Dissertation
Towards a Framework for DHT Distributed Computing

Andrew Rosen

Georgia State University

May 13th, 2016

Table of Contents

Georgia State University

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.

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.

- 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.
- This means we need two things:
- A way of easily abstracting DHTs
- A way to make sure that we can distribute work effectively across a DHT

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.

Distributed Computing platforms experience these challenges:

Scalability As the network grows, more resources are spent on maintaining and organizing the network.

To review now. Remember, computers aren't telepathic. There's always an overhead cost. It will grow. The challenge of scalability is designing a protocol 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 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.

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.

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. Nodes leaving is treated as a 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 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.

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

Distributed Key/Value Stores

Distributed Hash Tables are mechanisms for storing values associated with certain keys.

- Values, such as filenames, data, or IP/port combinations are associated with keys.
- These keys are generated by taking the hash of the value.
- We can get the value for a certain key by asking any node in the network.

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.

At their core, Distributed Hash Tables are giant lookup tables. Given a key, it will return the value associated with that key, if it exists. These keys, or hash keys, are generated by a hash function, such as SHA1 or MD5. These hash functions use black magic using prime numbers and modular arithmetic to return a close to unique identifier associated with a given input. The key about the keys is the same input will always produce the same output. From a design standpoint, they are distributed uniformly at random. **This is a lie and has a huge impact on the effectiveness of distributing computing using DHT**.

Strengths of DHTs

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

- Scalable
- Fault-Tolerant
- Load-Balancing

Strengths of DHTs

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

- Scalable
- Fault-Tolerant
- Load-Balancing

- 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.
  - This allows a large-scale failure, like California being hit by a massive earthquake, to be absorbed throughout the network, rather than a contiguous portion being knocked out.
  - Nodes are responsible for the data closest to it.
  - The space is large enough to avoid Hash collisions.

Table of Contents

Georgia State University

Required Attributes of DHT

One of our contributions is that we reduced a DHT down into its core components. A DHT needs:

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

Required Attributes of DHT

One of our contributions is that we reduced a DHT down into its core components. A DHT needs:

- A distance 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. This isn't just a matter of nesscicarily being closest to something, but what range you might be responsible for. For instance, Chord has been implemented with nodes being responsible for keys between their predessor and themselves or themselves and their successors. This can normally be covered by distance.
- 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.

Terms and Variables

- Network size is n nodes.
- Keys and IDs are m bit hashes, usually SHA1 with 160 bits.
- Peerlists are made up of:

  Short Peers The neighboring nodes that define the network's topology.

  Long Peers Routing shortcuts.

Terms and Variables

- Network size is n nodes.
- Keys and IDs are m bit hashes, usually SHA1 with 160 bits.
- Peerlists are made up of:
  - **Short Peers** The neighboring nodes that define the network's topology.
  - **Long Peers** Routing shortcuts.

- SHA1 is being depreciated, but this is trivial from our perspective. They al use cryptographic hash functions
- Short peers are actively maintained, long peers replaced gradually and are not actively pinged.

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

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

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

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

- 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

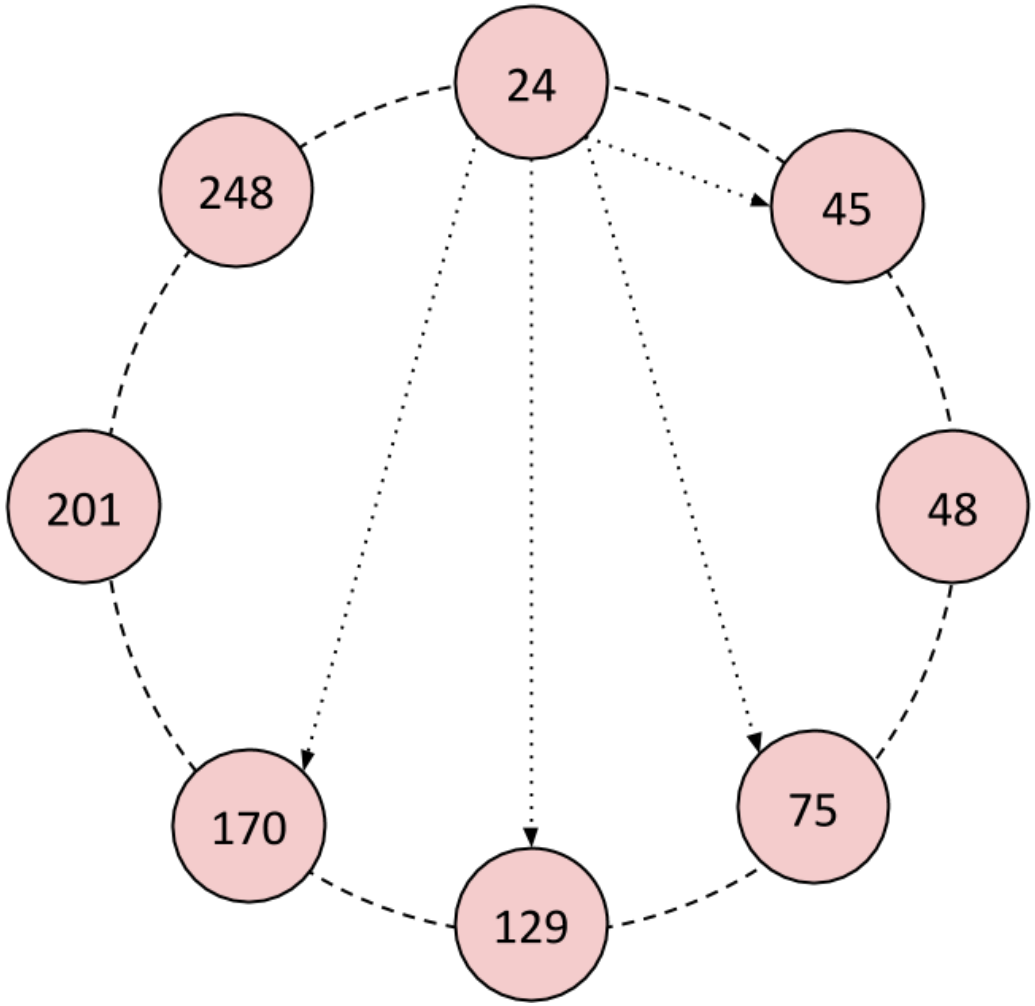


Figure: An 8-node Chord ring where $m = 8$. Node 24's long peers are shown.

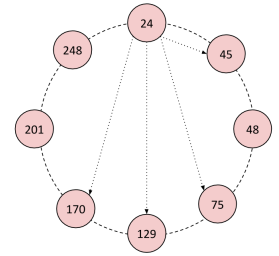


Figure: An 8-node Chord ring where $m = 8$. Node 24's long peers are shown.

- 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.
- Routing example 24 to 150.

Table of Contents

Georgia State University

Overarching Theme

My research has been focused on:

- Abstracting out DHTs.
- Distributed computation using DHTs.

Overarching Theme

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 DHT have in common, so that I could create a generic DHT.
- As I implied earlier, we all these DHTs, but they all have the same features and I want to generalize them.
- I focused on creating a more abstract framework for MapReduce, so I could move it out of the datacenter and into other contexts.

ChordReduce

Objective:

- Create an abstract MapReduce framework.
- Create a completely decentralized, fault-tolerant, distributed computing framework.

Results:

- We succeeded.
- Our network could handle nodes leaving, as well as assign joining nodes work.
- We found anomalous results with Churn.

ChordReduce

Objective:

- Create an abstract MapReduce framework.
- Create a completely decentralized, fault-tolerant, distributed computing framework.

Results:

- We succeeded.
- Our network could handle nodes leaving, as well as assign joining nodes work.
- We found anomalous results with Churn.

- We leveraged Chord's fault tolerance mechanisms to handle tasks
- Why did churn do what it did? That's covered in Autonomous Load Balancing.

VHash and DGVH

What was it:

- Made a Voronoi-based DHT.
- A greedy heuristic.
- More detail later.

Results:

- New concept: nodes that move their position.
- DGVH.
- We started to think how about how can we completely abstract any DHT topology into a Voronoi/Delaunay relationship.

D^3NS

- Create a completely decentralized, distributed, DNS.
- Backwards compatible and completely reliable to the user.
- Used Blockchains and UrDHT.

Sybil Attack Analysis

- Studied the amount of resources needed to perform a Sybil attack.
- Examined how difficult it was for nodes to inject a Sybil into a specific location.

Table of Contents

Introduction

- Build on DGVH and VHash
- Create an abstract model of a DHT based on Voronoi/Delaunay
- Can be used as a bootstrapping network for other distributed systems
- Can emulate the topology of other DHTs

Goals

VHash and DGVH sprung from two related ideas:

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

VHash and DGVH sprung from two related ideas:

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

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

Goals

VHash and DGVH 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:

VHash and DGVH 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:

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

Goals

VHash and DGVH 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 and DGVH 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.

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 and DGVH 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.

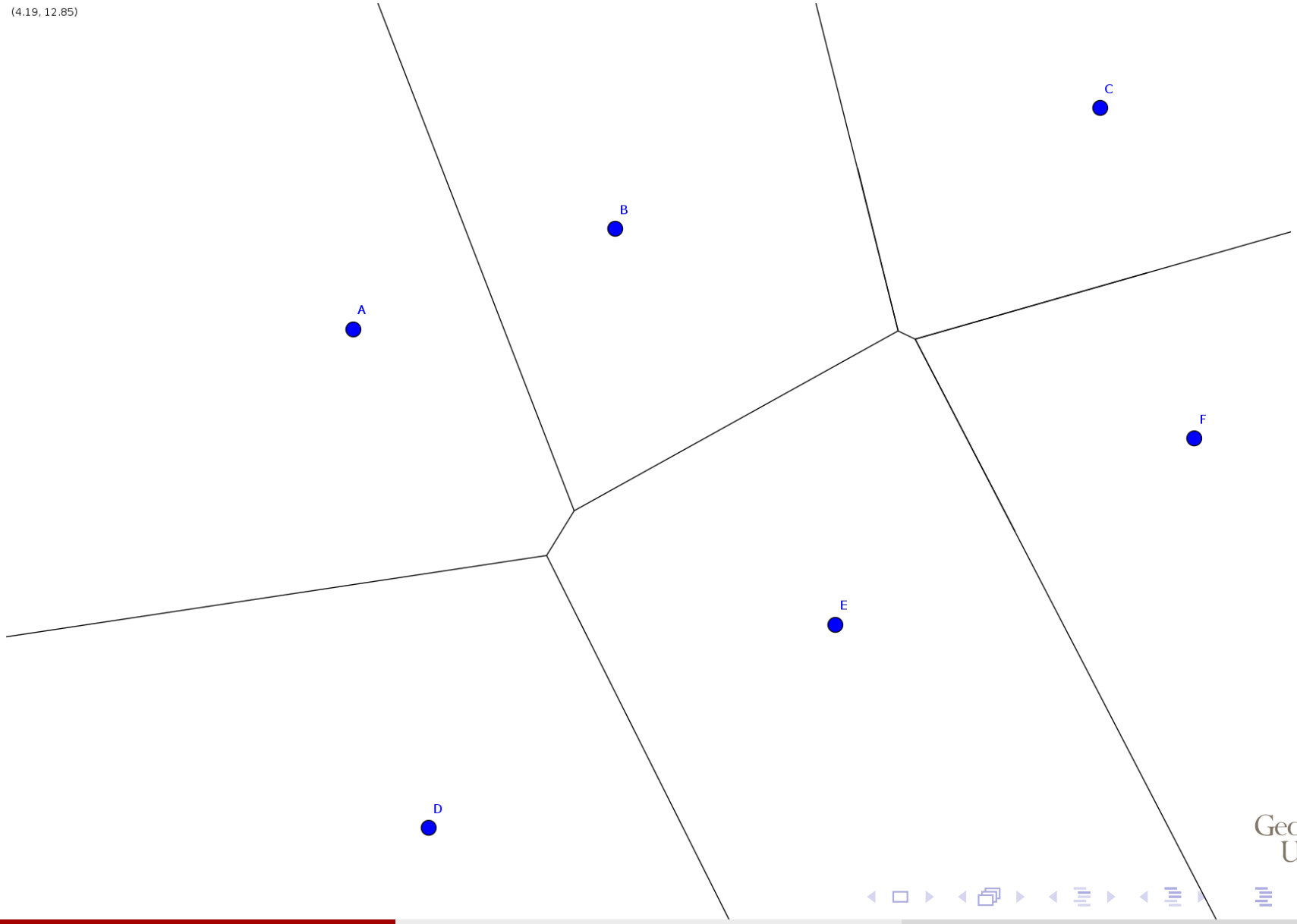
Goals

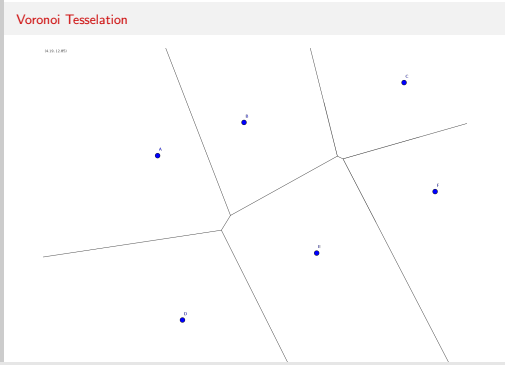
VHash and DGVH sprung from two related ideas:

- 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.

