

2016-05-11

Autonomous Load-Balancing

- └ Introduction
- └ Objective
- └ Objective

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

2016-05-11

Autonomous Load-Balancing

- └ Introduction
- └ Distributed Computing and Challenges
- └ Challenges of Distributed Computing

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.

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.

2016-05-11

Autonomous Load-Balancing

- └ Introduction
- └ Distributed Computing and Challenges
- └ Challenges of Distributed Computing

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.

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.

2016-05-11

Autonomous Load-Balancing

- └ Introduction
- └ Distributed Computing and Challenges
- └ Challenges of Distributed Computing

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.

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.

Autonomous Load-Balancing

- └ Introduction
 - └ What Are Distributed Hash Tables?
 - └ Distributed Key/Value Stores

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.

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.**

Autonomous Load-Balancing

- └ Introduction
 - └ What Are Distributed Hash Tables?
 - └ Strengths of DHTs

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

Autonomous Load-Balancing

- └ Introduction
 - └ What Are Distributed Hash Tables?
 - └ Strengths of DHTs

Strengths of DHTs

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

- Scalable
- Fault-Tolerant
- Load-Balancing

Autonomous Load-Balancing

- └ Background
 - └ The Components and Terminology
 - └ Required Attributes of DHT

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.

- 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 necessarily 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 predecessor 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.

Autonomous Load-Balancing

- Background
- The Components and Terminology
 - Terms and Variables

Terms and Variables

- Network size is n nodes.
- Keys and IDs are m -bit hashes, usually SHA1 with 160 bits.
- Peers 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 all use cryptographic hash functions
- Short peers are actively maintained, long peers replaced gradually and are not actively pinged.

Autonomous Load-Balancing

- Background
- Example DHT: Chord
 - Chord

Chord

- Ring Topology
- Short Peers: predecessor and successor in the ring.
- Responsible for keys between their predecessor and themselves.
- Long Peers: $\lg n$ nodes, where the node at index i in the peerlist is $\text{root}(i) = 2^{i-1} \bmod m$, $1 \leq i \leq m$.

- 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.

Autonomous Load-Balancing

- Background
- Example DHT: Chord
 - A Chord Network

A Chord Network



- 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.

Autonomous Load-Balancing

- Completed Work
- Overarching Theme

Overarching Theme

- My research has been focused on:
- Abstracting our DHTs
 - Distributed computation using DHTs.

- I want to get down to what the essence of a DHT is, find out what all DHTs 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.

2016-05-11

Autonomous Load-Balancing

Completed Work

ChordReduce

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 nodes joining nodes with.
- We found astonishing results with Churn.

- More nodes doing work was significantly faster than just one node, but then were diminishing returns due to maintenance.
- 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.

2016-05-11

Autonomous Load-Balancing

Completed Work

VHash and DGVH

VHash and DGVH

What was it:

- Made a Vortex-based DHT to operate in multiple dimensions.
- Wanted to embed information, such as item-node latency, into the position in the vortex.
- Vortex computations required 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 a DHT topology into a Vortex/Undersway relationship.

- The embedding stuff was Brendan's research

2016-05-11

Autonomous Load-Balancing

Completed Work

D³NS

D³NS

- Create a completely decentralized, distributed, DNS.
- Resolves completely and completely reliable to the user.
- Used Blockchain and UCDHT.

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

2016-05-11

Autonomous Load-Balancing

Completed Work

Sybil Attack Analysis

Sybil Attack Analysis

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

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

2016-05-11

Autonomous Load-Balancing

- └ UrDHT
 - └ Introduction
 - └ Introduction

Introduction

- Built on our DQDHT and VHash projects.
- Create an abstract model of a DHT based on Voronoi/Delaunay triangulations.
- Can be used as a bootstrapping network for other distributed systems.
- Can emulate the topology of other DHTs.

UrDHT differs from VHash in many ways. VHash is a DHT that has a toroidal topology and computes Voronoi and Delaunay triangulations 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

Goals

- VHash and DQDHT spring from two related ideas:
- We wanted a way to able optimize latency by embedding it into the routing overlay.

