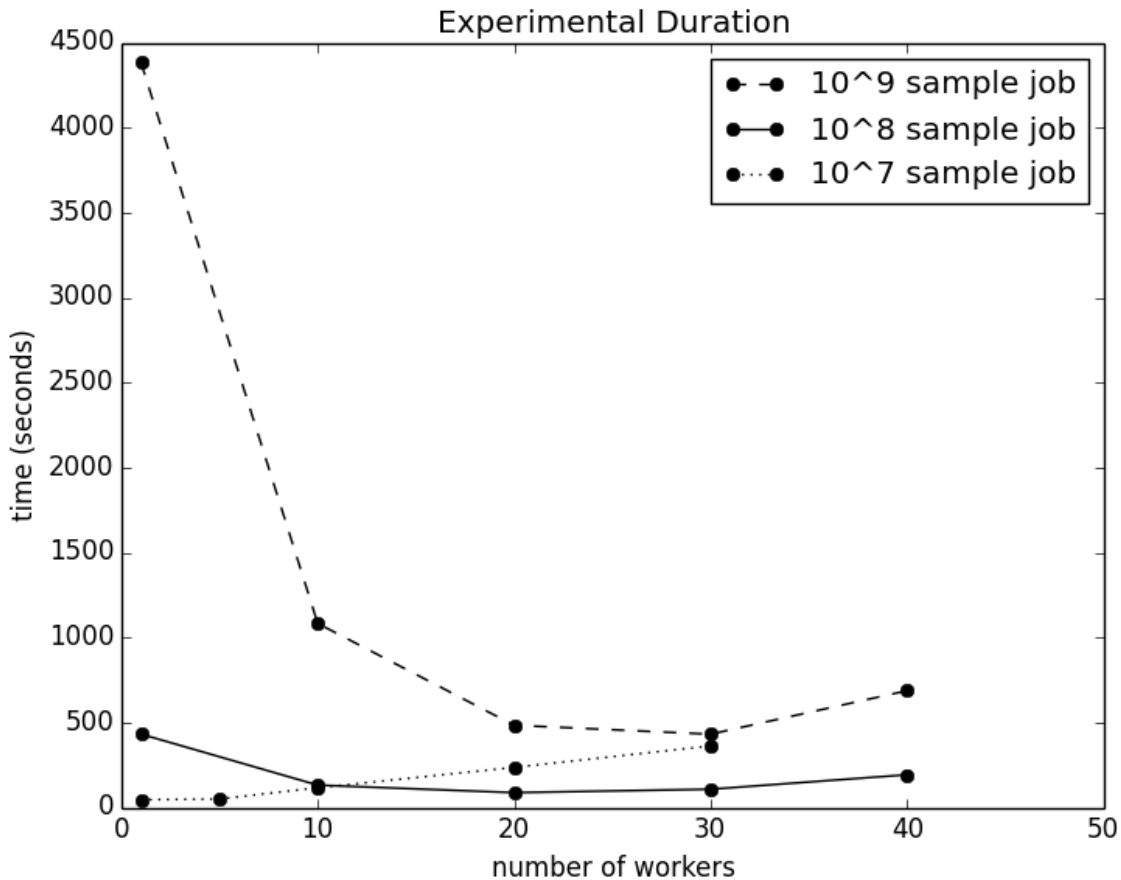


Figure: For a sufficiently large job, it was almost always preferable to distribute it.

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

- 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.

Analysis

The probability an attacker 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:

