Internship Summary Report
July 7, 2020-September 15, 2020

Edris Qarghah

MANIAC LAB

Enrico Fermi Institute
University of Chicago

The Problem

The Large Hadron Collider (LHC) at CERN produces exabytes of data that is disseminated to high-energy physics labs around the world for analysis. The network that supports this transmission is decentralized and consists of around $6,000$ individual nodes[1].

To get a sense of the activity on the network, certain endpoints (around 420 of them[2]) are configured as perfSONAR (PS) nodes. These regularly transmit data to one another[3] and record certain characteristics of those transmissions, such as one-way delay, packet loss and bandwidth.

With this limited visibility, if there is a disruption somewhere along the network, it can be hard to pin down where the problem is, potentially leaving entire regions of researchers with slow or limited access to other regions for months at a time.

Internship Goals

The primary goal of this internship has been to develop better methods of determining the source of disruptions on the network used by the high-energy physics community. To address this, we need to:

- Identify that there *is* an issue.
  - This is done via **anomaly detection**, the identification of events that are outside the norm. In our case, this would mean looking at packet loss, one-way delay and other metrics to see whether any abnormalities may indicate there is a problem that needs to be addressed.
- Identify *where* the issue is occurring.
  - To do this, we need to have an understanding of the topology of the network, which is achieved via the use of **network tomography**, which is the study of the shape, state and other characteristics of a network using only data gathered from limited set of endpoints (i.e., perfSONAR nodes).

I spent the entirety of my internship focused on the latter problem, which I tackled in three steps:

1. Create a toy network to use as a model.
2. Develop tomography strategies using the toy model.
3. Create a model of the real network that mirrors the toy model, so that tools/strategies can be adapted for use with real data.

Toy Network

Motivation

We have limited visibility into the real network we are working with, as we only have the data collected by perfSONAR nodes. Such data is incomplete, complicated and messy, so it does not make an ideal training ground for learning about networks, trying to understand how they are structured and determining causal relationships in network phenomena.

This is what motivates the construction of a toy network, one where I define all nodes and their connections. With such a model, I can make changes, observe the impact on various facets of the network and be certain this impact was caused by my changes. Real data, on the other hand, is subject to change for reasons that are sometimes unknowable and often completely outside our control.

Not only can the toy network provide a better understanding of behavior on the real network, it can also be a testing ground for network tomography strategies that can then be applied to the real network.

[1] A survey of trace routes from 7/7/2020 to 7/14/2020 found 5968 unique nodes.
[2] The aforementioned survey found 423 perfSONAR nodes.
[3] The aforementioned survey found trace routes between $24,503$ pairs of PS nodes.

Initial Parameters

It is impractical to create a toy network on the same scale and with the same level of detail as the real one, so I started with the following parameters:

- The network consists of 100 nodes.
- 10 random nodes are selected to "host" perfSONAR.
- Each node is a coordinate on the x, y plane.
- Each node has a random number of connections (up to 4) to its nearest neighbors.
- The 'latency' for each link is their geometric distance.