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

2016-05-11

Autonomous Load-Balancing

- └ UrDHT
 - └ DGVH
 - └ Goals

Goals

- VHash and DQDHT spring from two related ideas:
- We wanted a way to 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.

2016-05-11

Autonomous Load-Balancing

- └ UrDHT
 - └ DGVH
 - └ Goals

Goals

- VHash and DQDHT spring from two related ideas:
- We wanted a way to 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.

2016-05-11

Autonomous Load-Balancing

- └ UrDHT
- └ DGVH
- └ Goals

Goals

Vflood and DGVH spring from two related ideas:

- We wanted a way to able optimize latency by embedding it into the routing queries
- We wanted to create a DHT based off of Voronoi tessellations

Unfortunately:

- Distributed algorithms for this problem don't really exist.
- Existing optimization algorithms were inaccurate.

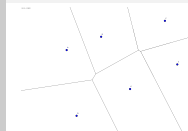
Simple approximations have no guarantees of connectivity, which is very bad for a routing topology. Better algorithms that existed for this problem technically ran in constant time, but had a prohibitively high sampling. So to understand what I'm talking about here, let's briefly define what a Voronoi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

2016-05-11

Autonomous Load-Balancing

- └ UrDHT
- └ DGVH
- └ Voronoi Tessellation

Voronoi Tessellation



Define

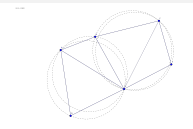
- A Voronoi tessellation or Voronoi diagram divides a space into regions, where each region encompasses all the points closest to Voronoi generators (points)
- Voronoi generators
- Voronoi Region
- Voronoi Tessellation/ Diagram

2016-05-11

Autonomous Load-Balancing

- └ UrDHT
- └ DGVH
- └ Delaunay Triangulation

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.

2016-05-11

Autonomous Load-Balancing

- └ UrDHT
- └ DGVH
- └ DHT and Voronoi Relationship

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.

Autonomous Load-Balancing

└ UrDHT
└ DGVH
└ DGVH Heuristic

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
   for all  $c$  in candidates do
5.   if Any node in peers is closer to  $c$  than  $n$  then
6.     Reject  $c$  as a peer
7.   else
8.     Remove  $c$  from candidates
9.   Add  $c$  to peers
10. end if
11. 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 to 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

Autonomous Load-Balancing

└ UrDHT
└ DGVH
└ DGVH Time Complexity

DGVH Time Complexity

For k candidates, the cost is:

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

However, the expected maximum for k is $O(\frac{\log^3 n}{\log^2 \log n})$, which gives an expected maximum cost of

$$O(\frac{\log^3 n}{\log^2 \log n})$$

or

$$O(\frac{\log^3 n}{\log^2 \log n})$$

Depending on whether we group with a single neighbor or all neighbors

DGVH 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 $O(\frac{\log^3 n}{\log^2 \log n})$, 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 $\Omega(1)$.

Autonomous Load-Balancing

└ UrDHT
└ DGVH
└ Summary

Summary

- DGVH is simple approximation for Delaunay Triangulation that generates 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 Vitath to test this.
- UrDHT fully implements this and see

- 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 it in a 100-dimensional euclidean space in $O(n^2)$ time rather than $O(n^{50})$ time (maybe we should have opened with this...)

Autonomous Load-Balancing

└ UrDHT
└ UrDHT
└ What is UrDHT

What is UrDHT

- Abstract framework for implementing DHTs in various topologies
- Three Components
 - Storage
 - Networking
 - Logic
 - Protocol
 - Query Math

- We implement these topologies by abstracting them to the Voronoi Delaunay
- 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.

2016-05-11

Autonomous Load-Balancing

└─UrDHT
└─UrDHT
└─The Protocol

The Protocol

- Consists of
 - Node information
 - Short peers
 - Long peers
 - The functions we use
- Replaced `lookup` with `seek`
- Maintenance is group based using functions provided by the Space Math
- Short peer selection is done by `getClosest` 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.