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

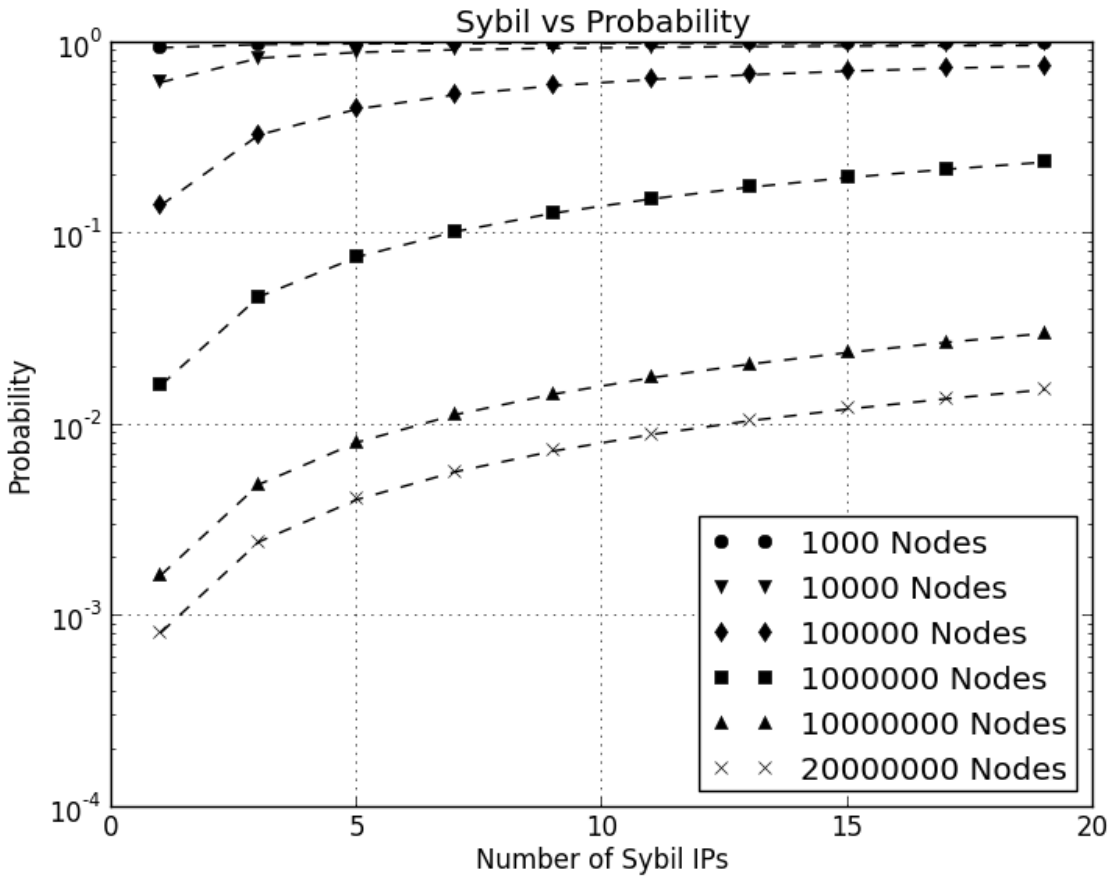


Figure: Our simulation results.

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

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

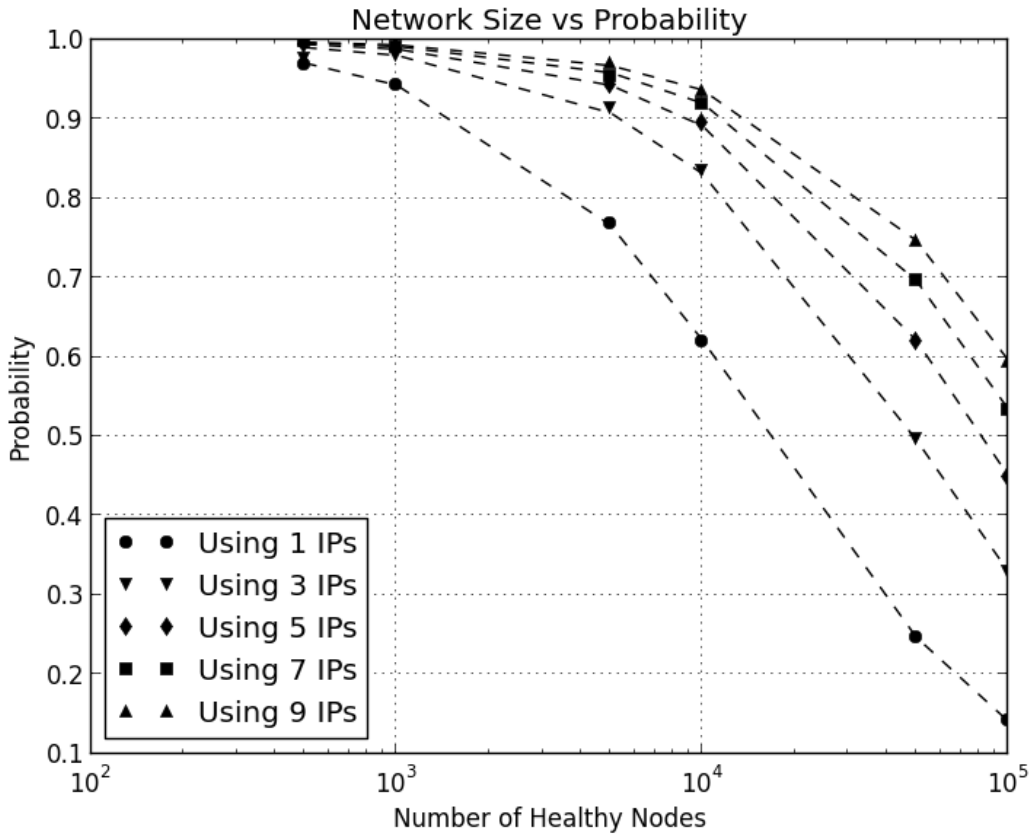


Figure: This graph shows the relationship between the network size and the probability a particular link, adjacent or not, can be mashed.

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

Table of Contents

Georgia State University

Introduction

- In this project, we set out to confirm the results of ChordReduce
- Objectives:
  - Confirm that high levels of churn can help a DHT based computing environment.
  - Develop better strategies than randomness

Distribution of Work in a DHT

Table: The median distribution of tasks (or files) among nodes. We can see the standard deviation is fairly close to the expected mean workload ($\frac{tasks}{nodes}$). Each row is the average of 100 trials. Experiments show there is practically little deviation in the median load of the network.

Nodes	Tasks	Median Workload	σ
1000	100000	69.410	137.27
1000	500000	346.570	499.169
1000	1000000	692.300	996.982
5000	100000	13.810	20.477
5000	500000	69.280	100.344
5000	1000000	138.360	200.564
10000	100000	7.000	10.492
10000	500000	34.550	50.366
10000	1000000	69.180	100.319

