Autonomous Load Balancing in Distributed Hash Tables Using Churn and The Sybil Attack

Andrew Rosen Brendan Benshoof Robert W. Harrison Anu G. Bourgeois
Department of Computer Science
Georgia State University
Atlanta, Georgia
rosen@cs.gsu.edu bbenshoof@cs.gsu.edu rharrison@cs.gsu.edu anu@cs.gsu.edu

***Abstract*—.....**

***Index Terms*—Distributed Computing; Decentralized Systems; Peer-to-Peer Networks; Distributed Hash Tables; Self-Organizing Networks; Load-Balancing; Sybil Attack;**

I. Introduction

What are DHTs

Distributed Hash Tables (DHTs) are a class of well researched decentralized key-value storage systems. DHTs are most frequently used to construct the overlays of P2P file-sharing systems, such as the incredibly popular BitTorrent [3]. DHTs have also been used in many unorthodox applications, such as machine learning [6] and, most relevant for our discussion, distributed computing [11].

Our previous research [11] examined using a DHT as the organizing mechanism for a distributed platform. This enabled us to create an exceptionally fault tolerant distributed computing platform that was easy to setup and could be run in a completely decentralized P2P environment.

What is the significance of the hash function

One of key components of a Distributed Hash Table is a cryptographic hash function, most commonly SHA1 [5]. Distributed Hash Tables rely on this cryptographic hash function to generate identifiers for data stored on the network. The cryptographic hash of the filename or other identifier for the data is used as the location of the file or data in the network. It can also be used to generate the identifiers for nodes in the network.

Ideally, inputing random numbers into a cryptographic hash function should produce a uniformly distributed output. However, this is impossible in practice [2] [15].

In practice, that means given any DHT with files and nodes, there will be an inherent imbalance in the network. Some nodes will end up with a lion's share of the keys, while other will have few responsibilities (Table I).

This makes it especially disheartening to try and ensure as even a load as possible. We cannot rely on a centralized strategy to fix this imbalance, since that would violate the principles and objects behind creating a fully decentralized and distributed system.

Therefore, if we want to create strategies to act against the inequity of the load distribution, we need a strategy that individual nodes can act upon autonomously. These strategies need to make decisions that a node can make at a local level, using only information about their own load and the topology they can see.

Motivation

The primary motivation for us is creating a new viable type of platform for distributed computing. Most distributed computing paradigms, such as Hadoop [1], assume that the computations occur in a centralized environments. One enormous benefit is a centralized system has much greater control in ensuring load-balancing.

However, in an increasingly global world where computation is king and the Internet is increasingly an integral part of everyday life, single points of failure failures quickly become more and more risky. Users expect their apps to work regardless of any power outage affecting an entire region. Customers expect their services to still be provided regardless of any. The next big quake affecting the the San Andreas fault line is a matter of when, not if. Thus, centralized systems with single points of failure become a riskier option and decentralized, distributed systems the safer choice.

Our previous work in ChordReduce [11] focused on creating a decentralized distributed computing framework based off of the Chord Distributed Hash Table (DHT) and MapReduce. ChordReduce can be thought of a more generalized implementation of the concepts

of MapReduce. One of the advantages of ChordReduce can be used in either a traditional datacenter or P2P environment.[1] Chord (and all DHTs) have the qualities we desire for distributed computing: scalability, fault tolerance, and load-balancing.

Fault tolerance is of particular importance to DHTs, since their primary use case is P2P file-sharing, such as BitTorrent [3]. These systems experience high levels of churn– disruptions to the network topology as a result of nodes entering and leaving the network. ChordReduce had to have the same level of robustness against churn that Chord did, if not better.

During our experiments with ChordReduce, we found that high levels of churn actually made our computations run *faster*. We hypothesized that the churn was effectively load-balancing the network.

Objectives

This paper serves to prove our hypothesis that inherent random churn can load balance a Distributed Hash Table. We also set out to show that we can use this in a highly controlled manner to greater effect.

We present three additional strategies that nodes can use to redistribute the workload in the network. Each of these three strategies relies on nodes creating virtual nodes, as a result they are represented multiple times in the network. The idea behind this is that a node with low work can create virtual nodes to seek out and acquire work from other nodes.

Consequently, we'll call our virtual nodes in this scenario Sybils for clarity and brevity. None of the strategies require a centralized organizer, merely a way for a node to check the amount of files or tasks it and its Sybils are responsible for.

We also want to show how distributed computing can be performed in a heterogeneous environment. In particular, we want to see if our strategies result in similar speedups in both homogeneous and heterogeneous environments.

II. How Work Is Distributed in DHTs: A Visual Guide

As we have previously mentioned, many DHTs use a cryptographic hash function to choose keys for nodes and data. However, no cryptographic hash function can uniformly distribute its outputs across its range. Table I shows the median workload for networks of different numbers of nodes and tasks. In this context, tasks and nodes are interchangeable, as each node in a DHT performing a distributed computation would receive one task per file or chunk of file.

[1]The other one being that new nodes could join during runtime and receive work from nodes doing computations.

Table I: 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

We can see that in a 1000 node/1,000,000 tasks network, the average median workload of a node is 692.3 tasks. If work were distributed equally among all nodes in this network, each node would have 1000 tasks. This means that 50% of the network has less significantly less tasks than the average number of tasks per node.

If we plot a histogram of how the probability of how work is distributed in this network, we gain a clearer picture (Figure 1). We see the bulk of the nodes have less than 1000 tasks and a few unfortunate nodes end up with more than 10,000 tasks.

Our final example is on a much smaller scale, but provides an excellent visual representation of a network. Figure 2 shows a visualization of 10 nodes and 100 tasks in a Chord overlay network.

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 and 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.

Figure 3 shows the same network, but the nodes are evenly distributed, rather than generated by the SHA1 hash function. 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

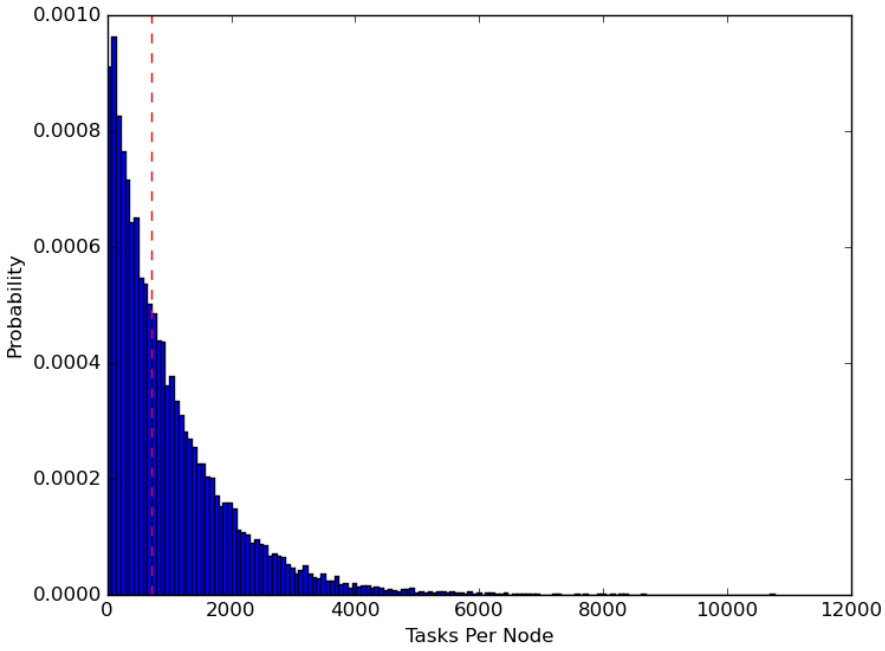

Figure 1: The probability distribution of workload in a DHT with 1000 nodes and 1,000,000 tasks or files. The vertical dashed line designates the median. Keys were generated by feeding random numbers into the SHA1 hash function [5], a favorite for many distributed hash tables.