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

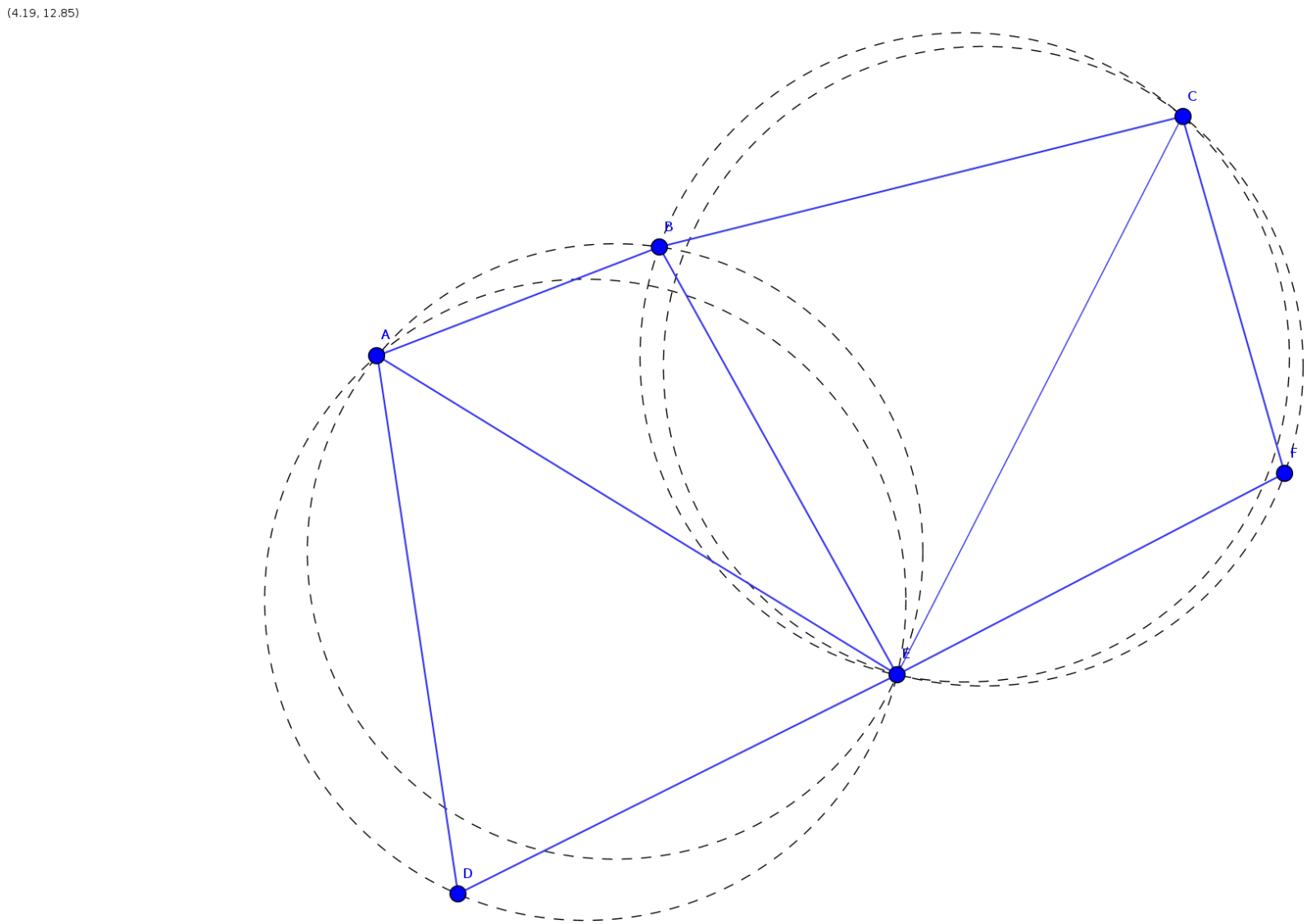


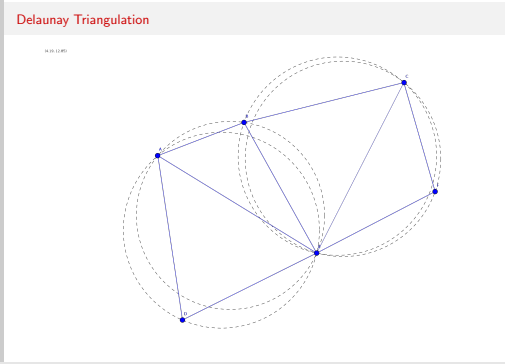


Define

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

Delaunay Triangulation





Define

- Delaunay Triangulation is a triangulation of a set of points with the following rule:
- No point falls within any of the circumcircles 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 its Delaunay neighbors.

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.

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.

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: **if** Any node in *peers* is closer to c than n **then**
7: Reject c as a peer
8: **else**
9: Remove c from *candidates*
10: Add c to *peers*
11: **end if**
12: **end for**

DGVH Heuristic

```
1:  Given node n and its list of candidates.
2:  peers ← 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:      if Any node in peers is closer to c than n then
7:          Reject c as a peer
8:      else
9:          Remove c from candidates
10:         Add c to peers
11:     end if
12: end for
```

1. We have n, the current node and a list of candidates.
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. Next, we iterate through the sorted list of candidates and either add them t the peers set or discard them.
6. If the candidate is closer to any peer than n, then it does not fall on the interface between the location's Voronoi regions.
7. in this case discard it
8. otherwise add it the current peerlist

DGVH Time Complexity

For k candidates, the cost is:

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

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

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

or

$$O(\frac{\log^4 n}{\log^4 \log n})$$

Depending on whether we gossip with a single neighbor or all neighbors.

DGVH Time Complexity

For k candidates, the cost is:

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

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

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

or

$$O(\frac{\log^4 n}{\log^4 \log n})$$

Depending on whether we gossip with a single neighbor or all neighbors.

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 then compares distances between all peers and all the candidates. This results in a cost of

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

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:

The expected worst case cost of DGVH is $O(\frac{\log^4 n}{\log^4 \log n})$, if we swap with all short peers. Otherwise the cost is $O(\frac{\log^2 N}{\log^2 \log N})$

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 $\Omega(1)$.

Summary

- DGVH is simple approximation for Delaunay Triangulation that guarantees a fully connected graph.
- Creates a fully connected subset of the Delaunay Triangulation.
- A DHT using DGVH can optimize over a metric such as latency and achieve superior routing speeds as a result.
- We built VHash to test this.
- UrDHT fully implements this and use

Georgia State University

Summary

- DGVH is simple approximation for Delaunay Triangulation that guarantees a fully connected graph.
- Creates a fully connected subset of the Delaunay Triangulation.
- A DHT using DGVH can optimize over a metric such as latency and achieve superior routing speeds as a result.
- We built VHash to test this.
- UrDHT fully implements this and use

- 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 opene with this...)

What is UrDHT

- Abstract framework for implementing DHTs or various topologies
- Three Components
  - Storage
  - Networking
  - Logic
    - Protocol
    - Space Math

What is UrDHT

- Abstract framework for implementing DHTs or various topologies
- Three Components
  - Storage
  - Networking
  - Logic
    - Protocol
    - Space Math

- We implement these topologies by abstracting them to the Voronoi Delauna
- Storage deals with file storage
- Networking deals with actual implementation of how nodes talk across the network
- Mostly we'll talk about the logic, which has two components
- You can change both, but changing space math is sufficient to create a DHT with a new topology.

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

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

- Seek is a single step of the lookup
- lookup can be done with iterative calls to seek
- While many protocols specify a recursive lookup, we find must actually implement an iterative, since it makes it easier to handle errors
- Don't have to change protocol, but you can. I implemented chord this way. Also, there may be some space DGVH might not work
- Only DHT we haven't been able to fully reproduce is CAN, because the insertion order matters in CAN, but not other DHTs.
- Handle long peers is discussed in a bit

Space Math

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

Space Functions

- `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`

Space Functions

- `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`

- Distance is not symetrical in every space
- Given a set of points, the candidates, and a center point, `getDelaunayPeers` calculates a mesh that approximates the Delaunay peers of center.
- `getClosest` returns the point closest to `center` from `candidates`. `seek` depends on `getClosest`
- `handleLongPeers` takes a liost of candidates and a center, returns a selection of long peers/
- Implementation should vary greatly, small world and the like should use a probability dist, Chord uses a structed distribution
- If long peers is too big, use a subset.

DHTs To Implement

We demonstrated how to implement

- Chord / Symphony
- Kademlia
- ZHT

Setup

- 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.

Setup

- 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.

- Two different types of experiments
- Tested to see if actual implementation worked
- Simulated creating larger networks using the smae exact logic we used for UrDHT
- This was one of the benefits to abstract this the way we did, we could pull the math out for easy simulation

Data Collected

- 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.

Data Collected

- 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.

- Everything is reachable
- The average degree of the network. This is the number of outgoing links and includes both short and long peers.
- Diameter is the worst case distance between two nodes using greedy routing
- Averages for distance are averages of 100 pairs of random source destinatio pairs (or network size if the network was smaller)
- Averages of degree were averages of all nodes.

Chord Degree

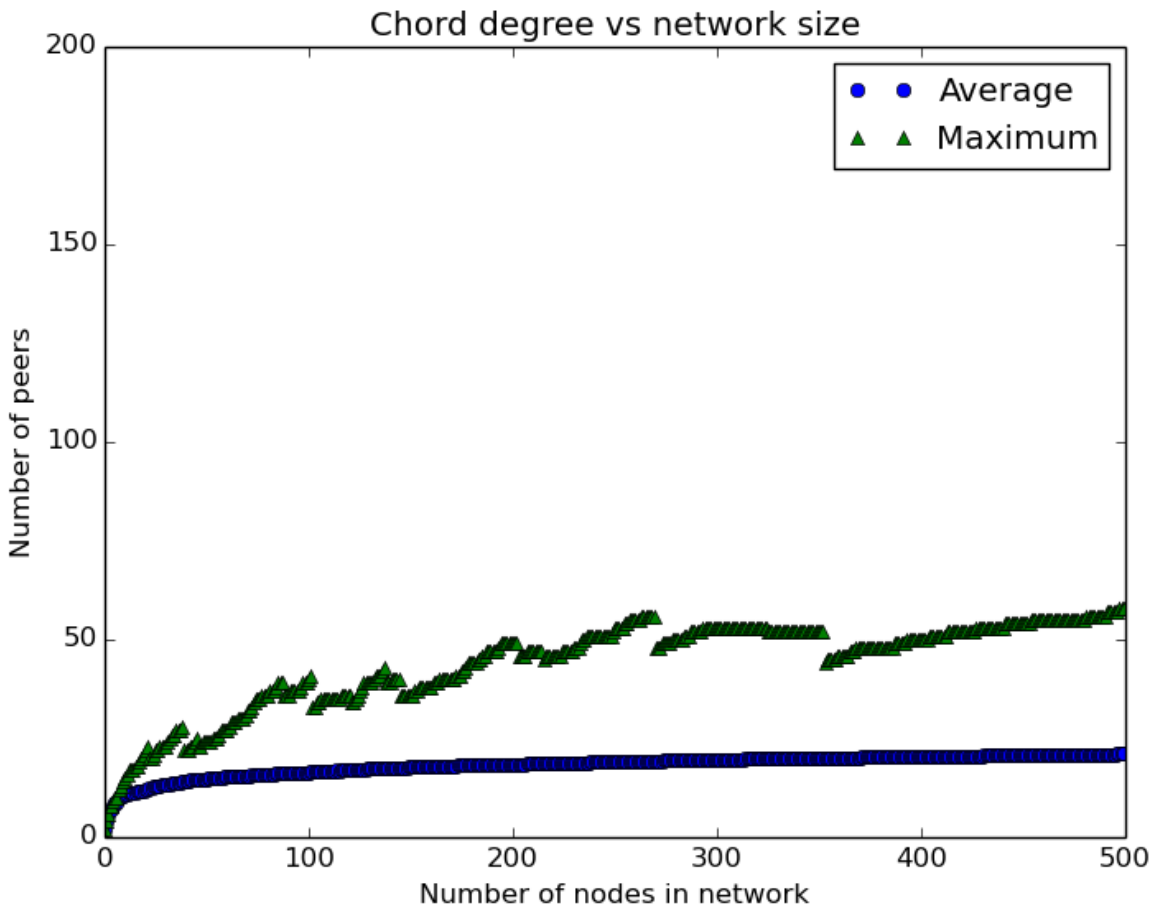


Figure: This is the average and maximum degree of nodes in the Chord network. This Chord network utilized a 120 bit hash and thus degree is bound at 122 (full fingers, predecessor and successor) when the network reaches 2^{120} nodes.