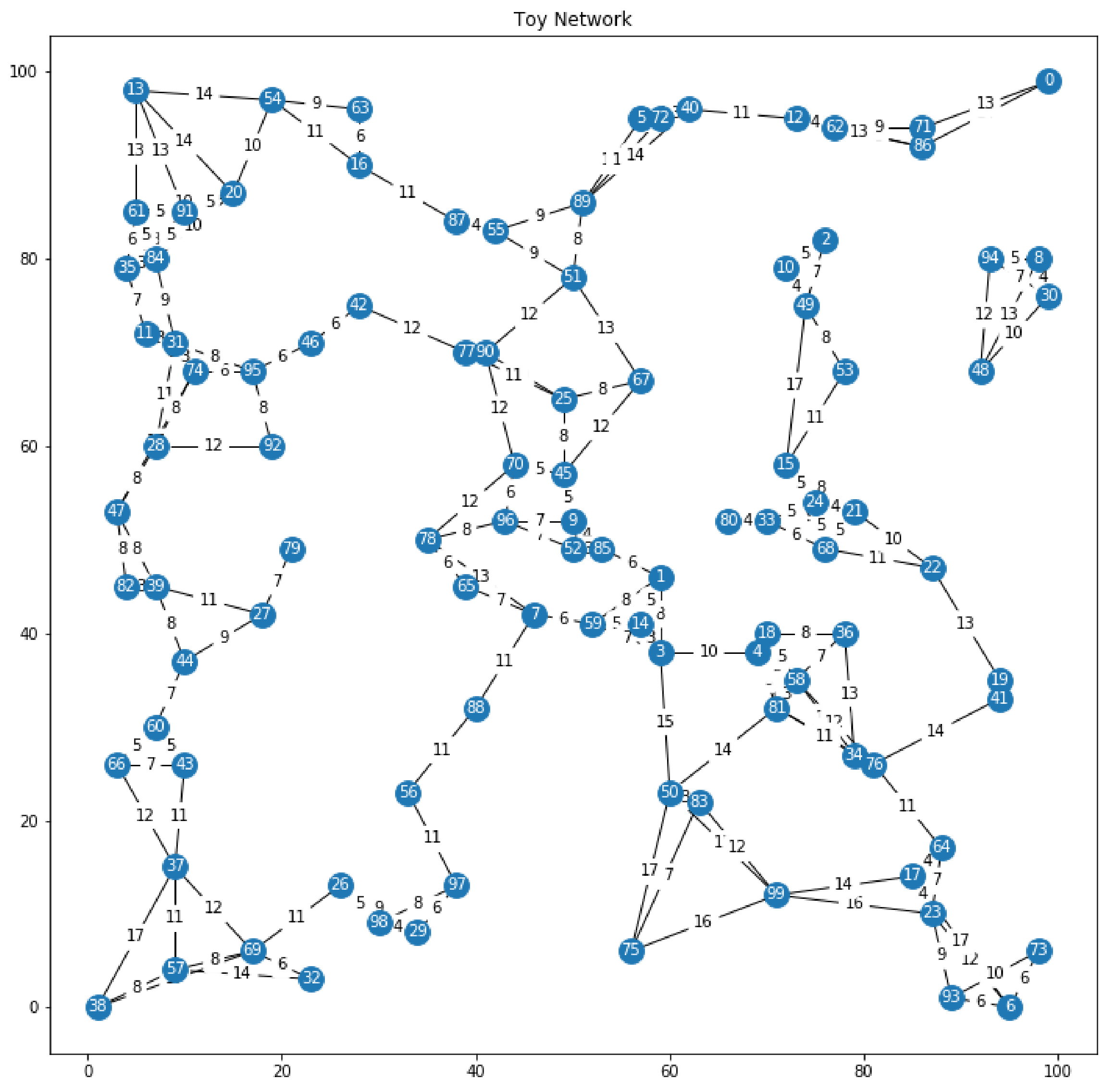

Figure 1: My first networks had no guarantee of being connected, as seen by nodes 94, 8, 48 and 30.

There were some issues with such a naive approach:

- There were no long edges (we know doesn't resemble reality even without exploring real data because of things like transatlantic cables).
- There was no guarantee that all components would be connected (for a cluster of nodes, the nearest neighbor to all those nodes may only be within that cluster).
- The toy network was not grounded in any information about the real world (i.e., it may not resemble the real network at all, in which case it wouldn't be a very good proxy).
- The "perfSONAR" nodes are indistinguishable from others and had no additional functionality.

Incremental Improvements

As I developed my toy network, I made a wide variety of incremental improvements. The list below is roughly chronological with breaks to provide figures at various stages in the process.

- I created hub nodes, that served as a backbone for the network, which were randomly placed in quadrants and quadrants within those quadrants, recursively (any number of layers deep, though ultimately 5). These hub nodes were connected by edges to the hub nodes within their respective sub-quadrants, ensuring that there were some longer edges and some structure to the network.
- I made sure that the network was k-connected (configurable, though I settled for 1-connected), which is to say that in order to separate a component from the rest of the network, k edges would need to be cut.
- I colored special nodes (i.e., hub and PS nodes).
- I found the shortest paths between perfSONAR nodes, using the Euclidean distances between nodes on the path.
- I gave each edge a packetloss (and displayed it) based on a distribution pulled from Kibana, but made the mistake of applying the distribution to individual edges and not entire paths.

