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 PS nodes. These regularly transmit data to one another[3] and record certain characteristics of those transmissions, such as one-way delay (OWD), packet loss and latency.

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, OWD 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).

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

1. Create a toy network to use as a model.
2. Develop network 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 PS 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 motivated the construction of a toy network, one where we could define all nodes and their connections. With such a model, we can make changes, observe the impact on various facets of the network and be certain this impact was caused by our 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 traceroutes from 7/7/2020 to 7/14/2020 found 5968 unique nodes.
[2] The aforementioned survey found 423 perfSONAR (PS) nodes.
[3] The aforementioned survey found traceroute 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 we started with the following parameters:

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

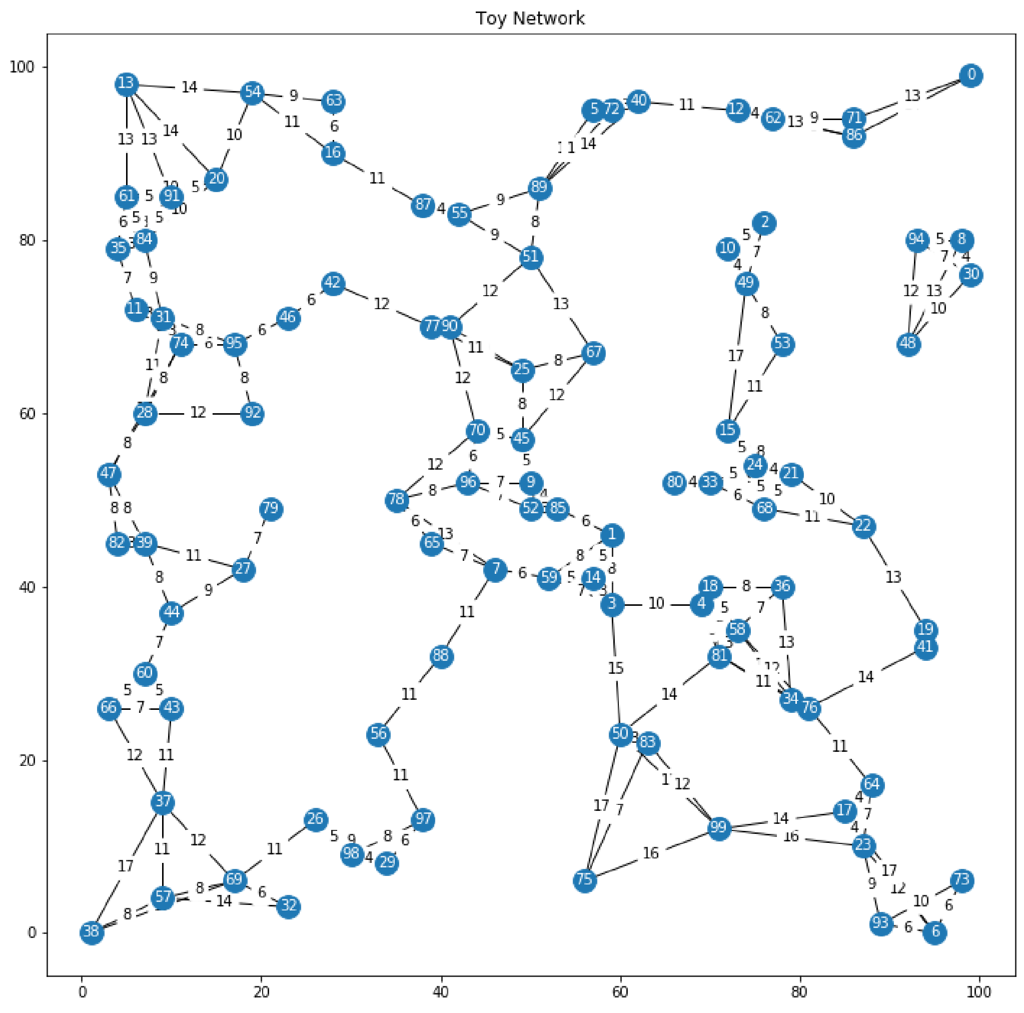

Figure 1: Our first networks had no guarantee of being connected, as seen by nodes 94, 8, 48 and 30.

There were some issues with such a naïve approach:

- There were no long edges (we know this doesn't resemble reality, even without exploring real data, because of 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" were indistinguishable from others and had no additional functionality.

Incremental Improvements

As we developed our toy network, we made a wide variety of incremental improvements. The list below highlights changes in roughly chronological order.

- We created hub nodes that served as a backbone for the network, which were randomly placed in quadrants and quadrants within those quadrants, recursively (it can be configured to any number of layers deep, though we ultimately settled on 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.
- We made sure that the network was k-connected (k could be configured, though we used 1-connected), which is to say that there is at least one component that could be separated from the rest of the network by cutting k edges.
- We colored special nodes (i.e., hub and PS nodes).
- We found the shortest paths between PS nodes, using the Euclidean distances between the nodes along the path.
- We gave each edge a packet loss (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.

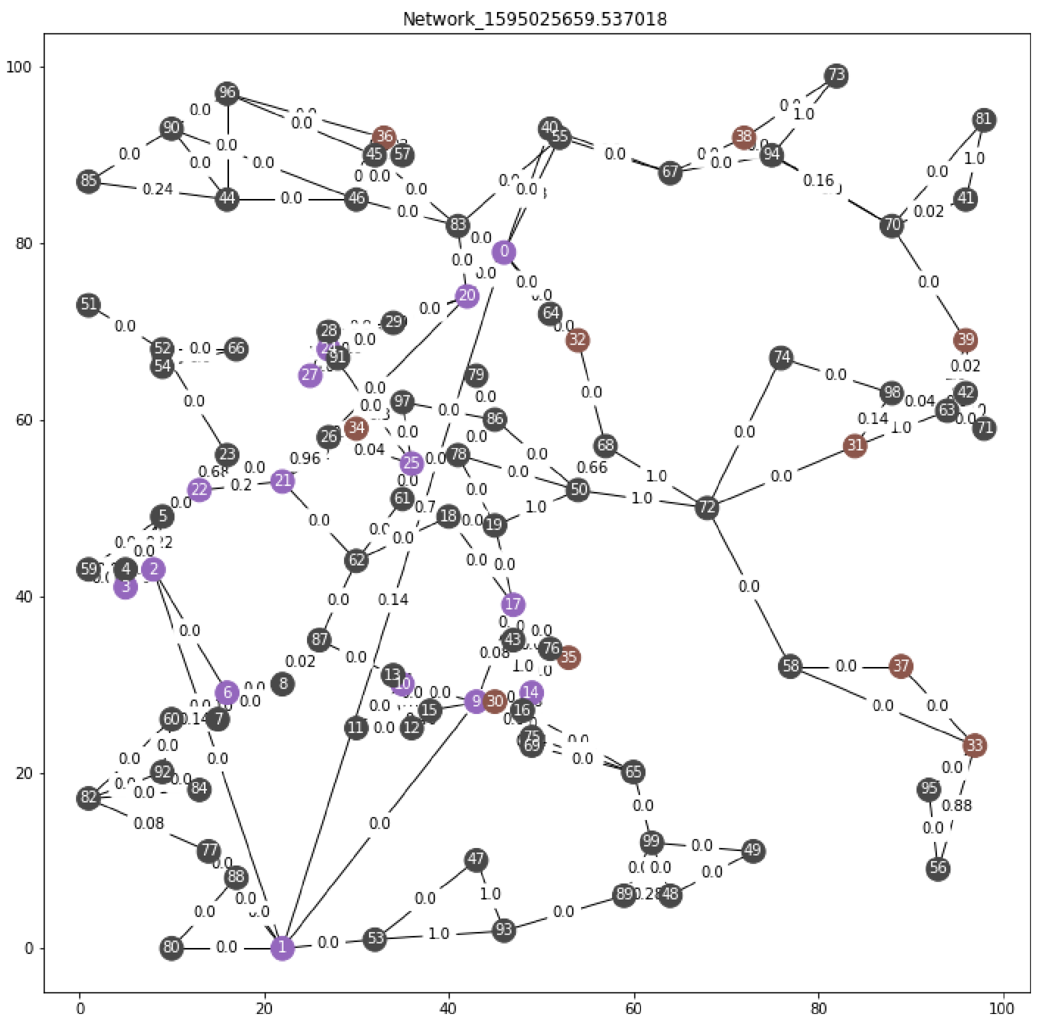

Figure 2: Hub (lilac) and PS nodes (brown) are colored and there are no unconnected components, but there are components without PS nodes (which would never have been seen/traversed on a real network) and the packet loss along the edges proved unrealistic.

