PATHly Visualizer

Exploring Anycast Stability in IPv4 and IPv6

Jack Campanale

Jasper Meggitt

Yash Patel

Cindy Trac

in collaboration with

fastly WPI

PATHly Visualizer: Exploring Anycast Stability in IPv4 and IPv6

A Major Qualifying Project submitted to the Faculty of
WORCESTER POLYTECHNIC INSTITUTE
in partial fulfillment of the requirements for the degree of Bachelor of Science

by

Jack Campanale
Jasper Meggitt
Yash Patel
Cindy Trac

December 16, 2022

Salman Saghafi, Stephen Strowes
Fastly
Professor Mark Claypool
Worcester Polytechnic Institute

ABSTRACT

Modern Internet services make significant use of IP Anycast, where multiple locations are identified by the same IP range, and the network distributes traffic across those locations. However, Anycast traffic management is difficult and IPv6 deployments are often less mature than their IPv4 counterparts. This project aimed to build a tool that utilizes publicly available measurement data to identify and correlate changes between IPv4 and IPv6 Anycast routing. We created PATHly Visualizer, a web-based traceroute pathing tool, to visualize paths that packets take from a source probe to Fastly Anycast destinations. PATHly Visualizer allows users to interactively display traceroute data and better understand and evaluate Anycast forwarding in IPv4 and IPv6.

ACKNOWLEDGMENTS

This project would not have been possible without the help and guidance of the following people. We'd like to extend special thanks to

- **Salman Saghafi** and **Stephen Strowes**, who served not only as our Fastly sponsors but also our mentors. They met with us every day during stand-up and were always readily available to help.
- **All the Fastlyans who took the time to chat with us:**
    - **James White,** who gave us a deeper look into Fastlyan culture and diversity.
    - **Glitch Team** for letting us pick your brains on all things Glitch.
    - **WPI alum** at Fastly for making time to speak with us and giving us insight into industry and life beyond WPI. Specifically, thank you to **Patrick McManus** for all giving us valuable feedback.
- **Professor Mark Claypool** for coordinating this experience and guiding us along the way.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF ACRONYMS

API	Application Programming Interface
AS	Autonomous System
ASN	Autonomous System Number
BGP	Border Gateway Protocol
CDN	Content Delivery Network
CI	Continuous Integration
CPU	Central Processing Unit
CSS	Cascading Style Sheets
DDoS	Distributed Denial of Service
DNS	Domain Name System
GCP	Google Cloud Platform
ICMP	Internet Control Message Protocol
IP	Internet Protocol
JS	JavaScript
JSON	JavaScript Object Notation
MQP	Major Qualifying Project
MUI	Material User Interface
NPM	Node Package Manager
POP	Point Of Presence
REST API	Representational State Transfer Application Programming Interface
RIPE NCC	Réseaux IP Européens Network Coordination Centre
RIS	Routing Information Service
RTT	Round Trip Time
SVG	Scalable Vector Graphic
TCP	Transmission Control Protocol
TTL	Time-To-Live
UI	User Interface
UML	Unified Modeling Language
VM	Virtual Machine

AUTHORSHIP

Section			Writer	Editor
Abstract			All	All
Introduction			Cindy	Jack
Background	Introduction		Jack, Cindy	Cindy
	Packet Routing		Jack, Cindy	Cindy
	IP Addresses: IPv4 and IPv6		Jack	Cindy
	Border Gateway Protocol		Jack	Cindy
	Confirming Network Connectivity through Ping		Yash	Jack, Cindy
	Path Tracing with Traceroute		Yash	Cindy
	Anycast and Its Use Cases for CDNs		Cindy	Jack, Cindy
Methodology	Introduction		Jack, Cindy	Cindy
	Evaluation of Anycast		Jack, Cindy	Cindy
	Project Management		Cindy	Cindy
	Security Concerns		Jack	Cindy
Implementation	Back-end	Technologies	Jasper	Cindy
		Learning Golang	Jasper	Cindy
		Service Model	Jasper	Jack
		REST API Routes	Jasper	Jack
		Traceroute Data Ingestion	Jasper	Jack
		Data Parsing and Storage of RIPE Atlas Probes	Yash	Jasper
		Cleanup Service	Yash	Jasper
	Front-end	Introduction	Cindy	Cindy
		Technologies	Jack	Cindy
		Data Representation	Jack	Cindy
		Editing Measurements	Jack	Cindy
		Hosting	Jack	Cindy
	Summary		Yash	Jack, Cindy
Future Work	Full Route Implementation		Jack	Cindy
	More Continuous Integration		Jack	Cindy
	Additional Public Data Sources		Jack	Cindy
	Better Visualize Difference Between IPv4 and IPv6		Jack, Cindy	Cindy
	Better Visualize Most Frequently Changing Paths		Jack	Cindy
Conclusion			Jack, Cindy	Jack, Cindy

1. INTRODUCTION

Modern Internet services make significant use of IP Anycast, where multiple locations are identified by the same IP range, and the network distributes traffic across those locations. If one location is taken offline, traffic will be forwarded to one of the remaining locations within the same IP range. Anycast allows for increased reliability and load balancing. The closer a network operates to the user, the faster the service. This enables service providers to process network data closer to users. This makes IP Anycast a valuable tool for *Content Delivery Networks (CDNs)* like Fastly.

While Anycast is beneficial, it is difficult to manage in a variety of ways. Packet routing is variable, meaning packets can take several path variations going from a source to a destination. It is never completely clear what path a packet is going to take in Anycast or when a sudden path change might be made. In addition, the difference in existing infrastructure for *Internet Protocol version 4 (IPv4)* versus *Internet Protocol version 6 (IPv6)* adds another layer of challenge. On the surface, the main difference between IPv4 and IPv6 seems to be that they support different bit-size addresses, 32-bit and 128-bit respectively. From this key difference, it might seem like IPv6 is simply an extension of IPv4. However, the two protocols are managed separately from one another, implying that a network may take different routing choices between IPv4 and IPv6. IPv6 is also not as widely deployed as IPv4, so fewer paths are available.

Our goal is to provide Fastly with a more concrete way of analyzing Anycast path visibility in both IPv4 and IPv6. To do so, we utilized RIPE Atlas, an open, global network of Internet connectivity measurement probes, as our main source of data. Although RIPE Atlas already provides raw data and insight into how the Internet operates, it is difficult to follow. In addition, raw traceroute data does not combine IPv4 and IPv6. At this time, network engineers have a difficult time analyzing these large files of text. Making comparisons between different paths is even more challenging. Our tool, *PATHly (Probe ASN Traffic Highway) Visualizer*, takes raw traceroute data and displays it graphically. PATHly visualizes the path a packet takes from a source probe to its destination. This provides a more user-friendly interface to analyze and gain further insight into Anycast forwarding.

Through this report, we detail our research, methodology, and implementation of PATHly. Chapter 2 of this report provides necessary background information on internet routing. Chapter 3 discusses the team's approach to evaluating Anycast and project management.

Chapter 4 describes our implementation of PATHly through documentation on both the back-end and front-end of the code. Finally, Chapters 5 and 6 wrap up with future applications and our concluding takeaways.

2. BACKGROUND

As technology continues to adapt and change, so must the systems currently in place to support and distribute it. By experimenting with the ways in which we are connected, we can gain insight into how to improve network systems. This background chapter describes various topics of computer network routing and how they come together to connect billions of users worldwide every day.

2.1 INTERNET ROUTING

Internet routing allows computers to connect with one another, enabling the sharing of information between different networks. Data sharing is only growing more prevalent as connectivity via messaging, social media, and other avenues become less of a luxury and more of a staple of everyday life. This growing need for fast and efficient communication serves as a motivator for more efficient computer network routing.

Packet Routing

Computer network routing enables networks to find the most efficient path for Internet packets to be sent from a source to a destination. *Packets* are small segments of larger messages, recombined at their destination to form the original message.[1] One goal of computer network routing is to find the best route packets can take. Routes can be measured by *Round Trip Time (RTT)*, which is the time it takes a packet to go from source to destination and receive an acknowledgment that it reached its destination. Routing a packet can follow one of four delivery schemes[2]:

1. **Unicast**: A method of *one-to-one* communication, where one source is sending information to one specified destination (e.g. a device with IP address 151.101.17.1 is announcing a message specifically for IP address 101.112.19.2).
2. **Broadcast**: A method of *one-to-all* communication, where one source is sending information to all destinations associated with a broadcast endpoint. Local networks allow for Broadcast routing, but Broadcast traffic networks do not allow Broadcast traffic to the full Internet.

[1] From Cloudflare. (n.d.). *What is an autonomous system? | What are ASNs?* Cloudflare. Retrieved November 8, 2022, from https://www.cloudflare.com/learning/network-layer/what-is-an-autonomous-system/

[2] From Wikimedia. (n.d.). *Routing*. Wikipedia. Retrieved November 8, 2022, from https://en.wikipedia.org/wiki/Routing

3. **Multicast**: A method of *one to many* communication, where one or more sources are sending information to a subset of destinations associated with a multicast endpoint.
4. **Anycast**: A method of *one to one of many communication*, where there is one source and multiple potential receivers all identified by the same IP range. The network forwards traffic to only one of the receivers.