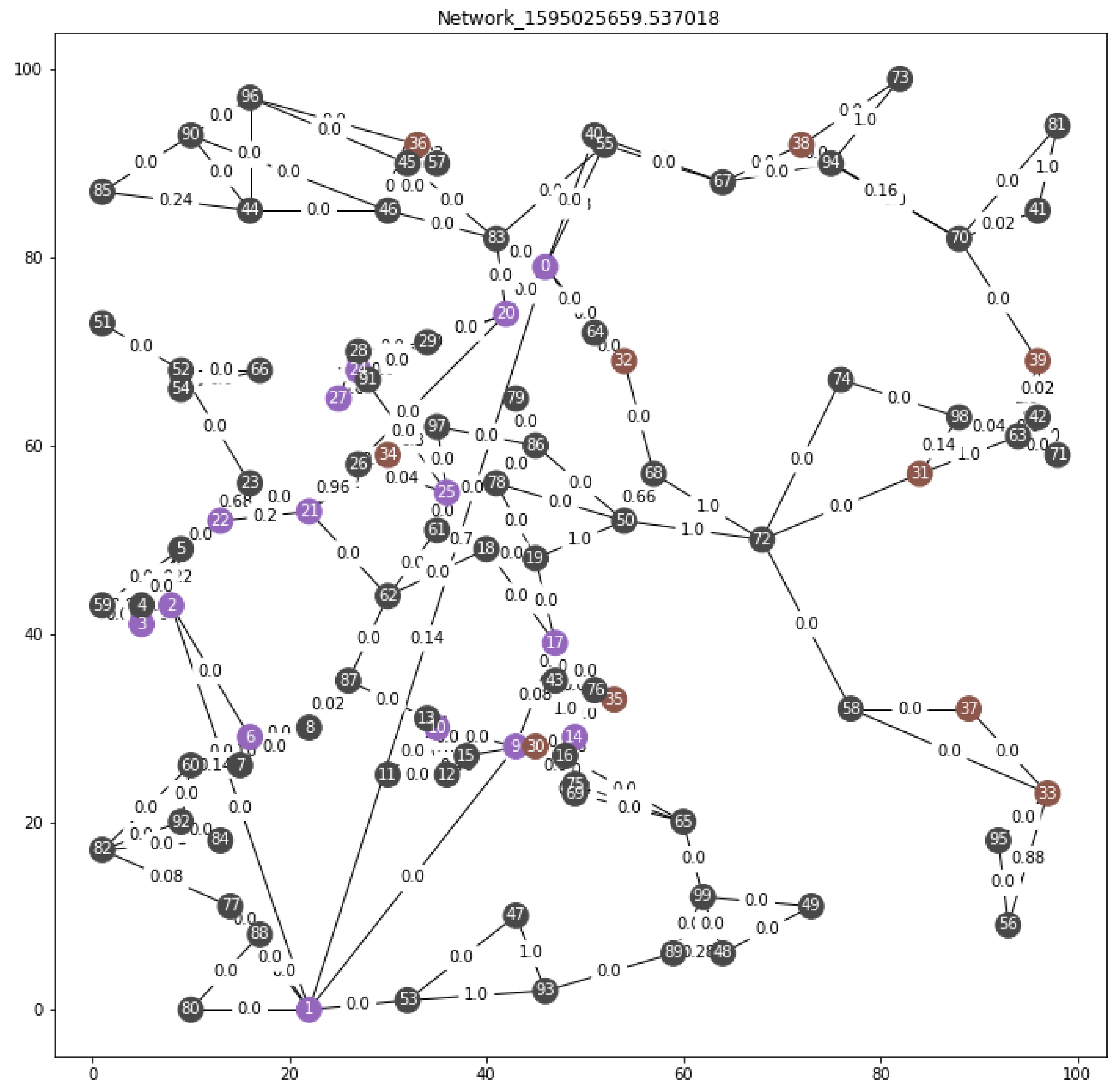

Figure 2: Hub (lilac) and PS nodes (brown) are colored and there are no unconnected components, but there are components without PS nodes (which would never have been seen/traversed on a real network) and the packetloss along the edges were unrealistic.

- Ensured that all degree 1 nodes are perfSONAR nodes (if a node is only connected to the network by a single edge, then the only way that node would ever be seen is if that node is itself a PS node), but this didn't account for components that didn't contain a PS node.
- Made sure that all bridge-connected components (i.e., components that could be separated from the rest of the network by removing a single edge) have at least perfSONAR node (otherwise there would be no reason to ever traverse this component).
- Used low Communicability Betweenness Centrality (roughly a measure of how many paths would be disrupted if a node is removed) as a perfSONAR selection criteria to ensure that boundary nodes (ones connecting a component to the rest of the network) are not selected.
- Distributed PS nodes to components proportional to the size of the component (to improve dispersion of nodes).