Distribution of Work in A DHT

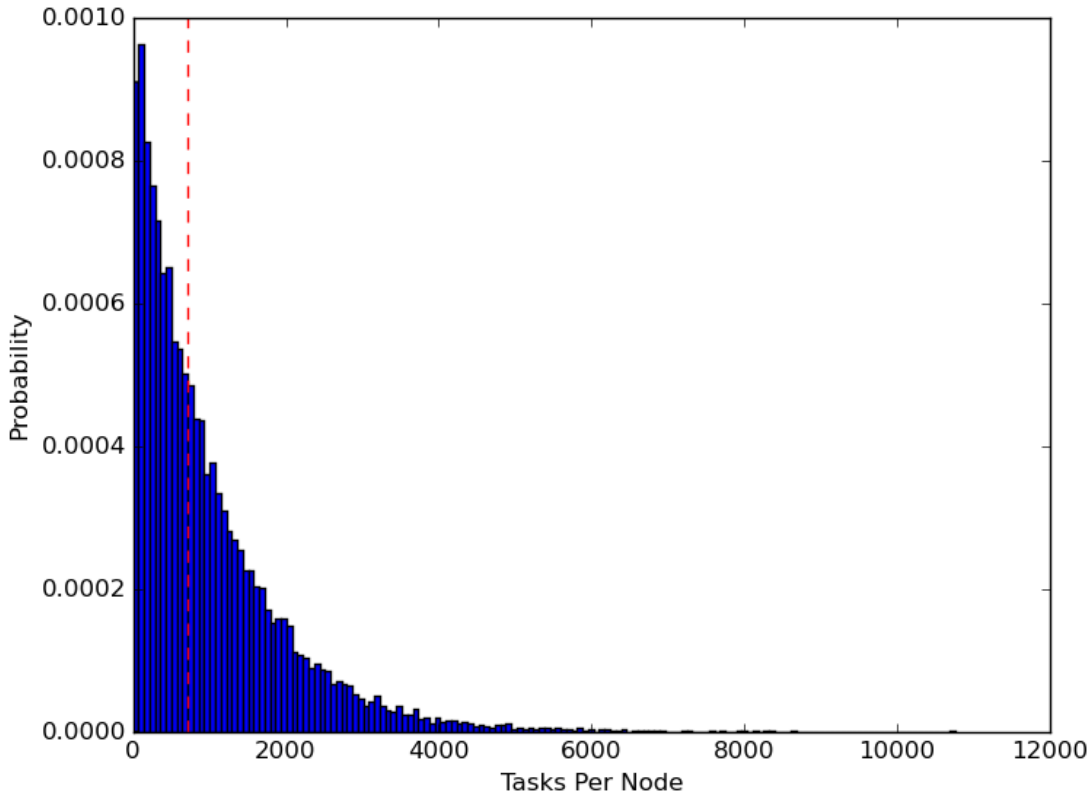


Figure: The probability distribution of workload in a DHT with 1000 nodes and 1,000,000 tasks or files. The vertical dashed line designates the median.

Distribution of Work in Chord

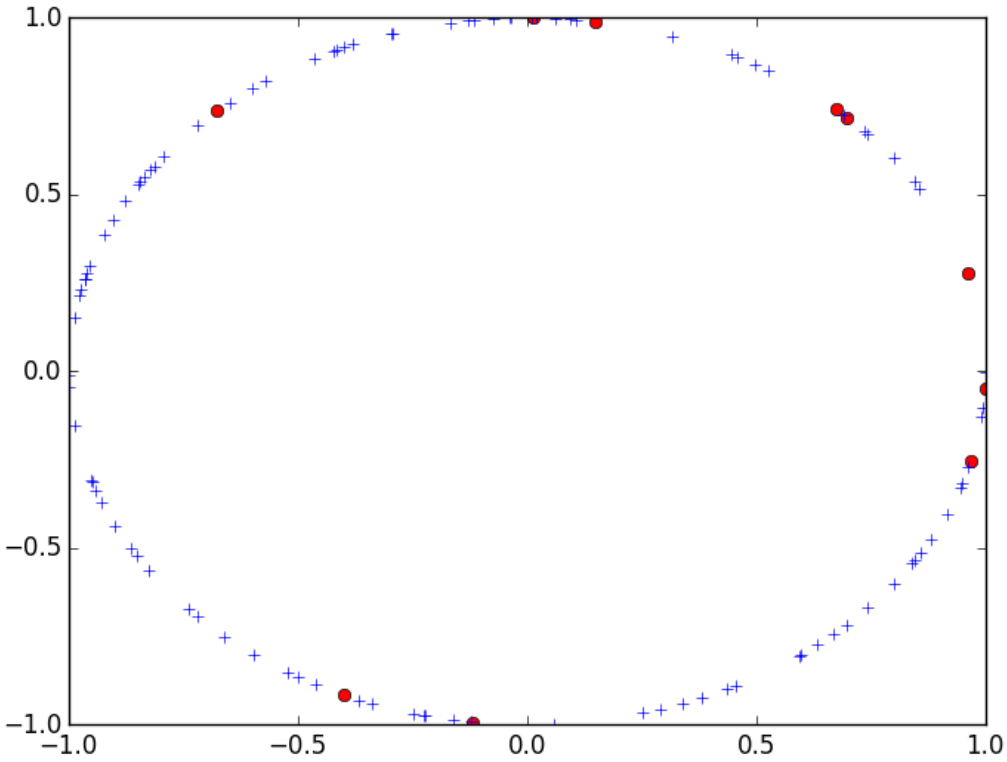


Figure: A visual example of data and nodes in a Chord DHT with 10 nodes (represented by red circles) and 100 tasks (blue pluses).

Terms and Assumptions

- Time is measured in ticks.
- A tick is enough time to perform aggressive, reactive maintenance[1]
- Jobs are measured in tasks; each task can correspond to a file or a piece of a file

[1] This has been implemented and tested.

Variables

- Strategy
- Homogeneity
- Work Measurement
- Number of Nodes
- Number of Tasks
- Churn Rate
- Max Sybils or Node Strength
- Sybil Threshold
- Number of Successors

Output

- Ideal Runtime
- Runtime
- Runtime Factor
- Task Distribution

Strategy

- The network load-balances using churn
- `churnRate` chance per tick for each node to leave network
- Pool of potentially joining joins at the same rate

Runtime

Table: Runtime factor of networks of varying sizes and number of tasks, each using the Churn strategy to load-balance. Each result is the average of 100 trials. The networks are homogeneous and each node consumes one task per tick. A runtime of 1 is the ideal and target.

