

Table of Contents

1

Introduction

- Objective
- Distributed Computing and Challenges
- What Are Distributed Hash Tables?
- Why DHTs and Distributed Computing

2

Background

- The Components and Terminology
- Example DHT: Chord

3

Previous Work

- ChordReduce

4

UrDHT

- Introduction
- DGVH
- UrDHT
- Experimental Data
- Conclusion

5

Autonomous Load-Balancing

- Introduction
- Distribution of Keys in A DHT
- Experimental Setup
- Churn
- Random Injection
- Neighbor Injection
- Invitation
- Conclusion

6

Conclusion

- Conclusion

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

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

Or

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.

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.

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

Uses For DHT Distributed Computing

The generic framework we are proposing would be ideal for:

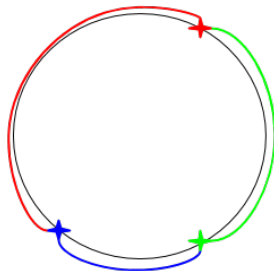
- Embarrassingly Parallel Computations
 - Any problem that can be framed using Map and Reduce.
 - Brute force cryptography.
 - Genetic algorithms.
 - Markov chain Monte Carlo methods.
- Use in either a P2P context or a more traditional deployment.

Table of Contents

- 1 Introduction
 - Objective
 - Distributed Computing and Challenges
 - What Are Distributed Hash Tables?
 - Why DHTs and Distributed Computing
- 2 Background
 - The Components and Terminology
 - Example DHT: Chord
- 3 Previous Work
 - ChordReduce
- 4 UrDHT
 - Introduction
 - DGVH
 - UrDHT
 - Experimental Data
 - Conclusion
- 5 Autonomous Load-Balancing
 - Introduction
 - Distribution of Keys in A DHT
 - Experimental Setup
 - Churn
 - Random Injection
 - Neighbor Injection
 - Invitation
 - Conclusion
- 6 Conclusion
 - Conclusion

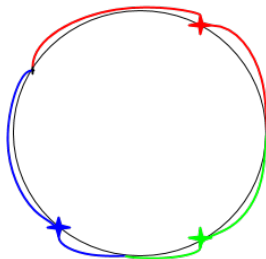
Chord's Closest Metric.

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.



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

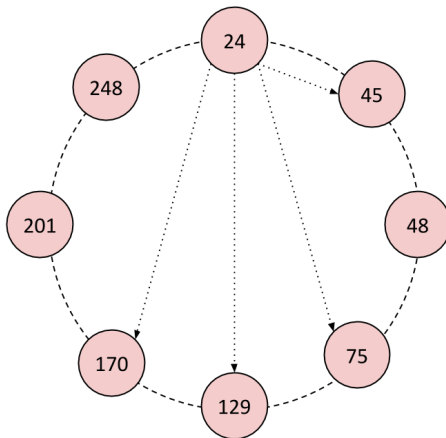


Figure: An 8-node Chord ring where $m = 8$. Node 24's long peers are shown.

Fault Tolerance 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

Table of Contents

- Objective
- Distributed Computing and Challenges
- What Are Distributed Hash Tables?
- Why DHTs and Distributed Computing

- The Components and Terminology
- Example DHT: Chord

3 Previous Work

- ChordReduce

- Introduction
- DGVH
- UrDHT
- Experimental Data
- Conclusion

- Introduction
- Distribution of Keys in A DHT
- Experimental Setup
- Churn
- Random Injection
- Neighbor Injection
- Invitation
- Conclusion

- ## ● Conclusion

Overarching Goal

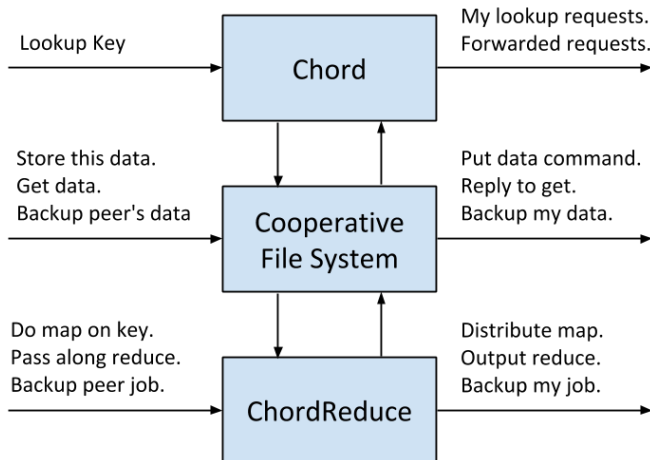
My research has been focused on:

- Abstracting out DHTs.
- Distributed computation using DHTs.