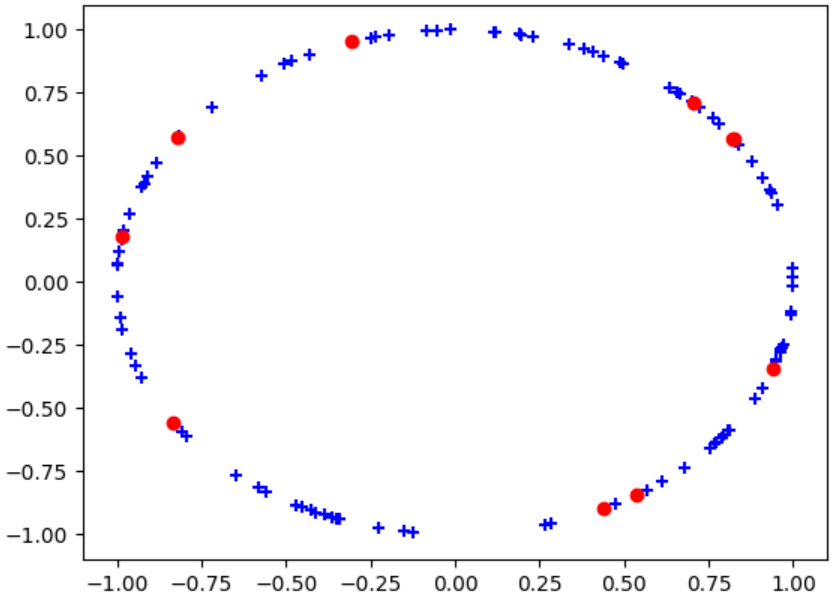

Figure 2: A visual example of data and nodes in a Chord DHT with 10 nodes (represented by red circles) and 100 tasks (blue pluses). Nodes are responsible for all the tasks that fall along the perimeter between themselves and their predecessor, which is the closest node counterclockwise.

for nodes to choose their own IDs in some DHTs [7], 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.

From these examples, we can see that nodes and data in a DHT are not distributed uniformly at random, but rather their distribution is better represented by an exponential distribution. In this setting, a few nodes are responsible for the bulk of the work, and in a distributed computation, it is these nodes that will account for the majority of the runtime in the network. As we will see in our simulations, if nodes with little or no work can acquire some of this work, it will have a significant impact on the network's runtime.

III. Simulation

We simulate our strategies on a Chord distributed hash table [14], although it is fairly straightforward to implement our strategies for other, more complex DHTs. Nodes in a Chord ring are given an ID, drawn from a cryptographic hash function, typically SHA1 [5]. Any data in the network is given a key in a similar manner. Nodes are responsible for all the keys between their predecessor's ID and their own.

We assume that the network starts our experiments stable and the data necessary is already present on the nodes and backed-up. The following analysis and simulation relies on an important assumption about DHT behavior often assumed but not necessarily implemented.

We assume that nodes are active and aggressive in creating and monitoring the backups and the data they are responsible for. We have demonstrated the effectiveness and viability of implementing an active backup strategy in other work [11] [13]. As previously mentioned, we also assume nodes have the ability to examine the amount of work they have and know how many tasks or pieces of data exist for a particular job.

Another assumption is that nodes do not have control in choosing their IDs from the range of hash values. This means that nodes cannot automatically change their spacing to ensure they are all evenly spread out across the network. Likewise, nodes cannot create necessarily create Sybils exactly where they would like, but would have to search for an appropriate ID in between two other nodes. We discussed this process in previous work and showed that it is extremely quick for a node to do so [12].

A. The Parameters

Our simulations relied on a great number of parameters and variables. We present each of them below.

1) Constants and Terminology: First, we'll define informally define the vocabulary we use to discuss our simulations.

Tick: In a simulation, normal measurements of time, such as a second, are completely arbitrary. We will be using an abstract *tick* to measure time. We consider the tick to be the amount of time it takes a node to complete one task (or more depending on our variables, see below) and perform the appropriate maintenance to keep the network consistent and healthy.

Maintenance: We assume nodes use the active, aggressive strategy from ChordReduce and UrDHT [11] [13]. Every maintenance cycle, each node checks and updates its list of neighbors (successors and predecessors in Chord) and responds appropriately. We assume that a tick is enough time to accomplish at least one maintenance cycle.

Task: We measure the size of a distributed computing job in tasks. Each task has a key that corresponds to the key of a file or chunk of a file stored on the network. We assume that it takes a tick for a node to complete at least one task.

IDs and Keys: We will be using SHA-1 [5], a 160-bit hash function, to generate node IDs and keys for tasks. We assume each task's key corresponds to some piece of data with the same key, a scheme we used in our previous work on ChordReduce [11].

2) Experimental Variables: We tested a number of strategies and variables that could affect each strategy. While we believed the overall strategy would result in largest differences in runtime, we wanted to see what effects, if any, each of the variables would have on a particular strategy.

Strategy: We use several different strategies: churn, random injection, neighbor injection, and invitation. Each of these strategies differs in how nodes attempt to autonomously balance the workload of the network. None of the strategies require centralized organization.

Homogeneity: This variable controls whether the network is homogeneous or not. In a homogeneous network, each node has the same strength, which dictates how much work is completed per a tick and how many Sybils it can create. In a heterogeneous network, each node has a strength chosen uniformly at random from 1 to `maxSybils`. The default value is homogeneous.

Work Measurement: This variable dictates how much work is consumed in a tick. Each node can either

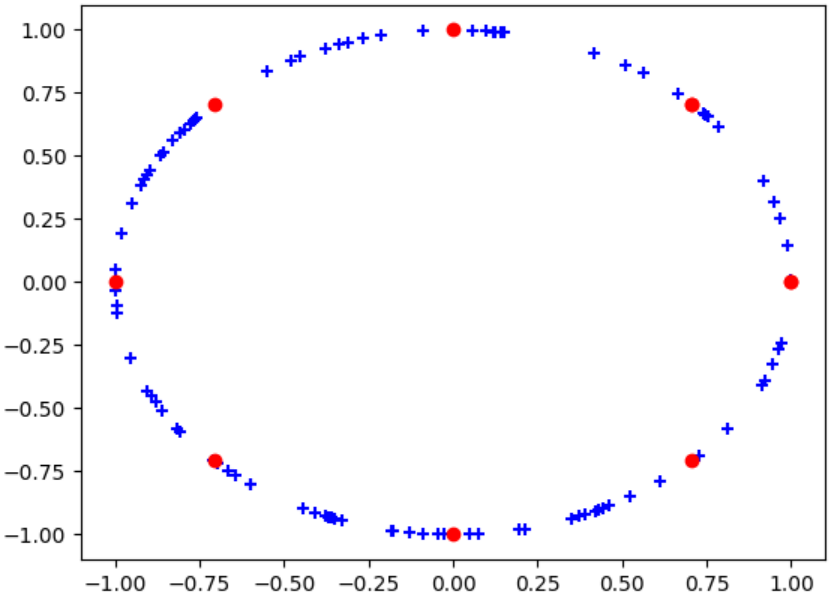

Figure 3: A visual example of data and nodes in a Chord DHT with 10 evenly distributed nodes (represented by red circles) and 100 tasks (blue pluses).

complete a single task per a tick or their strength's worth of tasks per a tick, with the former being the default.

Network Size: How many nodes start in the network. We assume that this can grow during the experiment, either via churn or by creating Sybils. We discuss how this works in Sections IV-A and IV-B.

Number of Tasks: We measure the size of of a job in tasks. This number is typically a few orders of magnitude greater than the network size.

Churn Rate: The churn rate is a float between 0 and 1. This represents both the chance each individual node has of leaving (or joining) the network each tick. It also corresponds to the fraction of nodes we expect to leave the network each tick. Churn can be self induced or a result of actual turbulence in the network. Like most analyses of churn [8], we assume churn is constant throughout the experiment and that the joining and leaving rate are equal. The default value is 0.

Max Sybils: `maxSybils` is the maximum number of Sybils a node can have at once. We tested 5, the default, and 10.

Sybil Threshold: The `sybilThreshold` dictates the number of tasks a node must have before it can create a Sybil. The default setting is having no work (0 tasks).

Successors: The number of successors each node keeps track of, either 5 or 10, with 5 being the default. Nodes also keep track of the same number of predecessors.

We also considered using a variable noted as the `AdaptationRate`, which was the interval at which nodes decided whether or not to create a Sybil. Preliminary experiments showed `AdaptationRate` to have a minimal effect on the runtime, so it was removed.

3) Outputs: The most important output was the number of ticks it took for the experiment to complete. We compared this runtime to the what we call the "ideal runtime," which is our expected runtime if the every node in the network was given an equal number of tasks and performed them without any churn or creating any Sybils. For example, consider a network with 1,000 nodes and 100,000 tasks, where each node consumes a single task each tick. This network has an ideal runtime of 100 ticks ($\frac{100000}{1000} = 100$).

We used these to calculate a "runtime factor," the ratio of the experimental runtime compared to the ideal runtime. For example, if 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.

We also collected data on the average work per

tick and statistical information about how the tasks are distributed throughout the network. We additionally performed detailed observations of how the workload is distributed and redistributed throughout the network during the first 50 ticks.

IV. Strategies

For our analysis, we examined four different strategies for nodes to perform autonomous load balancing. We first show the effects of churn on the speed of a distributed computation. We then look at three different strategies, in which nodes take a more tactical approach for creating churn and affecting the runtime.

Specifically, nodes perform a limited and controlled Sybil attack [4] on their own network in an effort to acquire work with their virtual nodes. Our strategies dictate when and where these Sybil nodes are created.