Churn Rate	10^3 nodes, 10^5 tasks	10^3 nodes, 10^6 tasks	100 nodes, 10^4 tasks	100 nodes, 10^5 tasks	100 nodes, 10^6 tasks
0	7.476	7.467	5.043	5.022	5.016
0.0001	7.122	5.732	4.934	4.362	3.077
0.001	6.047	3.674	4.391	3.019	1.863
0.01	3.721	2.104	3.076	1.873	1.309

Churn vs Runtime factor

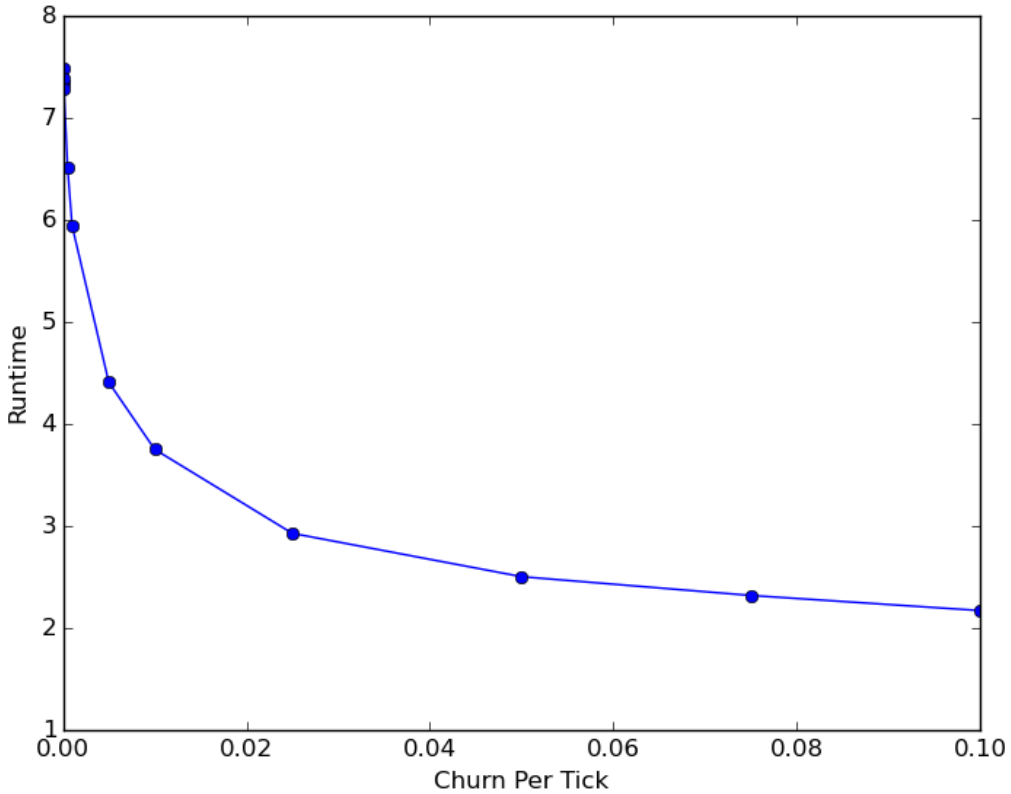


Figure: This graph shows the effect churn has on runtime in a distributed computation. Runtime is measured as how many times slower the computation runs than an ideal computation, where each node receives an equal number of tasks. The lower the runtime factor, the closer it is

Churn vs Average Work Per Tick

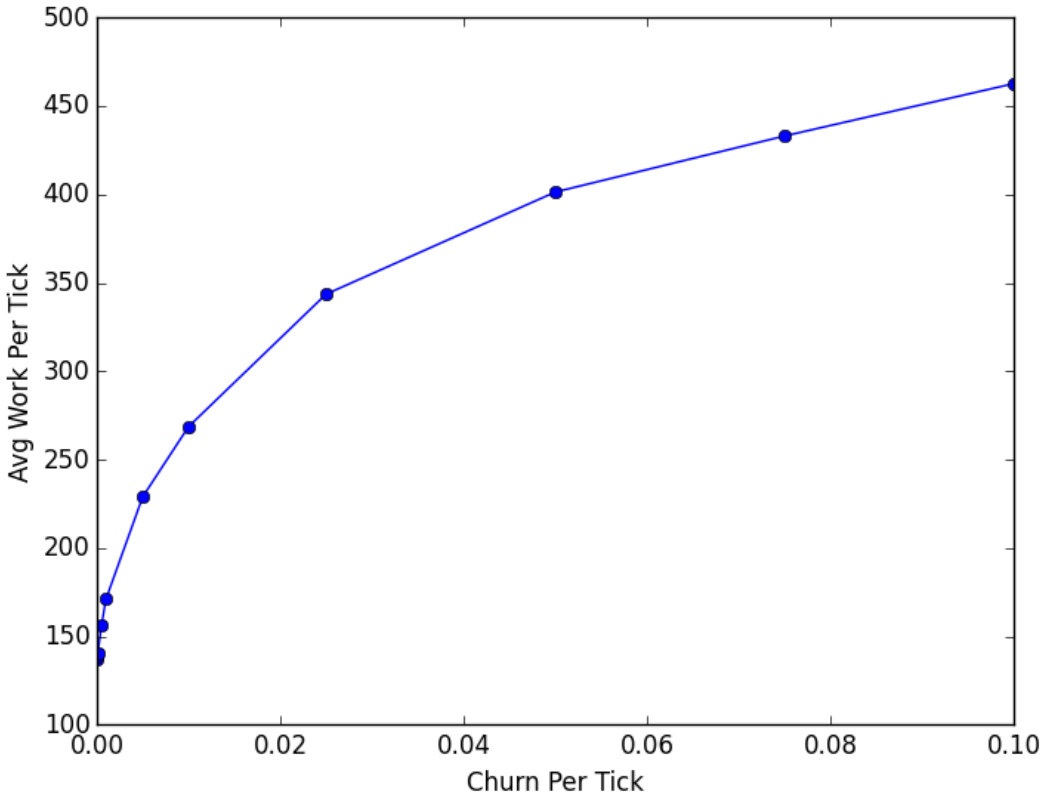


Figure: With more and more churn in the network, new nodes have a higher chance of joining the network and acquiring work from overloaded nodes. This results in more average work being done each tick, as there are less nodes simply idling.