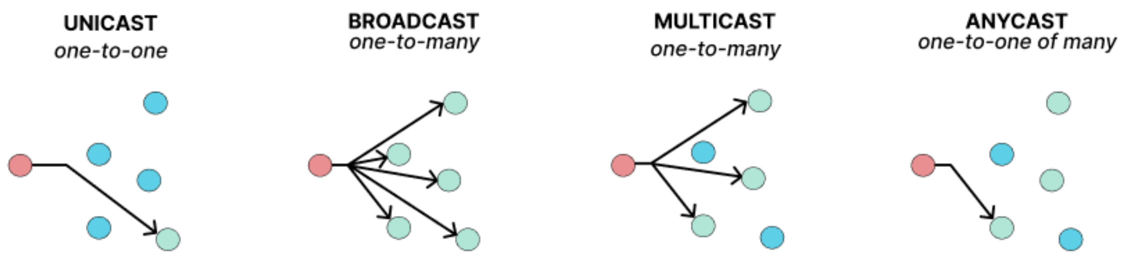


Figure 1: Packet Routing Schemes.

These four delivery schemes have their own uses and importance within today's network topology. Our project focused on traffic utilizing Unicast and Anycast. In order to direct information through any delivery scheme, routers utilize routing tables, which store routes to get to specific destinations. In the past, routers only required one routing table for IPv4 addresses. Now, modern routers need the ability to have two routing tables stored, for IPv4 and IPv6 respectively.

IP Addresses: IPv4 and IPv6

IP (Internet Protocol) addresses standardize the way computers communicate with one another. Computers identify and differentiate themselves through IP addresses. Under IPv4, there are 4.3 billion unique addresses because of its 32-bit address limit. With the rise in the number of network-capable devices, IPv6 was created in order to combat the declining amount of unassigned IPv4 addresses. Under IPv6, addresses have a 128-bit size limit, allowing for 3.4 x 10^{38} unique addresses. For now, this allows every device in the world to have its own unique IPv6 address.

With the introduction of IPv6 came a need for new routing technologies. One example of how developers are adjusting to IPv6 is by having separate IPv6 routing tables. In addition, because IPv4 addresses are limited, most modern devices that have an IPV6 address also have an IPv4 address.

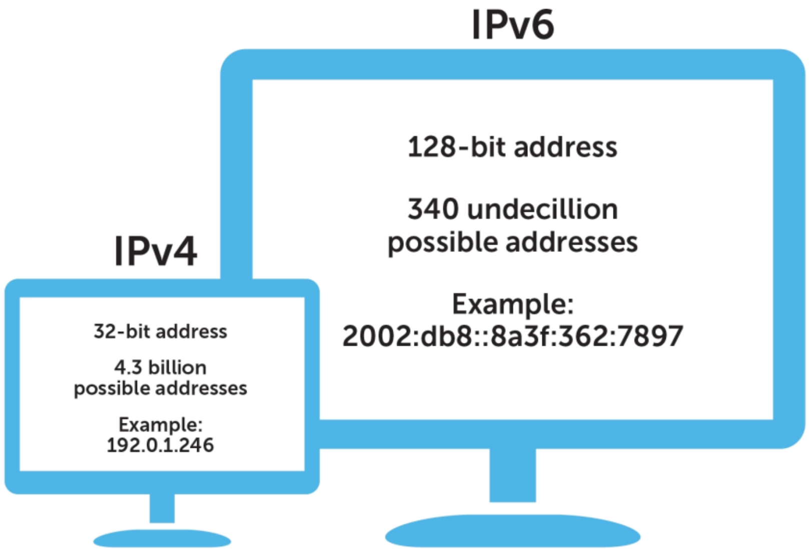

IPv6
128-bit address
340 undecillion possible addresses
Example: 2002:db8::8a3f:362:7897
IPv4
32-bit address
4.3 billion possible addresses
Example: 192.0.1.246

Figure 2: IPv4 v. IPv6.[3]

Looking back at the four routing delivery schemes, IPv4 only natively supports Unicast and Broadcast. However, Anycast being a special kind of Unicast meaning it supports that as well. Likewise, IPv6 does not support Broadcast natively but does support Multicast.[4] This, however, can be alleviated through workarounds. For example, IPv4 can utilize the *Border Gateway Protocol* (BGP) to assign multiple hosts to the same unicast address and announce routes through BGP.[5]

Border Gateway Protocol

Border Gateway Protocol (BGP) is an internet routing protocol that aims to determine the best route a packet can take between autonomous systems. *Autonomous Systems (AS)* are big networks that make up the internet, or in other words, large groups of networks with unified routing policies.[6] BGP can connect a path between a source and destination AS using a graph constructed by information exchanged between BGP routers.[7] A visual demonstration of BGP can be seen below in Figure 3.

[3] From Taylor, R. (2021, July 22). *IPv4 vs IPv6: What's the difference? – BlueCat Networks*. BlueCat Networks. Retrieved November 8, 2022, from https://bluecatnetworks.com/blog/ipv4-vs-ipv6-whats-the-difference/

[4] From *Differences between IPv4 and IPv6*. (2022, June 28). GeeksforGeeks. Retrieved November 8, 2022, from https://www.geeksforgeeks.org/differences-between-ipv4-and-ipv6/

[5] From *How Anycast Works - An Introduction to Networking*. (2018, October 4). KeyCDN. Retrieved November 8, 2022, from https://www.keycdn.com/support/anycast

[6] From Cloudflare. (n.d.). *What is the network layer? | Network vs. Internet layer*. Cloudflare. Retrieved November 8, 2022, from https://www.cloudflare.com/learning/network-layer/what-is-the-network-layer/

[7] From GeeksforGeeks. (2022, July 1). *Border Gateway Protocol (BGP)*. GeeksforGeeks. Retrieved November 8, 2022, from https://www.geeksforgeeks.org/border-gateway-protocol-bgp/

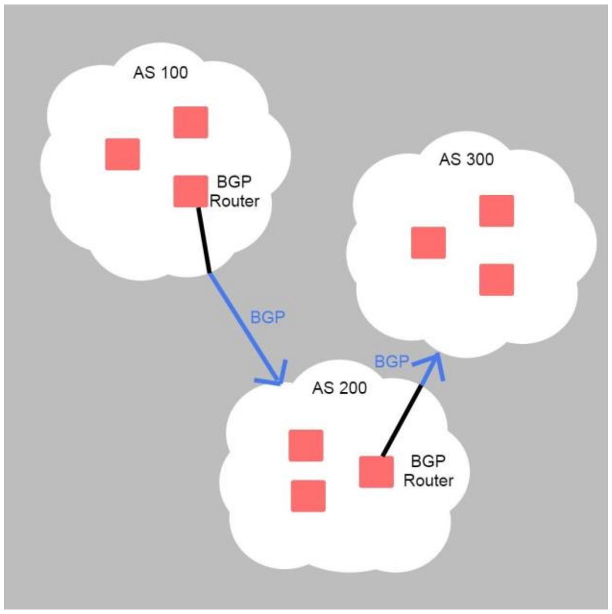


Figure 3: BGP Visualization.

BGP peers, routers designated for exchanging BGP information, initialize connection through a Transmission Control Protocol (TCP) handshake, verifying they both seek to be in communication with each other. They can then regularly announce how to reach certain networks.[7] BGP's ability to connect larger networks together makes it the base routing protocol of the Internet.