Chord Distance

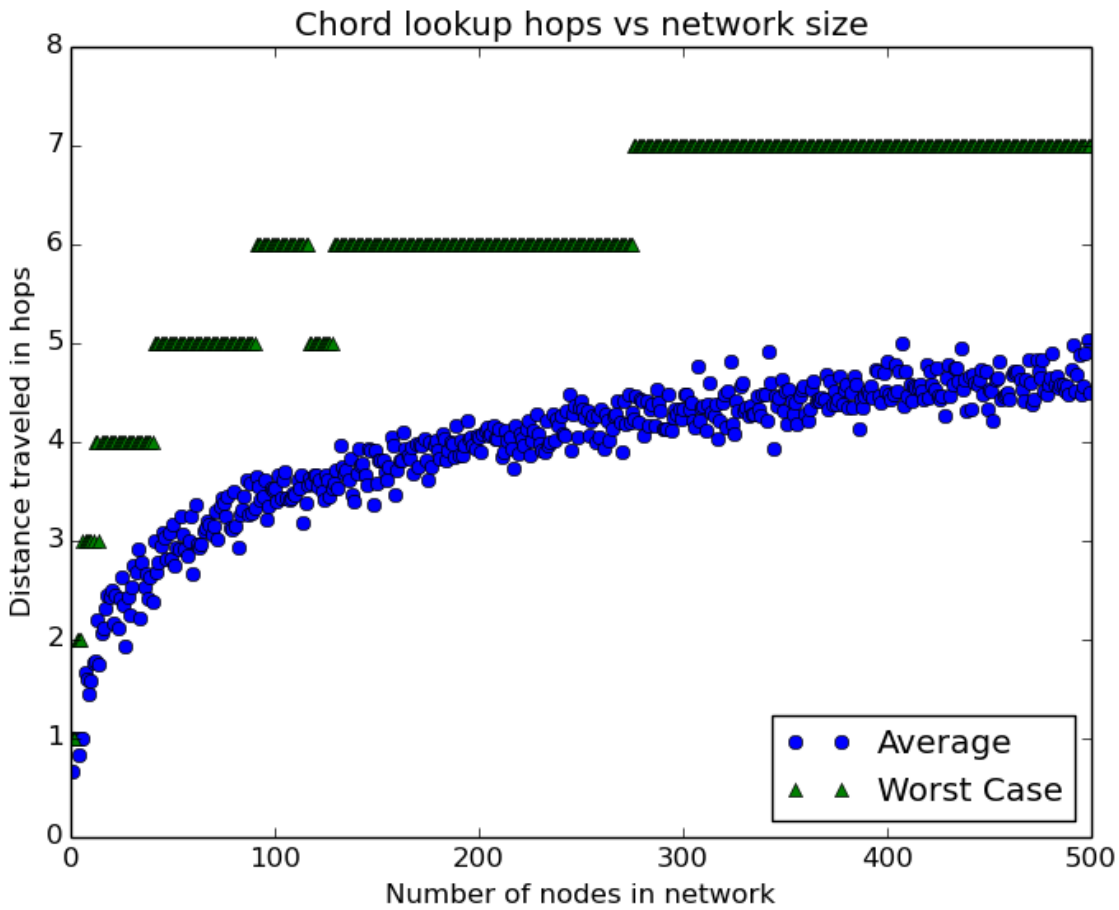


Figure: This is the number hops required for a greedy routed lookup in Chord. The average lookup between two nodes follows the expected logarithmic curve.

Kademlia Degree

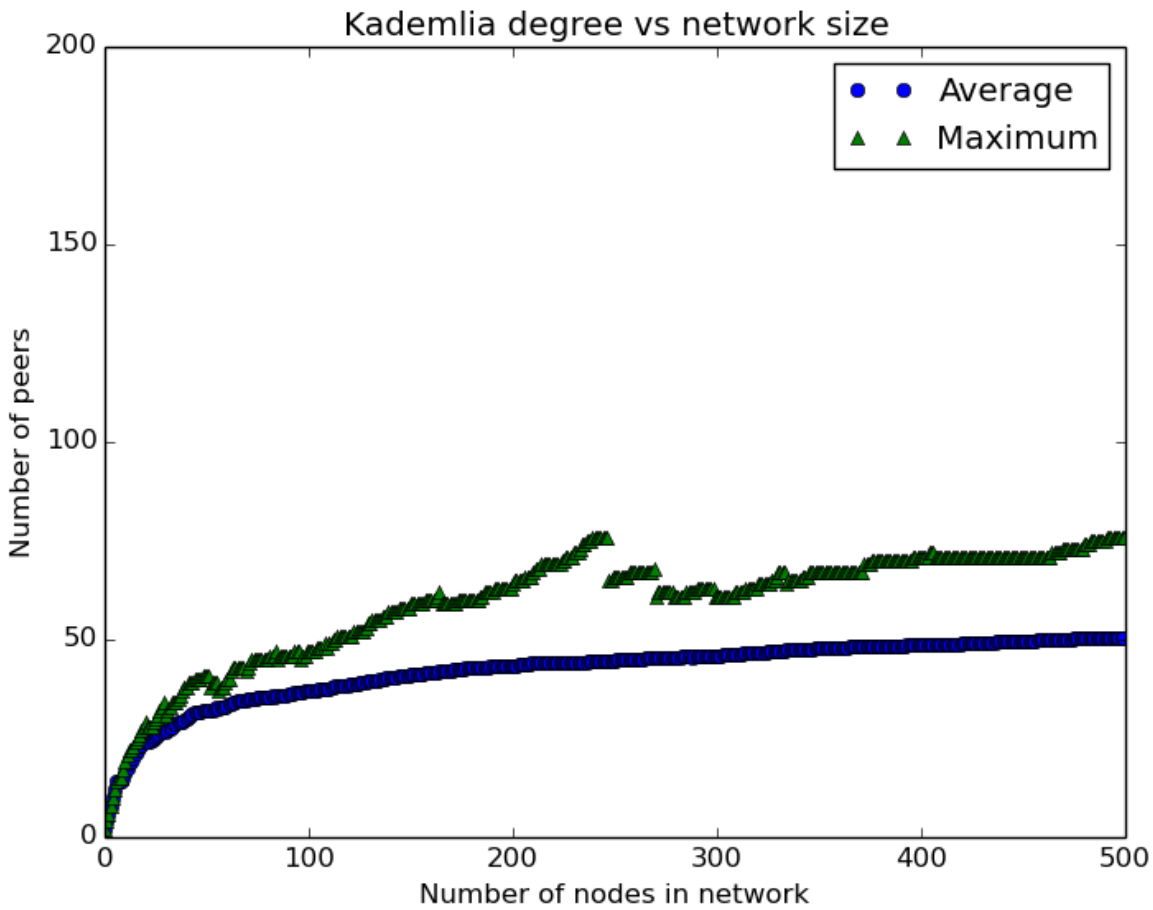


Figure: This is the average and maximum degree of nodes in the Kademlia network as new nodes are added. Both the maximum degree and average degree are $O(\log n)$.

Kademlia Distance

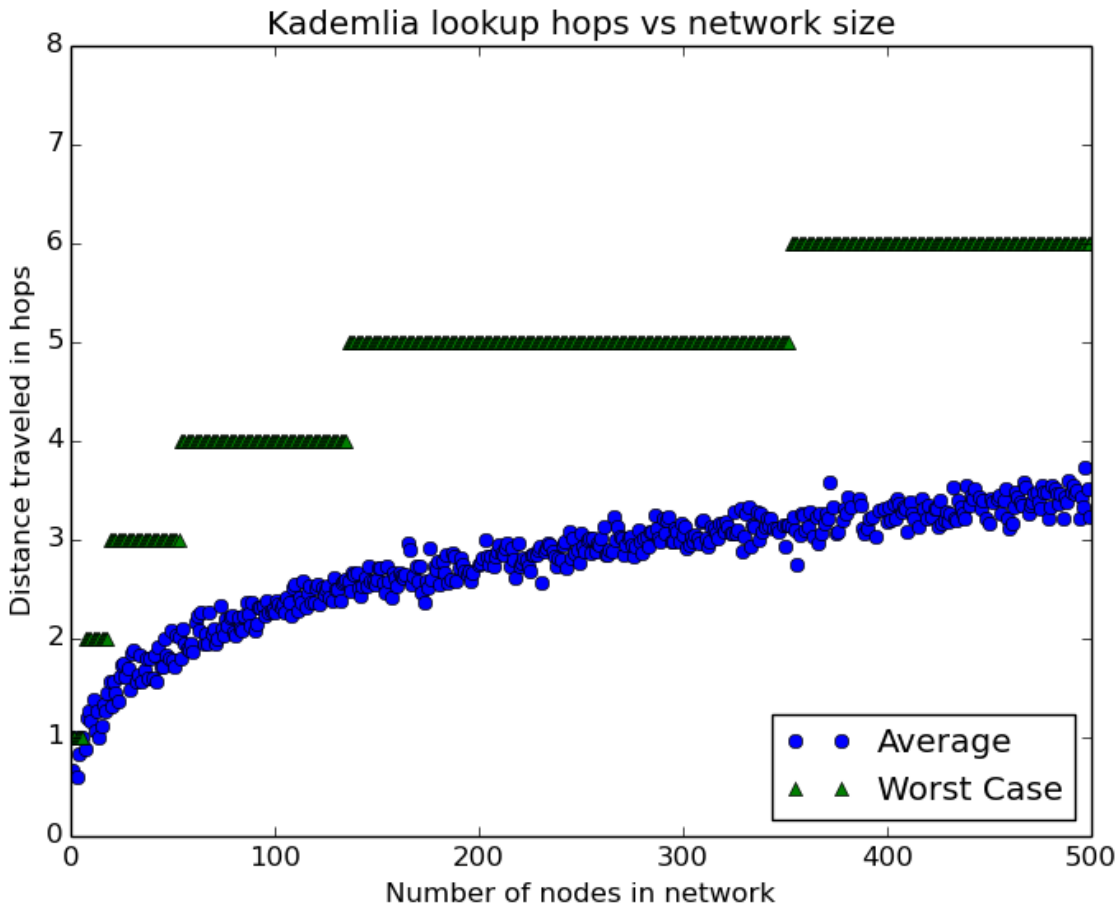


Figure: Much like Chord, the average degree follows a distinct logarithmic curve, reaching an average distance of approximately three hops when there are 500 nodes in the network.

Euclidean Degree

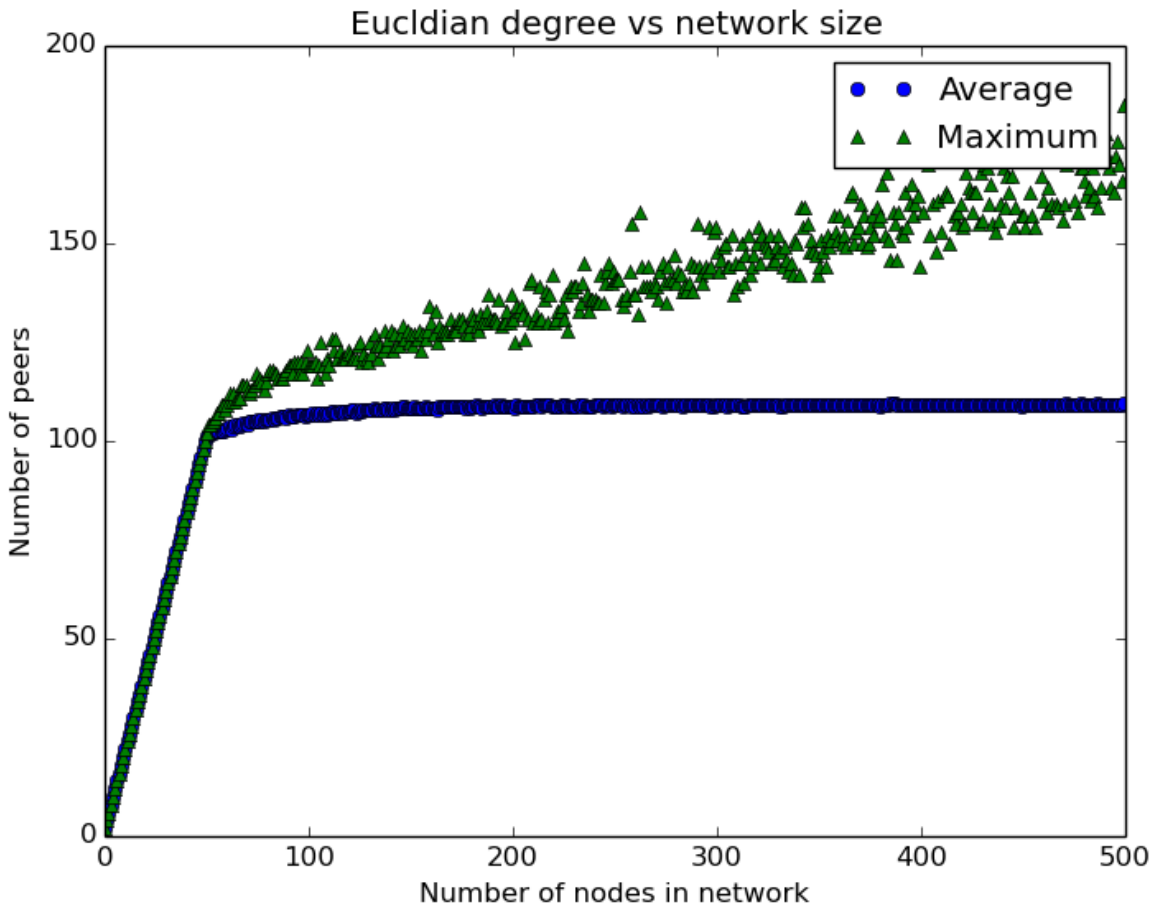


Figure: Because the long peers increase linearly to the maximum value (49), degree initially rises quickly and then grows more slowly as the number of long peers ceases to grow and the size short peers increases with network size.

