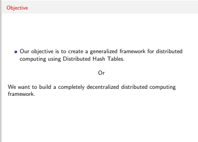
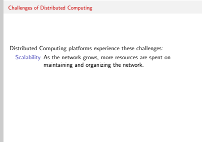
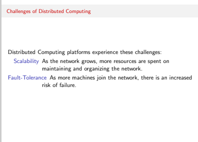


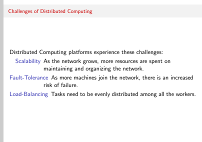
- We want to build a completely decentralized distributed computing framework based on distributed hash tables, or DHTs.
- Doing this will require a generic framework for creating distributed hash tables and distributed applications.
- This means we need two things:
- A way of easily abstracting DHTs
- A way to make sure that we can distribute work effectively across a DHT



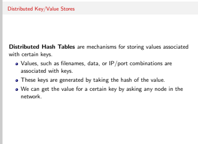
To review now. Remember, computers aren't telepathic. There's always an overhead cost. It will grow. The challenge of scalability is designing a protocol in which this cost grows at an extremely slow rate. For example, a single node keeping track of all members of the system might be a tenable situation up to a certain point, but eventually, the cost becomes too high for a single node.



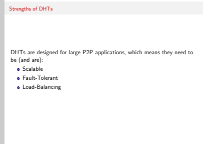
Failure Hardware failure is a thing that can happen. Individually the chances are low, but this becomes high when we're talking about millions of machines. Also, what happens in a P2P environment. Nodes leaving is treated as a failure.



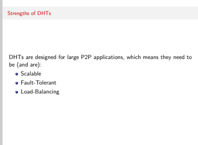
If we are splitting the task into multiple parts, we need some mechanism to ensure that each worker gets an even (or close enough) amount of work.



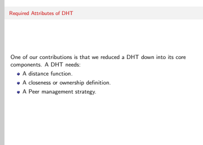
At their core, Distributed Hash Tables are giant lookup tables. Given a key, it will return the value associated with that key, if it exists. These keys, or hash keys, are generated by a hash function, such as SHA1 or MD5. These hash functions use black magic using prime numbers and modular arithmetic 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**.



- 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



- 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.



- 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.

- 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 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

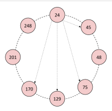


Figure: An 8-node Chord ring where $m = 8$. Node 24's long peers are shown.

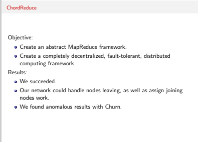
- The dashed lines are the short links; each node keeps track of its successor and predecessor.
- The dotted lines are node 24's long links; since $m = 8$ there's 8, but since the network is so small, 4 are duplicates.
- Traffic travels clockwise.
- Routing example 24 to 150.

Overarching Theme

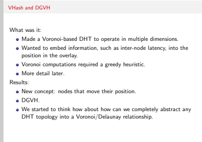
My research has been focused on:

- Abstracting out DHTs.
- Distributed computation using DHTs.

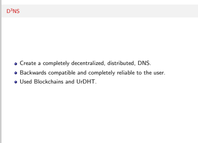
- I want to get down to what the essence of a DHT is, find out what all DHT have in common, so that I could create a generic DHT.
- As I implied earlier, we all these DHTs, but they all have the same features and I want to generalize them.
- I focused on creating a more abstract framework for MapReduce,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



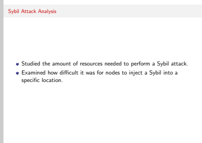
- More nodes doing work was significantly faster than just one node, but ther were diminishing returns due to maintanence.
- We did very small networks (30 - 50), and very weak nodes (cheapest on Amazon).
- We leveraged Chord's fault tolerance mechanisms to handle tasks.
- Why did churn do what it did? That's covered in Autonomous Load Balancing.



- The embedding stuff was Brendan's research



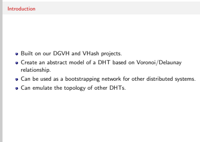
- Intended to replace TLD servers
- Intended to be completely transparent to users (no extra domains or extensions like Namecoin)



- Easy to inject where you need to in a range
- quick to precompute.