We discuss the effectiveness of each of the strategies and present the results of our simulations in Section V.

A. Induced Churn

Our first strategy, *Induced Churn*, relies solely on churn to perform load balancing. This churn can either be a product of normal network activity or self-induced. By self-induced churn, we mean that each node generates a random floating point number between 0 and 1. If the number is $\leq churnRate$, the node shuts down and leaves the network and gets added to the pool of nodes waiting to join.

Similarly, we have a pool of waiting nodes that begins at the same initial size as the network. When they generate an appropriate random number, they join the network and leave the waiting pool. We assume that nodes enter and leave the network at the same rate. Because the initial size of the network and the pool of waiting nodes is the same and nodes move between one another at the same random rate, the size of either does not fluctuate wildly.

As we have previously discussed, nodes in our network model actively back up their data and tasks to the a number of successors in case of failure. In addition, when a node joins, it acquires all the work it is responsible for. While this model is rarely implemented for DHTs, it is discussed [9] and often assumed to be the way DHTs operate. We have implemented it in ChordReduce[11] and UrDHT[13] and demonstrated that the network is capable from recovering from quite catastrophic failures and handling incredibly high amounts of churn.

The consequences of this are that a node suddenly dying is of minimal impact to the network. This is because a node's successor will quickly detect the loss of the node and know to be responsible for the node's work. Conversely, a node joining in this model can be a potential boon to the network by joining a portion of the network with a lot of tasks and immediately acquiring work.

This strategy acts as a baseline with which to compare the other strategies, as it is no more than a overcomplicated way of turning machines off and on again. All strategies below are attempts to do better than random chance. However, this strategy also serves to confirm the speedup phenomenon we observed in our previous work on a distributed, decentralized MapReduce framework [11].

B. Random Injection

Our second strategy we dubbed *Random Injection*. In this strategy, once a node's workload was at or below the `sybilThreshold`, the node would attempt to acquire more work by creating a Sybil node at a random address.

A node compares its workload to the `sybilThreshold` and decides whether or not to make a Sybil. This check occurs every 5 ticks. A node can also have multiple Sybils, up to `maxSybils` in a homogeneous network or the node's `strength` in a heterogeneous network.[2] If a node has at least one Sybil, but no work, it has its Sybils quit the network. We limit each node to creating a single Sybil during this decision to avoid overwhelming the network.

C. Neighbor Injection

Neighbor Injection also creates Sybils, but in this case, nodes act on a more restricted range in an attempt to limit the network traffic. Nodes with `sybilThreshold` or less tasks attempt to acquire more work by placing a Sybil among it's successors. Specifically, it looks for the biggest range among its successors and creates a Sybil in that range.

This range strategy of finding the largest range and injecting assumes that the node with the largest range of responsibility will have been allocated the most work. This is a sensible assumption since the larger the range of a node's responsibility, the more *potential* tasks it can receive. We compare this estimation strategy to actually querying the neighbor and asking how many tasks they have. An estimation, if accurate, would be preferable to querying the nodes as an estimation can be done without any communication with the successor nodes.

[2]No benefit was shown by increasing `maxSybils` beyond 10.

To avoid constant spamming of a range, once a node creates a Sybil, but does not acquire work, it may be advisable to mark that range as invalid for Sybil nodes so nodes don't keep trying the same range repeatedly.

D. Invitation

The *Invitation* strategy is the reverse of the Sybil injection strategies. In this strategy, nodes with a higher than desired level of work announces it needs help to its predecessors. The predecessor with least amount of tasks less than or equal to the `sybilThreshold` creates a Sybil to acquire work from the node. It is possible for an invitation to get more work to be refused by anyone if no predecessor is at the `sybilThreshold` or each predecessor has too many Sybils. Nodes determine whether or not they are overburdened using the `sybilThreshold`.

V. Results Of Experiments

We now briefly summarize the results of our simulation before discussing each more in depth below. First, we confirmed that churn, even at low levels, can speed up the execution of a distributed computation by dynamically load balancing the network. However, our best strategy was random injection, which managed to have a runtime factor which approached 1.

All examples are compared against a baseline network of the same size and initial configuration of nodes. The only difference is that the baseline never uses a strategy or experiences any churn.

All the raw data can be found online [10]. Files are sorted by the strategy used, the network size, and number of tasks and include the both the results of each individual run and the averages of each 100 trials for a particular set of variables. Each trial is linked with a seed for the pseudorandom number generator and can be fully reproduced.

We will be referring to a **base** runtime factor. This means the network had the defaults for variable. We suspect the worse runtime factors in homogeneous networks don't necessarily imply the method is weaker. It may imply that runtime factor is not the best metric, but it is the best simple one we have.

A. Churn

Our results confirmed our hypothesis from Chord-Reduce [11]. Churn has a profound and significant impact on the network's computation, and this effect is more pronounced with higher rates of churn. Our results in Table II show the effects of churn in networks of varying sizes and loads on the runtime factor.

Table II: 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

We see that for each network size and load, even small amounts of churn have a noticeable impact on performance. 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 gains from churn are most strongly related the number of tasks the network has, with the more tasks there are, the greater the gains of churn. A network with 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.

We ran additional experiments on rates of churn for a 1000 node network and 100,000 tasks, in order to give a fuller picture of the effects of churn. We tested this network with a wider variety of churn rates. The results of this are shown on Figure 4, which plots the runtime of the distributed computation against the level of churn in the network.

We note the significantly diminishing returns that occur after a churn rate of 0.01. One facet not captured by our simulations, but is significant, is the rising maintenance costs after that point. This makes any amount of churn after a certain point prohibitively expensive. Determination of this point is requires implementation on a real network.

This speedup is reflected in the average number of tasks consumed in a tick, shown in 5. In a stable network with no churn, we would expect a good portion of the network to end up with little work, which is quickly consumed. These nodes must then idle, waiting for the few nodes which received a large number of tasks to finish. When there is churn, however, nodes have a chance of joining the network and acquiring more work, thus making it possible for more work to be done per a

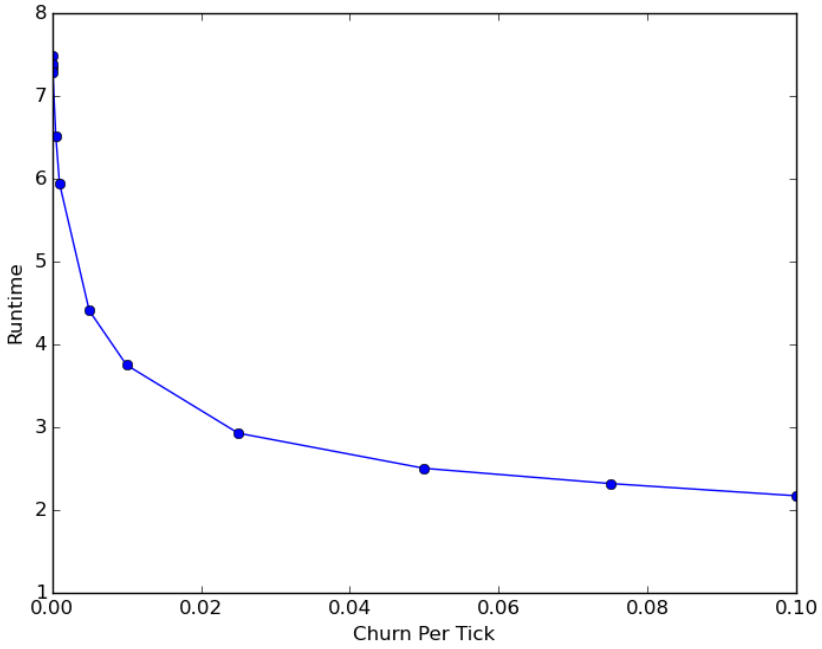

Figure 4: This graph shows the effect churn has on runtime in a distributed computation. Runtime is measured as how many times slower the computation runs than an ideal computation, where each node receives an equal number of tasks. The lower the runtime factor, the closer it is to an ideal of 1. Neither the homogeneity of the network nor work measurement had a significant effect on the runtime.

tick. The higher the rate of churn, the more this happens.

A more detailed view of how this happens is shown in Figures 6, 7, and 8. These figures show histograms of workload distribution for two networks at different ticks. One network uses 0.01 churn per tick to autonomously load balance the workload, the other uses no strategy at all.

Both networks [3] start with the 1000 nodes and 100,000 tasks distributed identically throughout the network. Figure 6 shows the distribution of the workload for this initial configuration.

As the network begins to work, we see the distributions diverge from one another. Figure 7 shows that after merely 5 ticks, the network using the churn strategy has noticeably fewer nodes that have smaller workloads. We also see more nodes with greater workloads. The effect becomes even more pronounced after 35 ticks, shown in Figure 8, where more have had a chance to finish their work and more churn has had the opportunity to occur.