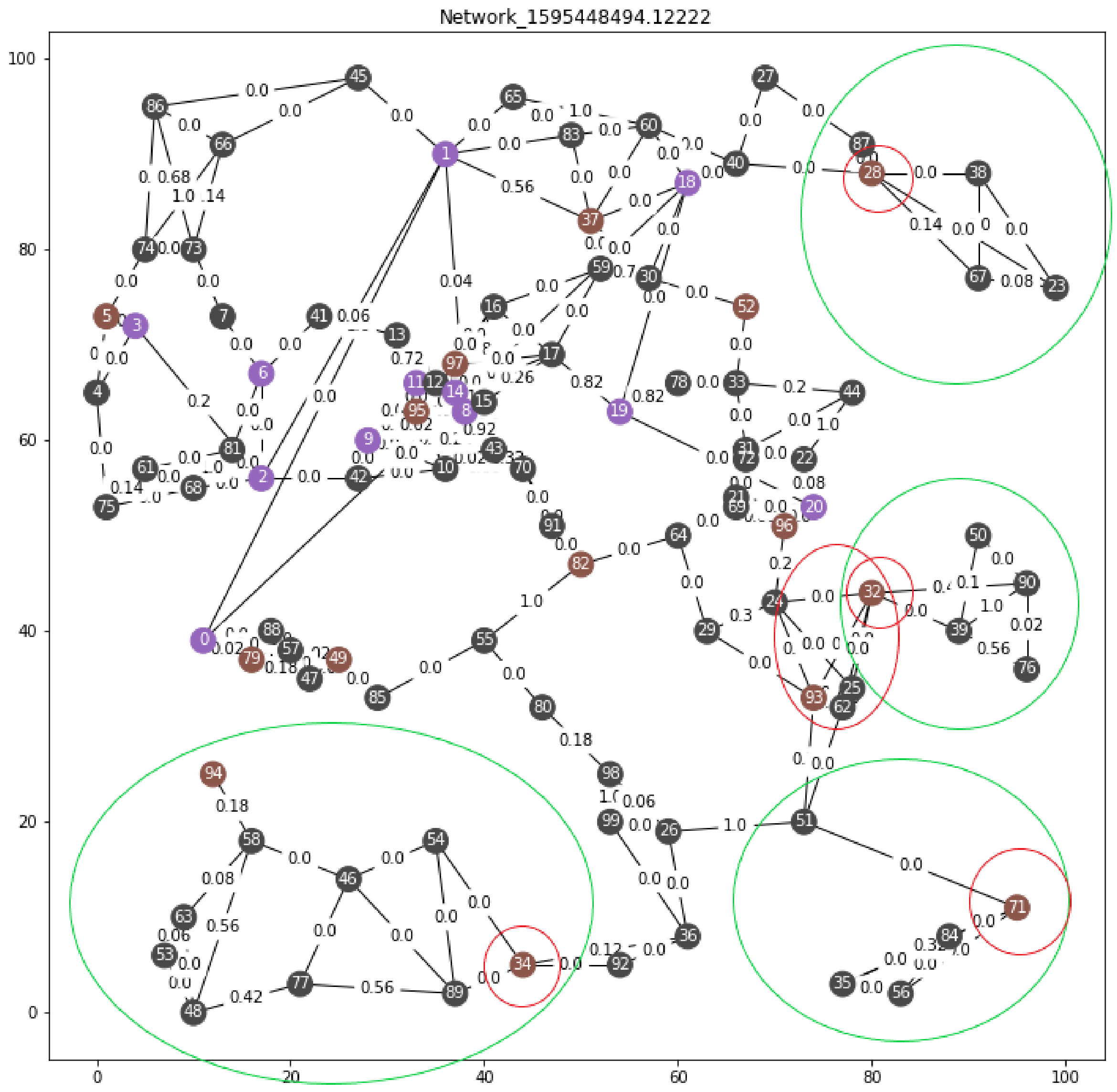

Figure 3: The components (circled green) have a PS node (circled red), but that node is the boundary, so there is no reason the rest of that component would ever be traversed.

- Added versioning to graphs, so multiple versions of the same graph can be compared.
- Made various improvements to increase graph readability and interpretability:
  - Increased font size.
  - Changed PS nodes to blue.

  - Thickened edges and added banded colors along paths between PS nodes, so that you can see where they diverge.
  - Removed edge labels.