Confirming Network Connectivity through Ping

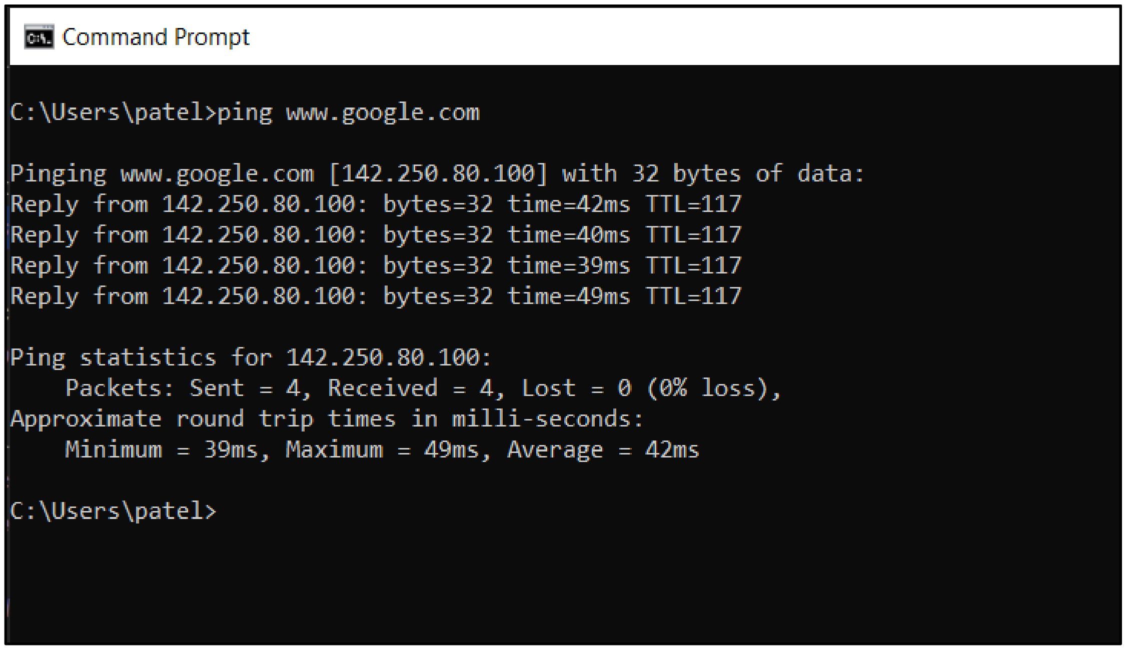
Ping is a simple command that is used for troubleshooting, information gathering, and preventing cyberattacks. Ping confirms if there is network connectivity between a source and destination node.[8] The command relies on *Internet Control Message Protocol (ICMP)* at the internet layer TCP/IP.[9] Ping sends an ICMP echo request and it expects an ICMP echo response back.

Here is an example of a ping command:

```
ping 10.1.1.42
```

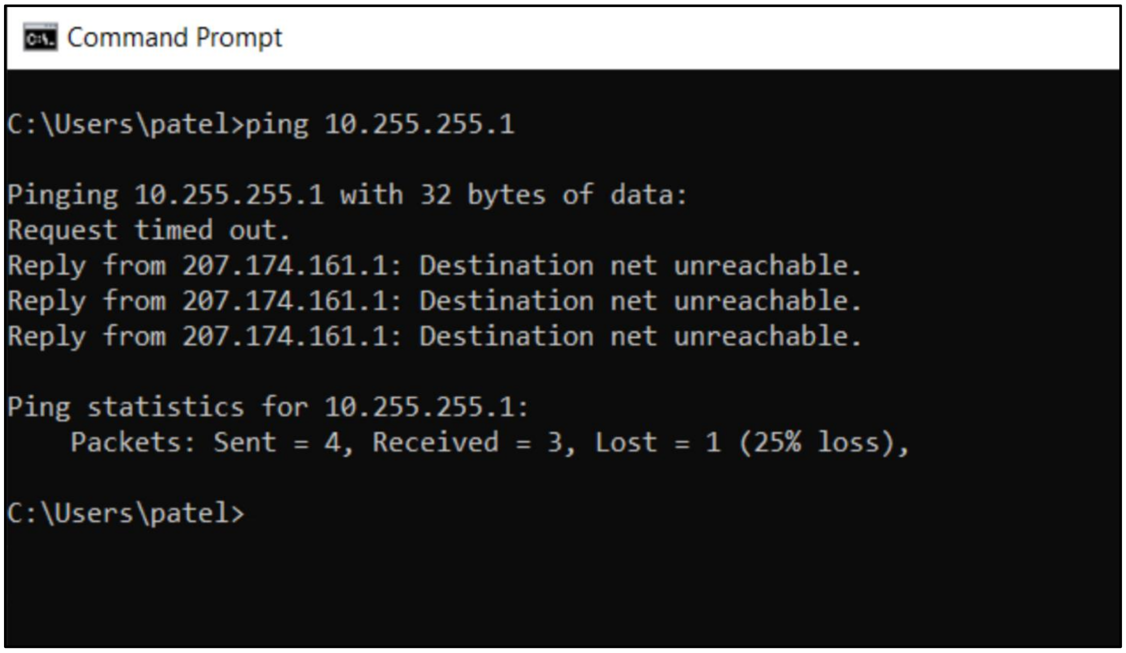
[8] From Garn, D. M. (2022, April 4). *The Ping Command | Computer Networking*. CompTIA. Retrieved November 8, 2022, from https://www.comptia.org/blog/understanding-ping-command-results

If the command returns an ICMP echo message, then there is network connectivity between the devices. If anything else is returned, the command has failed.



```
Command Prompt

C:\Users\patel>ping www.google.com

Pinging www.google.com [142.250.80.100] with 32 bytes of data:
Reply from 142.250.80.100: bytes=32 time=42ms TTL=117
Reply from 142.250.80.100: bytes=32 time=40ms TTL=117
Reply from 142.250.80.100: bytes=32 time=39ms TTL=117
Reply from 142.250.80.100: bytes=32 time=49ms TTL=117

Ping statistics for 142.250.80.100:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 39ms, Maximum = 49ms, Average = 42ms

C:\Users\patel>
```

Figure 4: Example of Successful Ping.

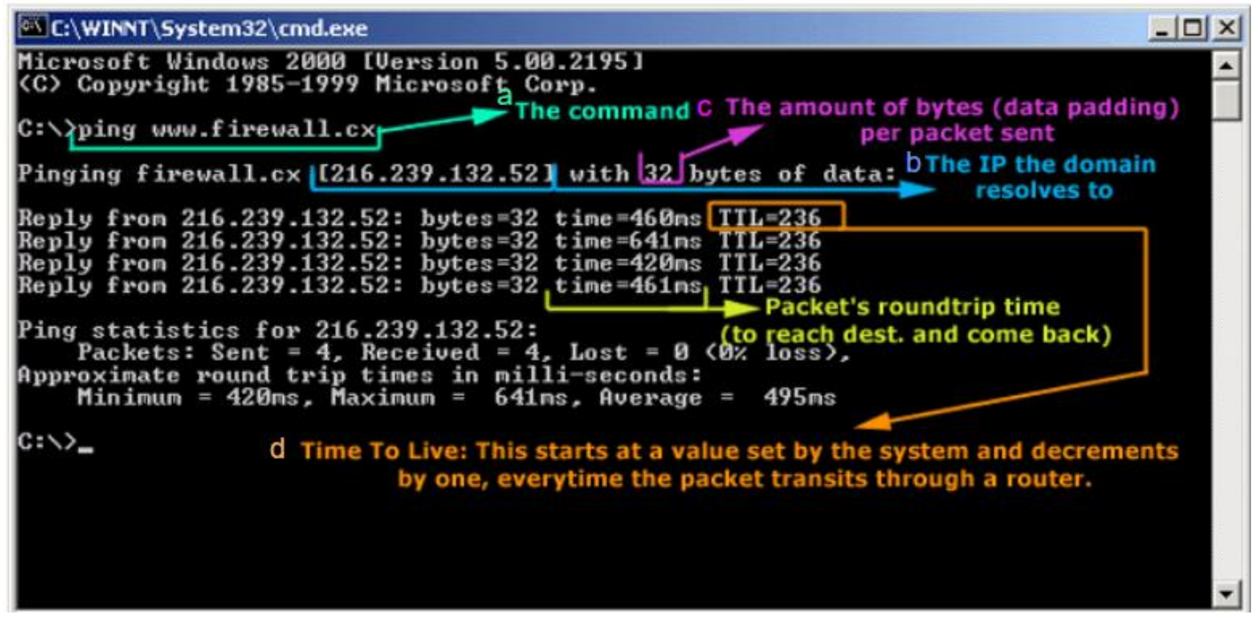


```
Command Prompt

C:\Users\patel>ping 10.255.255.1

Pinging 10.255.255.1 with 32 bytes of data:
Request timed out.
Reply from 207.174.161.1: Destination net unreachable.
Reply from 207.174.161.1: Destination net unreachable.
Reply from 207.174.161.1: Destination net unreachable.

Ping statistics for 10.255.255.1:
    Packets: Sent = 4, Received = 3, Lost = 1 (25% loss),

C:\Users\patel>
```

Figure 5: Example of Failed Ping Request.

A ping program can be called using either a hostname or IP address. The hostname is a label assigned to a device on a network.[9] On DNS servers, these are simpler, easy-to-remember names compared to the entire IP address. If you send a ping to a hostname, it will resolve the hostname to get an IP address that it can use to run the ping. A name resolution issue is where the DNS server cannot correlate the hostname to a corresponding IP address. If both IP address and hostname fail, then there is a network connectivity problem.[10]

An ICMP echo message can either be a ping request or ping reply. A ping request is the message that gets sent to the destination. The ping reply is the message that the destination sends back to the source. Figure 6 gives an example of a successful ping reply when sending a ping to www.firewall.cx.



```
C:\WINNT\System32\cmd.exe
Microsoft Windows 2000 [Version 5.00.2195]
(C) Copyright 1985-1999 Microsoft Corp.

C:\>ping www.firewall.cx

Pinging firewall.cx [216.239.132.52] with 32 bytes of data:

Reply from 216.239.132.52: bytes=32 time=460ms TTL=236
Reply from 216.239.132.52: bytes=32 time=641ms TTL=236
Reply from 216.239.132.52: bytes=32 time=420ms TTL=236
Reply from 216.239.132.52: bytes=32 time=461ms TTL=236

Ping statistics for 216.239.132.52:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 420ms, Maximum =  641ms, Average =  495ms

C:\>_
```

a The command
c The amount of bytes (data padding) per packet sent
b The IP the domain resolves to
Packet's roundtrip time (to reach dest. and come back)
d Time To Live: This starts at a value set by the system and decrements by one, everytime the packet transits through a router.