Euclidean Distance

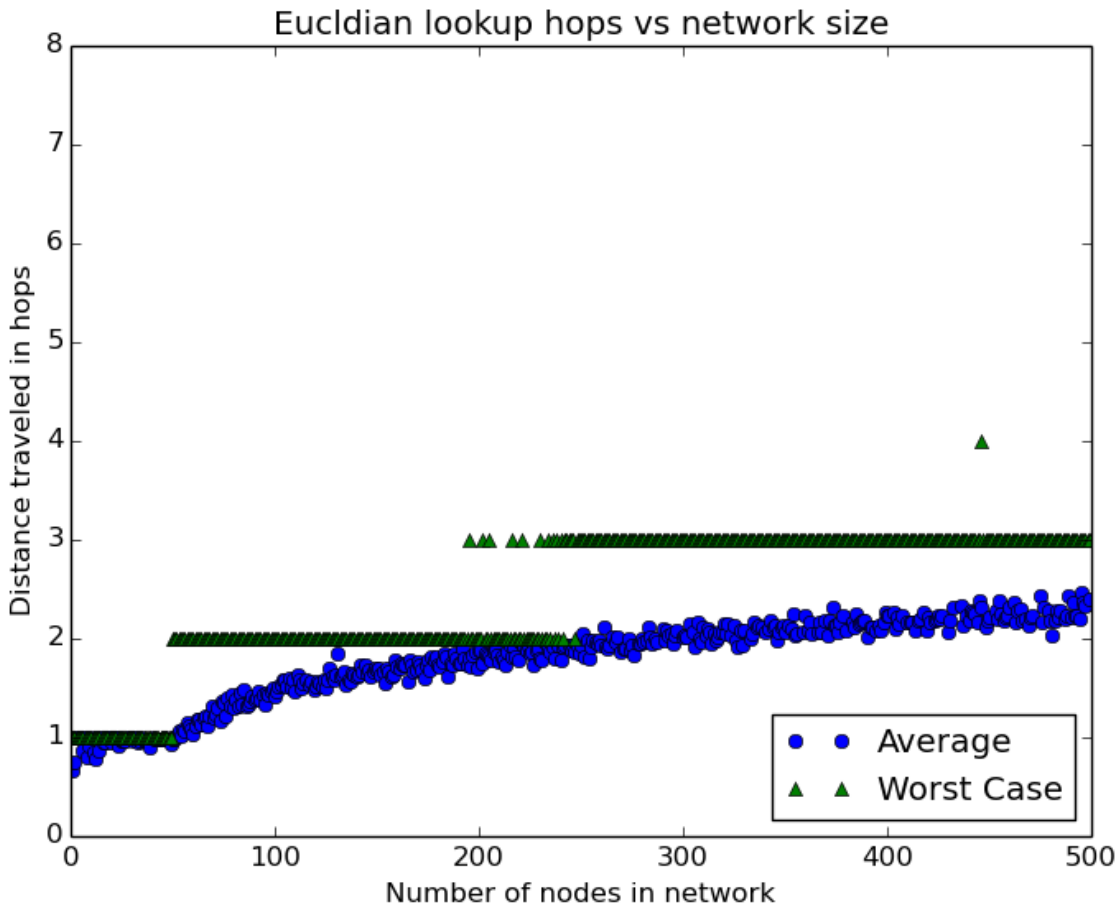


Figure: The inter-node distance stays constant at 1 until long peers are filled, then rises at the rate of a randomly connected network due to the distribution of long peers selected

Hyperbolic Degree

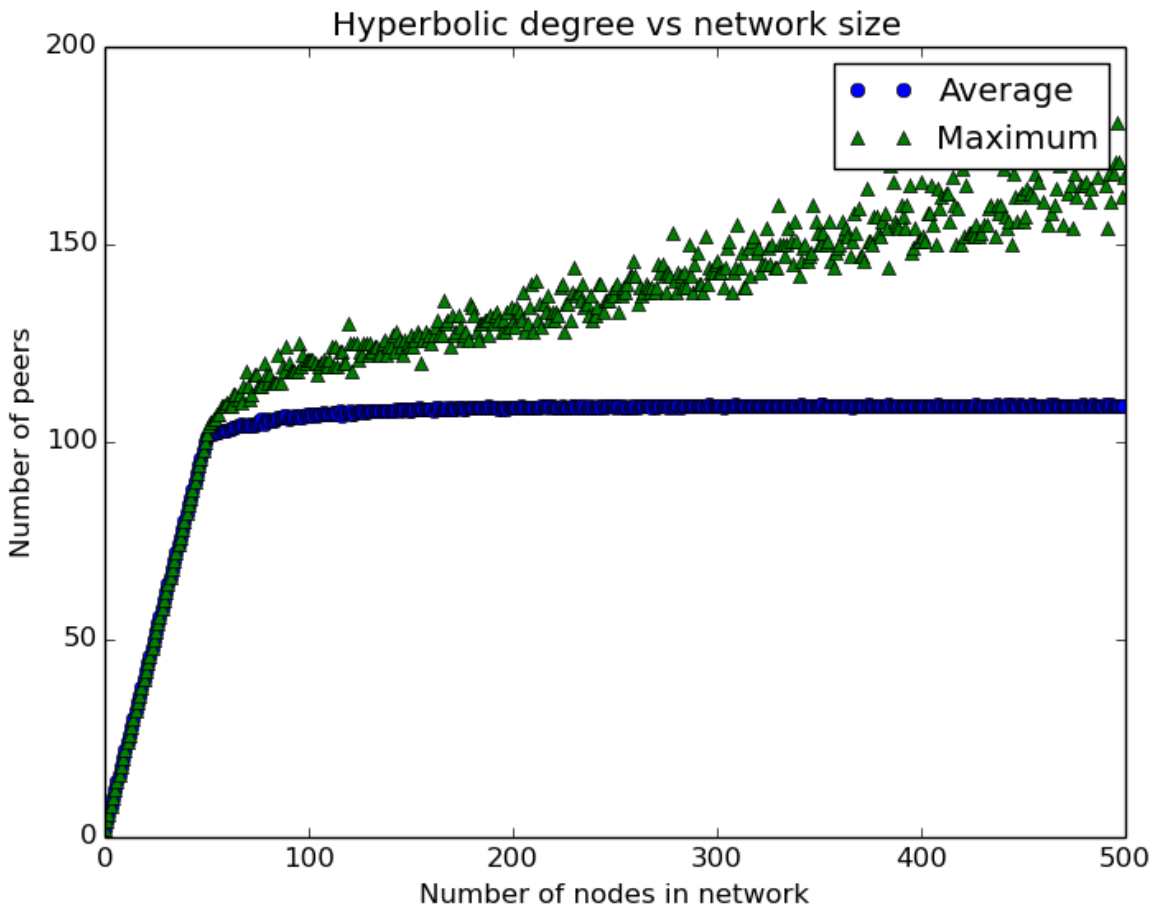


Figure: The Hyperbolic network uses the same long and short peer strategies to the Euclidean network, and thus shows similar results.

Hyperbolic Distance

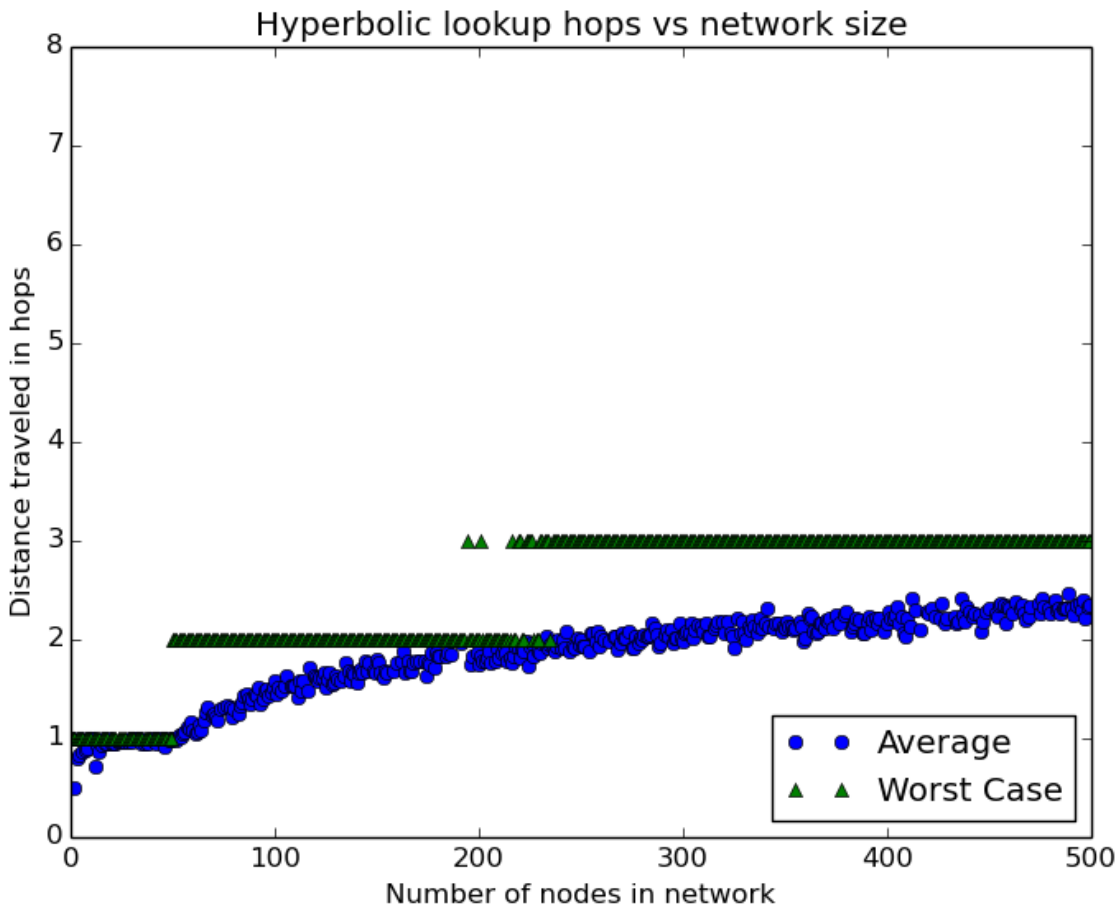


Figure: Like the Euclidean Geometry, our Poincarè disc based topology has much shorter maximum and average distances.

Conclusion

- UrDHT abstracts DHTs.
- UrDHT can emulate the topology and performance of various DHTs.
- We can use UrDHT to create a multidimensional DHT.
- The maintenance algorithm can nodes moving in the overlay.

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

Strategies

- Churn
- Random Injection
- Neighbor Injection
- Invitation

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 ($\frac{tasks}{nodes}$). 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

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

- This means 50% of the network has significantly less tasks than the averag
  Probably more.

Distribution of Work in A DHT

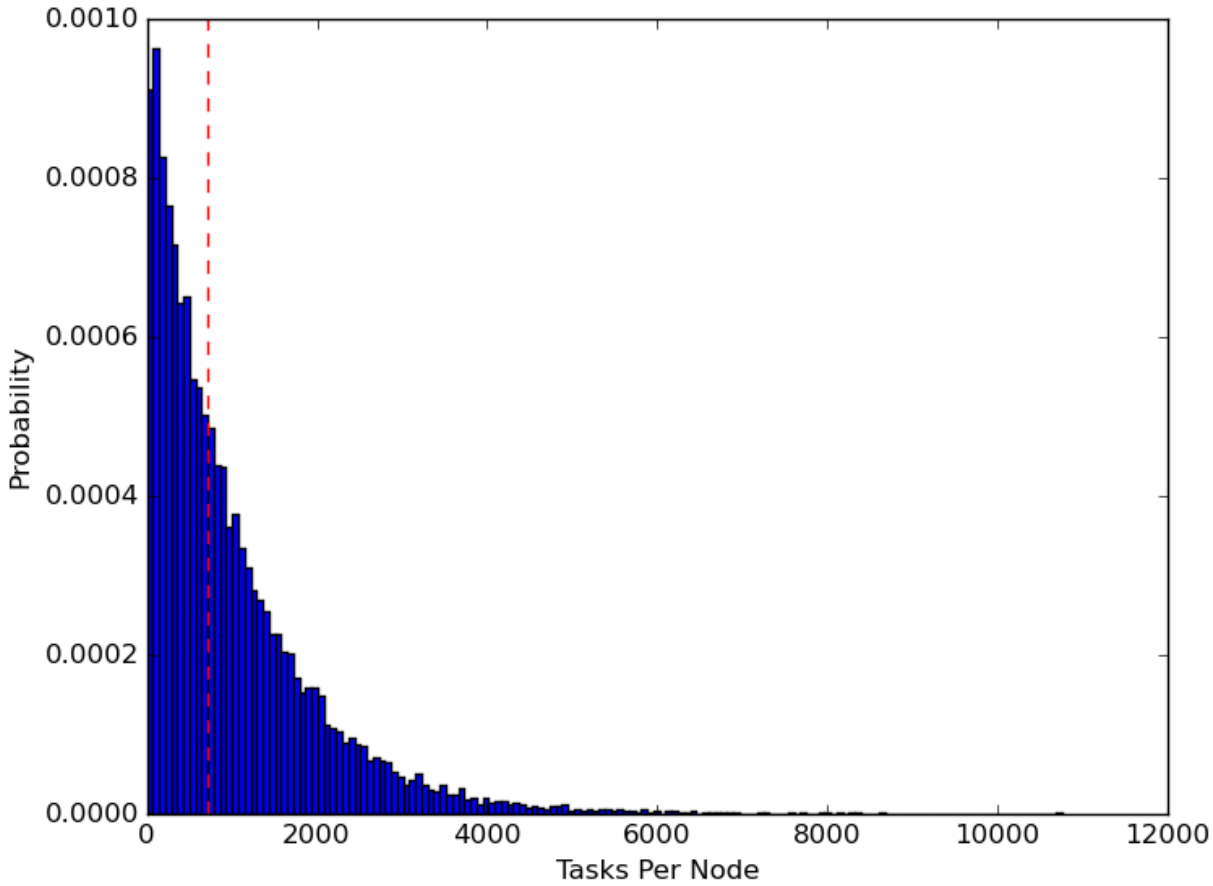


Figure: The probability distribution of workload in a DHT with 1000 nodes and 1,000,000 tasks or files. The vertical dashed line designates the median.