- Ensured that all degree 1 nodes are PS 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 PS node (otherwise there would be no reason to ever traverse this component).
- Used low Betweenness Centrality (roughly a measure of how many paths would be disrupted if a node is removed) as a PS 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).

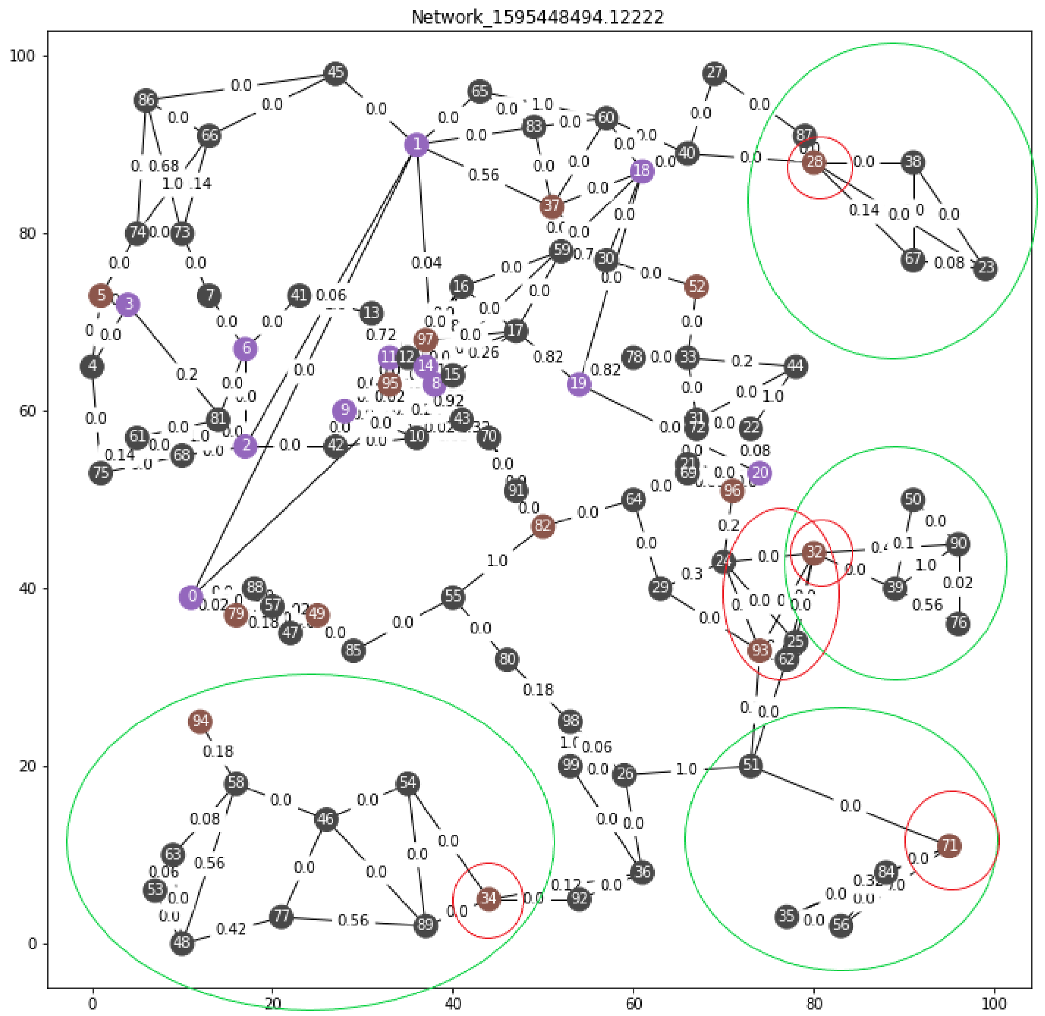

Figure 3: The components (circled green) have a PS node (circled red), but that node is the boundary, so there is no reason the rest of that component would ever be traversed.

- 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.
  - Added ability to specify the paths to highlight.
  - Removed edge labels.

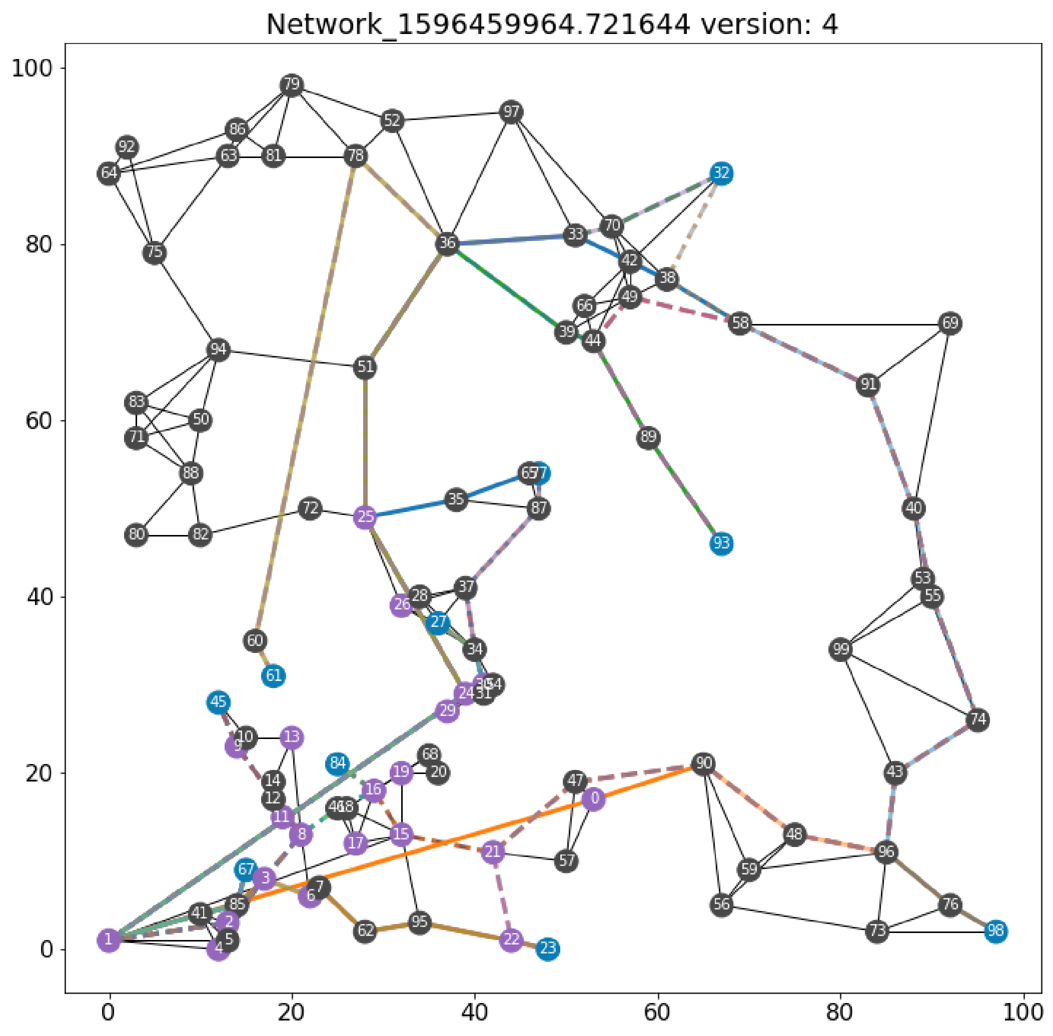

Figure 4: Making graphs more readable was helpful in providing a means of visually validating results and troubleshooting as we worked on network tomography.