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



Reducing Results of Data

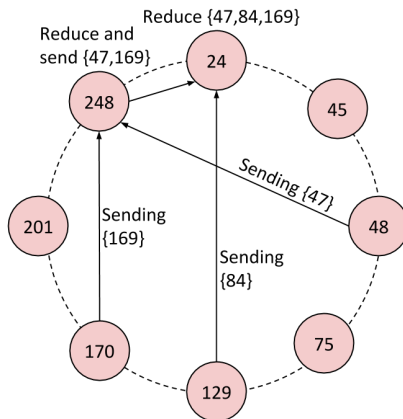


Figure: Results are sent back via the overlay. If a node receives multiple results, they are reduced before being sent on.

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.

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

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.

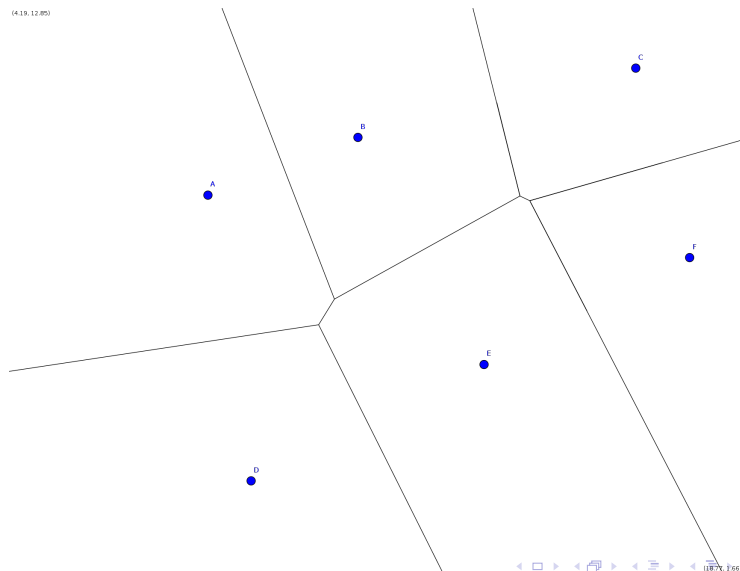
Goals

VHash sprung from two related ideas:

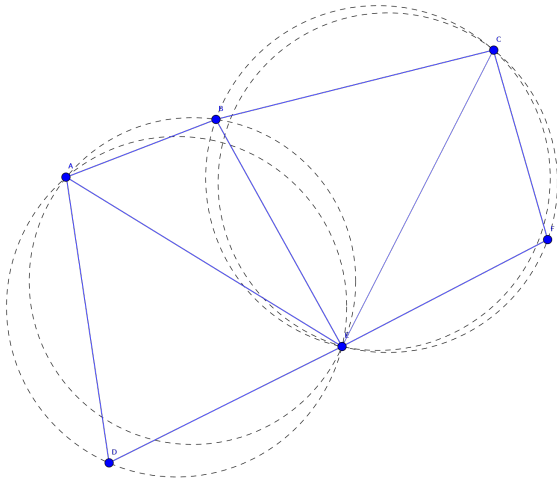
- We wanted a way be able optimize latency by embedding it into the routing overlay.
- We wanted to create a DHT based off of Voronoi tessellations. Unfortunately:
 - Distributed algorithms for this problem don't really exist.
 - Existing approximation algorithms were unsuitable.

Voronoi Tessellation

(4.19, 12.85)



(4.19, 12.85)



- 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.