2016-05-11

Autonomous Load-Balancing

└─UrDHT
└─UrDHT
└─Space Functions

Space Functions

- `getDistance` takes two, maps it to a point in space.
- `distance` outputs the shortest distance between `a` and `b`
- `getClosestPeers` which is DHTN
- `getClosest`
- `handleLongPeers`

- Distance is not symmetrical 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 list 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 structured distribution
- If long peers is too big, use a subset.

2016-05-11

Autonomous Load-Balancing

└─UrDHT
└─Experimental Results
└─Setup

Setup

- Tested four different topologies
 - Chord
 - Kademlia
 - Gnutella
 - Hypercube
- We create a 100 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 same 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

2016-05-11

Autonomous Load-Balancing

└─UrDHT
└─Experimental Results
└─Data Collected

Data Collected

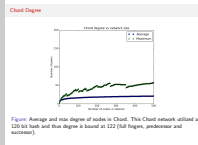
- Reachability
- The average degree of the network.
- The worst case degree of the network.
- The average number of hops between nodes using greedy routing
- The diameter of the network.

- 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 destination pairs (or network size if the network was smaller)
- Averages of degree were averages of all nodes.

2016-05-11

Autonomous Load-Balancing

- UrDHT
- Experimental Results
- Chord Degree

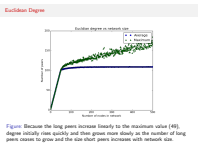


Jaggedness of maximum is most likely the result of nodes joining and occluding previous connections.

2016-05-11

Autonomous Load-Balancing

- UrDHT
- Experimental Results
- Euclidean Degree



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.

2016-05-11

Autonomous Load-Balancing

- Autonomous Load-Balancing
- Introduction
- Introduction

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 random chance

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.

2016-05-11

Autonomous Load-Balancing

- Autonomous Load-Balancing
- Introduction
- Strategies

Strategies

- Churn
- Random Injection
- Neighbor Injection
- Isolation

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

Autonomous Load-Balancing

└ Autonomous Load-Balancing

└ Distribution of Keys in A DHT

└ Distribution in Networks of Different Sizes

Distribution in Networks of Different Sizes

Table: The median distribution of tasks (or files) among nodes. We can see the expected deviation is fairly close to the expected mean workload (1000). Each row is the average of 100 trials. Experiments show there is practically little deviation in the median load of the network.

Nodes	Nodes	Median Workload	σ
1000	1000000	444.625	117.371
1000	5000000	240.510	406.589
1000	10000000	610.300	406.582
5000	1000000	110.820	20.477
5000	5000000	60.280	100.544
5000	10000000	130.380	200.564
10000	1000000	1.000	14.462
10000	5000000	24.000	50.306
10000	10000000	60.180	100.539

- 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 work.
- 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 average. Probably more as we'll see.

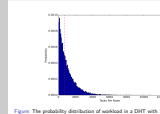
Autonomous Load-Balancing

└ Autonomous Load-Balancing

└ Distribution of Keys in A DHT

└ Distribution of Work in A DHT

Distribution of Work in A DHT



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

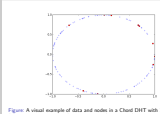
Autonomous Load-Balancing

└ Autonomous Load-Balancing

└ Distribution of Keys in A DHT

└ Distribution of Work in Chord

Distribution of Work in Chord



- 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 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.

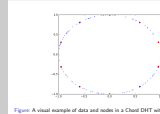
Autonomous Load-Balancing

└ Autonomous Load-Balancing

└ Distribution of Keys in A DHT

└ Distribution of Work in Chord with Evenly Distributed Nodes

Distribution of Work in Chord with Evenly Distributed Nodes



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.

2016-05-11

Autonomous Load-Balancing

- Autonomous Load-Balancing
 - Experimental Setup
 - Output

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.