Incorporating Real Distributions

Once we had a functioning toy network, we had to determine whether it was in any way a reasonable representation of the real network. In order to do this, we 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 B: Tools Used).

There were three primary metrics we looked at, which all pertained to the paths between PS nodes (because that is the only kind of information we 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 of a path).

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, we discovered that we were nearing 100% packet loss, because we failed to account for the multiplicative nature of 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.

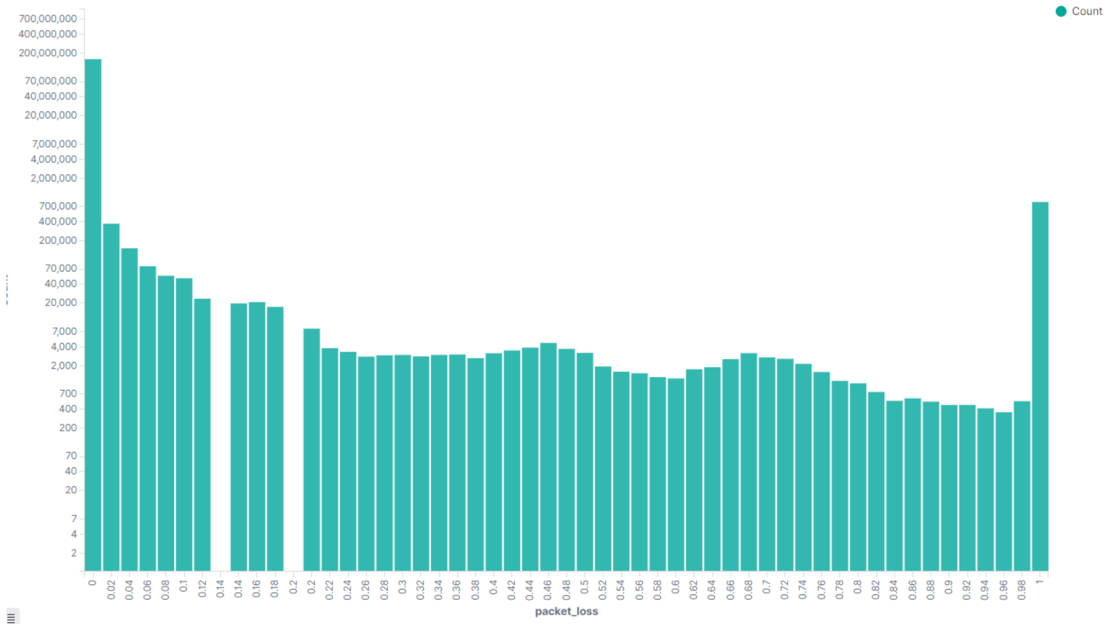

Figure 5: The count of paths is on a logarithmic scale, so is more heavily skewed than it appears at first glance.

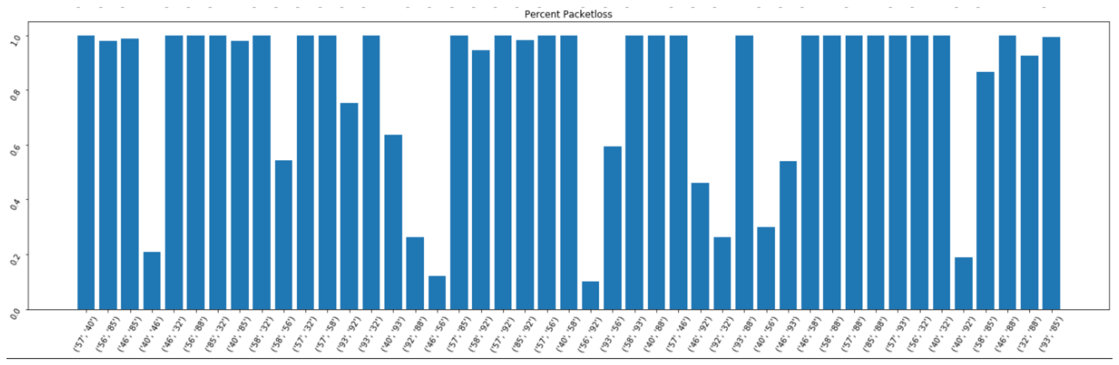

Figure 6: Most paths in our network were originally approaching 100% packet loss.

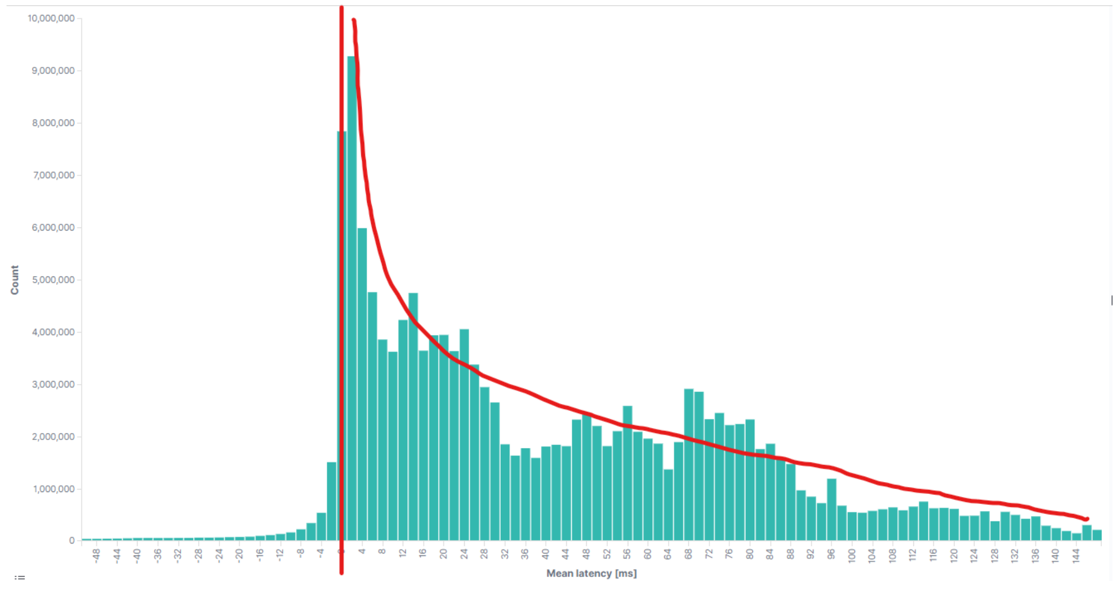

Figure 7: The real network had 0 and negative latencies, which is impossible, accompanied by what appeared to be exponential decay.

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, we were able to generate some toy networks that roughly resembled this modified distribution.