Figure 6: Example Ping Reply.[11]

A ping on Windows sends four ping requests, shown in Figure 6 with four replies and four packets sent. On Linux and macOS, the ping requests are continuous and only stop when the user asks it to.[11] Referencing Figure 6 part (a), is the ping command with the hostname. Part (b) is the IP address that the hostname resolves to. Part (c) is the number of bytes in each packet.

[9] From Fisher, T. (2022, March 8). *What Is a Hostname? (Host Name Definition)*. Lifewire. Retrieved December 6, 2022, from https://www.lifewire.com/what-is-a-hostname-2625906

[10] From Garn, D. M. (2022, April 4). *The Ping Command | Computer Networking*. CompTIA. Retrieved November 8, 2022, from https://www.comptia.org/blog/understanding-ping-command-results

[11] From *ICMP - Echo / Echo Reply (Ping) Message*. (n.d.). Firewall.cx. Retrieved December 6, 2022, from https://www.firewall.cx/networking-topics/protocols/icmp-protocol/152-icmp-echo-ping.html

This is the message that is sent and contains information on whether it is a request or reply, code, header checksum, identifier, sequence numbers, and variable data. Each reply has a *Round Trip Time (RTT)*, or the total amount of time in milliseconds it took for the message to reach the destination and come back to the source. Part (d) is the *Time to Live (TTL)***,** which is the limit on how long it should take the message to reach its destination. The value decrements by one for each time a packet goes through a router.[12]

A ping can fail for multiple reasons. For example, the local machine could not determine the destination to send the ping to, resulting in a "destination host unreachable" error.[13] The "request timed out," which happens when a network drops ICMP messages, so no response is received and the ping client gives up or times out. Although Ping is one of the simplest commands used in network connectivity, it can be expanded and used with other commands such as Traceroute.

Path Tracing with Traceroute

Traceroute is a command-line tool for tracing the path an IP packet takes across one or more networks.[14] System administrators and network engineers use traceroute to view traffic flows within their organization and identify irregular or suboptimal paths. From there, corporate networks can use this insight to block or filter traceroute packets to mitigate attackers. While Ping is used to test connectivity between the source and destination, traceroute can trace an IPv6 network and check the paths between both addresses.[14]

Traceroute utilizes the TTL of an IP packet to keep track of *hops*, the different nodes or servers that are in the path to the destination. The TTL (IPv4) / hop limit (IPv6) is a mandatory field in the packet header and is always set with regular traffic. The routing decisions can be transient and the TTL field is intended to flush the networks of packets where a forwarding loop is formed. Traceroute starts when the user invokes the traceroute command with a specific target hostname or IP address.

Traceroute then sends a data packet with a TTL of 1. The first router will decrement the TTL by 1, which will trigger a TTL exceeded limit message that gets sent back to the source. Once the source gets the error message, it sends a new data packet with a TTL of 2. This

[12] From *ICMP - Echo / Echo Reply (Ping) Message*. (n.d.). Firewall.cx. Retrieved December 6, 2022, from https://www.firewall.cx/networking-topics/protocols/icmp-protocol/152-icmp-echo-ping.html

[13] From Garn, D. M. (2022, April 4). *The Ping Command | Computer Networking*. CompTIA. Retrieved November 8, 2022, from https://www.comptia.org/blog/understanding-ping-command-results

[14] From Grimmick, R. (2022, June 25). *What is Traceroute? How It Works and How to Read Results*. Varonis. Retrieved December 6, 2022, from https://www.varonis.com/blog/what-is-traceroute

means that the second router will set the TTL to 0 and send back a TTL exceeded message. This will repeat itself until the destination is reached or an upper limit of hops is reached. Traceroute will then print all the hops on the path with the Round Trip Time or RTT.[15]

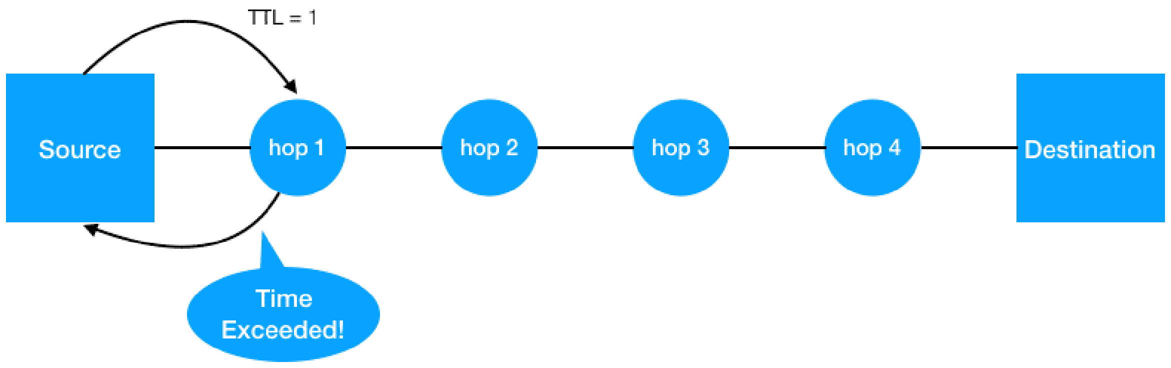
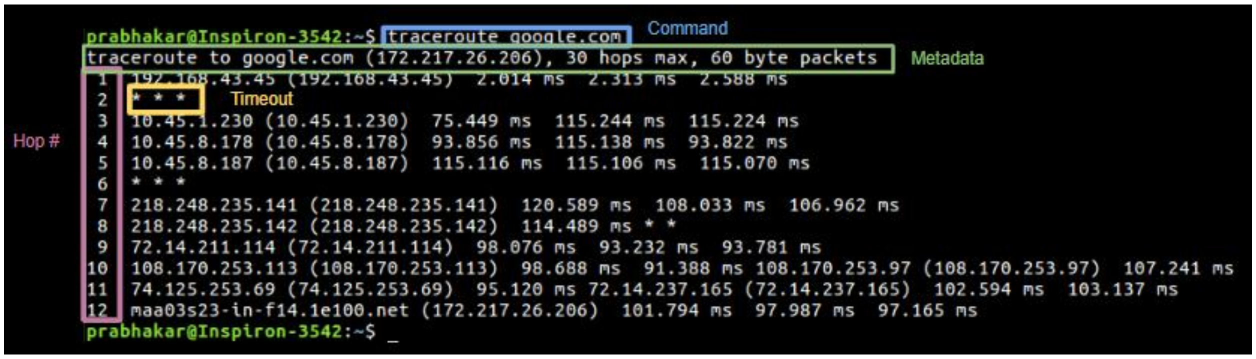


Figure 7: Visualization of Traceroute Hops to the Destination.[16]

Traceroute output, as shown in Figure 8, gives the path from the source to the destination. Immediately after the command, the metadata lists the destination IP address, upper limit of hops, and size of the data packets. Each hop is listed and numbered, starting at 1, and contains the IP address of the router that responded and the RTT. By default, traceroute sends three data packets, hence the three RTT samples per hop.[17] As mentioned before, some corporations block or filter traceroute so their routers do not send a response. When there is no response from a router, traceroute uses an asterisk (*) as an output. Since there are three data packets being sent, there is no guarantee that all three packets will take the same paths. This would be why some hops list two different IP addresses, such as hop 11 in Figure 8.[18]

[15] From Grimmick, R. (2022, June 25). *What is Traceroute? How It Works and How to Read Results*. Varonis. Retrieved December 6, 2022, from https://www.varonis.com/blog/what-is-traceroute

[16] From Gridelli, S. (2019, November 27). *Traceroute Benefits and Limits*. NetBeez. Retrieved December 3, 2022, from https://netbeez.net/blog/traceroute/



```
prabhakar@Inspiron-3542:~$ traceroute google.com
traceroute to google.com (172.217.26.206), 30 hops max, 60 byte packets
 1  192.168.43.45 (192.168.43.45)  2.014 ms  2.313 ms  2.588 ms
 2  * * *
 3  10.45.1.230 (10.45.1.230)  75.449 ms  115.244 ms  115.224 ms
 4  10.45.8.178 (10.45.8.178)  93.856 ms  115.138 ms  93.822 ms
 5  10.45.8.187 (10.45.8.187)  115.116 ms  115.106 ms  115.070 ms
 6  * * *
 7  218.248.235.141 (218.248.235.141)  120.589 ms  108.033 ms  106.962 ms
 8  218.248.235.142 (218.248.235.142)  114.489 ms * *
 9  72.14.211.114 (72.14.211.114)  98.076 ms  93.232 ms  93.781 ms
10  108.170.253.113 (108.170.253.113)  98.688 ms  91.388 ms 108.170.253.97 (108.170.253.97)  107.241 ms
11  74.125.253.69 (74.125.253.69)  95.120 ms 72.14.237.165 (72.14.237.165)  102.594 ms  103.137 ms
12  maa03s23-in-f14.1e100.net (172.217.26.206)  101.794 ms  97.987 ms  97.165 ms
prabhakar@Inspiron-3542:~$ _
```

Figure 8: Traceroute Output to google.com.[17]

We can view the geolocation of each IP address by referencing its ASN. Using geolocation, we can map out on a world map how far our data is traveling. Traceroute is an important tool to be able to view how and where we are connecting to our destination.