2016-05-11

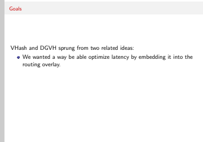
Autonomous Load-Balancing
└UrDHT
 └Introduction
  └Introduction



UrDHT differs from VHash in many ways. VHash is a DHT that has a toroidal topology and computes Voronoi and Delaunay trinagulations to figure the network overlay. It does not emulate or recreate other topologies. UrDHT seeks to be an abstract DHT that we can build all other DHT topologies by defining the appropriate parameters. One way to think about it is that VHash is a default setting for UrDHT.

2016-05-11

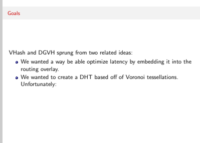
Autonomous Load-Balancing
└UrDHT
 └DGVH
  └Goals



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

2016-05-11

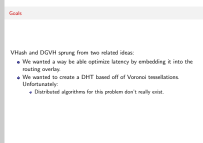
Autonomous Load-Balancing
└UrDHT
 └DGVH
  └Goals



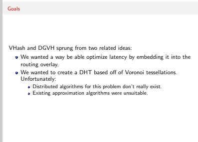
We discovered a mapping between Distributed Hash Tables and Voronoi/Delaunay Triangulations.

2016-05-11

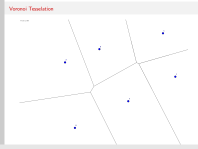
Autonomous Load-Balancing
└UrDHT
 └DGVH
  └Goals



I lie, they do exist, but they all are "run the global algorithm on your local subset. And if we move out of or above 2D Euclidean space, as Brendan wanted to, no fast algorithms exist at all. We quickly determined that solving was never really a feasible option. So that leaves approximation. A distributed algorithm would be helpful for some WSNs solving the boundary coverage problem.

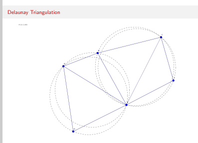


Simple approximations have no guarantees of connectivity, which is very bad for a routing topology. Better algorithms that existed for this problem technically ran in constant time, but had a prohibitively high sampling. So to understand what I'm talking about here, let's briefly define what a Voronoi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



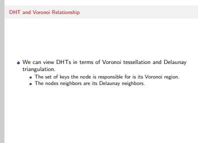
Define

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

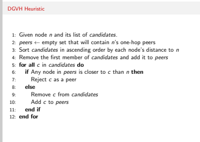


Define

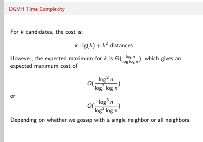
- Delaunay Triangulation is a triangulation of a set of points with the following rule:
- No point falls within any of the circumcircles for every triangle in the triangulation,
- The Voronoi tessellation and Delaunay Triangulation are dual problems.
  - Solving one yields the other.
  - We can get the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



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.



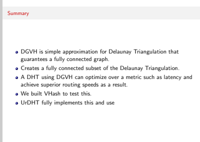
1. We have n, the current node and a list of candidates.
2. peers is a set that will build the peerlist in
3. We sort the candidates from closest to farthest.
4. The closest candidate is always guaranteed to be a peer.
5. Next, we iterate through the sorted list of candidates and either add them t the 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



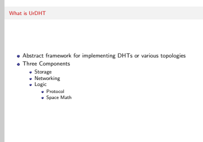
DVGH is very efficient in terms of both space and time. Suppose a node n is creating its short peer list from k candidates in an overlay network of N nodes. The candidates must be sorted, which takes $O(k \cdot \lg(k))$ operations. Node n then compares distances between all peers and all the candidates. This results in a cost of

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

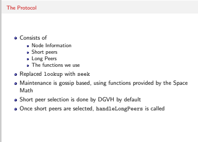
Since k is bounded by $\Theta(\frac{\log N}{\log \log N})$ (the expected maximum degree of a node), we can translate the above to:
The expected worst case cost of DGVH is $O(\frac{\log^3 n}{\log^3 \log n})$, if we swap with all short peers. Otherwise the cost is $O(\frac{\log^2 N}{\log^2 \log N})$
In 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 $\Omega(1)$.