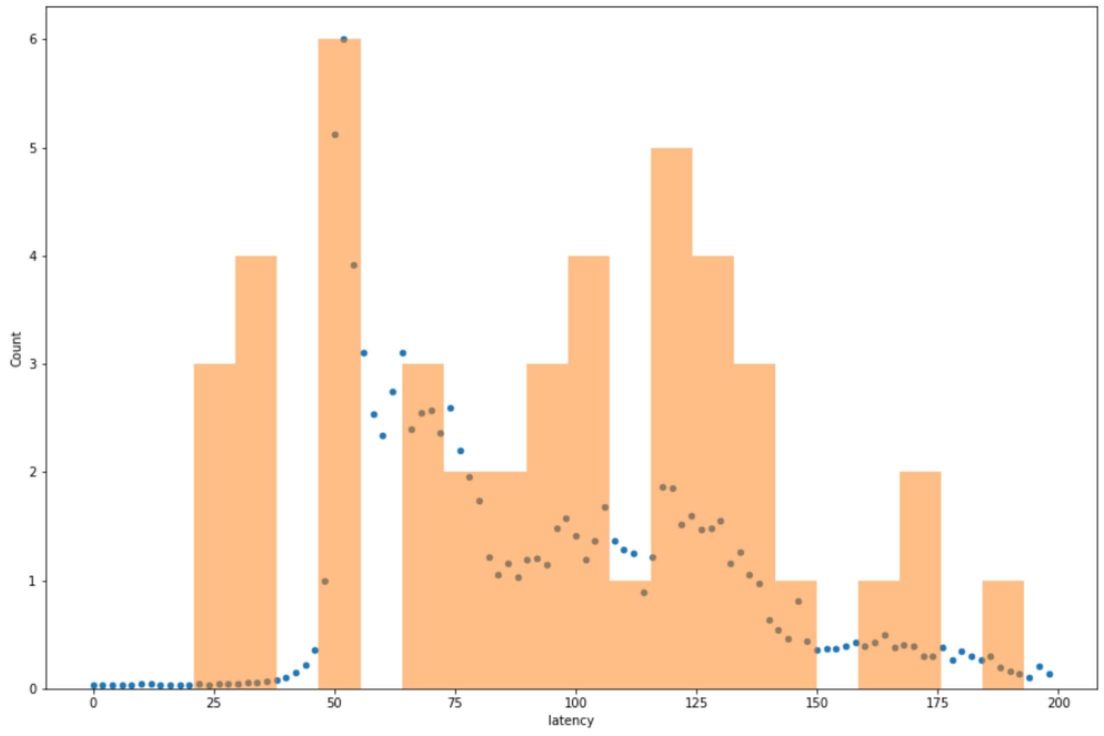

Figure 8: The blue dots represent the square root of the counts of the real network paths with a given latency, normalized to a max of 6. The orange histogram represents the count of paths in a given latency range.

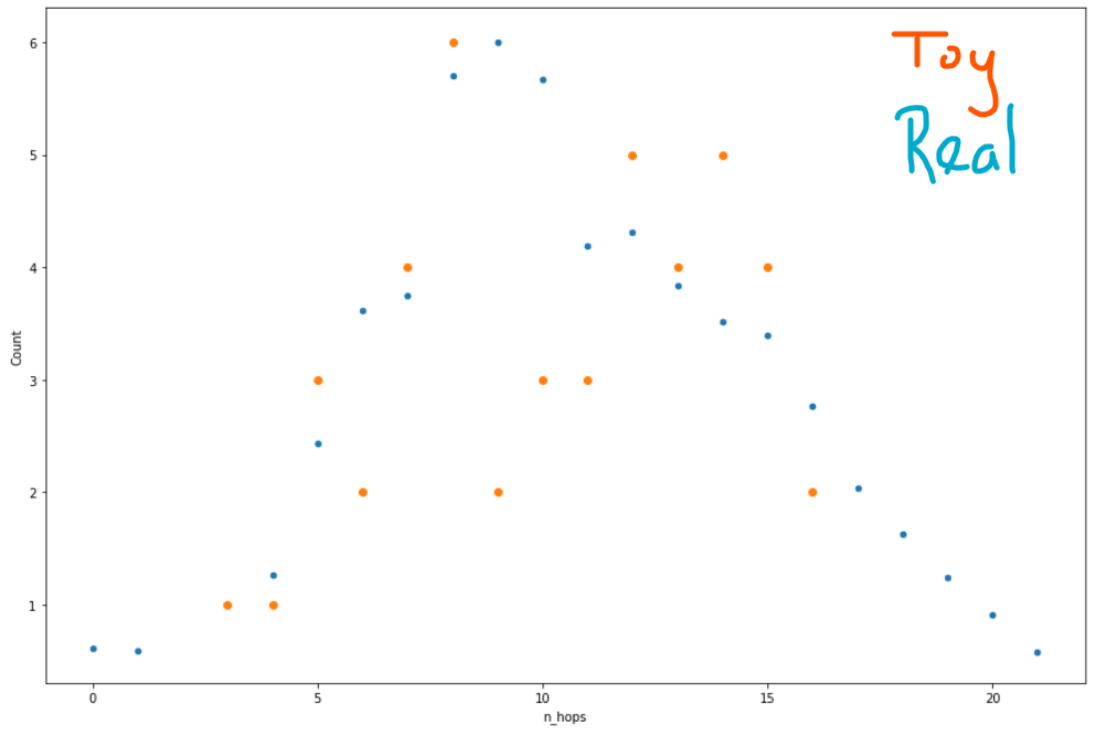

Figure 9: Though the counts required normalization, the average number of hops between PS nodes was actually a fairly decent match for what we was emerging from our procedurally generated toy networks.

Tomography on the Toy Network

Once I'd built out a fairly robust toy network, that at least somewhat resembled the real one, and the tools to manipulate it (e.g., remove edges, calculate shortest paths, etc.), we were able to manipulate the network, observe changes in network metrics and determine whether it was possible to infer what had been manipulated from those results. The primary case we considered was edge deletion.