Base Work Distribution

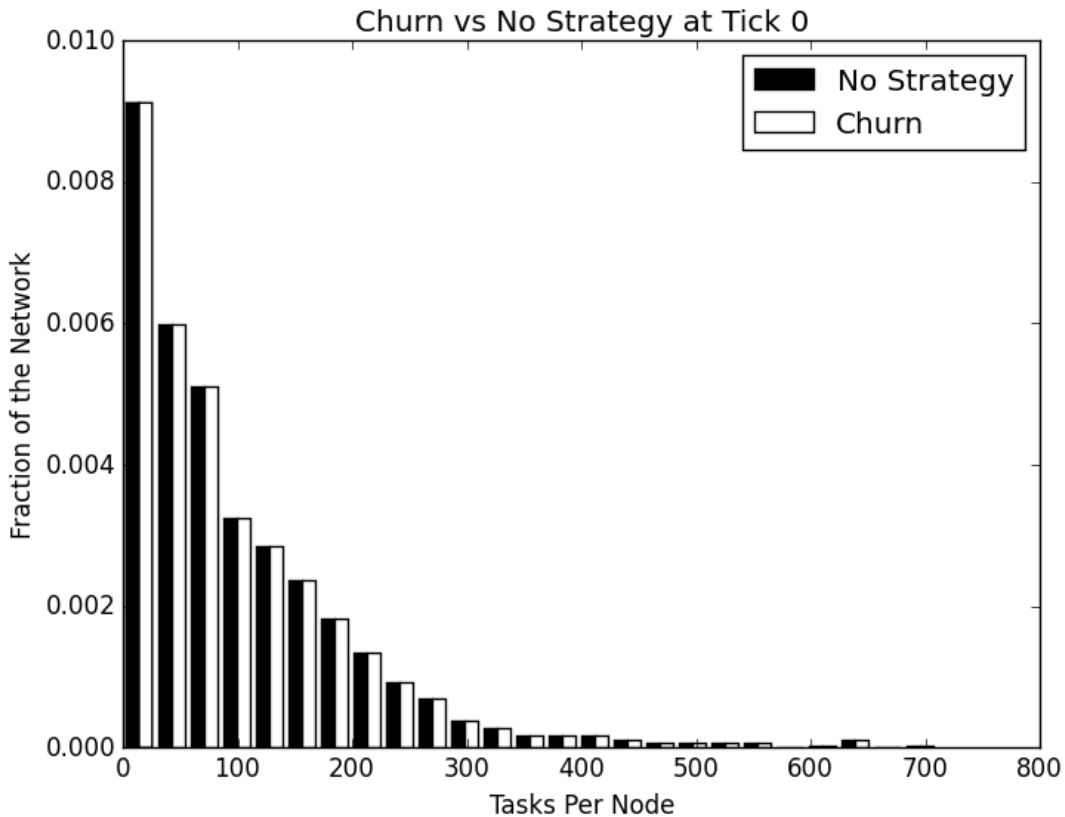


Figure: The initial distribution of the workload in both networks. As both networks start with same initial configuration, the distribution is currently identical. This greatly resembles the distribution we saw in Figure 13.