B. *Random Injection*

The goal behind the churn strategy was to effectively have nodes join in the proper place with random insertions. However, churn relies on two factors that are completely random. The network nor its members have any control over when node joins happen or where these joins occur. In addition, churn removes nodes, which can be detrimental to the network.

The random injection strategy removes some of the randomization that the churn strategy deals with, namely *when* new nodes join, albeit not where. As we discussed earlier, random injection does this by means of the Sybil Attack [4], with nodes creating virtual Sybil nodes when they are underutilized.

We found that the strategy of having under-utilized nodes randomly creating Sybil nodes works phenomenally well, approaching very close to the ideal time. Figures 9 and 8 compare two networks with identical starting configurations as the network preforms the computations.[4] These networks have 1000 nodes and

[3]Indeed, all networks we do this assessment for start with the same initial configuration

[4]We omit a figure comparing the two networks at tick 0, since it identical to Figure 6.

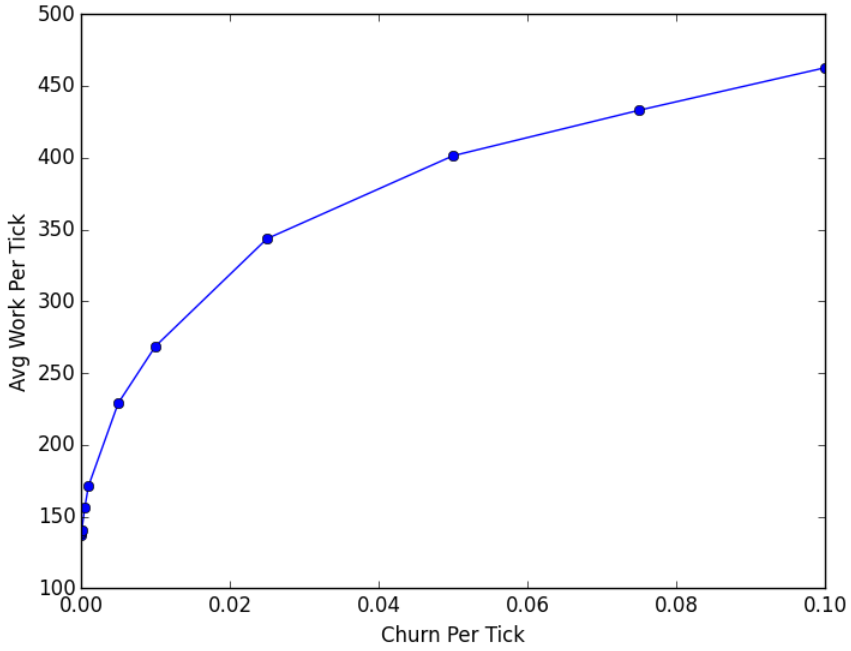

Figure 5: With more and more churn in the network, new nodes have a higher chance of joining the network and acquiring work from overloaded nodes. This results in more average work being done each tick, as there are less nodes simply idling.

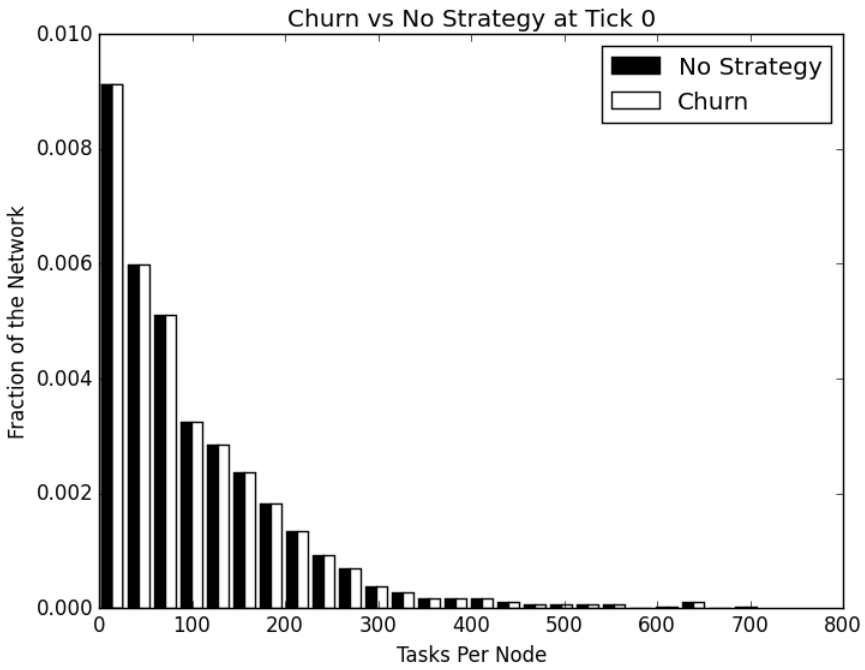

Figure 6: The initial distribution of the workload in both networks. As both networks start with same initial configuration, the distribution is currently identical. This greatly resembles the distribution we saw in Figure 1.

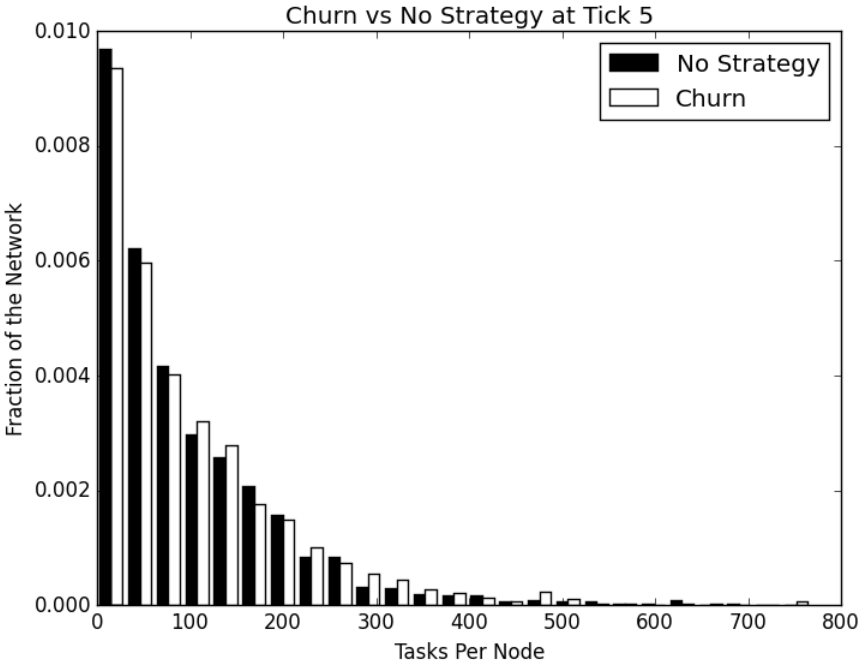

Figure 7: This is the distribution of the workloads at the beginning of tick 5. We can see the network using 0.01 churn has fewer nodes with less work and more nodes with a greater workload.

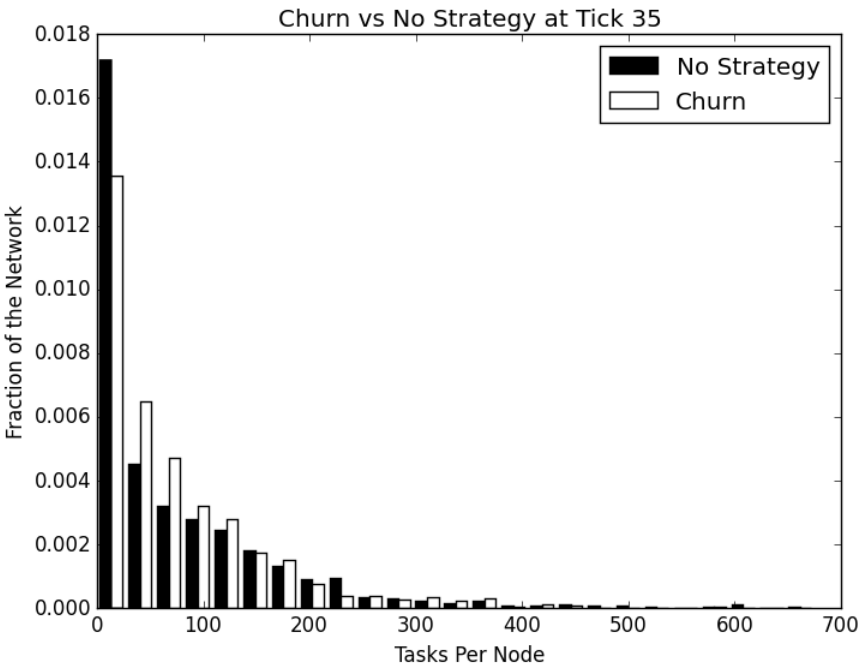

Figure 8: After 35 ticks, the effects of churn on the workload distribution become more pronounced. More nodes have consumed all their tasks and are simply idling, but significantly less in the network using churn.