2016-05-11

Autonomous Load-Balancing

- Autonomous Load-Balancing
 - Churn
 - Strategy

Strategy

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

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

2016-05-11

Autonomous Load-Balancing

- Autonomous Load-Balancing
 - Churn
 - Runtime

Runtime

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

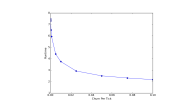
Nodes	Tasks	Runtime Factor
10	10	1.00
10	100	1.00
10	1000	1.00
10	10000	1.00
100	10	1.00
100	100	1.00
100	1000	1.00
100	10000	1.00
1000	10	1.00
1000	100	1.00
1000	1000	1.00
1000	10000	1.00
10000	10	1.00
10000	100	1.00
10000	1000	1.00
10000	10000	1.00

2016-05-11

Autonomous Load-Balancing

- Autonomous Load-Balancing
 - Churn
 - Churn vs Runtime factor

Churn vs Runtime factor



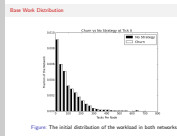
- The magnitude of churn's effect varies based on the size of the network and the number of tasks.
- In networks where there are fewer nodes, we see the base runtime factor is smaller.
- The more tasks there are, the greater the gains of churn.
- A network 100 nodes and 1 million tasks, on average, has 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.

- 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



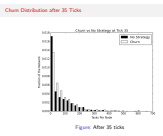
As both networks start with same initial configuration, the distribution is currently identical. This greatly resembles the distribution we saw in Figure 10.

This will be the same for every strategy.

2016-05-11

Autonomous Load-Balancing

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



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

- Consolidating returns.
- Maintenance nodes 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 $< \text{splitThreshold}$ 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 similar result with no disruption if a node asks if it would be stealing any work.
- So query before joining

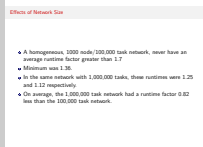
2016-05-11

Autonomous Load-Balancing

Autonomous Load-Balancing

Random Injection

Effects of Network Size



- Heterogeneous networks also saw significantly 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.

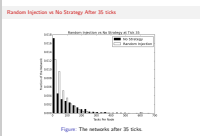
2016-05-11

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 nodes and substantially more nodes with some or lots of work.
- We see even better improvement here.

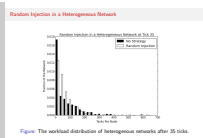
2016-05-11

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.

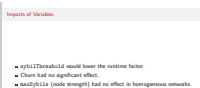
2016-05-11

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.

2016-05-11

Autonomous Load-Balancing
└ Autonomous Load-Balancing
└ Random Injection
└ Remarks

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

2016-05-11

Autonomous Load-Balancing
└ Autonomous Load-Balancing
└ Neighbor Injection
└ Strategy

Strategy

- Rather than creating Sybils randomly, nodes create one in their successors.
- Preferring node of ones smaller.
- Estimating which successor has most work.
- Tested estimation against smart method.

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

2016-05-11

Autonomous Load-Balancing
└ Autonomous Load-Balancing
└ Neighbor Injection
└ Base Runtime

Base Runtime

- The base runtime in a 1000 node/100,000 task homogeneous network was 5.523.
- 2.4 base time no strategy
- Base runtime in a homogeneous runtime was worse
- Inefficiency improves the runtime factor (5.5 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.

2016-05-11

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

Neighbor Injection after 35 Ticks

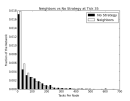


Figure: Neighbor injection vs no strategy after 35 ticks.

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

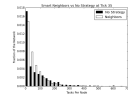


Figure: Smart Neighbor Injection vs no strategy after 35 ticks.

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
- Producers with smaller workload and below `splitThreshold` create a Sybil
- Practice vs Practice

- 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

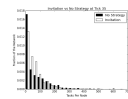


Figure: Invitation vs no strategy after 35 ticks.

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

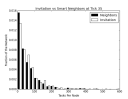


Figure: Invitation vs smart neighbor injection after 35 ticks.

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.