Anycast and Its Use Cases for CDNs

Anycast is a routing scheme designed to make a particular IP Address available from multiple locations. Packets then can be routed to one available device within the specified location. This means that content is able to be distributed to many locations while directing traffic to the closest data center for low latency and better performance. It can be considered a one-to-nearest mapping.[18] Anycast allows *Content Delivery Networks (CDN)* like Fastly to process network requests more efficiently by routing incoming network traffic to the closest data center. Anycast allows CDNs to have better system performance, security, and load balancing.

System Performance

Since multiple points of presence (POP) can have the same IP range, a POP can be withdrawn from the Anycast group and the network will still have destinations to send traffic. This is ideal for CDNs, which aim to give end users the most direct route to their destination. Latency is reduced because of the greater availability and variety of possible connection points. More widespread points, both in terms of location and number, allow the user to connect to their destination faster.

[17] From codstar. (2019, May 27). *traceroute command in Linux with Examples*. GeeksforGeeks. Retrieved December 6, 2022, from https://www.geeksforgeeks.org/traceroute-command-in-linux-with-examples/

[18] From Kesavan, A. (2016, May 19). *Using Anycast for Internet Services*. ThousandEyes. Retrieved December 5, 2022, from https://www.thousandeyes.com/blog/using-anycast-for-internet-services

System Security

The backup, failsafe system when connection points go down also affords Anycast systems more protection when it comes to distributed denial-of-service (DDoS) attack mitigation. In the event that a server is targeted or compromised, there are measures in place so that an adversary cannot maliciously flood servers with excessive traffic. Anycast as a load balancing method can be used as a tool to not only combat DDoS attacks but also to control nonmalicious traffic spikes (e.g. a sports website during the Super Bowl or news sites during a Presidential Election).

System Load Balancing

For a CDN of Fastly's scale, it is important to be able to handle many incoming requests. Especially when dealing with high-volume traffic customers like Reddit or The New York Times, it is vital for Fastly to be able to spread incoming traffic amongst many servers. If there is a traffic spike, load balancing allows the incoming requests to be divided up among different servers. This prevents one server from being overloaded.

3. METHODOLOGY

This methodology chapter discusses the team's approach to tackling different parts of the project. For the technical portion, we discuss the tools and data sources needed to effectively evaluate Anycast traffic. We outline the team's approach to project planning and management. Finally, we address potential security concerns as a result of the project.

3.1 EVALUATION OF ANYCAST

The evaluation of Anycast traffic involves many different factors. While a CDN, such as Fastly, can know its own specific network policies and configurations, it has no jurisdiction over that of other providers. In order to successfully and efficiently deliver its traffic, Fastly relies on thousands of other players. To begin to understand the paths that inbound connections to Fastly take, we can use external resources. Measuring external data sources can provide Fastly with more information on how the network is actually behaving rather than just what Fastly is dictating. This can allow Fastly to better optimize how the network is flowing into its Anycast servers. In order to measure from external sources, the team looked into publicly available data sources.

Public Data Sources

RIPE Atlas is an open, global network of Internet connectivity measurement probes, dedicated to providing a better understanding of the Internet as a whole.[19] RIPE Atlas users can utilize active probes to monitor specific measurements like ping or traceroute, test IPv6 connectivity, and investigate and troubleshoot network issues.[20] To create a measurement, a user must have credits, which they can earn by hosting or sponsoring one or more probes.[20] The team ultimately chose RIPE Atlas as PATHly's main public data source because of sponsor recommendations and credit allowance. This decision afforded the team more help avenues when it came to troubleshooting and also a no-cost option for data.

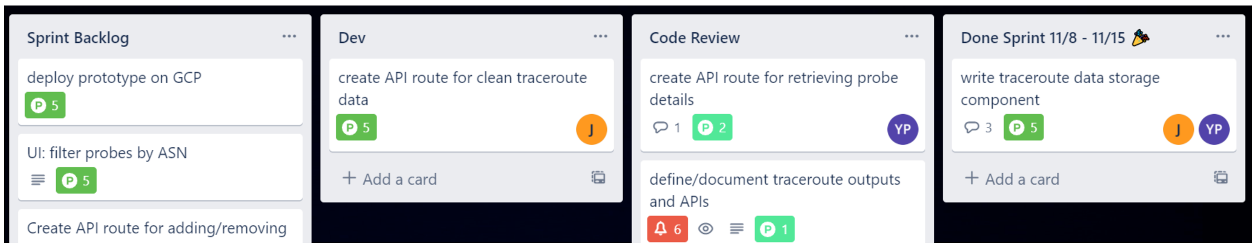
[19] From *What is RIPE Atlas?* (n.d.). RIPE Atlas. Retrieved December 8, 2022, from https://atlas.ripe.net/about/

3.2 PROJECT MANAGEMENT

The team utilized an *agile methodology framework* for project management. An agile approach enabled the team to break up the project into several one-week sprints. Through this iterative process, the team focused on key tasks each week and reflected on our progress. The following agile methodology components allowed us to better manage both the project and the team's resources:

Sprints and Sprint Planning

The team broke down the overall project by sprints. We planned and organized our sprints on *Trello*, a visual tool for task organization. Trello allowed us to have a centralized task management system. At the beginning of the project, we brainstormed all deliverables and the steps to them. This became our *Backlog*, which we referenced each sprint. Before every sprint, our team met with our product owner to choose tasks from the Backlog. Our *Sprint Backlog* consisted of the tasks we would focus on for the week. Each task would be assigned a Fibonacci number, a *Story Point* amount, to indicate the amount of time and effort we thought would go into that task. During the sprint, team members would assign themselves to *Story Cards* and move them through the board as they progressed through the task.


Sprint Backlog
deploy prototype on GCP
5
UI: filter probes by ASN
5
Create API route for adding/removing

Dev
create API route for clean traceroute data
5
J
+ Add a card

Code Review
create API route for retrieving probe details
1 2
YP
define/document traceroute outputs and APIs
6 1

Done Sprint 11/8 - 11/15
write traceroute data storage component
3 5
J YP
+ Add a card

Figure 9: Screenshot of Trello Board.

Stand-Up

Due to the many moving parts of the project, communication was key to the team's success. This took the form of daily check-in meetings, scheduled for 20 minutes. Each team member, along with the product owner and sponsors, would state what work they did, what work they're doing, and what they will be doing. Team members would also discuss any issues that may be blocking progress so that the team would be able to help if applicable. These quick Stand-Up meetings allowed the team to have a better idea of each member’s progress and the overall project.

Retrospective

At the conclusion of each sprint, the team reflected on what went well and not as well in the *Retrospective*. The team would take 5 minutes to categorize their thoughts and feedback on the week into three categories: Good, Okay, and Bad. Afterward, the team would *dot vote* by upvoting five retro cards that they agreed with. This allowed the team to cover top discussion items. The Retrospective was a chance for all team members to let their voices be heard. Retrospectives provided an avenue for the team to reflect and change course as necessary. It opened us up to constant feedback in order to avoid being stuck in unproductive habits for too long.

3.3 SECURITY CONCERNS

Our project presented little to no security concerns. The application is hosted on a virtual machine managed by fastly, running inside the *Google Cloud Platform (GCP)*. This application is openly available on the internet, but because it’s being hosted on GCP, access to any internal services through GCP are restricted to only those with public key access, being the team and Fastly Employees. In addition, all data sources utilized in our project are open-source, meaning all data used in PATHly is accessible to the general public.

4. IMPLEMENTATION

In this section, we break down the implementation of PATHly Visualizer. For both the front-end and back-end components of PATHly, we describe the technologies used and their applications.

4.1 BACK-END

For the backend of the project, we used the Golang (Go) programming language and implemented a service model.

Service Model

We made use of a service model for our project when communicating between different systems, based on how we would need the ability to run many threads (goroutines in this case) and a uniform way of communicating between them. The design of the model is shown in Figure 10. We created a server that will store a list of all the services of the project and a pointer to an application state which contains the shared application state structure. While we considered making more use of channels, we found it did not fit our use case, as our REST API service would need to have access to nearly all data on the system to fulfill requests. So for important shared information, we used our shared thread-safe application state.