Distribution of Work in A DHT

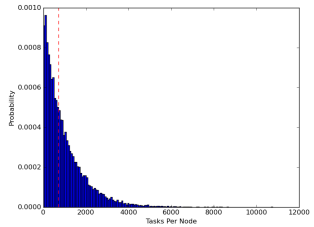


Figure: The probability distribution of workload in a DHT with 1000 nodes and 1,000,000 tasks or files. The vertical dashed line designates the median.

- Keys were generated by feeding random numbers into the SHA1 hash function, a favorite for many distributed hash tables.
- Nodes with small amounts of work (the first two buckets) will finish first.
- If each node consumes a task a tick, then that means the nodes the furthes right will dictate the runtime.

Distribution of Work in Chord

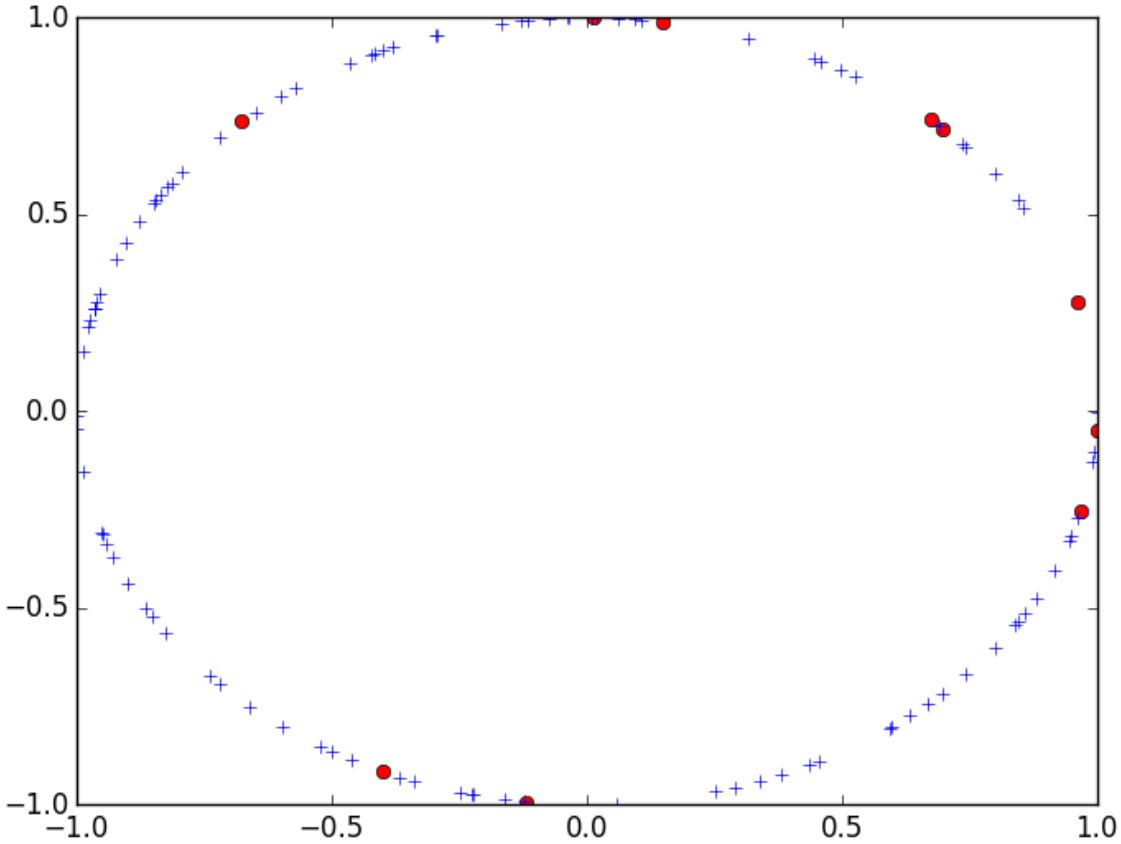


Figure: A visual example of data and nodes in a Chord DHT with 10 nodes (represented by red circles) and 100 tasks (blue pluses).

Distribution of Work in Chord

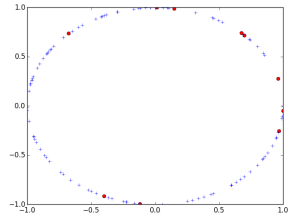


Figure: A visual example of data and nodes in a Chord DHT with 10 nodes (represented by red circles) and 100 tasks (blue pluses).

- Nodes are responsible for all the tasks that fall along the perimeter between themselves and their predecessor, which is the closest node counterclockwise
- Each node and task is given a 160-bit identifier `id` that is mapped to location (x, y) on the perimeter of the unit circle via the equations $x = \sin\left(\frac{2\pi \cdot id}{2^{160}}\right)$ and $y = \cos\left(\frac{2\pi \cdot id}{2^{160}}\right)$.
- Note that some of the nodes cluster right next to each other, while other nodes have a relatively long distance between each other along the perimeter.
- The most blatant example of this is the node located at approximately $(-0.67, 0.75)$, which would be responsible for all the tasks between that an the next node located counterclockwise.
- That node and the node located at about $(-0.1, -1)$ are responsible for approximately half the tasks in the network.

Distribution of Work in Chord with Evenly Distributed Nodes

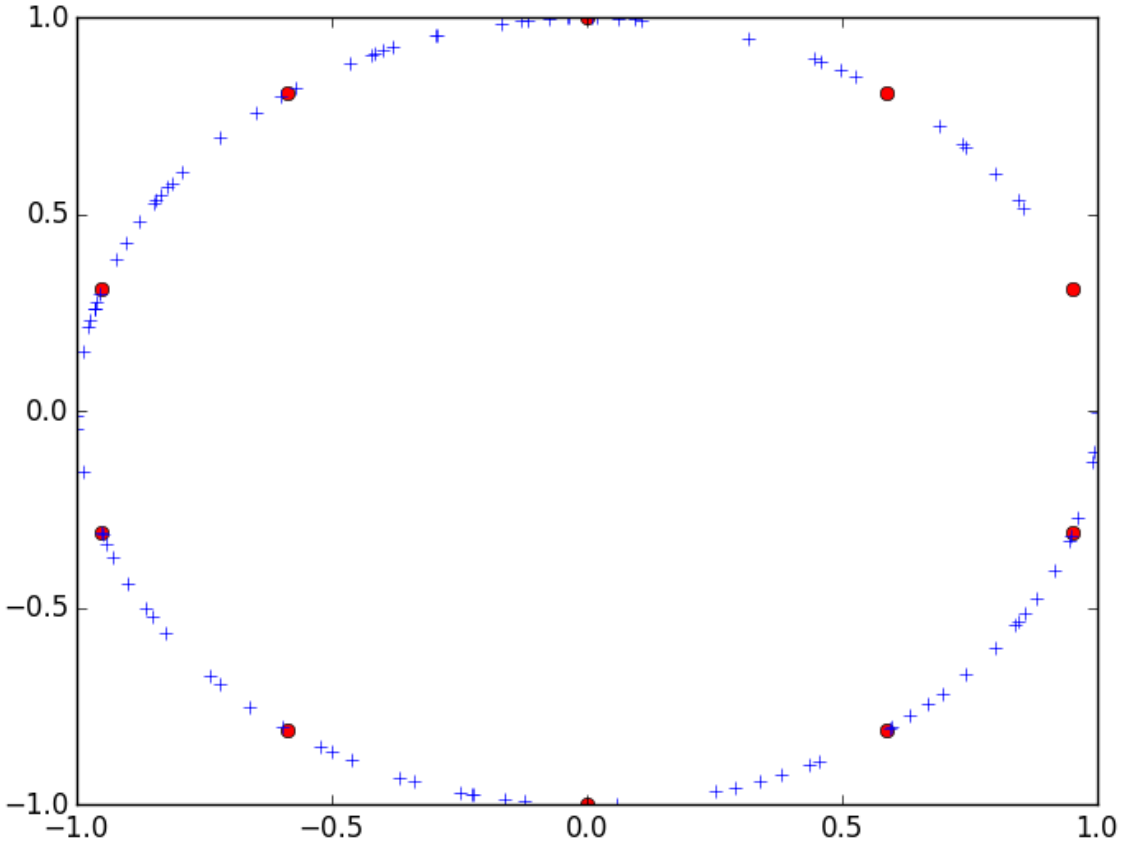


Figure: A visual example of data and nodes in a Chord DHT with 10 evenly distributed nodes (represented by red circles) and 100 tasks (blue pluses).

Distribution of Work in Chord with Evenly Distributed Nodes

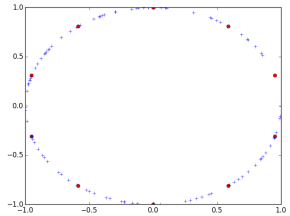


Figure: A visual example of data and nodes in a Chord DHT with 10 evenly distributed nodes (represented by red circles) and 100 tasks (blue pluses).

We see that while the network is better balanced, the files cluster and some nodes still end up with noticeably more work than others. It is possible for nodes to choose their own IDs in some DHTs, but the files will still be distributed according to a cryptographic hash function. In addition, it would require some additional centralization to make sure every single node covered the same range and may be impossible to coordinate such an effort in a constantly changing network.

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

We used these to calculate a "runtime factor," the ratio of the experimental runtime compared to the ideal runtime. For example, in the network from our previous example took 852 ticks to finish, its factor is 8.52. We prefer to use this runtime factor for comparisons since it allows us to compare networks of different compositions, task assignments, and other variables.

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

- 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

- Leaving nodes enter pool, joing nodes leave pool
- Check is done before work is done

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

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

- The magnitude of churn's effect varies based on the size of the network and the number of tasks.
- In networks where there are fewer nodes, we see the base runtime factor is smaller.
- The more tasks there are, the greater the gains of churn.
- A network 100 nodes and 1 million tasks, on average, has a runtime factor only 30% higher than ideal when churn is 0.01 per tick.
- The runtime for heterogeneous versus homogeneous networks had no significant differences. This won't always be the case.