- 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

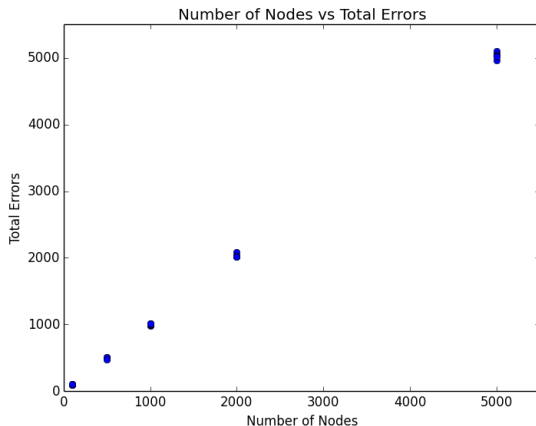
```

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

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

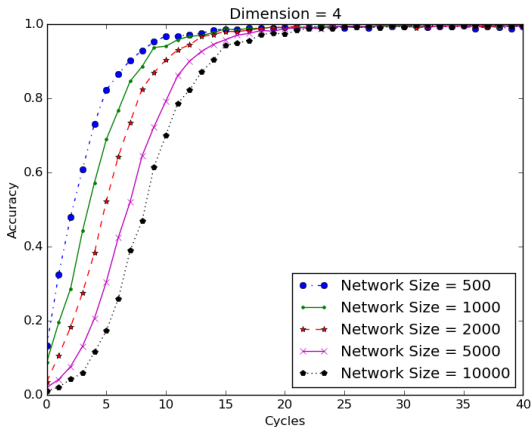
$$O\left(\frac{\log^2 n}{\log^2 \log n}\right)$$

## 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.





## 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.

# What is UrDHT

- Three Components
  - Storage
  - Networking
  - Logic
    - Protocol
    - Space Math

# The Protocol

- Consists of
  - Node Information
  - Short peers
  - Long Peers
  - The functions we use
- Replaced lookup with seek
- Maintenance is gossip based, using functions provided by the Space Math
- Short peer selection is done by DGVH by default
- Once short peers are selected, `handleLongPeers` is called

# Space Math

- Defines the DHT topology
- Requires a way to generate short peers and choose long peers

- `idToPoint` takes key, maps it to a point in space
- `distance` outputs the shortest distance between  $a$  and  $b$
- `getDelaunayPeers` which is DGVH
- `getClosest`
- `handleLongPeers`

We demonstrated how to implement

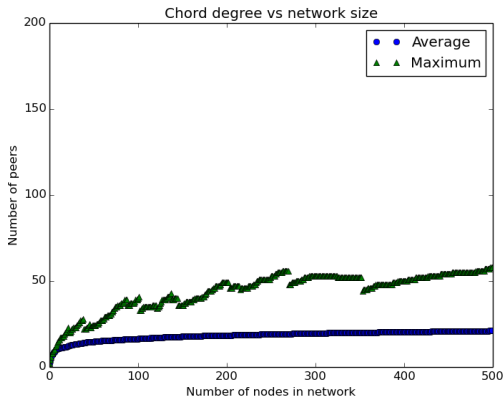
- Chord / Symphony
- Kademia
- ZHT

- Tested four different topologies
  - Chord
  - Kademlia
  - Euclidean
  - Hyperbolic
- We create a 500 node network, adding one node at a time and completing a maintenance cycle.



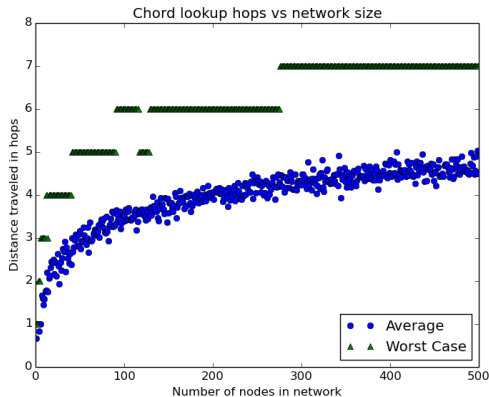
- Reachability
- The average degree of the network.
- The worst case degree of the network.