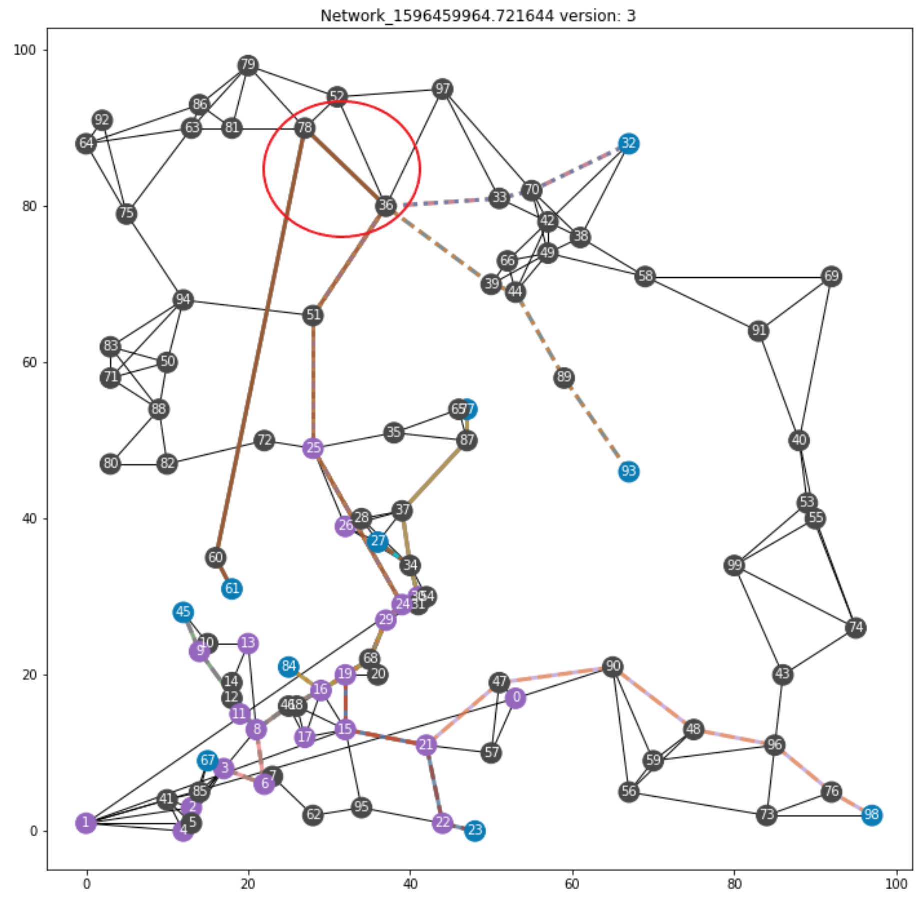

Figure 10: A toy network, G, before any edges have been removed.

We used the following process to determine which edges were deleted:

1. Determine the shortest paths between all pairs of PS nodes in a toy network, G.
2. Make a copy of that network, H.
3. Delete a single edge in H.
4. Determine the shortest paths between all pairs of PS nodes in H.
5. Compare the shortest paths in H to those in G:
    (a) Record edges on each path in G that aren't on the corresponding path in H (i.e., edges that were potentially removed).
    (b) Confirm that those edges were entirely removed from H (i.e., they are not on any other paths in H).
    (c) Determine how many different paths each edge was removed from.
    - It's likely that the edge removed from the most paths is the deleted edge.
    - If multiple edges were removed an equal number of times, there is ambiguity as to which was deleted.
6. Repeat 2-5 for each edge in the network.

While experimenting with this process, we also drew the network at each phase and highlighted paths, so that we could see the removed edge and the paths that were rerouted as a result. We plotted the impact of these changes on network metrics like latency, packet loss and the number of hops on paths.

Latency necessarily increased, as we determined shortest path based on latency, so an edge removed from that path could not be replaced by one with smaller latency. It was quite possible, however, for the total number of hops or the packet loss to go down as a result of moving edges, because in our toy network these were not factors in determining paths.

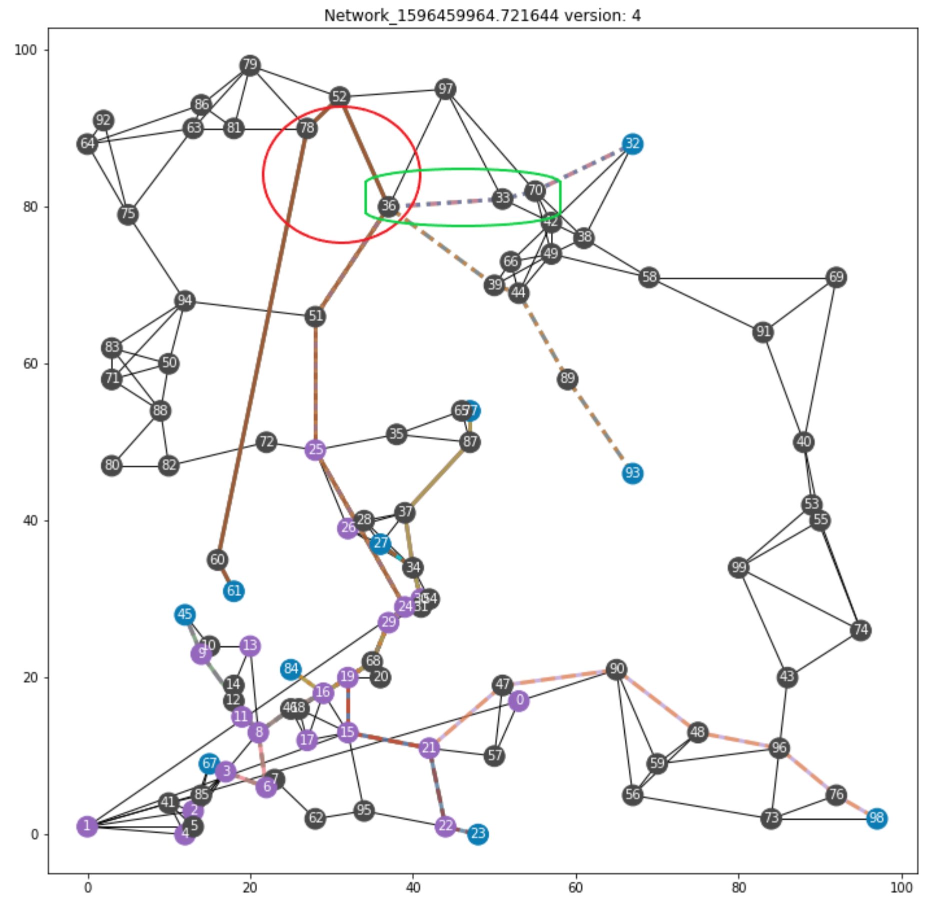

Figure 11: The toy network, H, a copy of G with the edge from 78 to 36 removed. Note that $(78, 52)$ and $(52, 36)$ are now on a path and that some paths are still using $(36, 33)$ and $(33, 70)$. The significance of that second point is highlighted in the next figure.

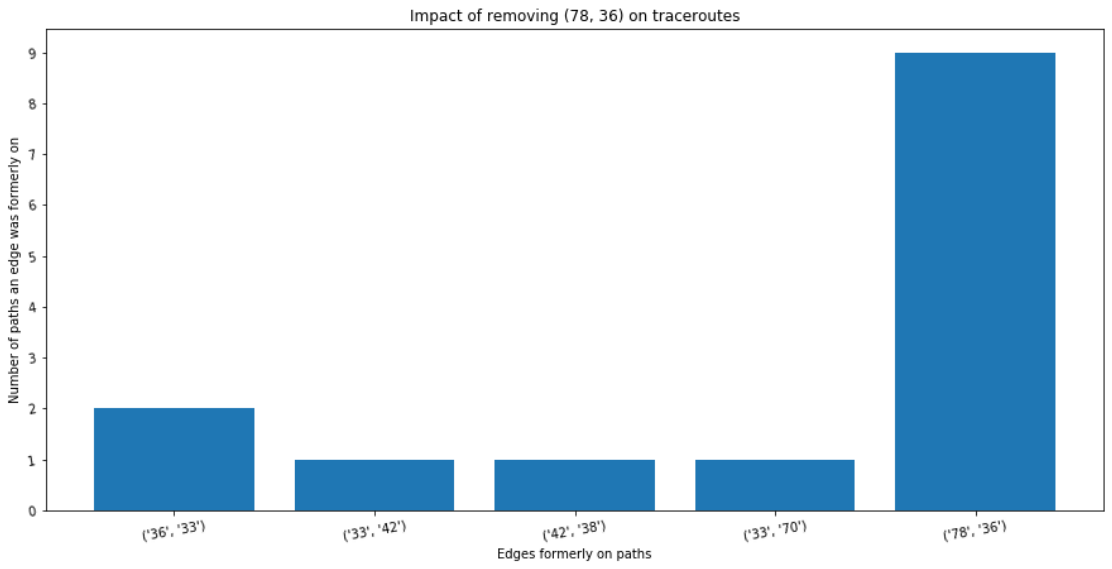

Figure 12: Only 5 edges were removed from paths as a result of the change. It is pretty clear that $(78, 36)$ is the removed edge, as it was removed from more than 4 times as many routes as any other. Furthermore, we can factor out edges that appear along some other path in H, such as $(36, 33)$ and $(33, 70)$.