100,000 tasks. The network using random injection fared significantly better than those using no strategy.

We can see in Figure 9, the results of a single load balancing operation. Each node has checked its workload, and if it was too low, created a single Sybil. If we compare this to Figure 6, we see that this single operation has a better workload distribution than what we had started with and is substantially better than using no strategy.

We can see these effects become even more pronounced at tick 35 (Figure 10), where seven load balancing operations have taken place. The network using random injection has many fewer nodes with little or no work and many more nodes with some work. Furthermore, we see significant difference between the effectiveness of random injection versus churn (Figure 11).

We found that a homogeneous, 1000 node/100,000 task network, where each node consumed a single task each tick would never have an average runtime factor greater than 1.7 and the average runtime was as fast as 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. This large jump is the result of how the network is impacted by heterogeneous nodes, explained further below.

Overall we found that networks with the same ratio of tasks to nodes would have relatively the same runtimes, but the smaller network would run slightly faster. For example, we compared two networks with 100 tasks per a node: a network with 100 nodes and 100,000 tasks and one with 1,000 nodes and 1,000,000 tasks. The smaller network had, on average, a mean runtime factor 0.086 less.

Heterogeneous networks also saw significantly better performance (Figure 12), 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 1000 tasks per node, compared to 4.052 on the smaller ratio networks with 100 tasks per node. However, most trials did not have a runtime factor greater than 3.

1) The Effects of Other Variables: The `sybilThreshold` had an effect in homogeneous networks but no effect in heterogeneous networks. In 1000 node/100,000 task and 100 node/10,000 experiment, a homogeneous network where each node completed one task per a tick, this amounted to an reduction of at least 0.1 runtime factor. The effect was more pronounced when networks could complete a number of tasks equal to their strength each tick, reducing the runtime factor by at least 0.2.

However, we saw no corresponding speedup in the 1000 node/1,000,000 task and 100 node/100,000 task networks. We also found the `sybilThreshold` had no discernible effect on the runtime of heterogeneous networks. This suggests the effect of a particular `sybilThreshold` is tied to the ratio of nodes to tasks and that the larger the ratio, the less room there is to improve.

Churn had no positive impact on the runtime factor. At higher levels, churn could begin having an extremely minor impact when churn is at 0.01 per tick, increasing the average runtime factor by approximately 0.06, not to mention any maintenance costs incurred by handling churn.

Our `maxSybils` variable had no noticeable effect on runtime in a heterogeneous network. `maxSybils` controlled both the number of tasks a node could consume and the maximum number of Sybils that node could make. Surprisingly, larger values for `maxSybils` had a negative impact on runtime in heterogeneous networks. Networks where nodes strength ranged 1 to 5 performed better than those where the range of `maxSybils` was 1 to 10. In other words, the greater the disparity between node strength in the network, the greater the detrimental effect to the heterogeneous network.

This impact was felt stronger in nodes with a lower node to task ratio. The runtime factor saw a 0.3 to 0.4 increase in the 1000 node/1,000,000 task and 100 node/100,000 task networks (1000 tasks per node). On the other hand, the 100 node/10000 task and 1000 node/100000 task networks saw an increase of about 1.

C. Neighbor Injection

The neighbor injection strategy was also effective at reducing the overall runtime factor, albeit not as greatly as the random injection strategy. Figures 13 and 14 show the workload distribution of the neighbor injection strategy in a 1000 node/100,000 task network. At the first glance, neighbor injection does not seem to be doing worse than no strategy; there are more nodes with less work than no strategy. However, if we look at the whole histogram, we see there are less nodes that have a large amount of work and noticeably more nodes with a small number of tasks. In fact, at tick 35, we see that the network with no strategy has a node with approximately

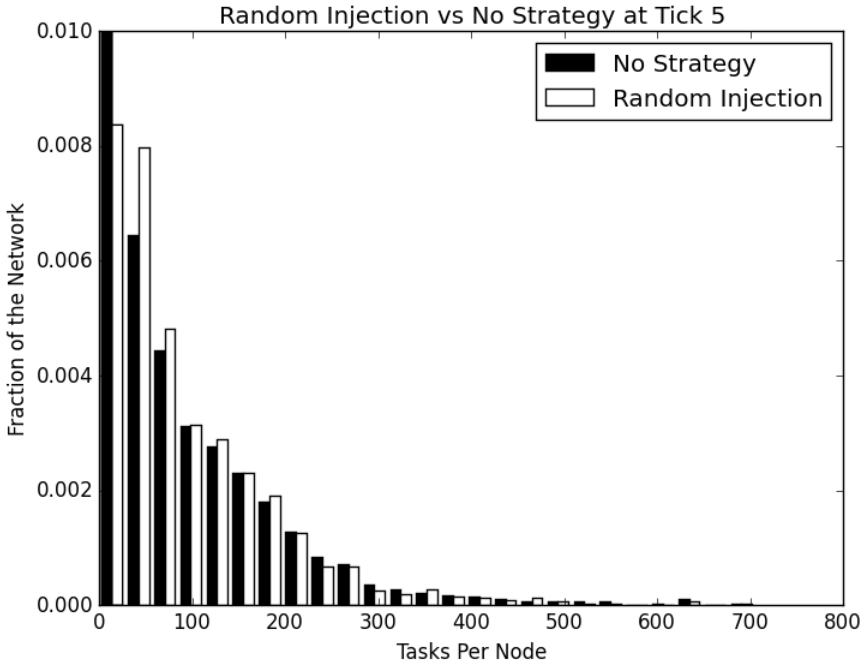

Figure 9: The networks after 5 ticks. The network using random injection has significantly less underutilized nodes, even after only 5 ticks.

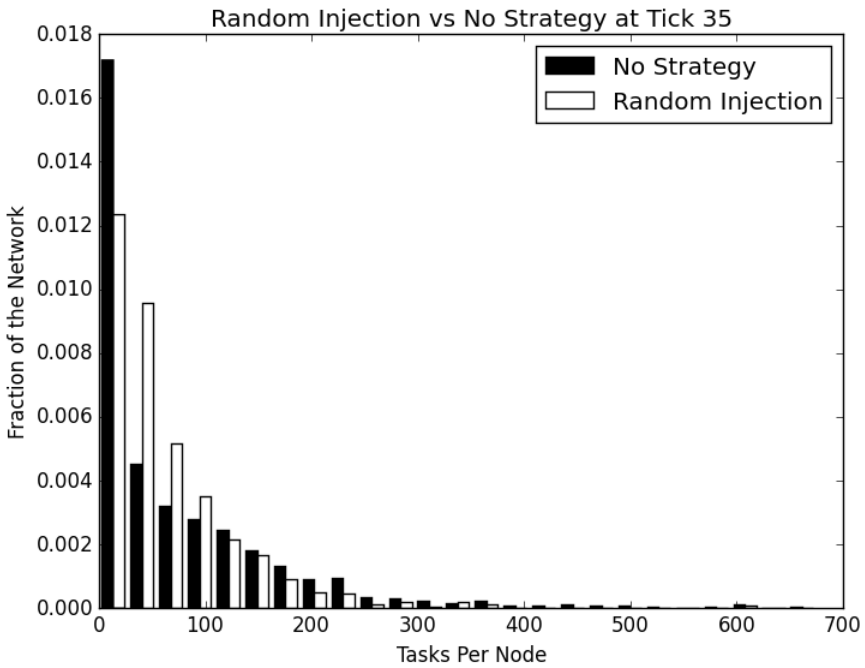

Figure 10: The networks after 35 ticks. The network using random injection has significantly less underutilized nodes and substantially more notes with some or lots of work.