- The average number of hops between nodes using greedy routing.
- The diameter of the network.

# Chord Degree

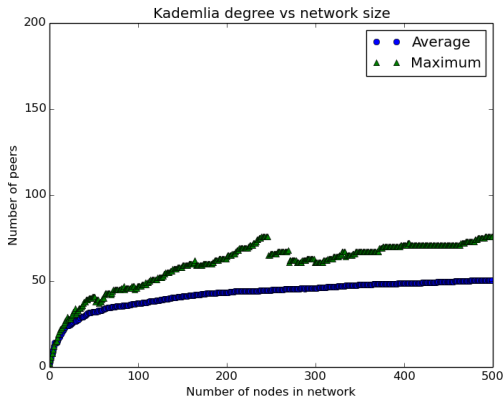


**Figure:** This is the average and maximum degree of nodes in the Chord network. This Chord network utilized a 120 bit hash and thus degree is bound at 122 (full fingers, predecessor and successor) when the network reaches  $2^{120}$  nodes.

# Chord Distance

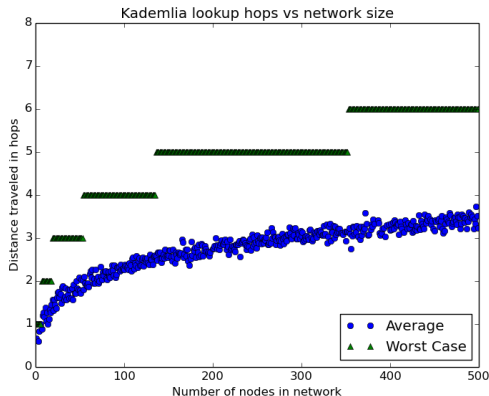


**Figure:** This is the number hops required for a greedy routed lookup in Chord. The average lookup between two nodes follows the expected logarithmic curve.



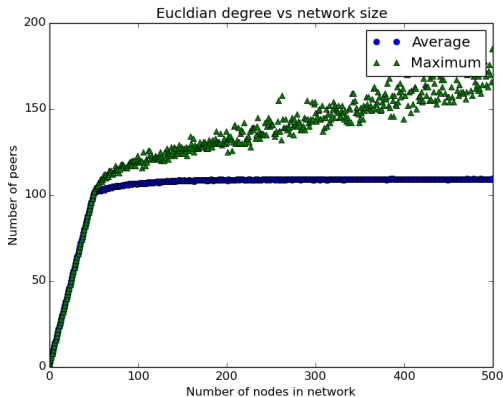
**Figure:** This is the average and maximum degree of nodes in the Kademlia network as new nodes are added. Both the maximum degree and average degree are  $O(\log n)$ .

# Kademlia Distance



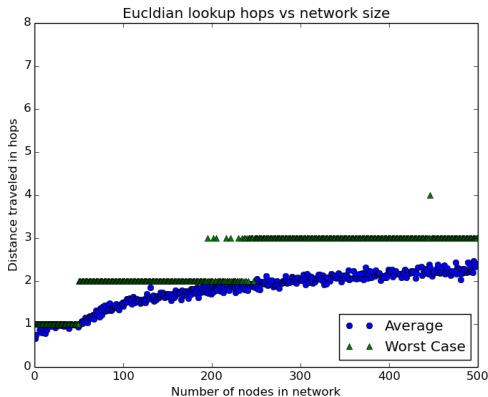
**Figure:** Much like Chord, the average degree follows a distinct logarithmic curve, reaching an average distance of approximately three hops when there are 500 nodes in the network.

# Euclidean Degree



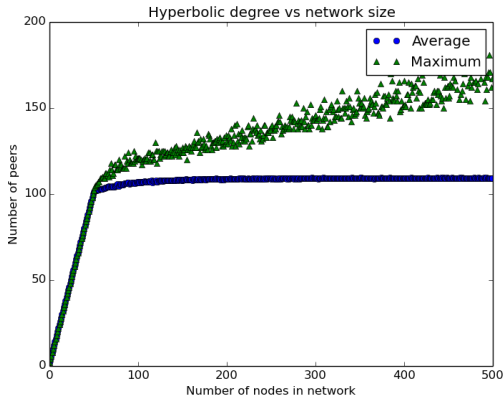
**Figure:** Because the long peers increase linearly to the maximum value (49), degree initially rises quickly and then grows more slowly as the number of long peers ceases to grow and the size short peers increases with network size.

# Euclidean Distance



**Figure:** The inter-node distance stays constant at 1 until long peers are filled, then rises at the rate of a randomly connected network due to the distribution of long peers selected

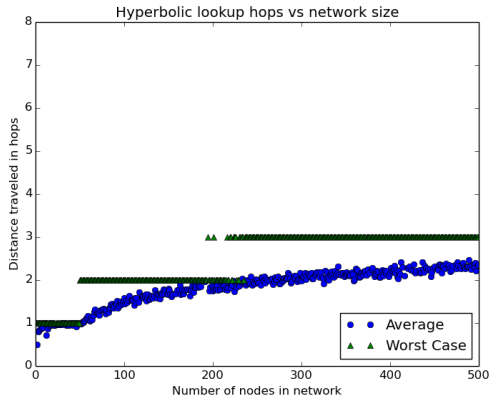