In a real network, a connection between successive hops is rarely completely severed. This strategy would need to be adapted to consider some performance threshold on the real network as being equivalent to an edge being deleted in the toy model.

Modelling the Real Network

In order to begin applying the lessons learned from the toy network to the real one, we first needed a comparable representation of the real network or some part of it. The data aggregated by real PS nodes are indexed several different ways in ElasticSearch to make it easier to track different characteristics. The two indices we were particularly concerned with were `trace_derived_v2`, which has summative data about all paths in the network over the entirety of our records, and `ps_trace`, home of the individual traceroute records the former is derived from.

From the aggregated data in `trace_derived_v2`, we were able to get a list of all PS nodes that have served as a source or a destination. Originally, we looked to whittle down these pairs to form a list of paths to look for in the `ps_trace` data. For example, we removed ones that had an average number of hops that was less than 1 (unlike clock sync, we can't really come up with a reason why this would even be recorded as such, but it is clearly impossible to get from one endpoint to another without taking any hops).

We ultimately scrapped the approach of excluding routes based on particular criteria in favor of collecting data on all routes over a given time period and building a core network from that. Having collected 7 days worth of `ps_trace` data, we looked at how long, on average, the path between pairs of PS nodes stayed stable.

The results were encouraging, as the majority of paths lasted the entire 168 hours without changing even once. When counting pairs of PS nodes that had shorter average path lives, the number of paths dropped precipitously as the life of path decreased.

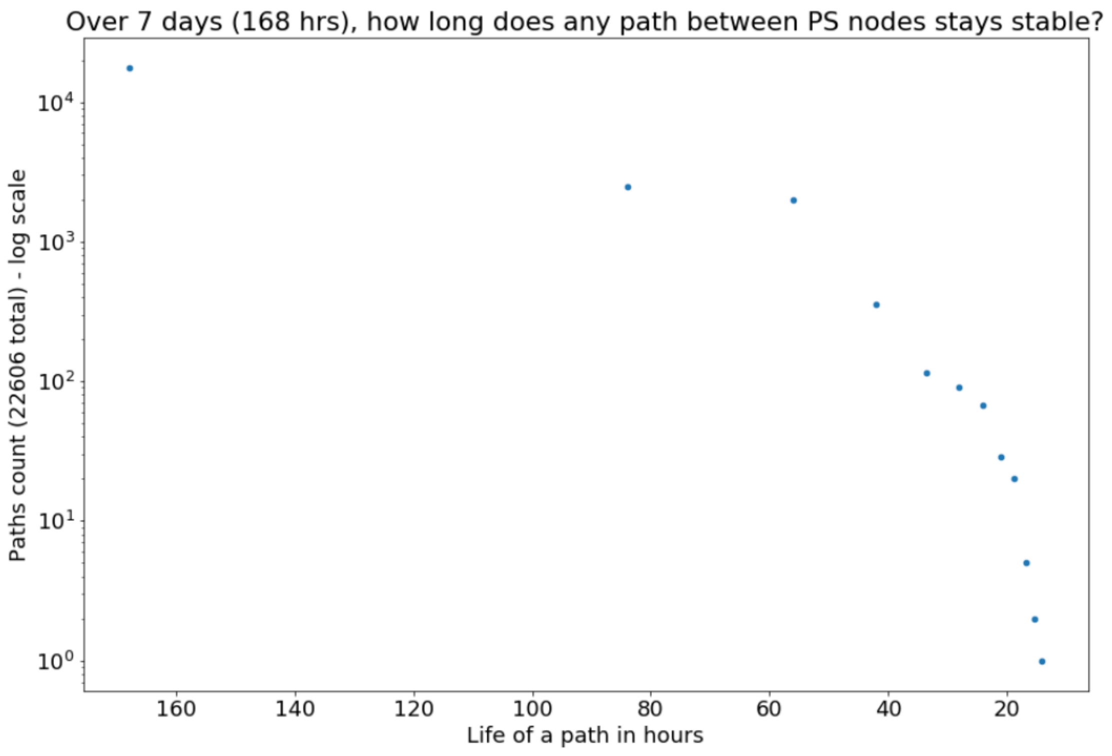

Figure 13: Most of the 22,606 pairs of PS nodes had a path that remained stable through the entire 7 days of data (from 7/7/20 – 7/14/20).

After determining that most routes are fairly stable, we determined what edges were most central to the network, as follows:

1. We created a list of all unique routes and their frequency.
2. We created a list of edges on each route, by combining all pairs of adjacent hops.
3. For each edge, we made a weighted sum of the number of unique routes it occurred on (weighted by the frequency of that route's occurrence).

4. We ordered put this list of edges in reverse order by count, to get a list of edges from the most frequent (a measure of centrality) to the least frequent.

After creating a list of the most frequent edges, we took two different approaches to visualizing them. For both we used the spring layout provided by the `networkx` module, but in the first case we graphed only the n most frequent edges, without creating a complete graph of the network. The results showed that the most central edges did tend to be connected, but that there were smaller high frequency clusters of edges that were presumably within a particular region (e.g., the PS nodes in the UK are all configured to send messages to each other more frequently than to other nodes on the network).

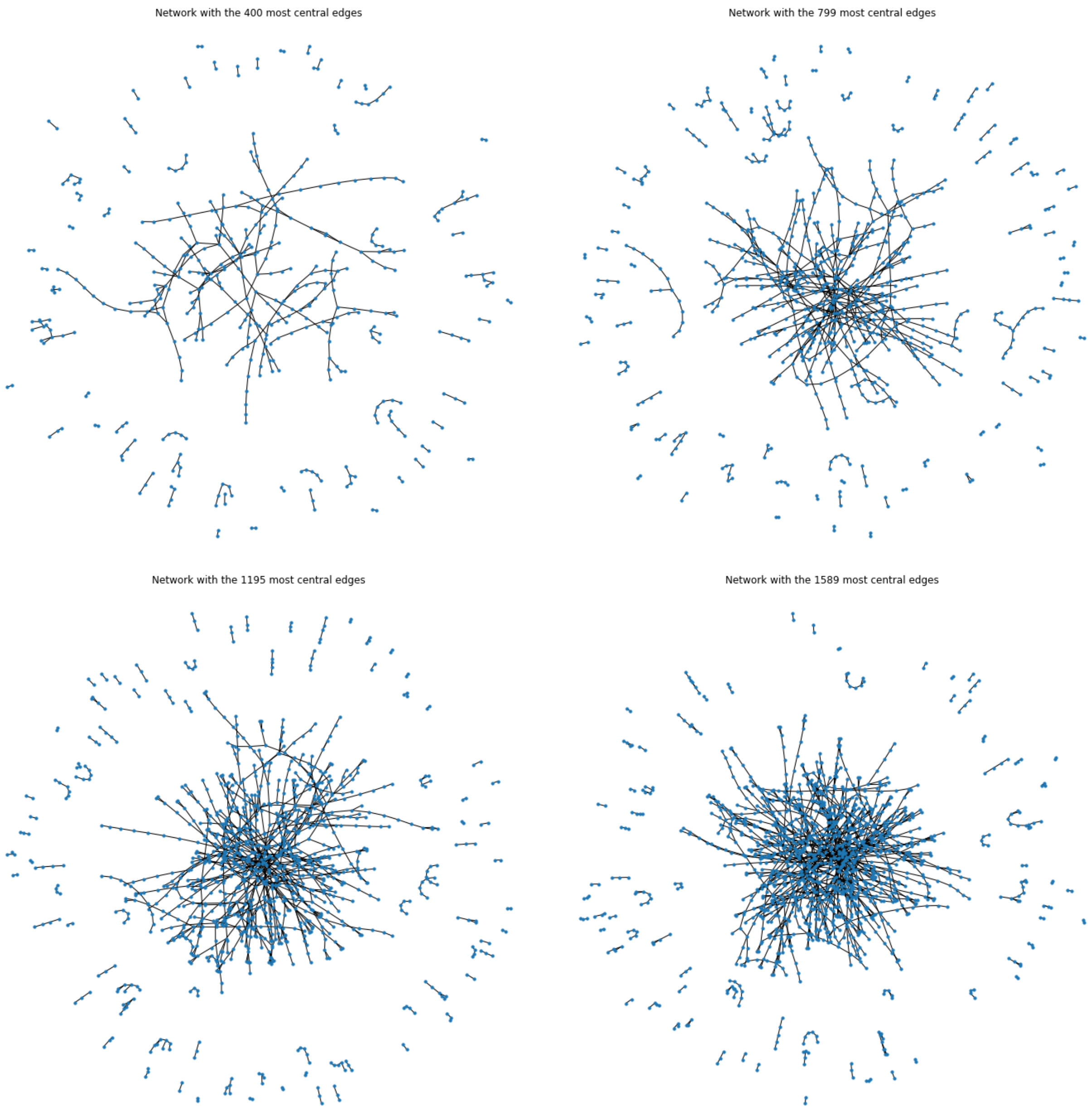
Network with the 400 most central edges
Network with the 799 most central edges
Network with the 1195 most central edges
Network with the 1589 most central edges