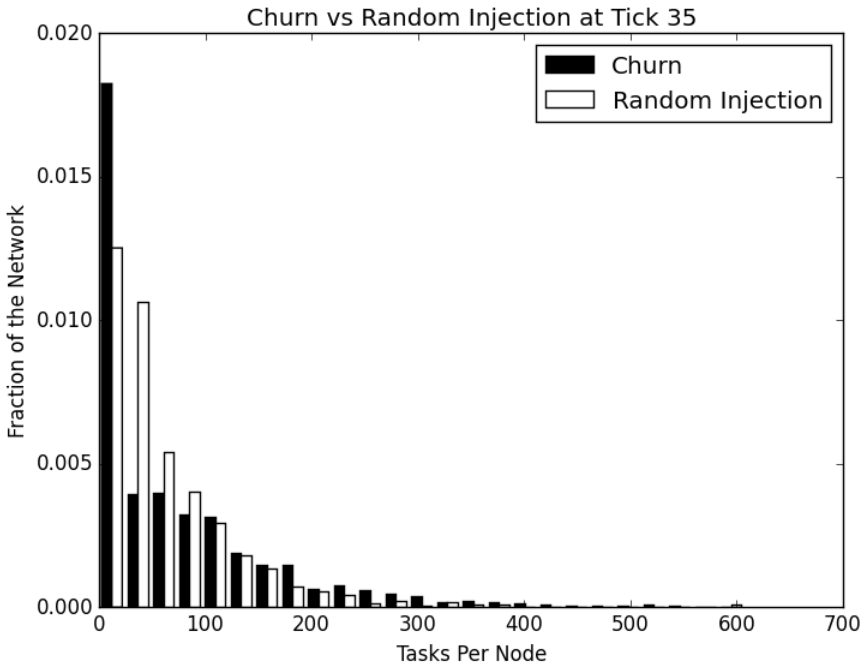

Figure 11: The networks after 35 ticks. The network using random injection load-balances significantly better than the network using Churn.

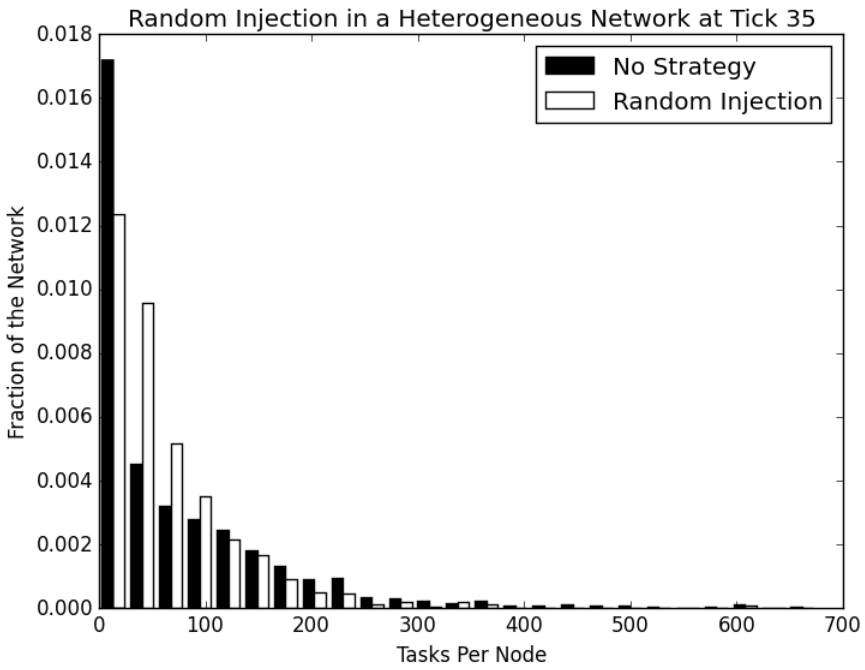

Figure 12: The workload distribution of heterogenous networks after 35 ticks. We can see the network using the random injection strategy is experiencing a better distribution of work.

650 tasks, while the most that the network using the neighbors strategy has is 450 tasks.

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 may not acquire any 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.[5]

The base runtime factor of the neighbor strategy in a 1000 node/100,000 task homogeneous network was 5.033, which is 2.4 lower than no strategy. In a 100 node/10,000 task network, this was lower – a runtime factor of 3.006, which was 2 lower than no strategy. However, the base runtime in a heterogeneous network was **worse** and, like in random injection, is exacerbated by a higher `maxSybil`.[6] We suspect this is because weaker nodes are acquiring work away from stronger nodes and then taking longer to finish.

Unsurprisingly, a larger `numSuccessors` improves the runtime factor. Our example 1000 node/100,000 task network experienced a decrease of about 0.3 to the runtime factor. However, `sybilThreshold` had no significant effect, nor did `maxSybil` for homogeneous networks.

To fully evaluate the effectiveness of the strategy, we compared the neighbor injection strategy, which estimates the node with the largest workload, to what we call smart neighbor injection. Smart neighbor injection actually queries the neighbors to find out their workload and injects a Sybil to take work away from the node with the most amount of work, not merely the node with the largest space of responsibility. The workload distributions of smart injection are shown in Figures 15 and 16. Both graphs show the same reduction in high amounts of workload that the base neighbor strategy provided, but with fewer idling nodes.

The strategy of probing each of the neighbors before inserting, rather than estimating, improved the runtime factor by 1.2 on average when we compare each strategy's mean homogeneous and heterogeneous runtimes. However, the estimation based neighbor injection requires fewer messages in an actual implementation. Overall, either neighbor injection strategy did not perform as well as the random injection strategy, but generates much less churn from joining nodes, since nodes can create their Sybils in a greatly reduced range.

[5]We suggest how to avoid this in a real implementation in Section IV-C

[6]Runtime was fine in the heterogeneous networks where each node could create a different number of Sybils, but still only complete one task per tick.

D. Invitation

Our final strategy is invitation based injection. What differentiates invitation based injection from the neighbor injection or random injection strategy is that the latter two are proactive strategies, while the former is reactive.

The proactive strategies work by having nodes with low work loads seek out work from other nodes. This is good because less nodes become idle. However, nodes acquire work from other workers, even if that other worker needed no help. In addition, nodes can spend many cycles trying to acquire work that may or may not exist, which significantly Invitation, however, is reactive. Nodes react to having too much work and ask the predecessors (the same nodes who would be injecting Sybils in the neighbor injection strategy) with the least work to come and help. This means queries of other nodes and Sybil injections only occur when they need to, greatly reducing the maintenance costs in an actual implementation.

Figures 17 and 18 show the work distribution of a network using the invitation based strategy. When we compare this strategy to the smart neighbor injection strategy (Figures 13 and 14), we see that invitation does a much better job of distributing the work.

The impact of the invitation strategy on the runtime factor was closely tied to the number of nodes in the network. For example, in a 100 node/ 100,000 task network (1000 tasks per node), the base average runtime factor was 3.749. However, a 1000 node/ 100,000 task network had a base average runtime of 5.673.

Other variables and the ratio of tasks to nodes appeared to have no effect on the runtime factor.[7] The single exception was when the network was heterogeneous and each node could complete a number of tasks equal to its strength. Like the neighbor injection strategy, networks under these conditions faired much worse, with our 1000 node/ 100000 tasks networks base average runtime factor being 6.097.

The invitation strategy load balances better than smart neighbors and uses less bandwidth than it and neighbor injection, since this strategy is reactive, rather than proactive, like random injection or neighbor injection.

[7]Churn was not tested on the invitation strategy, but we suspect it has the same effect as in the neighbor strategy.

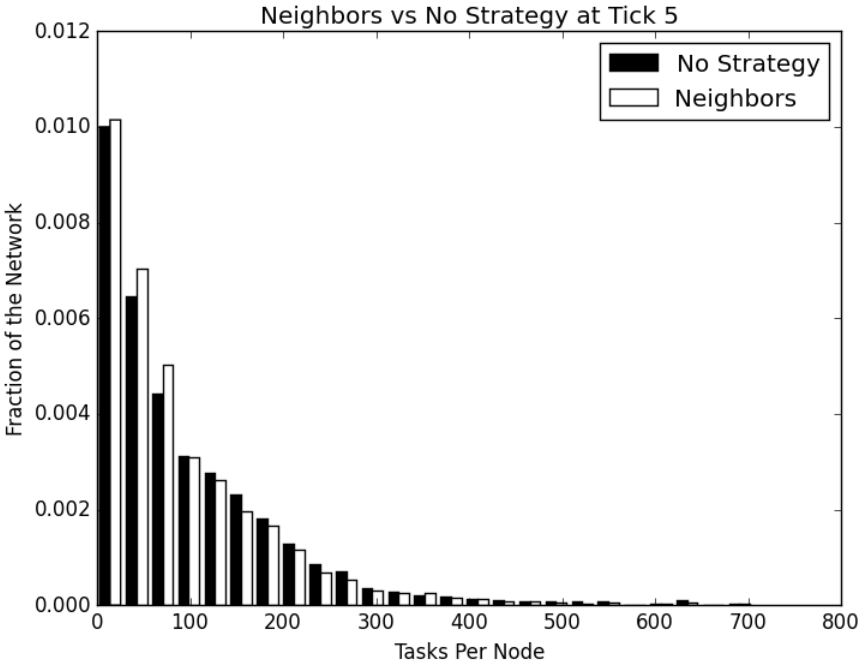

Figure 13: While the network no strategy appears to have less nodes idling, the active nodes in the network using neighbors are being better utilized.