Each of our services implement the Service interface which contains the name of the service, initialization function, and run function. This allows services to store their own data and allows for direct service to service communication by sharing channels during initialization. It also helped cut down on time spent debugging service-to-service interactions since services were only free to share information between themselves in two locations. By enforcing this service interface, we also benefited from standardized logging through a service's creation and initialization.

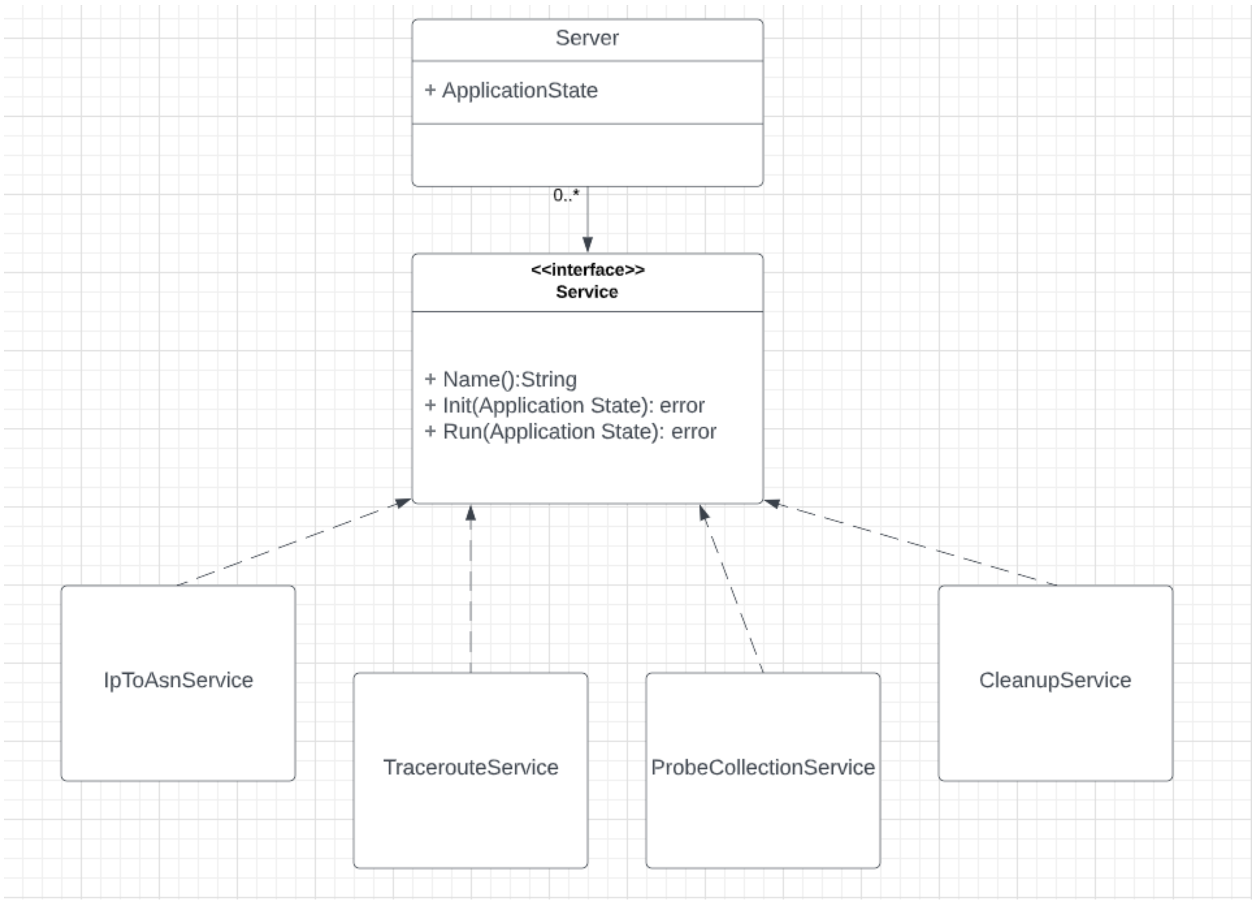
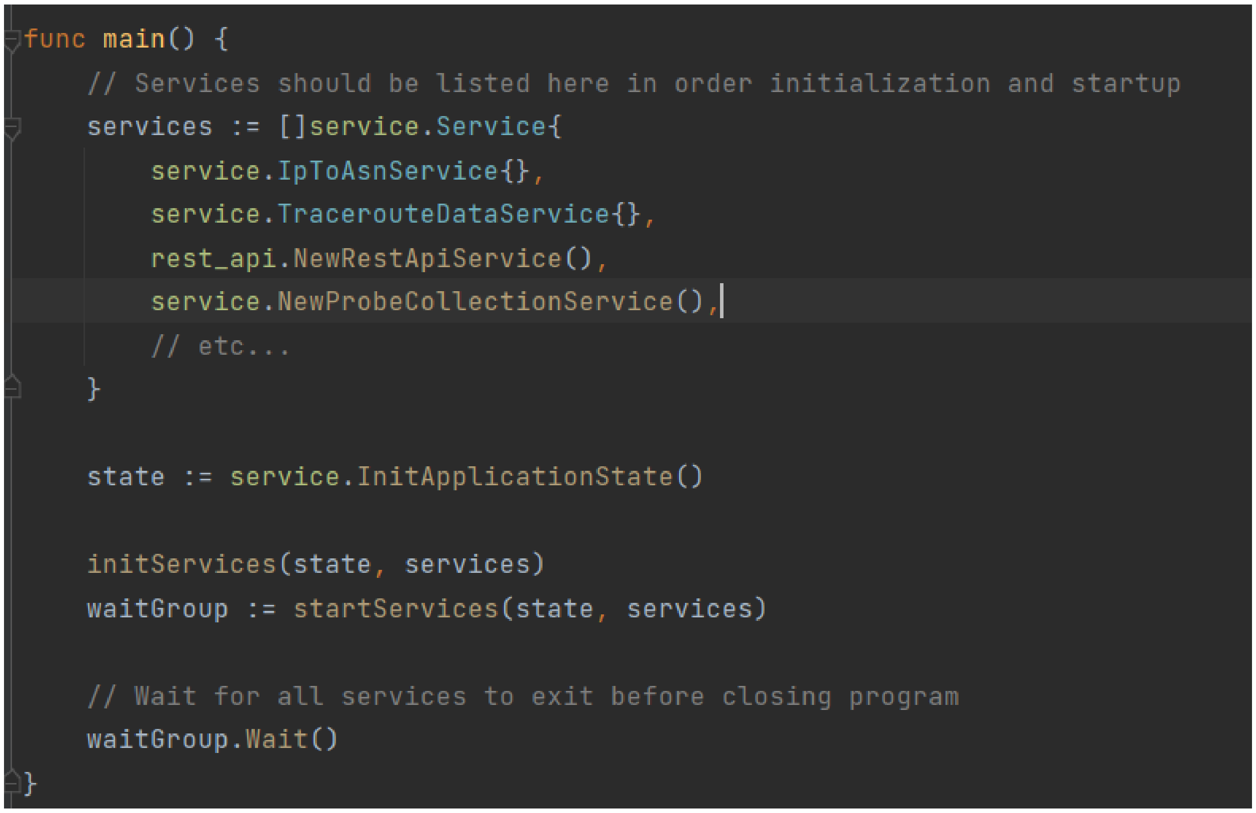


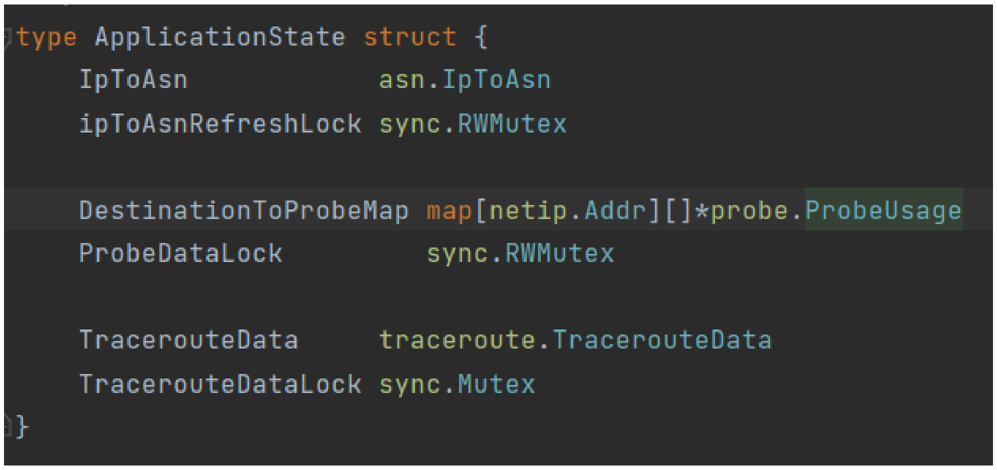
Figure 10: Simplified UML Diagram of the Service Model Used.