Figure 14: Redrawing the network with the 400 most central edges, the 800 most central and so on, shows that there are clearly clusters of central edges and ones on the periphery probably represent more localized networks.

This activity on the periphery eventually decreased (i.e., the clusters became connected to the larger network) as we increased the how many of the most frequent edges we considered. While this was expected, we also began to note a less expected phenomena.

The number of most central edges actually being plotted was not quite tracking with the number of edges being supplied. The deviance increased/became more apparent as the number of edges to be plotted increased. Upon investigation the reason for this appeared to be symmetric edges, which were being supplied twice (e.g., once as (a, b) and once as (b, a)), but only graphed once. We would have expected this to be a very common occurrence, but, upon further investigation, only about 2% of edges were symmetric.

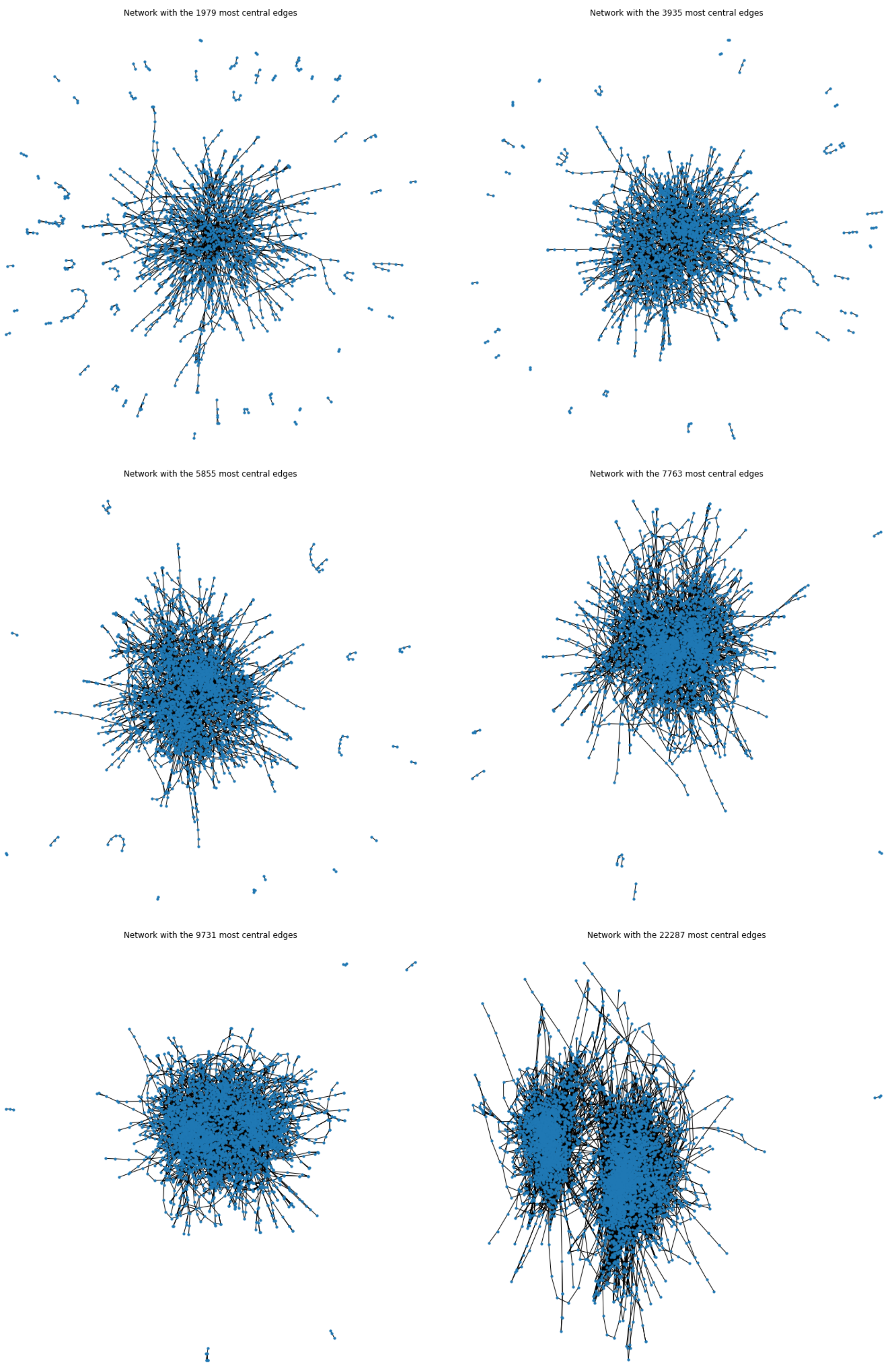

Figure 15: As the number of edges increased, the number of edges that remained unconnected decreased.

There was some interesting phenomena once we finally reached a complete graphing of the network, namely, the graph clearly split into two distinct clusters. This presumably represents the continental divide, but what was interesting about this is that the edges splitting the clusters were not of high enough frequency that the clusters became distinct early on.

To get an idea of how the network builds up as we increase the number of edges, it seemed prudent to repeat the process, but having fixed the entire graph layout before drawing any edges. This meant keeping the relative positions of all the nodes constant, but only drawing those incident to the n most frequent edges.