Churn Distribution after 5 Ticks

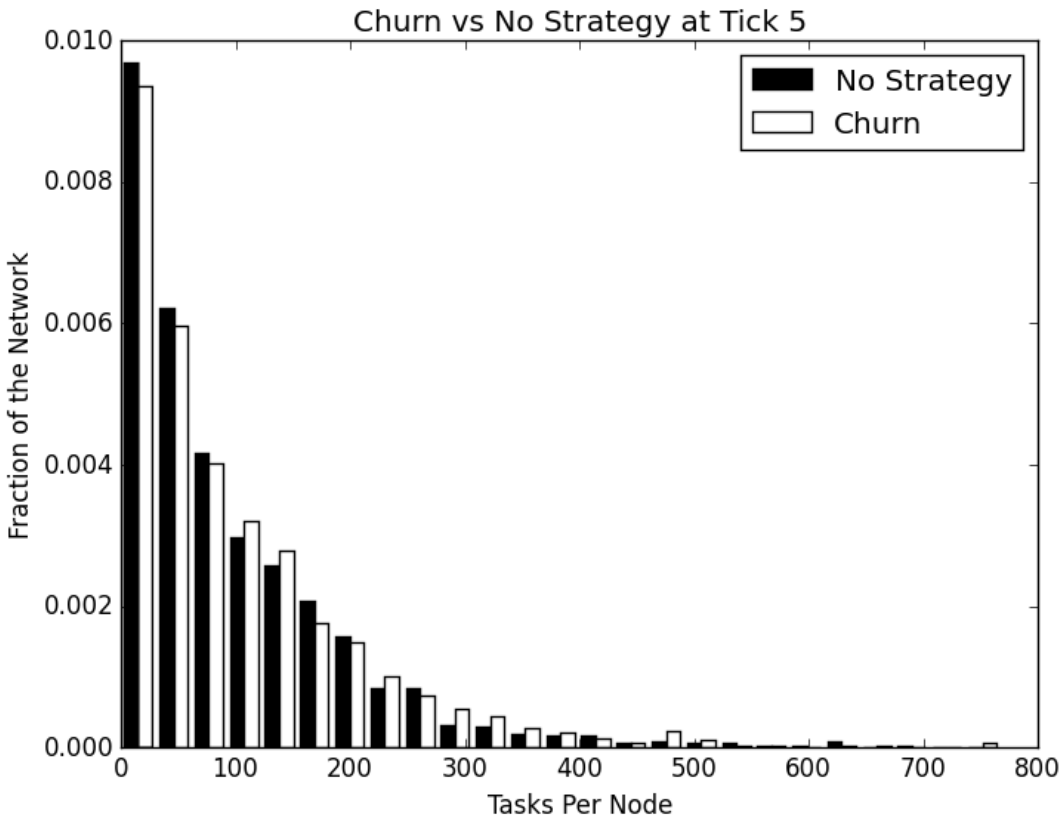


Figure: This is the distribution of the workloads at the beginning of tick 5. We can see the network using 0.01 churn has fewer nodes with less work and more nodes with a greater workload.

Churn Distribution after 35 Ticks

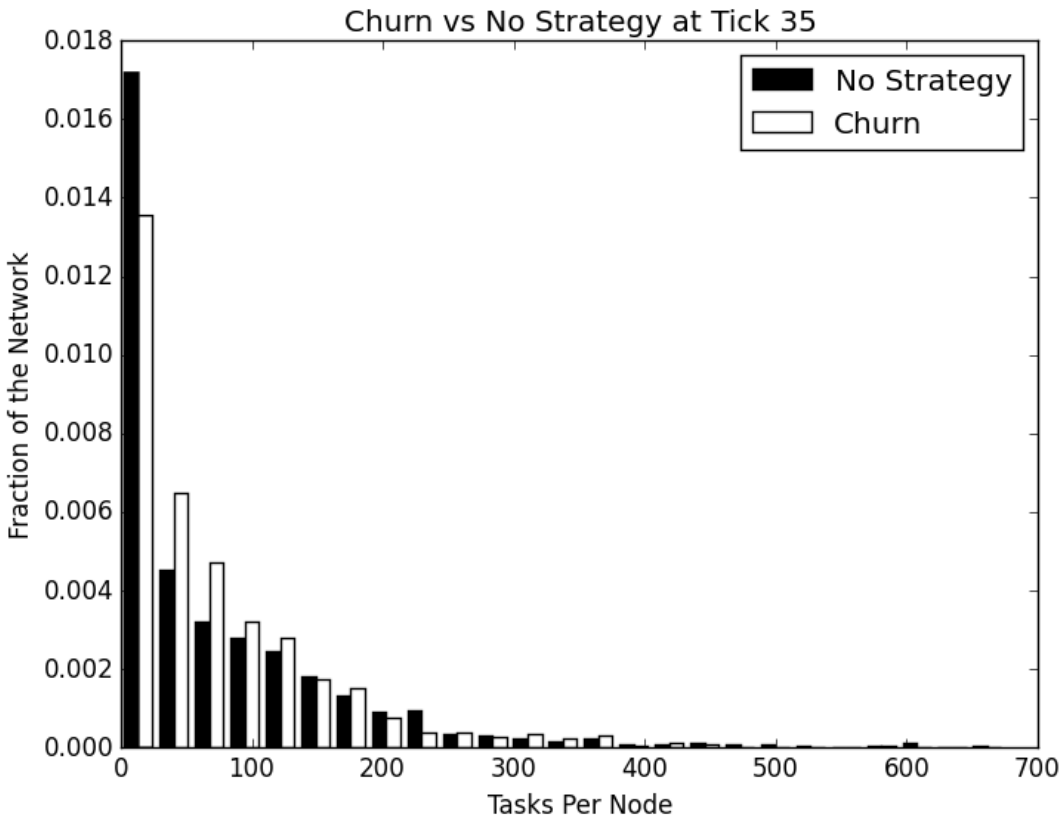


Figure: After 35 ticks, the effects of churn on the workload distribution become more pronounced. More nodes have consumed all their tasks and are simply idling, but significantly less in the network using churn.

Remarks

Strategy

- Nodes with loads $\leq$ `sybilThreshold` create Sybils
- These Sybils are randomly placed
- Act as a virtual node so the same node essentially exists in multiple locations
- Sybils are removed if the node that created it has no work