# Hyperbolic Degree



**Figure:** The Hyperbolic network uses the same long and short peer strategies to the Euclidean network, and thus shows similar results.



# Hyperbolic Distance



**Figure:** Like the Euclidean Geometry, our Poincaré disc based topology has much shorter maximum and average distances.

# Table of Contents

- Objective
- Distributed Computing and Challenges
- What Are Distributed Hash Tables?
- Why DHTs and Distributed Computing

- The Components and Terminology
- Example DHT: Chord

- ChordReduce

- Introduction
- DGVH
- UrDHT
- Experimental Data
- Conclusion

## 5 Autonomous Load-Balancing

- Introduction
- Distribution of Keys in A DHT
- Experimental Setup
- Churn
- Random Injection
- Neighbor Injection
- Invitation
- Conclusion

- ## ● Conclusion

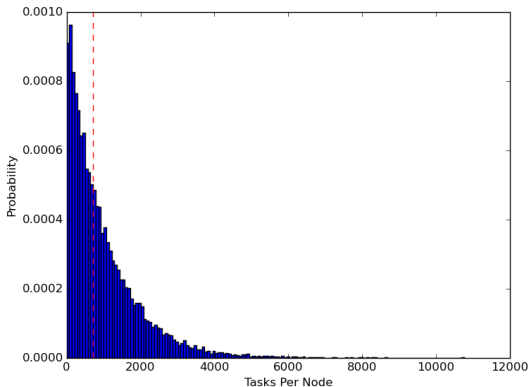
- In this project, we set out to confirm the results of ChordReduce
- Objectives:
  - Confirm that high levels of churn can help a DHT based computing environment.
  - Develop better strategies than randomness

# Distribution in Networks of Different Sizes

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

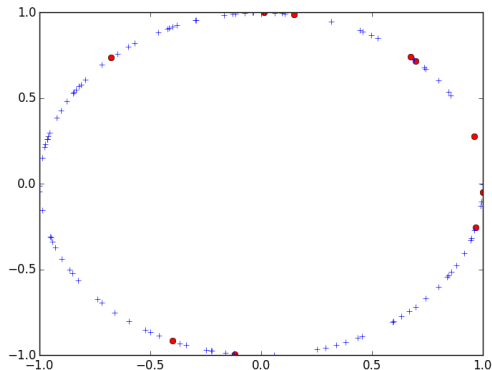
| Nodes | Tasks   | Median Workload | $\sigma$ |
|-------|---------|-----------------|----------|
| 1000  | 100000  | 69.410          | 137.27   |
| 1000  | 500000  | 346.570         | 499.169  |
| 1000  | 1000000 | 692.300         | 996.982  |
| 5000  | 100000  | 13.810          | 20.477   |
| 5000  | 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).