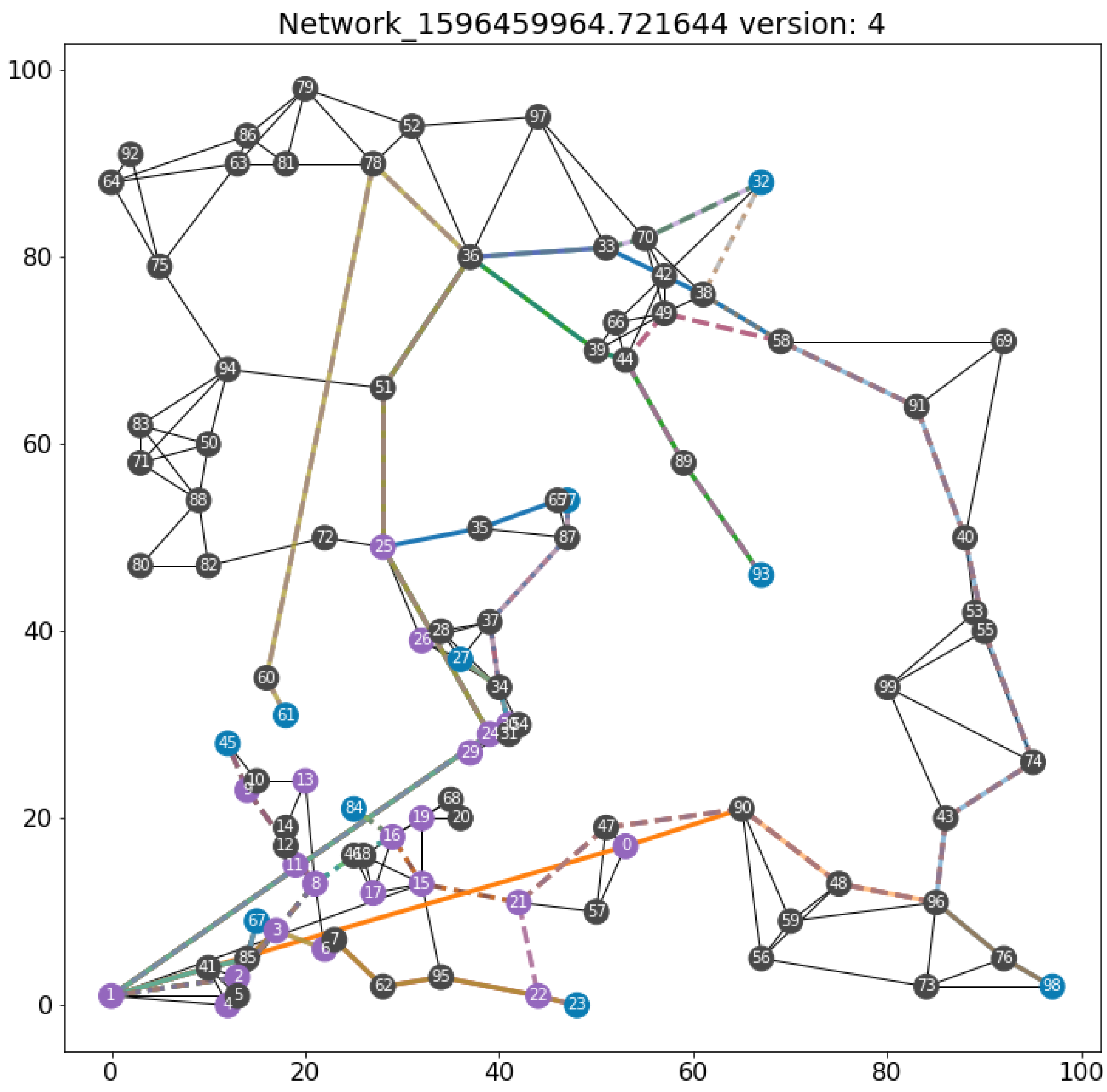

Figure 4: Making graphs more readable was helpful in providing a means of visually validating results and troubleshooting as I worked on network tomography.

Incorporating Real Distributions

Once I had a functioning toy network, I had to determine whether it was in any way a reasonable representation of the real network. In order to do this, I pulled information about the emergent characteristics of the toy network and the same information from the real network, first via Kibana and then directly using the `elasticsearch` client in Python (you can read more about these in Appendix A: Tools Used).

There were three primary metrics I looked at, which all pertained to the paths between perfSONAR nodes (because that is the only kind of information I have regarding the real network):

- The number of hops (how many nodes were along the path to any given destination from any given source).
- The total latency (sum of the edge lengths, which in the toy network was simply Euclidean distance).
- The percent packet loss (the product of packet loss along each edge).

In the real data, the number of routes with any given packet loss was heavily skewed toward 0, with a small spike at 100% (when a route had packet loss, it was much more likely to have lost all packets). By contrast, looking at all the PS pairs for one toy network, I discovered that we were nearing 100% packet loss, because I failed to account for the multiplicative nature of packet loss.

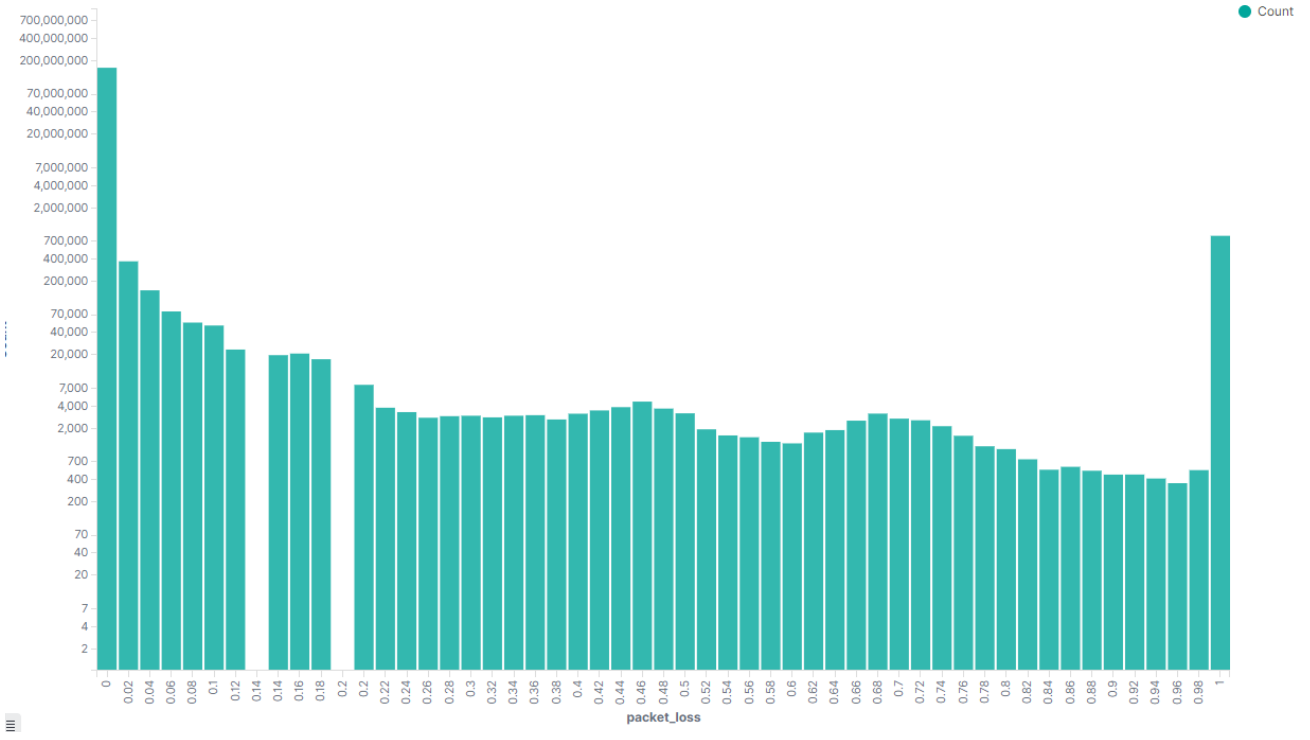

Figure 5: The count of paths is on a logarithmic scale, so is more heavily skewed than it appears at first glance.