The server starts the project by calling the Init() function in a thread on each of the services as shown in Figure 11. The purpose of the Init() functions is for each service to initialize their own data structures and load in any information prior to what the service is intended to do. Once all the services are initialized, the server starts each of the services with their Run() function in their own thread. This gets handled in the startServices() function in Figure 11. The purpose of the thread, or goroutine, is to ensure each of the services work concurrently. Services utilize the shared data in the application state pointer they are given in the Init() and Run() functions. The application state is thread safe by using mutexes for each of the data structures within it as shown in Figure 12.



```go
func main() {
    // Services should be listed here in order initialization and startup
    services := []service.Service{
        service.IpToAsnService{},
        service.TracerouteDataService{},
        rest_api.NewRestApiService(),
        service.NewProbeCollectionService(),
        // etc...
    }

    state := service.InitApplicationState()

    initServices(state, services)
    waitGroup := startServices(state, services)

    // Wait for all services to exit before closing program
    waitGroup.Wait()
}
```

Figure 11: Server Initializing and Starting Each of the Services.



```go
type ApplicationState struct {
    IpToAsn            asn.IpToAsn
    ipToAsnRefreshLock sync.RWMutex

    DestinationToProbeMap map[netip.Addr][]*probe.ProbeUsage
    ProbeDataLock         sync.RWMutex

    TracerouteData     traceroute.TracerouteData
    TracerouteDataLock sync.Mutex
}
```

Figure 12: Application State with Each Data Structure Having a Mutex.

With the service model outlined, we can now explain how each of the services work.

REST API Routes

After finishing our MQP, we are unable to continue providing updates to the project or maintain the codebase. This could become problematic for our sponsor if they need to extend the functionality or do some additional analysis on the data PATHly provides. Thus we designed our server around a few REST API that can be interfaced with directly to pull data or modify the existing server configuration. We used the Gin web framework[20] in Go for our API. From there, we split our REST API functionality into sections.The core routes we needed to focus on were configuring which measurements to collect, retrieving traceroute route graphs, and gathering information on which probes were collecting data.

For traceroute data, we needed to have 3 different routes. These routes correspond to retrieving the full, cleaned, and raw data. As the names imply, the full API route retrieves all of the data the server stores on a given route. The raw API route retrieves the raw data from RIPE Atlas that was used to build that route. As for the clean API route, it works very similarly to the full API route with some small tweaks to improve the readability of the output graph. The main change made to this route is the removal of timeout nodes. Depending on the stability, responsiveness of the nodes along a route, and the statistics period the data is stored for, timeout nodes can quickly add up. These nodes can begin to clutter the view and complicate the graph layout process when drawing boxes around autonomous systems. To remove them, we assume that the timeout nodes do not exist within the graph and connect disconnected layers. However, the next main issue we encountered with the clean data was the prevalence of false edges on the graph. These edges are frequently caused by the route changing midway through a measurement or an inconsistent decementation of the packet time to live value. These edges are almost always unhelpful and can make it harder for the viewer to pick out the primary edges at a glance. To combat this issue, we pruned edges from the graph which did not meet a minimum percentage of the occurrences out of all the edge occurrences from the source node. At first we attempted to use a flat percentage across all of our graphs, but we ran into some issues with nodes that connected to many children. By splitting the occurrences among so many children it became possible for none of the children to meet the threshold leading to many disconnected nodes being left in the graph. We were able to solve this problem by scaling the minimum threshold by the number of connected nodes. This led to a satisfactory result where most of the false edges were removed without hurting any of the main edges.

Next, we added routes for retrieving probes involved in a specific measurement. The purpose of this route was to enable selection on the front-end of valid nodes for a given destination. However, we quickly found it could be used for a number of other features. This included location information on each probe to allow for the ability to select probes by region

[20] From gin-gonic. (n.d.). *gin-gonic/gin*. GitHub. Retrieved December 15, 2022, from https://github.com/gin-gonic/gin

and metadata to enable searches to be done on probes to filter by ASN, network prefix, and route stability. Unfortunately, we did not get to implement many of these features on the frontend due to limited time, but this route does have most of the functionality so it can be used if future work is done on the project.

Lastly, we needed API routes to control which measurements were being monitored by the server. This was important since up to this point, the specific RIPE Atlas measurement IDs we were collecting data on were hard coded into the program. These API routes allowed a user to list which measurements are in use, add new measurements, and remove/stop existing measurements. Additionally, this added the differentiation between live collection and loading historical data. Using these routes a user can determine which type of collection is being performed on each measurement, switch between collection modes, or drop a measurement's data from the server.

Traceroute Data Ingestion

To load traceroute data, we used three systems throughout the project. This includes caching measurements data, pulling a measurement's history, and listening to live data as it is collected.

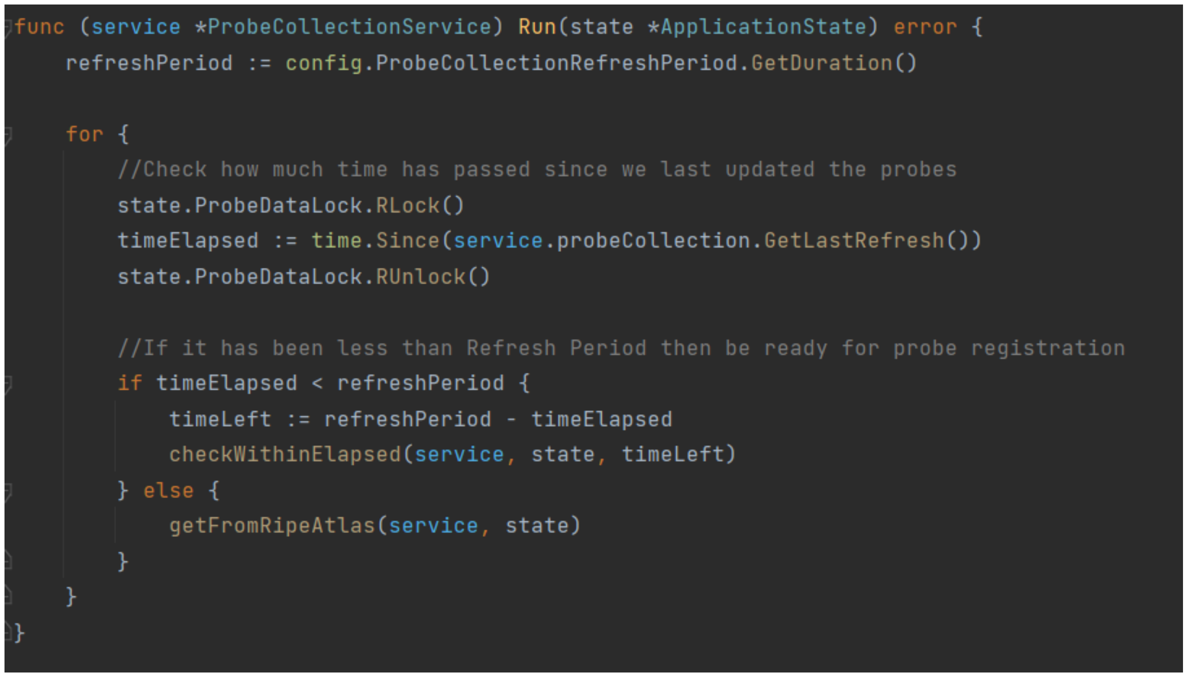
Through the project, our primary method for debugging and testing made use of our cached measurement API. The primary goal of this approach was to reduce the turnaround time of running tests and starting up the server. It was able to achieve this goal by storing measurement data locally as a file instead of performing the timely request to RIPE Atlas. Since the primary, and only, use case of this approach was for debugging, it was implemented with no global eviction strategy under the understanding that we only used a couple of hard-coded measurements for tests. Instead we used a time-to-live strategy for each individual cached entry based on when the entry was last refreshed. The cache period is configurable and allowed for data to be kept mostly up to date without slowing down production. As for the performance gains, we were successfully able to reduce the load time of a measurement from minutes to seconds. However, without any compression or cleanup of unused entries within the cache, one member of our team struggled with limited storage space for measurement data.

Next, we provided the option to pull a measurement's history. Unlike the previous option, this was intended for production to be used by users of our dashboard to fetch data for a quick inspection of a specific measurement. Unlike our observations when debugging, we saw little utility in attempting to cache this data since it is far less likely to see reuse and would likely eat through the available storage on our system. The storage concerns of caching could be mitigated if we were to spend the time upgrading to a least recently used cache model. But due to added complications due to concurrent requests, we did not think this would be an effective use of our time.

Lastly, the primary data collection route for our application is through live collection of measurements. Unlike the previous options, this approach simply receives new measurements instead of allowing for the retrieval of historical data. This approach closely lines up with the project’s user stories since it enables the ability to highlight interesting networking features as they appear in the output data.