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

<sup>1</sup>This has been implemented and tested.

- 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

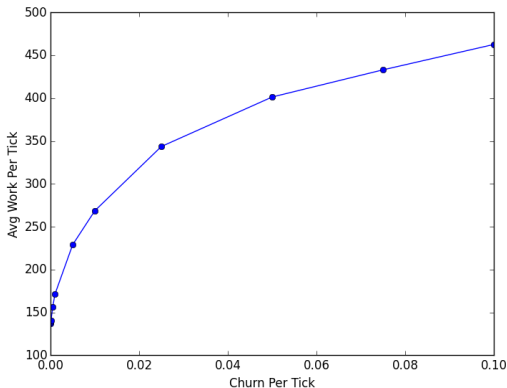


**Table:** Runtime factor of networks of varying sizes and number of tasks, each using the Churn strategy to load-balance. Each result is the average of 100 trials. The networks are homogeneous and each node consumes one task per tick. A runtime of 1 is the ideal and target.

| Churn Rate | $10^3$ nodes,<br>$10^5$ tasks | $10^3$ nodes,<br>$10^6$ tasks | 100 nodes,<br>$10^4$ tasks | 100 nodes,<br>$10^5$ tasks | 100 nodes,<br>$10^6$ 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                      |

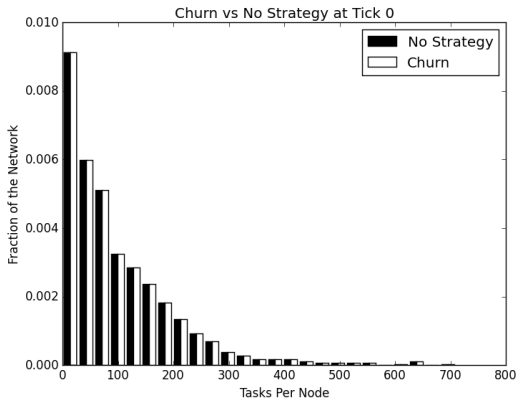


# Churn vs Average Work Per Tick



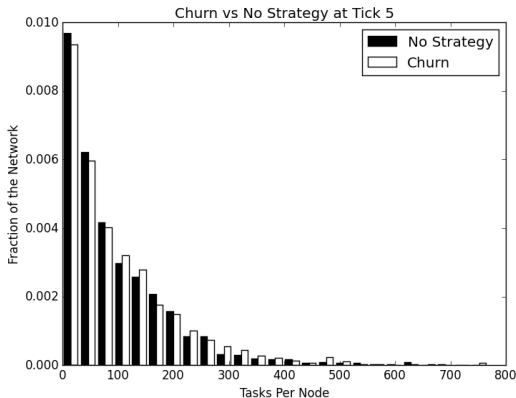
**Figure:** With more and more churn in the network, new nodes have a higher chance of joining the network and acquiring work from overloaded nodes. This results in more average work being done each tick, as there are less nodes simply idling.

## Base Work Distribution



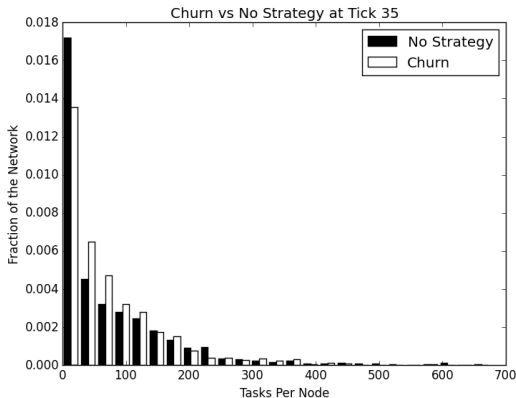
**Figure:** The initial distribution of the workload in both networks. As both networks start with same initial configuration, the distribution is currently identical. This greatly resembles the distribution we saw in Figure 17. <sup>Geo</sup>

## 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



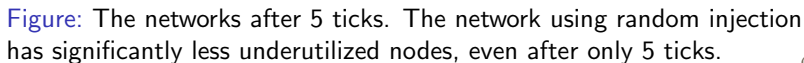