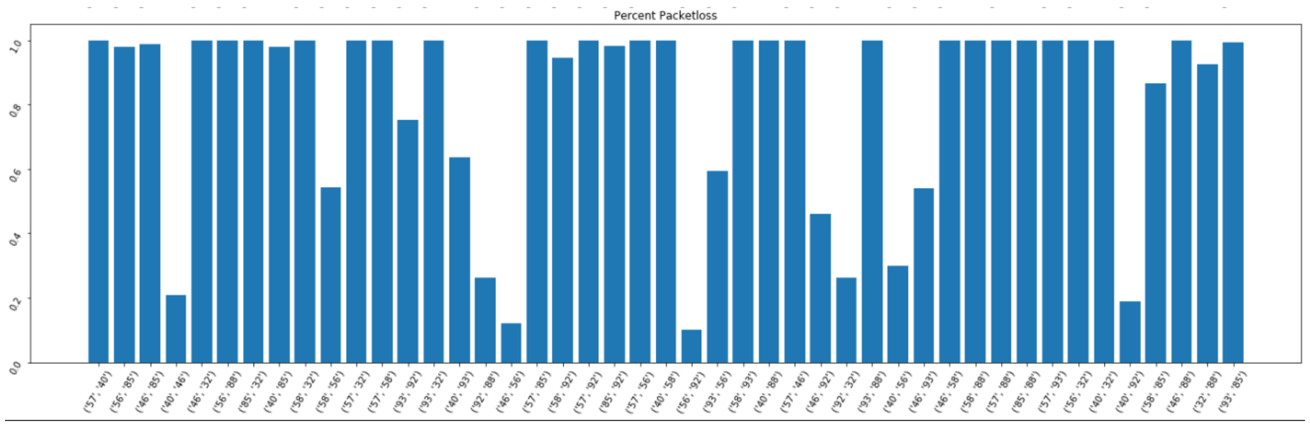

Figure 6: Most paths in my network were originally approaching 100% packet loss.

The latency in the real network proved problematic, as there were paths with negative latency, a peak near 0 latency and what looked exponential decay thereafter. It should be impossible to have even 0 latency, as there is inherently some delay in communicating information over any distance, so this data was clearly erroneous and likely a consequence of clock sync issues.