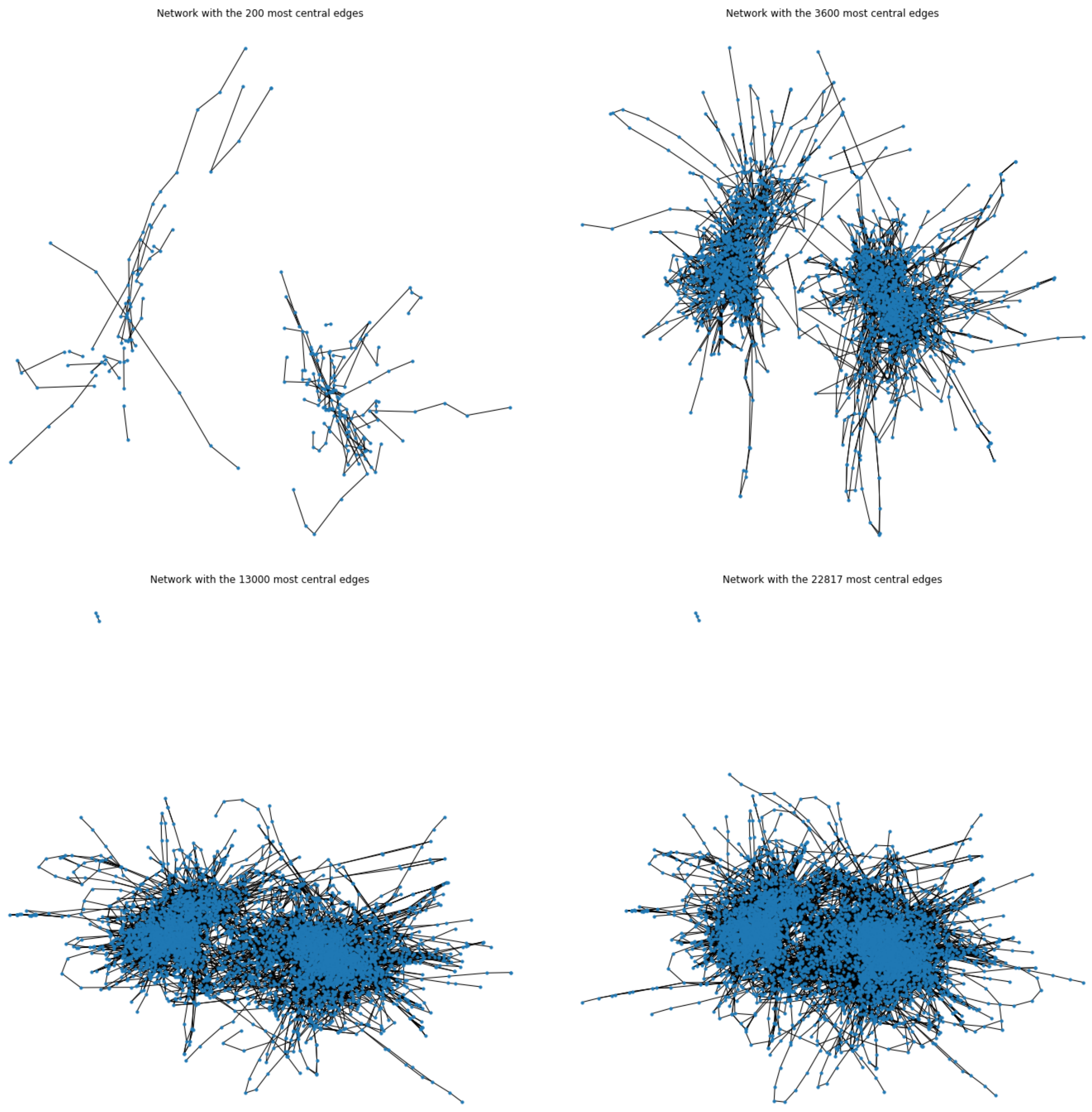
Network with the 200 most central edges
Network with the 3600 most central edges
Network with the 13000 most central edges
Network with the 22817 most central edges

Figure 16: The continental divide is more apparent when graphing edges in the context of their position in a spring layout for the entire network.

It appears that the majority of PS nodes are in most frequent communication with other nodes that are in the same region (as opposed to ones on the other side of the Atlantic), which explains why that divide is not apparent until very late when graphing without the context of the entire network.

Topics for Further Study

With the determination of which edges are most frequent and the ability to construct graphs of the real network in the same fashion we did graphs of the toy network, the door is open to further investigation and transference of lessons learned on the toy network to use with the real network. Below is a brief list of just a few outstanding questions/topics that would be worth exploring further:

- How significant is the impact of the asymmetry of edges on the network?
- Are there ways to account for this asymmetry?
- Similarly, there may be multiple representations of the same devices and PS nodes on the network, most notably IPv4 vs. IPv6 identifiers; can we map different representations of a device to the same device?
- Knowing that certain edges are very central to the network, can we monitor those specific edges for potential performance issues, to allow us to identify network issues sooner?
- Alternatively, could we use anomaly detection to identify when to investigate some portion of the network?
- Would the same strategy used with the toy network (determining edges that were removed from multiple impacted paths), allow us to pinpoint issues on the real network?

Appendix A: Terms

Glossary

***k*-connected** A graph is said to be *k*-connected, if removing *k* edges somewhere in the graph would result in the graph no longer being connected. 4, 16

anomaly detection The identification of events on the network that are outside the norm. 2, 15

Betweenness Centrality The centrality of an vertex in a graph can be defined in many ways. Betweenness centrality does so by calculating the number of paths along which a given node is essential (i.e., without the node, there would no longer be a connection between two other nodes). 5

boundary A node or set of nodes that are between two subgraphs. 5

bridge-connected A subgraph connected to the rest of the graph by a single edge (a bridge, which would become a separate component if that edge were removed. 5

component A cohesive subgraph containing any number of connected nodes. A connected graph has one component (the entire graph), but a *k*-connected graph would break into more, smaller components if particular sets of *k* edges were cut. 3, 4, 5, 16

connected A graph is said to be connected, if all nodes can be reached from all other nodes.. 3, 16

degree The number of edges connected to a node.. 4

endpoint The devices that serve as the source or destination of a transmission along the network. While any device could serve as an endpoint, the only endpoints we are concerned with (as they are the only ones regarding which we have data), are perfSONAR nodes. 2, 16

hop Each step along the network from one device to the next (and sometimes within the same device). The hop is usually documented via traceroute and is identified by the Internet Protocol, though usually synechdoche for IP address (IP) of the device the hop arrives at. 6, 8, 9, 10, 11, 16

hub A node with significantly more links than average. 4

IP address An Internet Protocol (IP) address is a 32-bit (IPv4) or 128-bit (IPv6) number that identifies a device on a network. 16, 17

latency The amount of time it takes for one bit of data to travel along a network from one endpoint to another. 2, 3, 6, 7, 8, 9, 10, 16

network tomography The study of the shape, state and other characteristics of a network using only data gathered from limited set of endpoints. 2, 6

node A device connected to the network which may serve as a hop between two endpoints. 2, 3, 16

one-way delay The amount of time it takes for data to be transmitted from a source PS node to a destination. This is measured by the difference in clock measurements between endpoints. 2, 17

packet loss The percentage of packets of data that failed to reach their destination. The Transmission Control Protocol (TCP) identifies and re-transmits lost packets, but this can slow transmission down to a trickle. 2, 4, 6, 7, 9, 10, 16

perfSONAR Short for performance Service-Oriented Network monitoring ARchitecture, it is a network measurement toolkit designed to provide federated coverage of paths and help to establish end-to-end usage expectations. 2, 3, 16, 17

perfSONAR node Network endpoints equipped with perfSONAR that send regular communications to one another in order to collect network measurements (i.e., packet loss, latency and OWD). 2, 3, 16

traceroute Data collected about the hops between two endpoints. 2, 11, 16

Acronyms

IP Internet Protocol, though usually synechdoche for IP address. 16

LHC Large Hadron Collider. 2

OWD one-way delay. 2, 16

PS perfSONAR. 2, 4, 5, 6, 8, 9, 11, 12, 14, 15, 16

TCP Transmission Control Protocol. 16

Appendix B: Tools Used

Describing the network: `networkx`

For creating an underlying representation of the network, we 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.

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