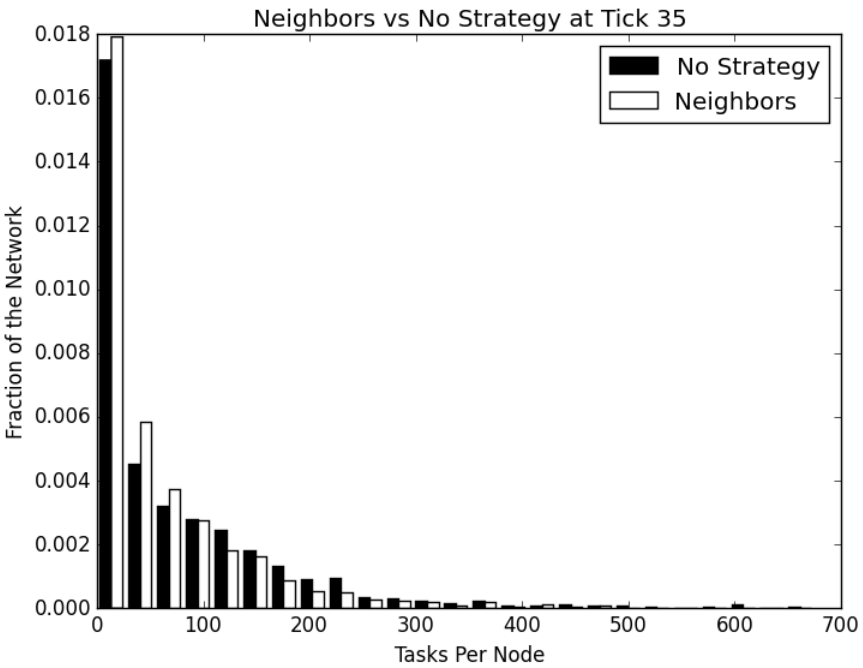

Figure 14: Despite have more idling nodes, we see that the nodes using the neighbor injection strategy have acquired smaller workloads and have effectively shifted part of the histogram left.

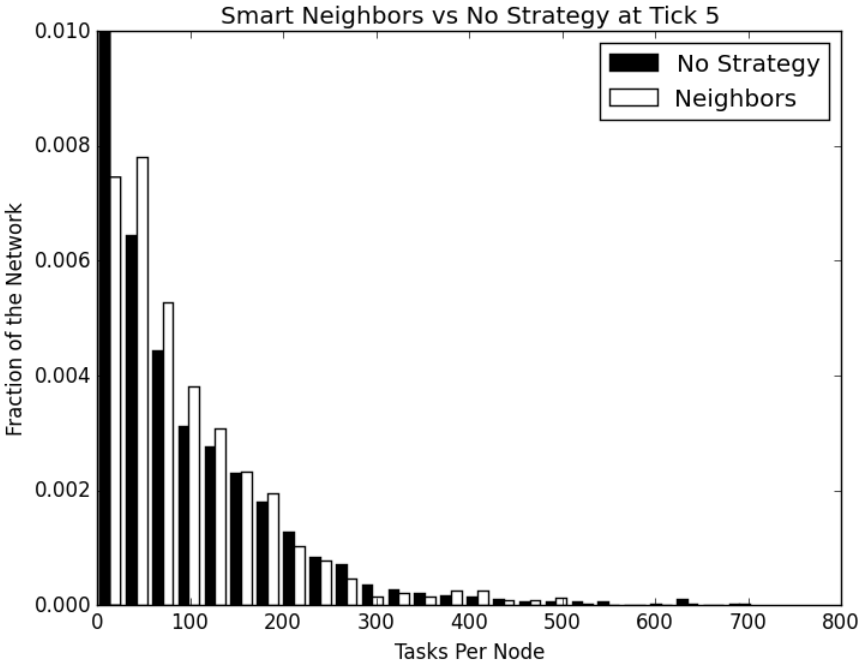

Figure 15: The workload in the networks after 5 ticks. We can that the network using neighbor injection has directed more nodes that would be idling to perform work. This is especially apparent in comparison to Figure 13.

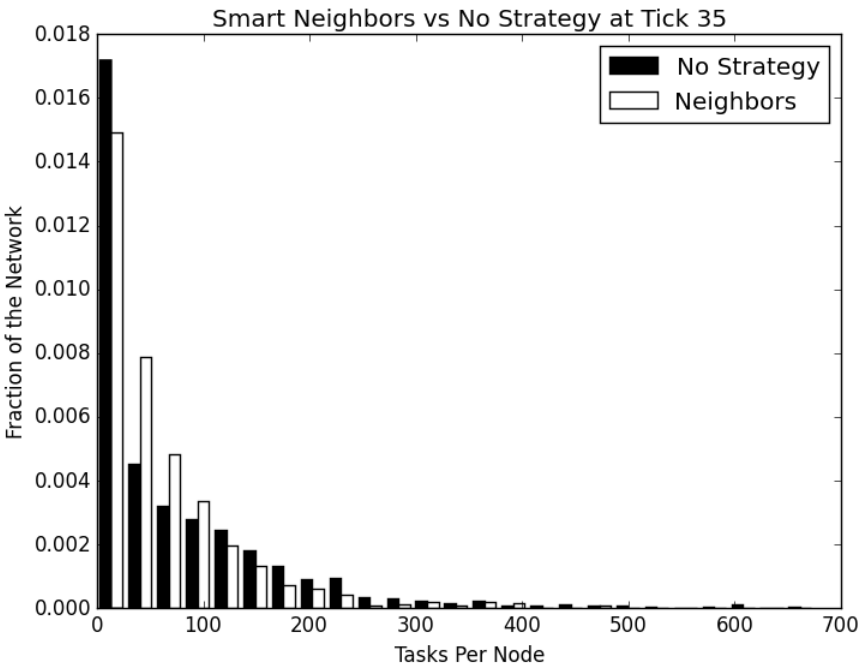

Figure 16: After 35 ticks, we see the network using the smart neighbor injection strategy has significantly less nodes with little or no work, more nodes with smaller amounts of work, and less nodes with large amounts of tasks.

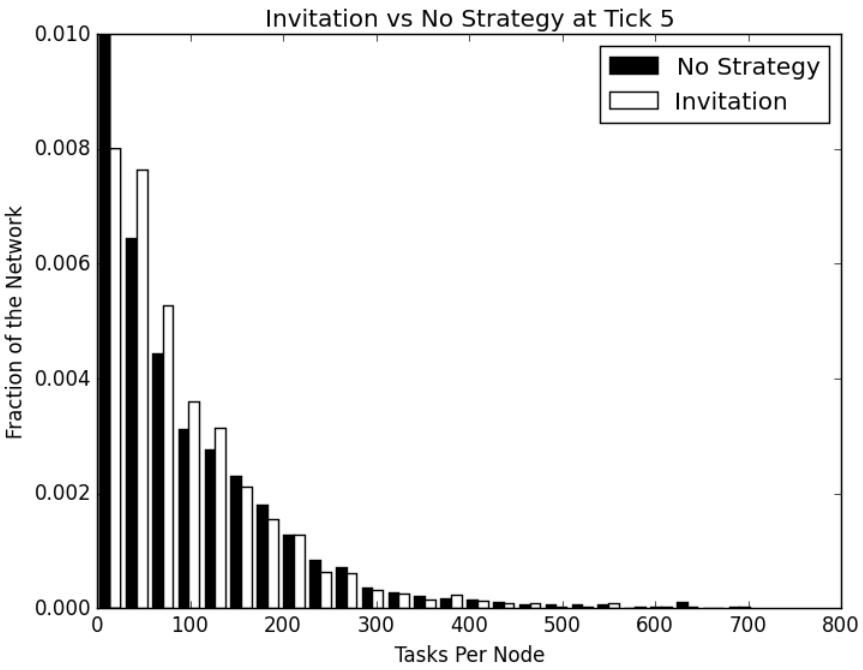

Figure 17: Here, we see that the invitation strategy performs markedly better at distributing the work than no strategy. At tick 5, there are many more nodes with small work loads. There are also few nodes with large amounts of work.