Churn vs Runtime factor

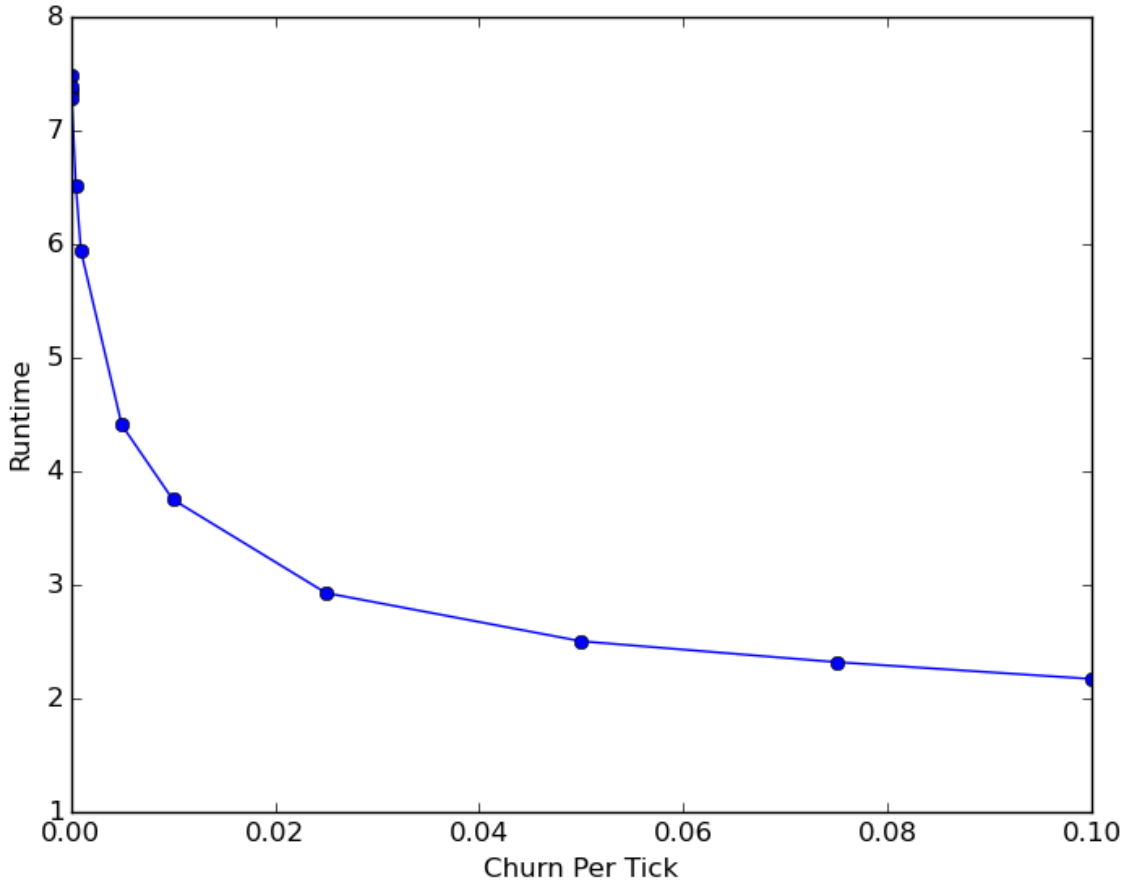


Figure: This graph shows the effect churn has on runtime in a distributed computation. Runtime is measured as how many times slower the computation runs than an ideal computation, where each node receives an equal number of

Base Work Distribution

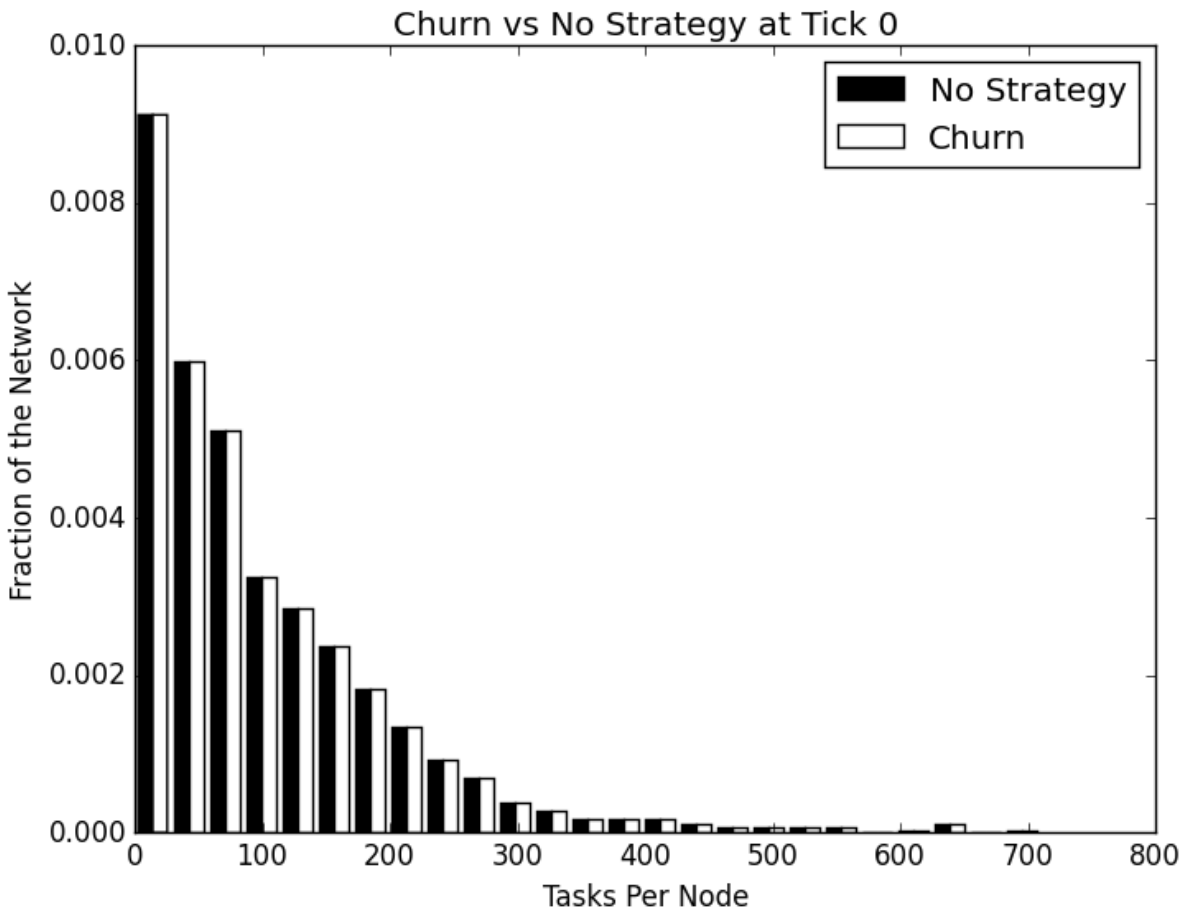


Figure: The initial distribution of the workload in both networks. As both networks start with same initial configuration, the distribution is currently identical. This greatly resembles the distribution we saw in Figure 10.

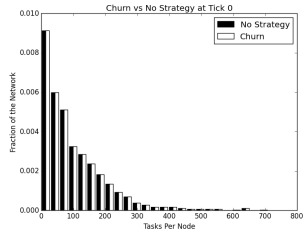


Figure: The initial distribution of the workload in both networks. As both networks start with same initial configuration, the distribution is currently identical. This greatly resembles the distribution we saw in Figure 10.

This will be the same for every strategy.

Churn Distribution after 35 Ticks

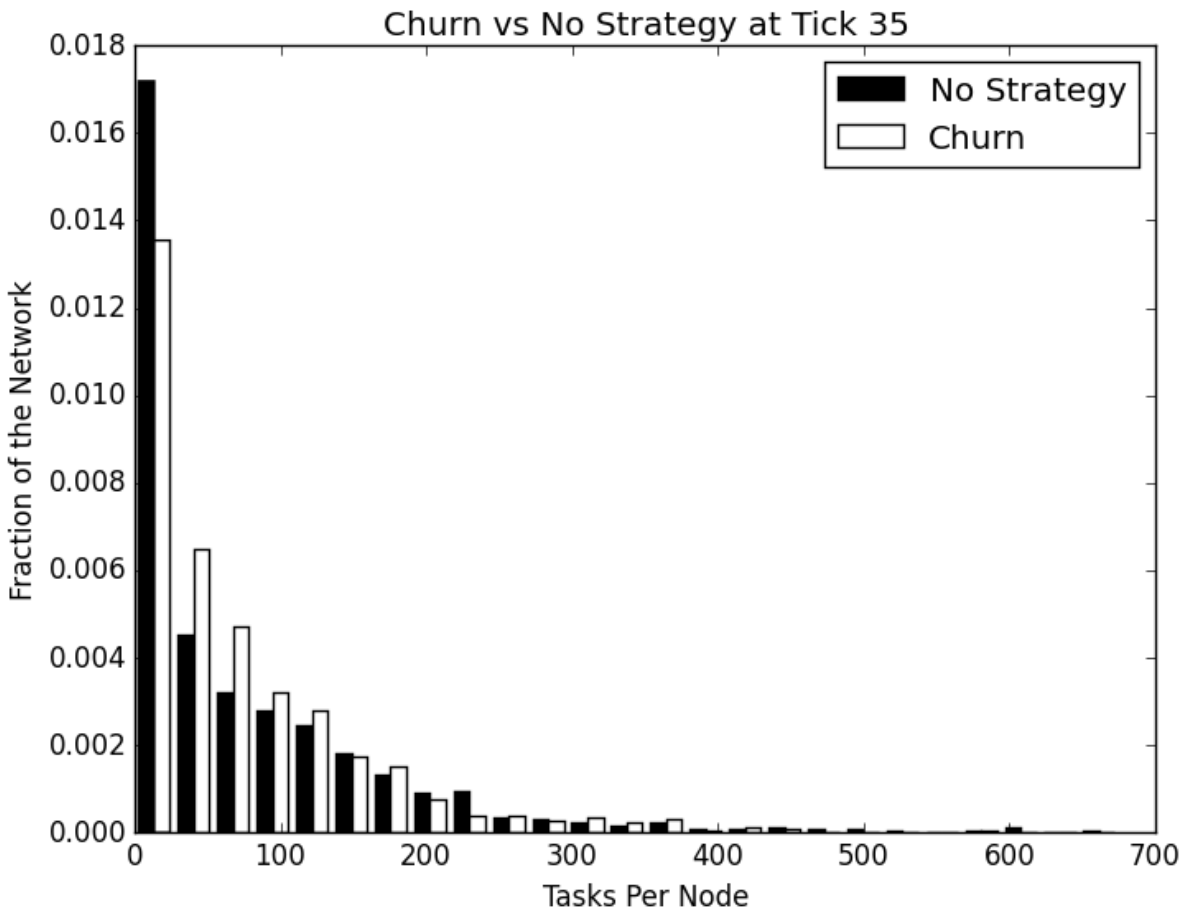


Figure: After 35 ticks, the effects of churn on the workload distribution become more pronounced. More nodes have consumed all their tasks and are simply idling, but significantly less in the network using churn.

Remarks

- Diminishing returns
- Maintenance costs can get excessive
- We don't actually have to kill nodes, most of the speedup is from joining.

Strategy

- Nodes with loads $\leq$ `sybilThreshold` create Sybils
- This check occurs every 5 ticks before work is performed
- 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

Strategy

- Nodes with loads $\leq$ `sybilThreshold` create Sybils
- This check occurs every 5 ticks before work is performed
- 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

- Leaving nodes enter pool, joing nodes leave pool
- Sybil removal is instantaneous if no qork was acquired. We can get a simila result with no disruption if a node asks if it would be stealing any work.

Effects of Network Size

- 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.

Effects of Network Size

- 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.

- Heterogeneous networks also saw signifcantly better performance , but the gains were not as great as in homogeneous networks.
- However, the larger ratio networks handled heterogeneity much better, with the worst average heterogeneous run time being 1.955 in networks with 100 tasks per node, compared to 4.052 on the smaller ratio networks with 100 tasks per node.
- Most trials did not have a runtime factor greater than 3.
- Random injection handled heterogeneity the best

Random Injection vs No Strategy After 35 ticks

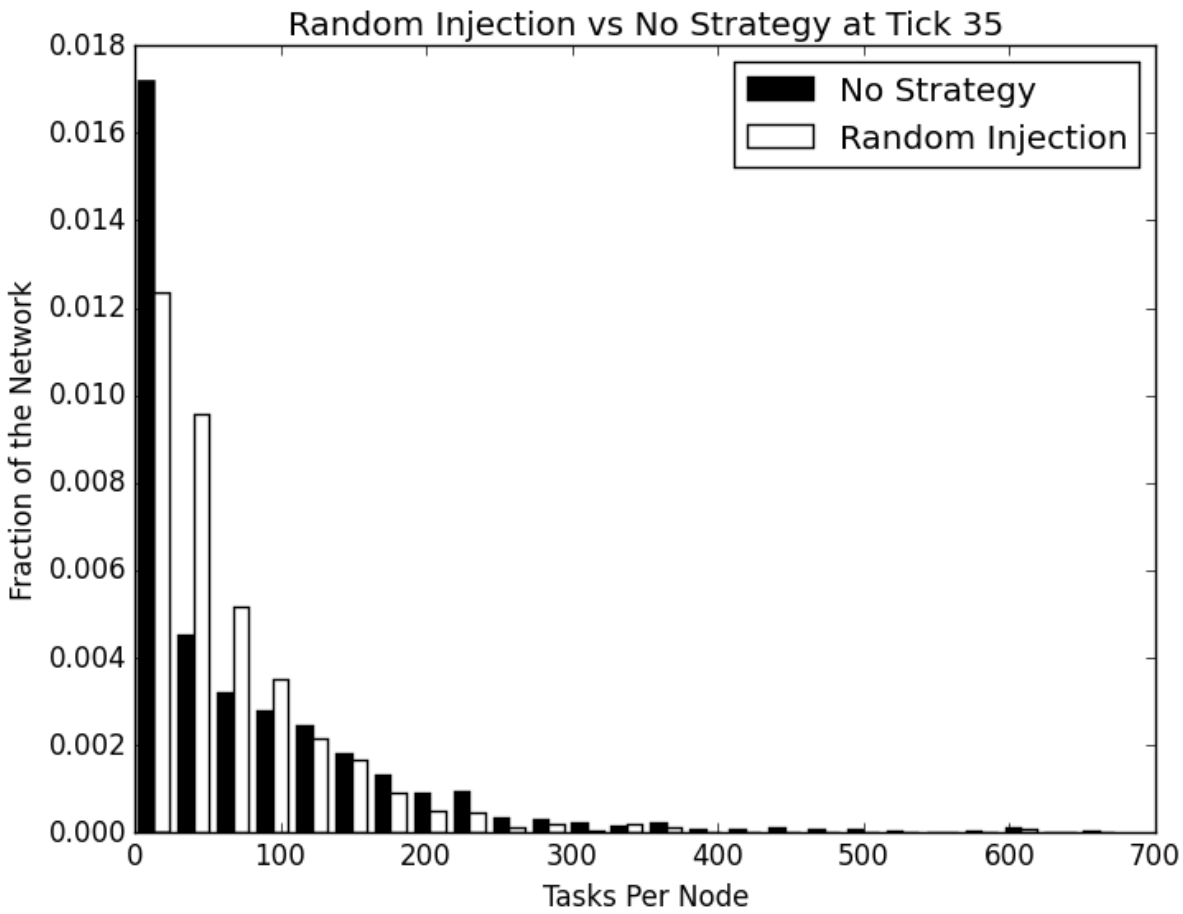


Figure: The networks after 35 ticks. The network using random injection has significantly less underutilized nodes and substantially more notes with some or lots of work.