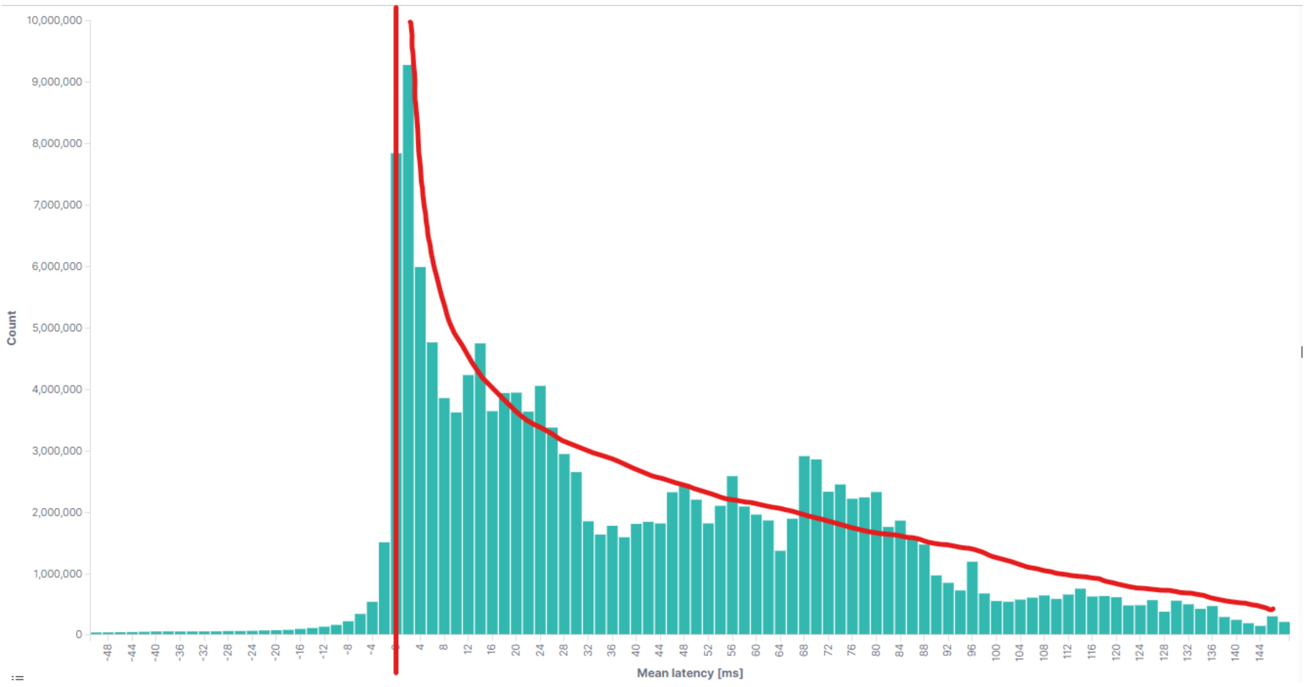

Figure 7: The real network had 0 and negative latency, which is impossible, accompanied by what appeared to be exponential decay.

As a stop-gap measure, for want of a better solution, we worked under the (probably incorrect) assumption that all latencies were simply offset by roughly 50 ms. We also took the square root of the counts, to create a better comparison with the scale of the toy network. Though not perfect, I was able to generate some toy networks that roughly resembled this modified distribution.

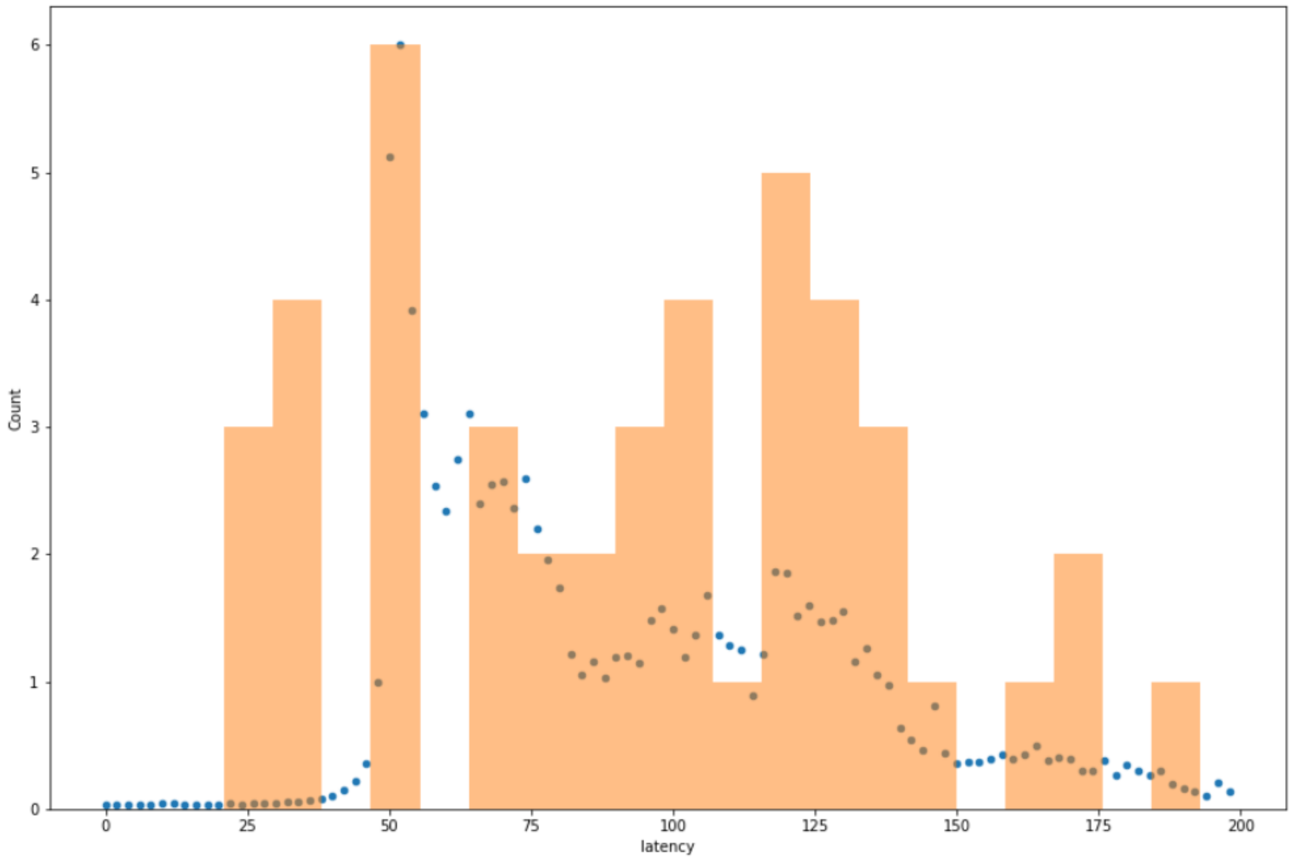

Figure 8: The blue dots represent the square root of the counts of the real network paths with a given latency, normalized to a max of 6. The orange histogram represents the count of paths in a given latency range.