**Figure:** After 35 ticks, the effects of churn on the workload distribution become more pronounced. More nodes have consumed all their tasks and are simply idling, but significantly less in the network using churn.



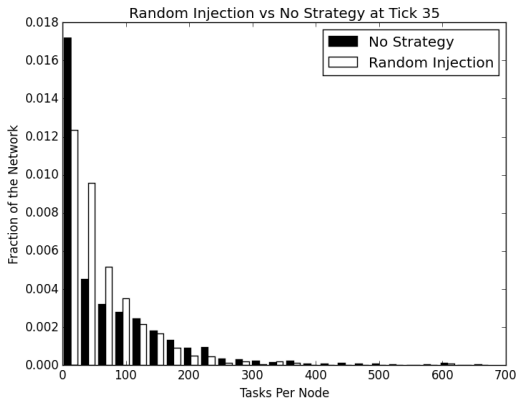




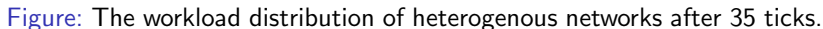
- A homogeneous, 1000 node/100,000 task network, never have an average runtime factor greater than 1.7
- Minimum was 1.36.
- In the same network with 1,000,000 tasks, these runtimes were 1.25 and 1.12 respectively.
- On average, the 1,000,000 task network had a runtime factor 0.82 less than the 100,000 task network.



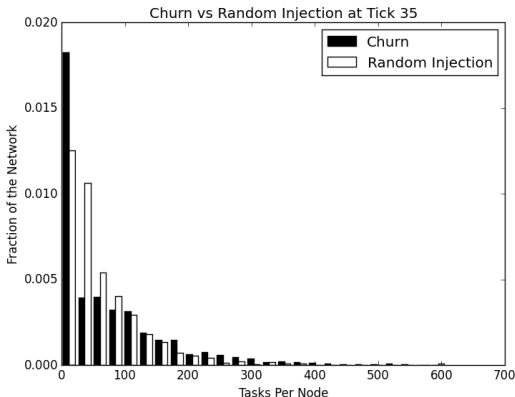
# Random Injection vs No Strategy After 35 ticks



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



# Random Injection VS Churn



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

- sybilThreshold would lower the runtime factor.
- Churn had no significant effect
- Maxsybils (node strength) had no effect in homogeneous networks

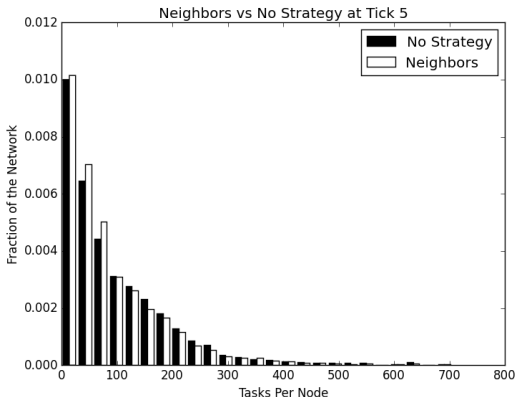






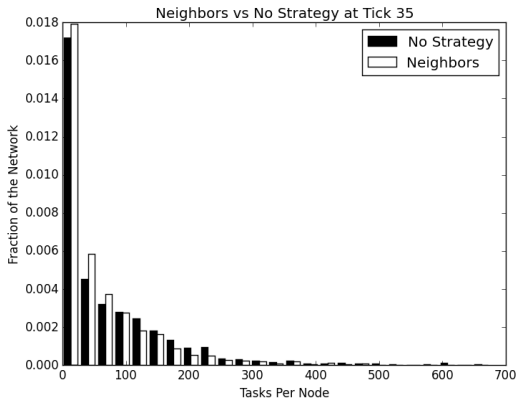


# Neighbor Injection after 5 Ticks

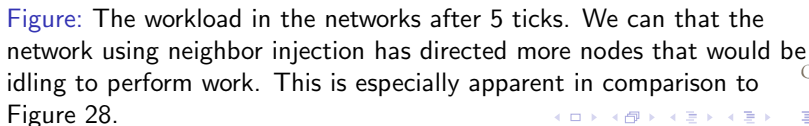


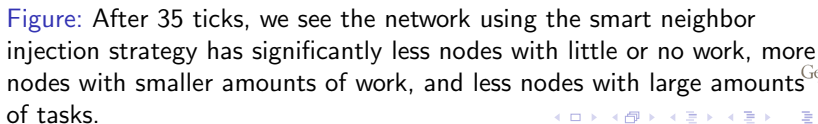
**Figure:** While the network no strategy appears to have less nodes idling, the active nodes in the network using neighbors are being better utilized.

## Neighbor Injection after 35 Ticks



**Figure:** Despite have more idling nodes, we see that the nodes using the neighbor injection strategy have acquired smaller workloads and have effectively shifted part of the histogram left.

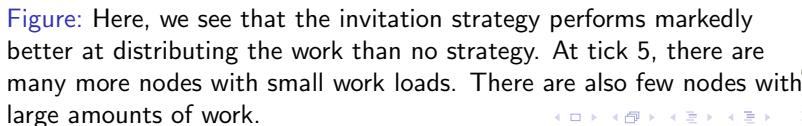




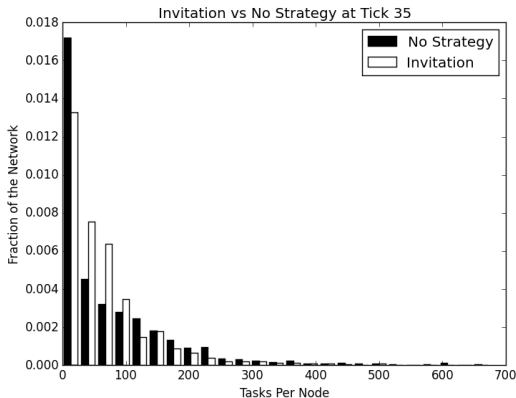