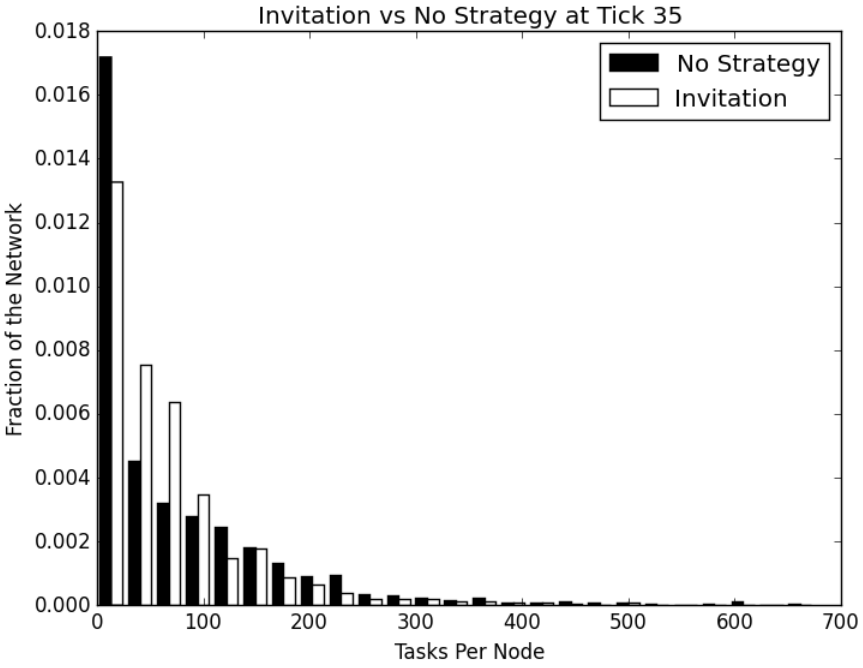

Figure 18: At 35 ticks, we can see the network using the invitation strategy perform markedly better than the network using no strategy. The highest load is around 500 tasks in the network using invitation, compared to approximately 650 tasks in the network using no strategy.

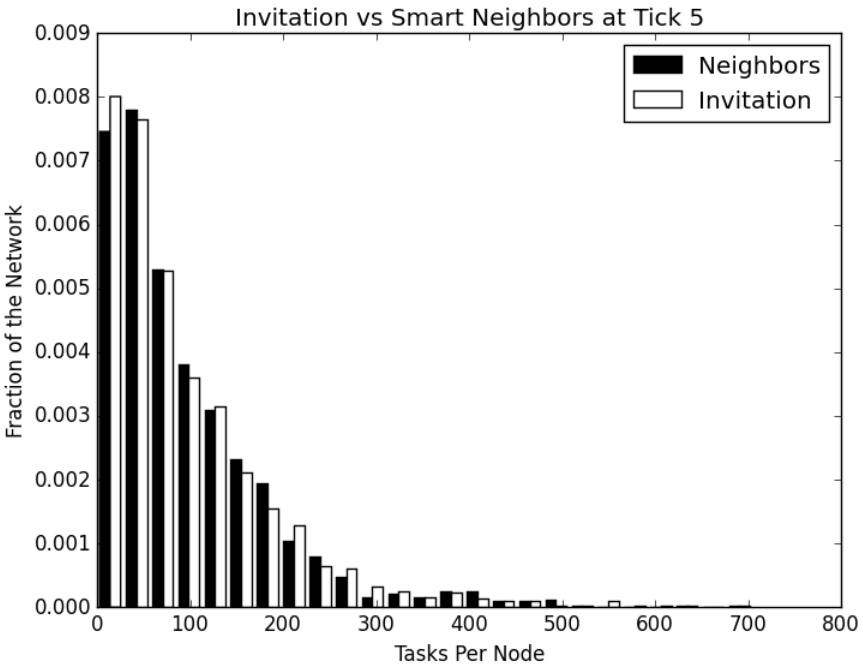

Figure 19: A comparison of a network using the smart neighbors strategy and a network using the invitation strategy. Invitation has slightly more nodes on the performing less work, but no significant differences have emerged.

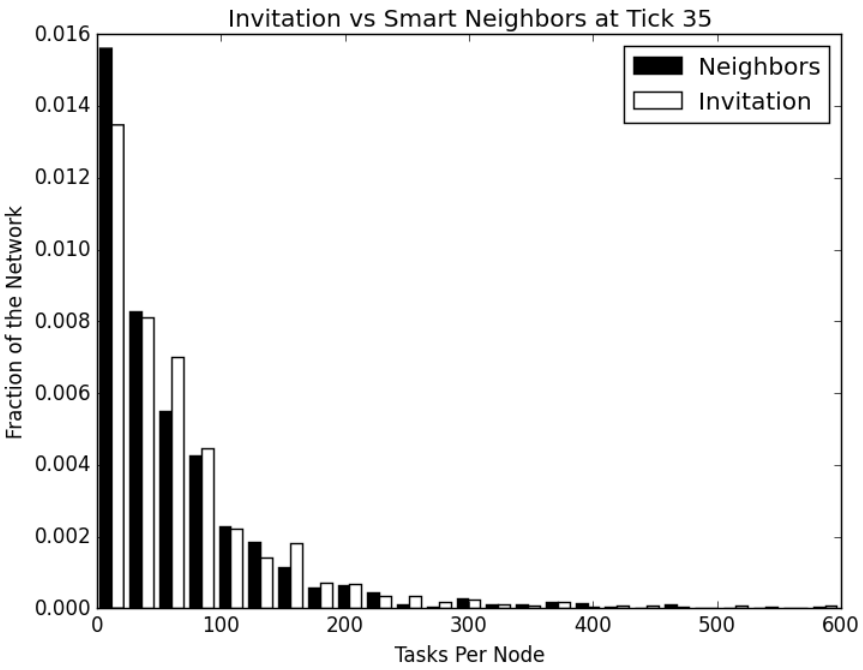

Figure 20: After 35 ticks, differences between the two strategies have emerged. The network using the invitation strategy has significantly less nodes with a small work load and many more with large work loads.

VI. Conclusions and Future Work

Our simulation results are notable for a number reasons. First, as we have previously mentioned, is the confirmation that churn can have a net positive effect on the runtime of the network. Second, we presented strategies based on the Sybil attack that performed significantly better than churn and would have a much less disruptive impact on the network's structure. We found that our random injection strategy performed exceptionally well, approaching as close to ideal times as we could hope.

The fact that these strategies can be used as new nodes join the network can not be understated. Each joining node is another member of the network that can fully participate in the computation, despite not being present at the beginning of the computation.

However, our strategies were not nearly as successful in a heterogeneous network, even though the work was better distributed throughout the network. We suspect this is the result of weaker nodes acquiring more work from stronger nodes, leading to an overall longer runtime, despite the workload being better balanced. In other words, the workload is balanced in the heterogeneous network, but the efficiency is not improved. An avenue for future work can take the node strength into account or use another, smarter strategy.

As mentioned in Section III, we made the assumption that nodes cannot choose their own ID and must rely on the strategies described in Chapter **??** [12] for creating a Sybil with the appropriate ID. However, if this assumption was removed, this presents even more strategies for nodes to autonomously load-balance.

References

[1] Hadoop. http://hadoop.apache.org/.

[2] Michael Buddingh. The distribution of hash function outputs. http://michiel.buddingh.eu/distribution-of-hash-values.

[3] Bram Cohen. Incentives build robustness in bittorrent. In *Workshop on Economics of Peer-to-Peer systems*, volume 6, pages 68–72, 2003.

[4] John R Douceur. The sybil attack. In *Peer-to-peer Systems*, pages 251–260. Springer, 2002.

[5] Donald Eastlake and Paul Jones. Us secure hash algorithm 1 (sha1), 2001.

[6] Mu Li, Li Zhou, Zichao Yang, Aaron Li, Fei Xia, David G Andersen, and Alexander Smola. Parameter server for distributed machine learning.

[7] Andrew Loewenstern and Arvid Norberg. BEP 5: DHT Protocol. http://www.bittorrent.org/beps/bep_0005.html, March 2013.

[8] Fabrizio Marozzo, Domenico Talia, and Paolo Trunfio. P2P-MapReduce: Parallel Data Processing in Dynamic Cloud Environments. *Journal of Computer and System Sciences*, 78(5):1382–1402, 2012.

[9] Petar Maymounkov and David Mazieres. Kademlia: A peer-to-peer information system based on the xor metric. In *Peer-to-Peer Systems*, pages 53–65. Springer, 2002.

[10] Andrew Rosen. Autonomous load-balancing raw data. https://github.com/abrosen/thesis/tree/master/data/done.

[11] Andrew Rosen, Brendan Benshoof, Robert W Harrison, and Anu G. Bourgeois. Mapreduce on a chord distributed hash table. In *2nd International IBM Cloud Academy Conference*.

[12] Andrew Rosen, Brendan Benshoof, Robert W Harrison, and Anu G. Bourgeois. The sybil attack on peer-to-peer networks from the attacker's perspective.

[13] Andrew Rosen, Brendan Benshoof, Robert W Harrison, and Anu G. Bourgeois. Urdht. https://github.com/UrDHT/.

[14] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. *SIGCOMM Comput. Commun. Rev.*, 31:149–160, August 2001.

[15] Søren Steffen Thomsen and Lars Ramkilde Knudsen. *Cryptographic hash functions*. PhD thesis, Technical University of DenmarkDanmarks Tekniske Universitet, Department of Applied Mathematics and Computer ScienceInstitut for Matematik og Computer Science, 2005.