Random injection vs no strategy after 5 ticks

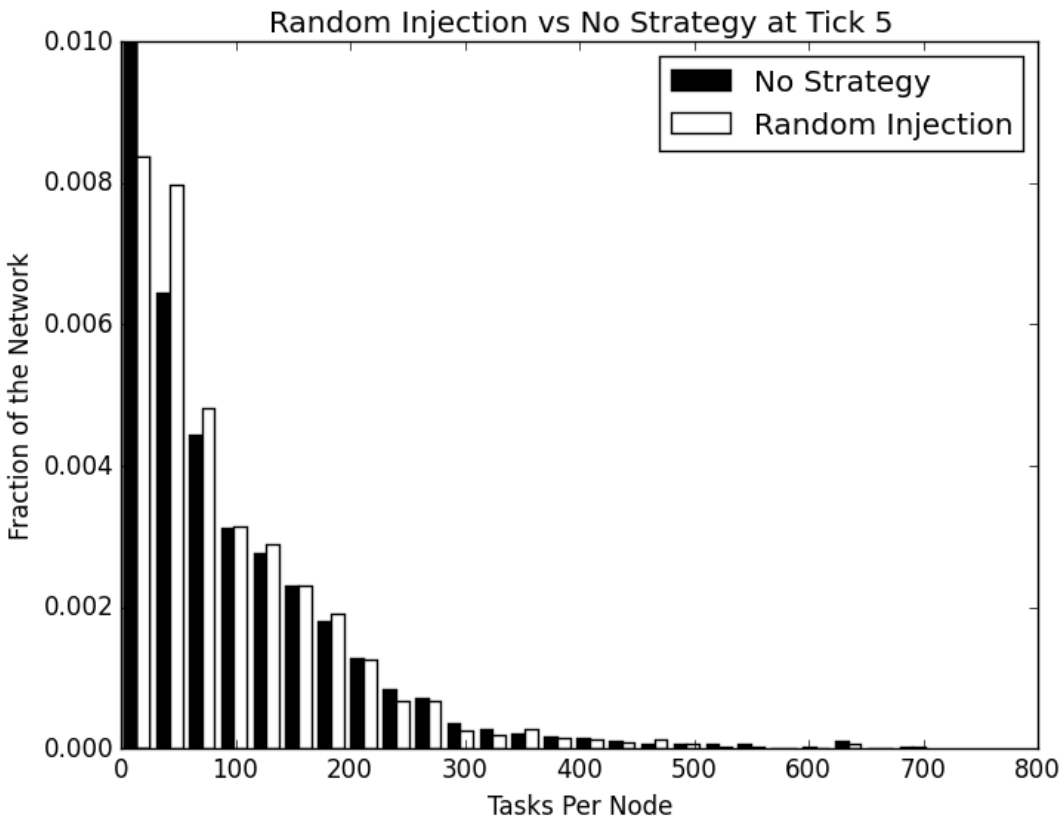


Figure: The networks after 5 ticks. The network using random injection has significantly less underutilized nodes, even after only 5 ticks.

Random Injection vs No Strategy After 35 ticks

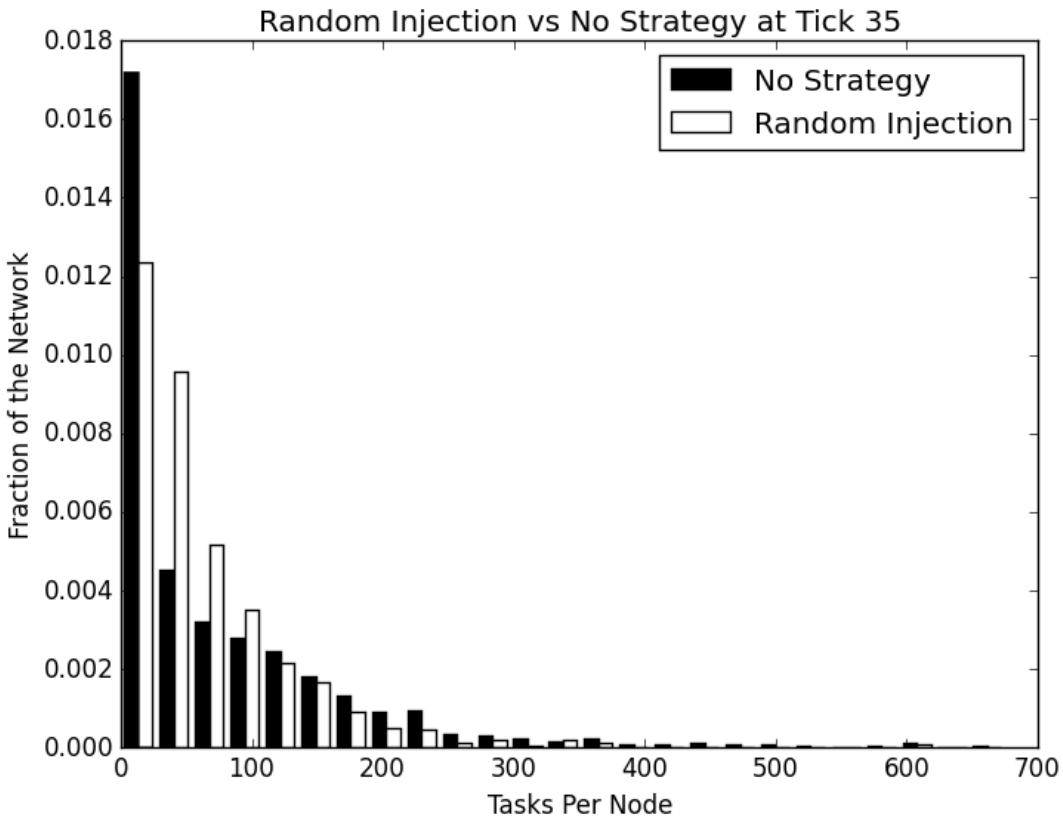


Figure: The networks after 35 ticks. The network using random injection has significantly less underutilized nodes and substantially more notes with some or lots of work.