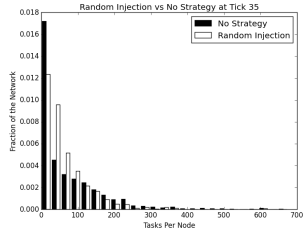


Figure: The networks after 35 ticks. The network using random injection has significantly less underutilized nodes and substantially more notes with some or lots of work.

- We see even better improvement here.

Random Injection in a Heterogeneous Network

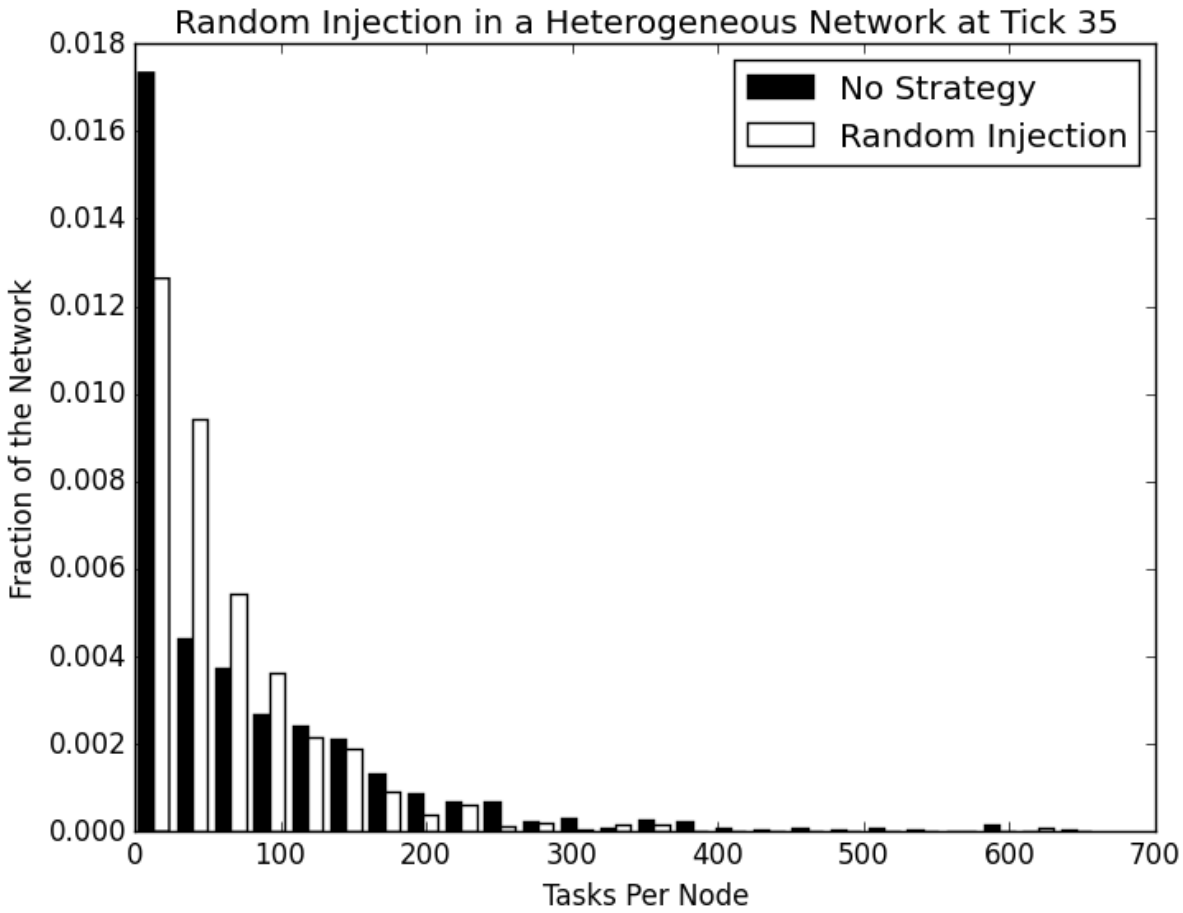


Figure: The workload distribution of heterogenous networks after 35 ticks.

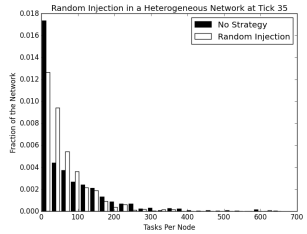


Figure: The workload distribution of heterogenous networks after 35 ticks.

- We can see the network using the random injection strategy is experiencing a better distribution of work.
- This did not translate to better runtime factor, as we'll in in other experiments.

Random Injection VS Churn

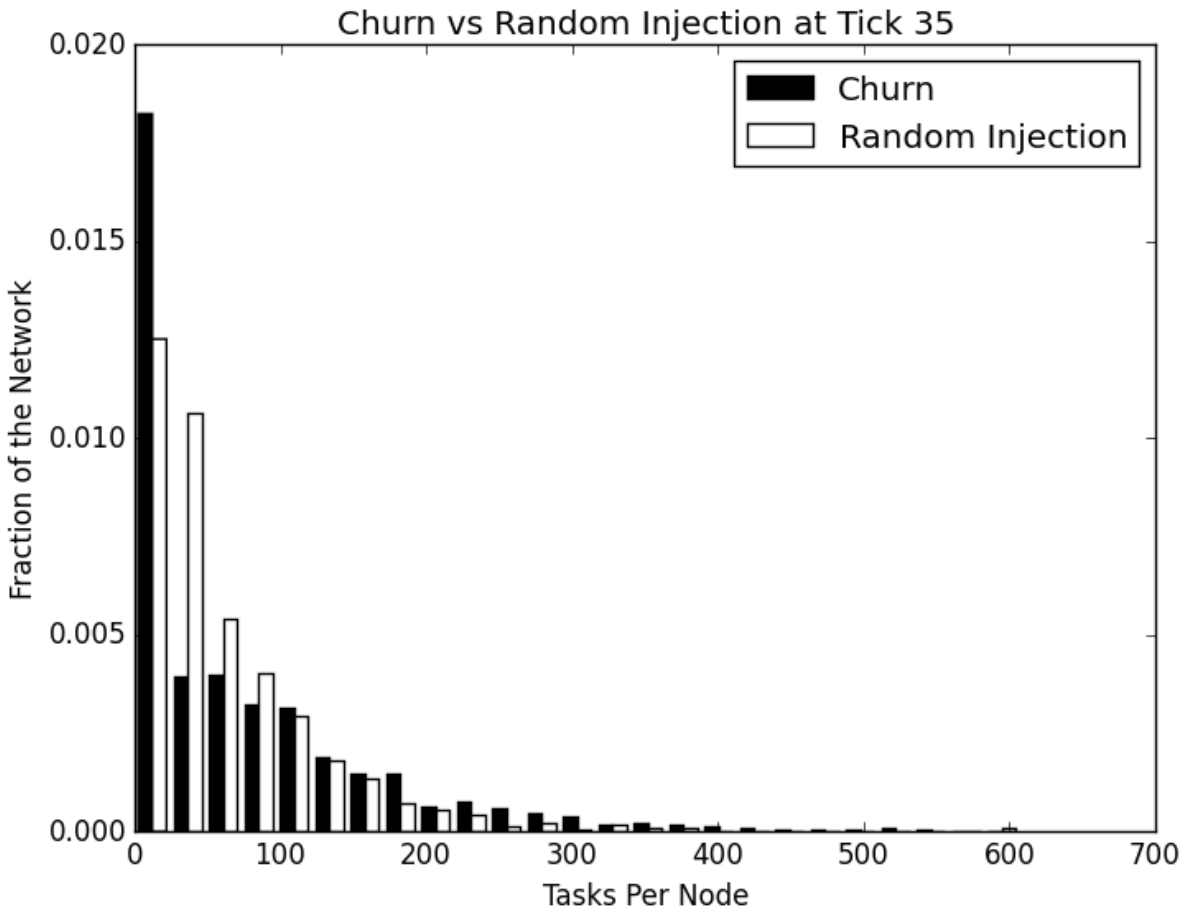


Figure: The networks after 35 ticks. The network using random injection load-balances significantly better than the network using Churn.

Impacts of Variables

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

Impacts of Variables

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

- `sybilThreshold` did nothing in heterogeneous networks. The improvemen was tied to the ratio of tasks to nodes.
- Heterogeneous networks were hurt by `maxSybils` being larger. The wider the disparity in strength, the more the network was hurt.

Remarks

- Best runtime factor of our experiments.
- Could still incur high maintenance costs, especially with nodes being deleted as soon as they are made.

- Gets extremely close to ideal.
- Query first

- Best runtime factor of our experiments.
- Could still incur high maintenance costs, especially with nodes being deleted as soon as they are made.

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.

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.

- Smart actually queuries
- Estimation can cause a node to keep checking same space each tick. Trivia to solve.
- Otherwise same as random injection

Base Runtime

- The base runtime in a 1000 node/100,000 task homogeneous network was 5.033
- 2.4 lower than no strategy
- Base runtime in a heterogeneous runtime was worse
- `numSuccessors` improves the runtime factor (0.3 for base network)
- Other variables had no significant effect.

- The base runtime in a 1000 node/100,000 task homogeneous network was 5.033
- 2.4 lower than no strategy
- Base runtime in a heterogeneous runtime was worse
- `numSuccessors` improves the runtime factor (0.3 for base network)
- Other variables had no significant effect.

This is because the network is finishing off the tasks with small amounts of work quicker. However, nodes are not able to acquire work outside their immediate vicinity and must idle. In addition, nodes always inject Sybils into the largest gap between their successors that they see, but if they acquire no work. That means in the simulation, it is possible for nodes to get into a loop of constantly checking the largest gap and miss other neighbors that do have work to acquire.

Neighbor Injection after 35 Ticks

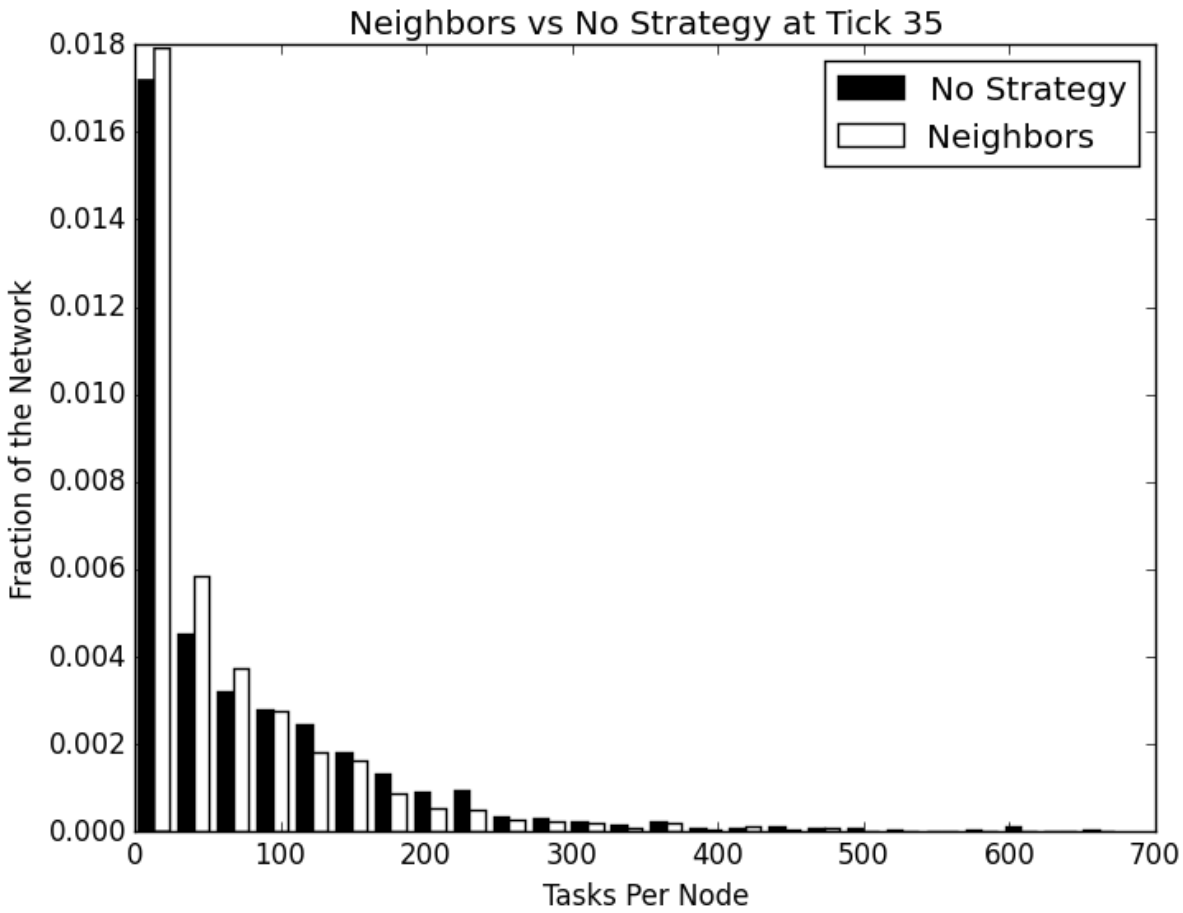


Figure: Despite have more idling nodes, we see that the nodes using the neighbor injection strategy have acquired smaller workloads and have effectively shifted part of the histogram left.