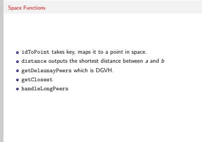
- DGVH is of similar complexity to picking k-nearest node or nodes in distance k.
- other methods don't guarantee full connectivity
- It caps out at $O(n^2)$ complexity, no matter how many dimensions or complexities of the metric space (unless calculating distance or midpoint is worse than $O(1)$)
-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



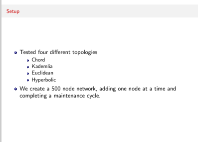
- We implement these topologies by abstracting them to the Voronoi Delauna
- Storage deals with file storage
- Networking deals with actual implementation of how nodes talk across the network
- Mostly we'll talk about the logic, which has two components
- You can change both, but changing space math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



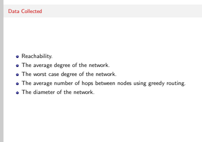
- `seek` is a single step of the `lookup`.
- `lookup` can be done with iterative calls to `seek`.
- While many protocols specify a recursive lookup, we find must actually implement an iterative, since it makes it easier to handle errors.
- Don't have to change protocol, but you can. I implemented chord 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.



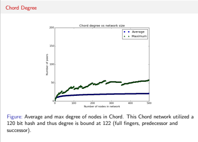
- Distance is not symetrical in every space
- Given a set of points, the candidates, and a center point, `getDelaunayPeers` calculates a mesh that approximates the Delaunay peers of center.
- `getClosest` returns the point closest to `center` from `candidates`. `seek` depends on `getClosest`
- `handleLongPeers` 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.



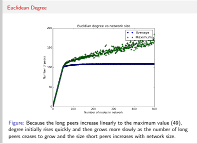
- Two different types of experiments
- Tested to see if actual implementation worked
- Simulated creating larger networks using the smae exact logic we used for UrDHT
- This was one of the benefits to abstract this the way we did, we could pull the math out for easy simulation



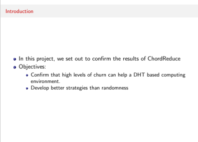
- Everything is reachable
- The average degree of the network. This is the number of outgoing links and includes both short and long peers.
- Diameter is the worst case distance between two nodes using greedy routin
- Averages for distance are averages of 100 pairs of random source destinatio pairs (or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



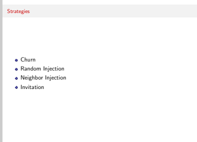
Jaggedness of maximum is most likely the result of nodes joining and occluding previous connections.



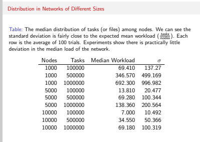
The Euclidean and Hyperbolic networks are 2d spaces. Our equation for peers from VHash was $3d + 1$ short peers minimum and $(3d + 1)^2$ long peers maximum. We limit the long peers because those could eventually encompass the entire network if we don't cap it. We don't cap short peers because it is possible, but unlikely, for a node to have many many many short peers.



We saw that churn could speed up a distributed computation. If it works, then essentially, we're saying turning machines off and on randomly speeds things up in a DHT. We also want to probably create some strategies better than random chance.



- These are all strategies that require no centralization, little or no coordination.
- Each node makes a lod balancing decision according to the strategy on its own.

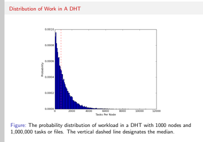


Distribution in Networks of Different Sizes

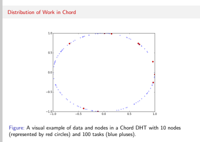
Table: The median distribution of tasks (or files) among nodes. We can see the standard deviation is fairly close to the expected mean workload ($\frac{tasks}{nodes}$). Each row is the average of 100 trials. Experiments show there is practically little deviation in the median load of the network.

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

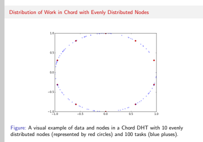
- Let's see how work is distributed in a DHT.
- In an ideal world, assume each node was the same strength and consumed single task per tick, we'd want to give every node the same amount of worl
- In a DHT, though, we find that work is not evenly distributed.
- Compare the median and standard deviation to the ideal average.
- This means 50% of the network has significantly less tasks than the averag Probably more as we'll see.



- The graph(and all graphs) are normalized. Taking the integral will give us 1
- 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 furthe right will dictate the runtime.



- Nodes are responsible for all the tasks that fall along the perimeter betwee themselves and their predecessor, which is the closest node counterclockwis
- 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.