Random Injection in a Heterogeneous Network

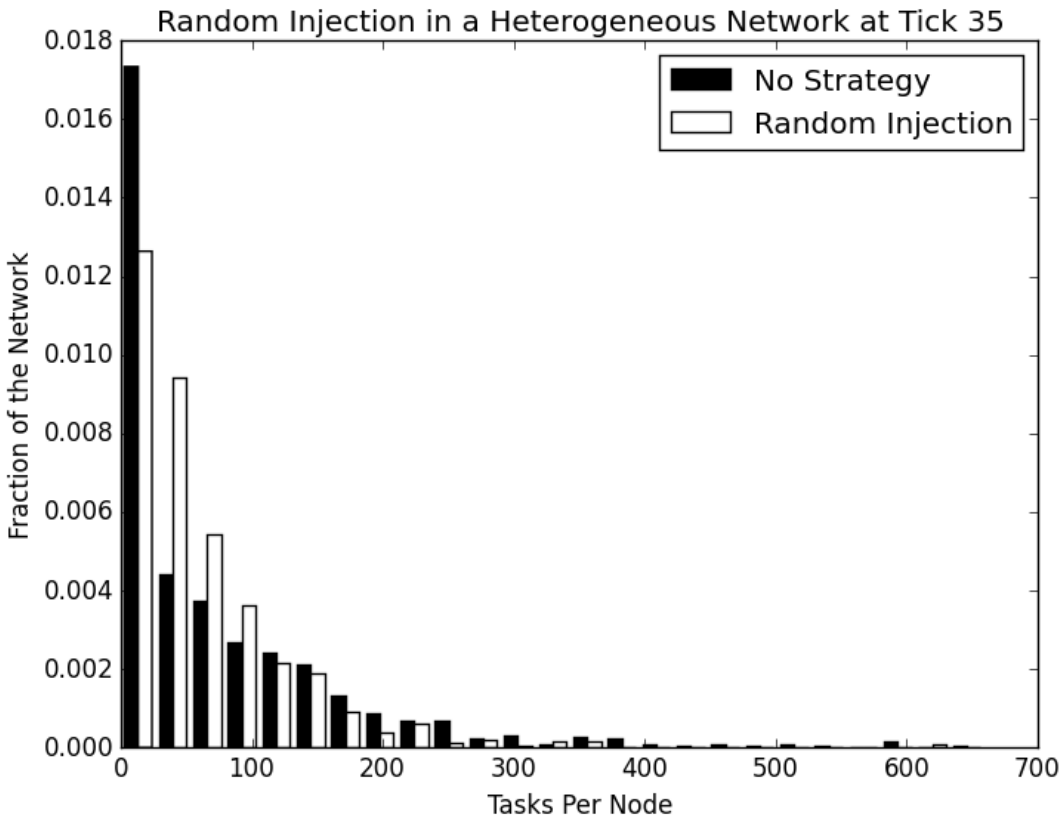


Figure: The workload distribution of heterogenous networks after 35 ticks. We can see the network using the random injection strategy is experiencing

Random Injection VS Churn

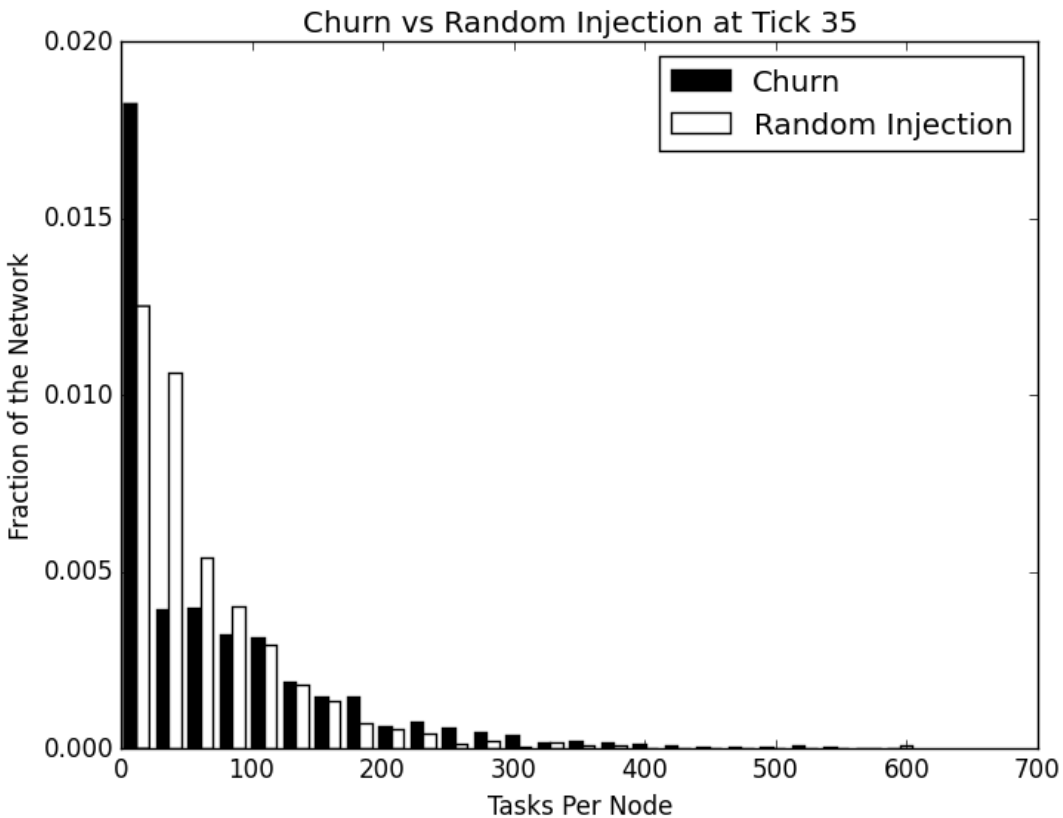


Figure: The networks after 35 ticks. The network using random injection load-balances significantly better than the network using Churn.

Remarks

- Best Results

Strategy

- Rather than creating Sybils randomly, nodes create one in their successors
- Finding node id uses mashes
- Estimates which successor has most work.
- Tested estimation against smart method

Remarks

Strategy

- Nodes with too much work ask for help
- Predecessor with smallest workload and below `sybilThreshold` creates a Sybil
- Reactive vs Proactive

Remarks

Summary

- Reactive vs Proactive

Table of Contents

Conclusion

- Load Balancing