Smart Neighbor Injection after 35 Ticks

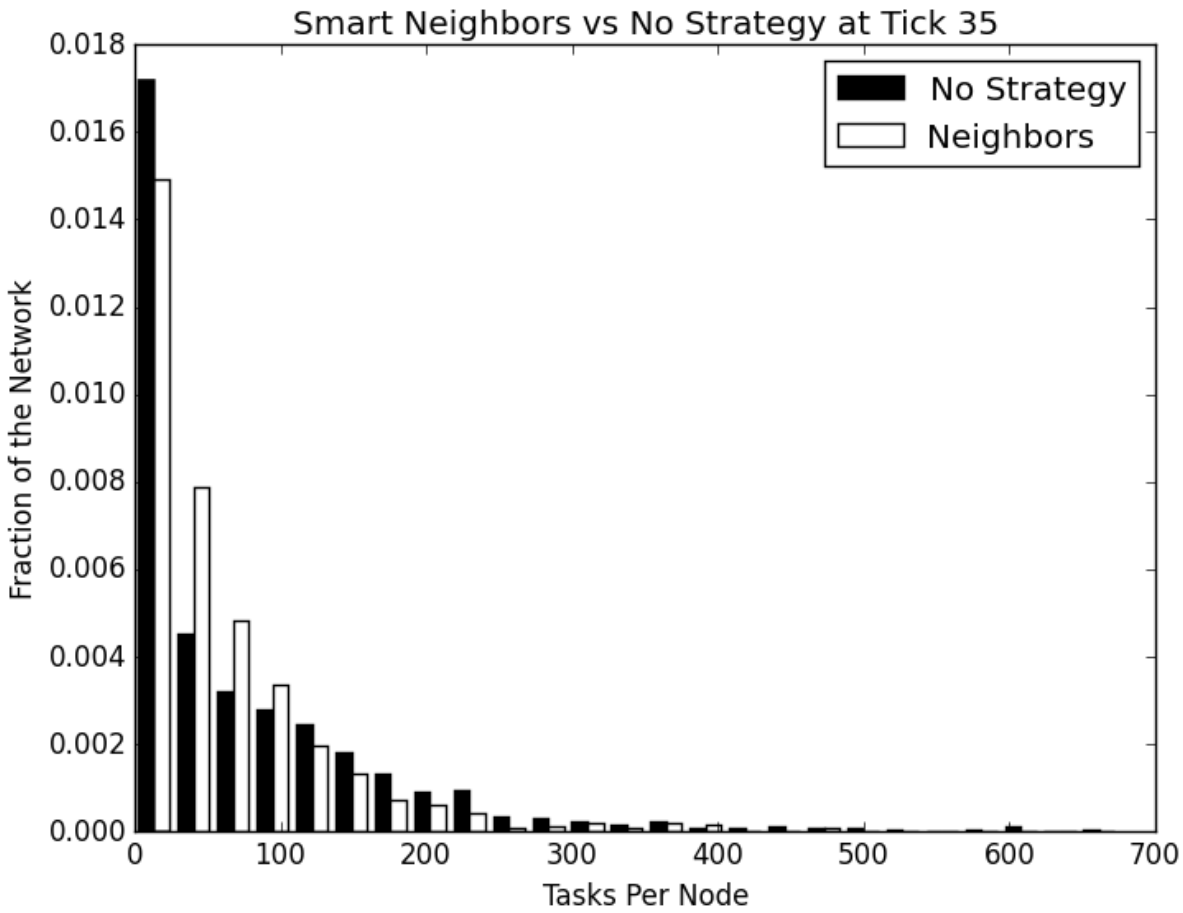


Figure: After 35 ticks, we see the network using the smart neighbor injection strategy has significantly less nodes with little or no work, more nodes with smaller amounts of work, and less nodes with large amounts of tasks.

Remarks

- Smart method improved runtime factor by 1.2 on average.
- Smart would require querying, "dumb" estimation still would provide improvement.
- Less churn of joining nodes.

Strategy

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

Strategy

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

- Only checks to create more work when a node is actually over loaded
- This means less traffic
- Random and neighbors try to acquire work proactively
- They spam new sybils in the hopes of finding a place to grab work from
- Invitation is reactive, overloaded nodes react and ask for sybils
- less churn

Invitation at 35 Ticks

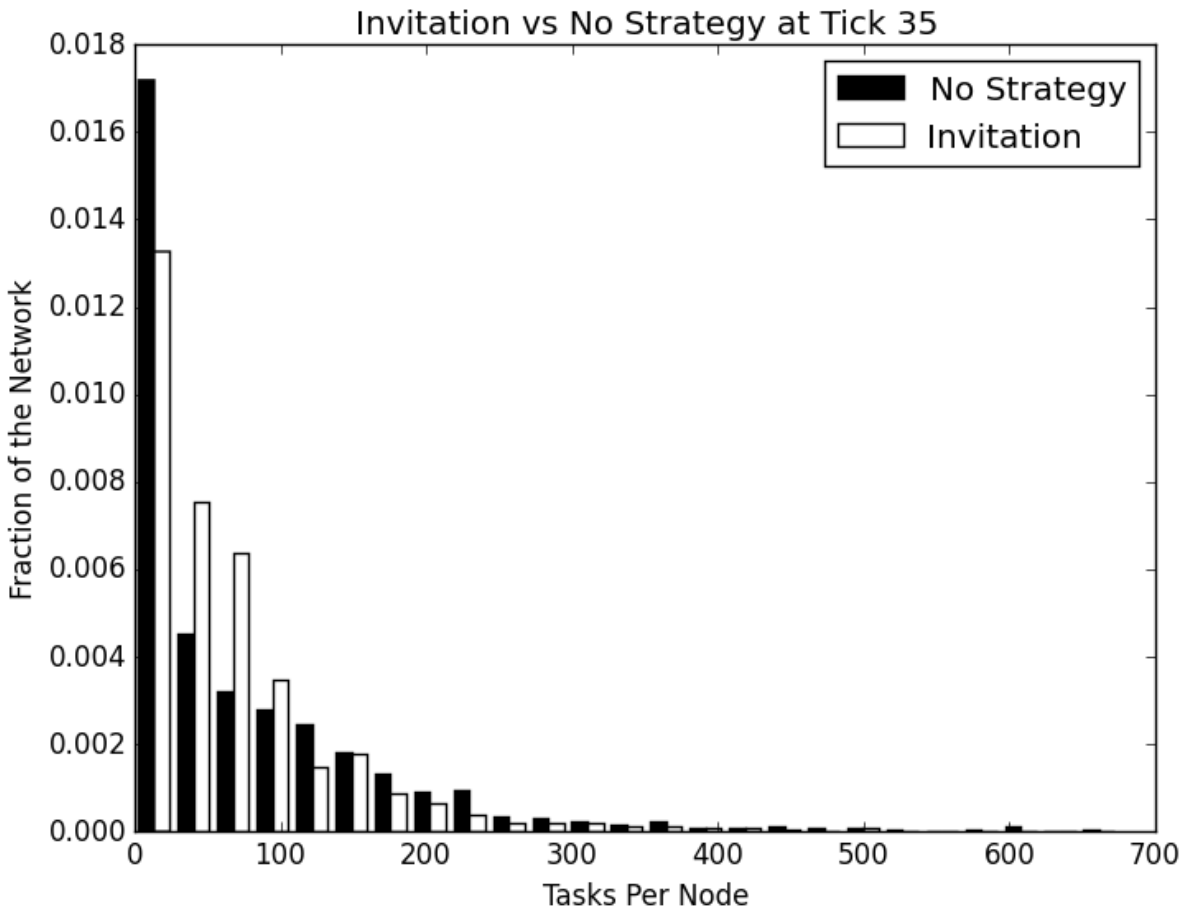


Figure: At 35 ticks, we can see the network using the invitation strategy perform markedly better than the network using no strategy. The highest load is around 500 tasks in the network using invitation, compared to approximately 650 tasks in

Invitation vs Smart Neighbors at Tick 35

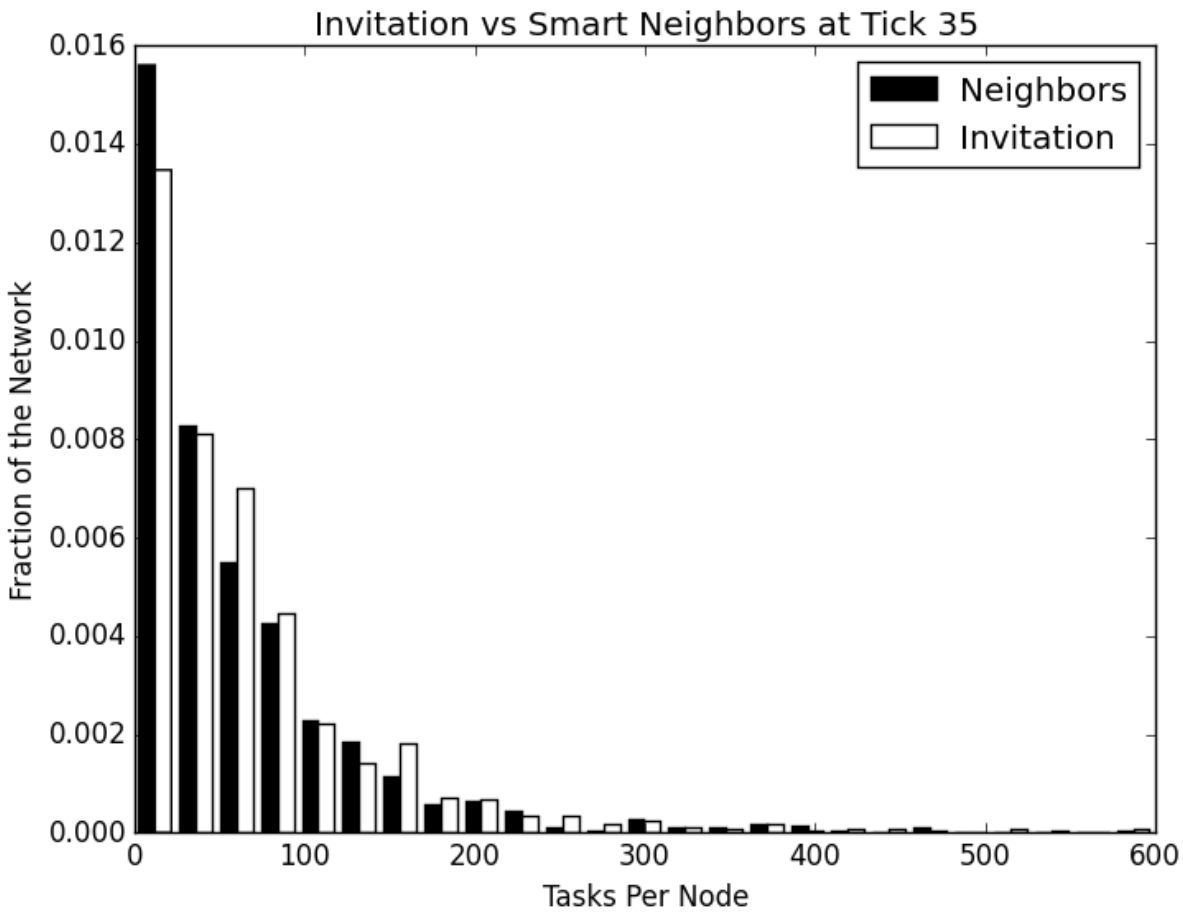


Figure: After 35 ticks, differences between the two strategies have emerged. The network using the invitation strategy has significantly less nodes with a small work load and many more with large work loads.

Remarks

- Impact of Invitation was closely tied to the number of nodes in the network.
  - 100 node/ 100,000 task network (1000 tasks per node), the base average runtime factor was 3.749.
  - 1000 node/ 100,000 task network had a base average runtime of 5.673.
- Performed poorer in heterogeneous networks, but better than base on average (6.097 vs 7.5).
- Better than smart neighbors and uses less bandwidth.

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

Conclusion

- Load Balancing

Publication List

- Load Balancing