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.

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

- 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

- The magnitude of churn's effect varies based on the size of the network an[d] 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

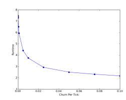


Figure: This graph shows the effect churn has on runtime in a distributed computation.

- Runtime is measured as how many times slower the computation runs than an ideal computation, where each node receives an equal number of 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.

2016-05-11

Autonomous Load-Balancing
└─Autonomous Load-Balancing
  └─Churn
    └─Base Work Distribution

Base Work Distribution

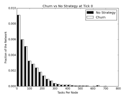


Figure: The initial distribution of the workload in both networks.

As both networks start with same initial configuration, the distribution is currently identical. This greatly resembles the distribution we saw in Figure 10.
This will be the same for every strategy.

2016-05-11

Autonomous Load-Balancing
└─Autonomous Load-Balancing
  └─Churn
    └─Churn Distribution after 35 Ticks

Churn Distribution after 35 Ticks

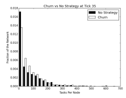


Figure: After 35 ticks

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

2016-05-11

Autonomous Load-Balancing
└─Autonomous Load-Balancing
  └─Churn
    └─Remarks

Remarks

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

- After 0.01 churn, we saw reduced improvement.
- We didn't simulate maintenance, but those are a factor
- Killing nodes provides no speedup

2016-05-11

Autonomous Load-Balancing
└─Autonomous Load-Balancing
  └─Random Injection
    └─Strategy

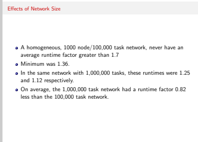
Strategy

- 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.

- The check interval used to be a variable, but preliminary results showed it did not have much of an effect.
- Sybil removal is instantaneous if no work was acquired. We can get a simila result with no disruption if a node asks if it would be stealing any work.
- So query before joining

2016-05-11

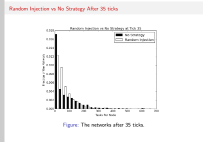
Autonomous Load-Balancing
└Autonomous Load-Balancing
└Random Injection
└Effects of Network Size



- 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.

2016-05-11

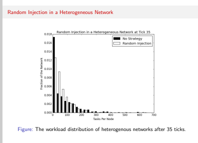
Autonomous Load-Balancing
└Autonomous Load-Balancing
└Random Injection
└Random Injection vs No Strategy After 35 ticks



- The network using random injection has significantly less underutilized node and substantially more notes with some or lots of work.
- We see even better improvement here.

2016-05-11

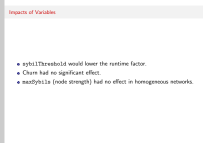
Autonomous Load-Balancing
└Autonomous Load-Balancing
└Random Injection
└Random Injection in a Heterogeneous Network



- 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.

2016-05-11

Autonomous Load-Balancing
└Autonomous Load-Balancing
└Random Injection
└Impacts of Variables



- `sybilThreshold` did nothing in heterogeneous networks. The improvement 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

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.

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

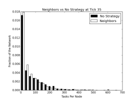


Figure: Neighbor injection vs no strategy after 35 ticks.

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.

2016-05-11

Autonomous Load-Balancing
└─Autonomous Load-Balancing
  └─Neighbor Injection
    └─Smart Neighbor Injection after 35 Ticks

Smart Neighbor Injection after 35 Ticks

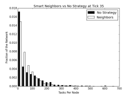


Figure: Smart Neighbor injection vs no strategy after 35 ticks.

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.

2016-05-11

Autonomous Load-Balancing
└─Autonomous Load-Balancing
  └─Invitation
    └─Strategy

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

2016-05-11

Autonomous Load-Balancing
└─Autonomous Load-Balancing
  └─Invitation
    └─Invitation at 35 Ticks

Invitation at 35 Ticks

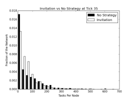


Figure: Invitiation vs no strategy after 35 ticks

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.

2016-05-11

Autonomous Load-Balancing
└─Autonomous Load-Balancing
  └─Invitation
    └─Invitation vs Smart Neighbors at Tick 35

Invitation vs Smart Neighbors at Tick 35

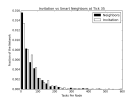


Figure: Invitation vs smart neighbor injection after 35 ticks.

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.