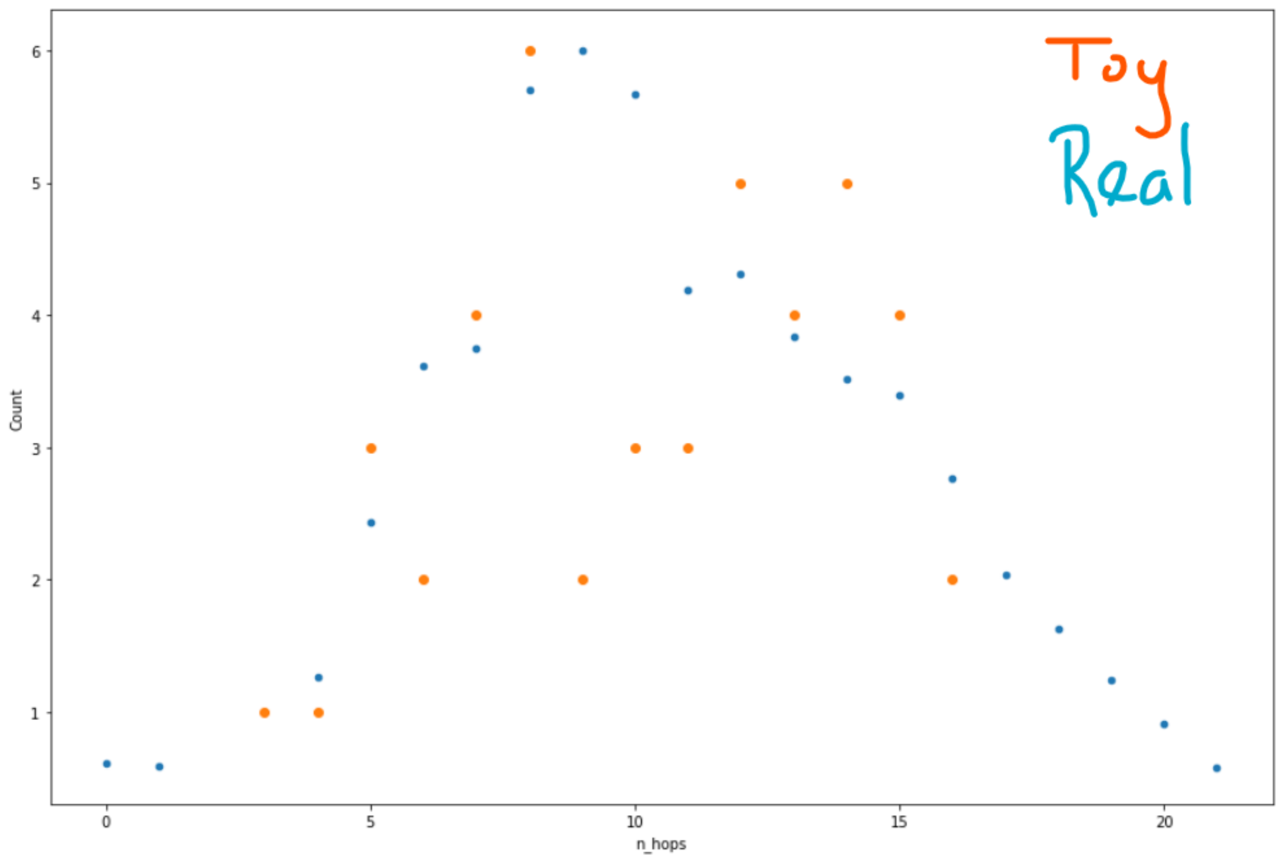

Figure 9: Though the counts required normalization, the average number of hops between PS nodes was actually a fairly decent match for what I was emerging from my procedurally generated toy networks.

Tomography on the Toy Network

Modelling the Real Network

Topics for Further Study

- –
- –
- –

Appendix A: Tools Used

Describing the network: `networkx`

For creating an underlying representation of the network, I used the `networkx` package in Python. This provided a graph object with various means of adding nodes, edges and metadata. It also provided tools for determining graph characteristics, such as bridge-connected components, k-connectivity and betweeness centrality.

Drawing the network and plotting data: `matplotlib`

For drawing the network, `networkx` integrates with `matplotlib`. This gave me a means of specifying (or not) the locations of nodes, as well as drawing, labeling and coloring specific nodes and edges.

I also used `matplotlib` in conjunction with `numpy` arrays and `pandas` dataframes in order to create various charts and histograms.

Exploring real network data: Kibana

For preliminary data exploration, I used Kibana, which provided visualizations such as histograms and tables. It also provided, via the Console and the Inspect tool (available on all visualizations), a means of testing and exploring how ElasticSearch's Query DSL works.

Querying real network data: ElasticSearch Query DSL

For pulling large amounts of data for analysis, I used the `elasticsearch` client in Python, which provides an API for ElasticSearch's Query DSL. More specifically, for small queries I used `search` which can provide only limited results. For larger amounts of data, I used `scan`, which is a helper for the `bulk` API.