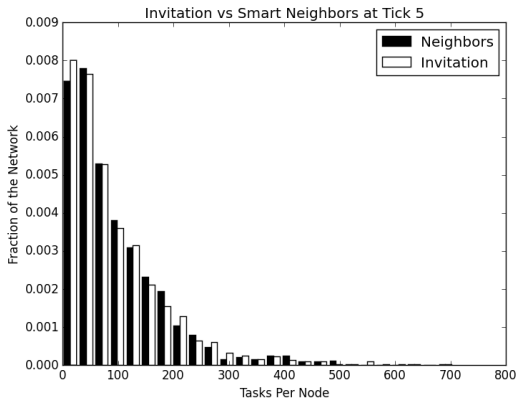


## Invitation at 35 Ticks



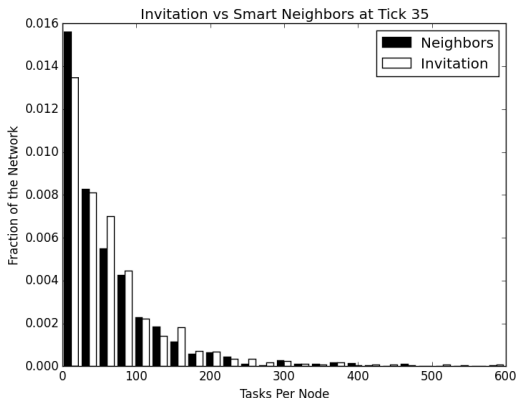
**Figure:** At 35 ticks, we can see the network using the invitation strategy perform markedly better than the network using no strategy. The highest load is around 500 tasks in the network using invitation, compared to approximately 650 tasks in the network using no strategy.

# Invitation vs Smart Neighbors at Tick 5



**Figure:** A comparison of a network using the smart neighbors strategy and a network using the invitation strategy. Invitation has slightly more nodes on the performing less work, but no significant differences have emerged.

# Invitation vs Smart Neighbors at Tick 35



**Figure:** After 35 ticks, differences between the two strategies have emerged. The network using the invitation strategy has significantly less nodes with a small work load and many more with large work loads.



## Summary

- Reactive vs Proactive
- Heterogeneity was best handled by Churn or Random Injection.
- Random injection was best overall
- Load balanced did not mean faster since node strength was not taken into account.

# Table of Contents

- Objective
- Distributed Computing and Challenges
- What Are Distributed Hash Tables?
- Why DHTs and Distributed Computing

- The Components and Terminology
- Example DHT: Chord

- ChordReduce

- Introduction
- DGVH
- UrDHT
- Experimental Data
- Conclusion

- Introduction
- Distribution of Keys in A DHT
- Experimental Setup
- Churn
- Random Injection
- Neighbor Injection
- Invitation
- Conclusion

## 6 Conclusion

- ## ● Conclusion