Data Parsing and Storage of RIPE Atlas Probes

The probe collection service has two main functions. The first is to obtain the metadata from RIPE Atlas. The second is to get information from the Traceroute measurements and update our destination-to-probe map. As shown in Figure 13, we have a refresh period which is the time we wait before we refresh our main probe map. The default is once a day. After a day has passed we retrieve the data from RIPE Atlas. If we are within the refresh period, we wait for probes from the traceroute measurements to add to our destination to probe map.



```
func (service *ProbeCollectionService) Run(state *ApplicationState) error {
    refreshPeriod := config.ProbeCollectionRefreshPeriod.GetDuration()

    for {
        //Check how much time has passed since we last updated the probes
        state.ProbeDataLock.RLock()
        timeElapsed := time.Since(service.probeCollection.GetLastRefresh())
        state.ProbeDataLock.RUnlock()

        //If it has been less than Refresh Period then be ready for probe registration
        if timeElapsed < refreshPeriod {
            timeLeft := refreshPeriod - timeElapsed
            checkWithinElapsed(service, state, timeLeft)
        } else {
            getFromRipeAtlas(service, state)
        }
    }
}
```

Figure 13: Probe Collection Service Run Function.

One unexpected challenge of this project was keeping an up-to-date list of collected. RIPE Atlas probes are constantly updated regardless of if they are connected, disconnected, or abandoned. We need to periodically check the RIPE Atlas servers for their active probes.

To solve this, we created a *Probe Collection Service* to continuously pull probes from the RIPE Atlas servers using the RIPE Atlas golang library released by DNS OARC[21]. This library is used to pull every probe and keep the metadata on each one. We then take in the metadata and parse out information about the probe ID, IPAddress, ASN, Country Code, Longitude, and Latitude. Ripe Atlas stores all of its probes on about 400 different web pages. The issue is that the library can only read data from one page of nodes at a time leaving hundreds of pages to parse through, causing a processing time of about 4 minutes. To solve this, we implemented worker goroutines for each CPU on the server to concurrently wait on and parse many pages at a time. This significantly dropped the processing time to an average between 8-25 seconds.

While we store all of the probes we still need to map them to the traceroute measurements they are used for. To do this, each IP address has a list of corresponding probes that they are connected to. While we ingest traceroute measurements, we send the destination IP and probe to the Probe Collection Service in a channel. We then add that information to our destination-to-probe map. The destination-to-probe map is stored in the application state because this is global information that is used by other services.

Cleanup Service

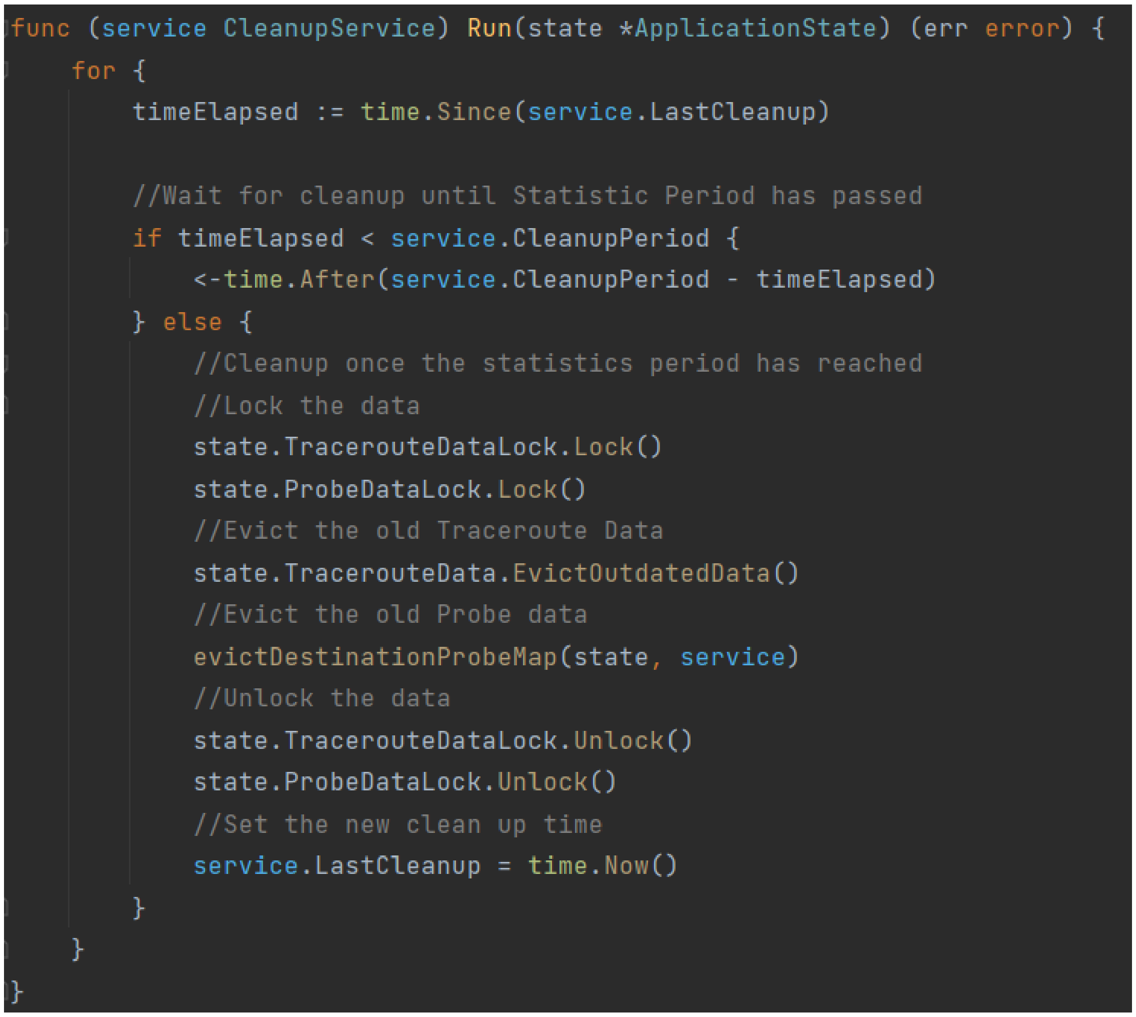
With the large ingestion of data, we need to periodically evict the old data. We use the Cleanup Service to handle this. We have configuration variables for the *Cleanup Period* and *Statistics Period*. The cleanup period is how often we need to clean up the data and by default is once a day. The statistics period is how far back we keep the data and by default is two weeks.

There are two main data structures that we need to clean. The first is the Traceroute Data. For each of the traceroute measurements, we remove the old nodes, edges, clean edges, and probe IPs. We check if they are old by checking the last used time in each of those attributes to see if they are before the statistics period. We also need to increase the upper bound of the nodes and edges for the moving statistics. The second structure we need to clean is the Destination to Probe Map. We iterate through each probe list in each destination IP and

[21] From *DNS-OARC/ripeatlas: Go bindings for RIPE Atlas API*. (n.d.). GitHub. Retrieved December 12, 2022, from https://github.com/DNS-OARC/ripeatlas

compare when the last time the probe was used. We create a new list of probes that are more recent and have that to be attached to the destination IP.

Figure 14 helps show how the Cleanup Service is implemented. We wait until the cleanup period is reached and then evict the data. Since we are modifying the shared data we have to lock each data structure with its corresponding mutex. We then update the last time the project was cleaned.



```go
func (service CleanupService) Run(state *ApplicationState) (err error) {
    for {
        timeElapsed := time.Since(service.LastCleanup)

        //Wait for cleanup until Statistic Period has passed
        if timeElapsed < service.CleanupPeriod {
            <-time.After(service.CleanupPeriod - timeElapsed)
        } else {
            //Cleanup once the statistics period has reached
            //Lock the data
            state.TracerouteDataLock.Lock()
            state.ProbeDataLock.Lock()
            //Evict the old Traceroute Data
            state.TracerouteData.EvictOutdatedData()
            //Evict the old Probe data
            evictDestinationProbeMap(state, service)
            //Unlock the data
            state.TracerouteDataLock.Unlock()
            state.ProbeDataLock.Unlock()
            //Set the new clean up time
            service.LastCleanup = time.Now()
        }
    }
}
```

Figure 14: Cleanup Service Run Function.

4.2 FRONT-END

The front-end component of our application is what the user sees and interacts with. To guide our design process, our sponsor provided us with user stories that outlined what a user should be able to do with the application (see Appendix A). From these user stories, we designed user interface (UI) mock-ups to provide a static visualization of what the front-end could look like (see Appendix B). This allowed us to get feedback from our sponsor and iterate the design process. This section covers the technologies and other considerations taken to implement these designs.

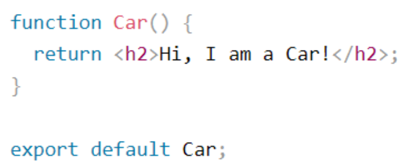
Technologies

We used *Figma* to design UI mock-ups and *React*, a Javascript framework, to develop the front-end of this application. React applications are built upon a *Node.js* web server. We utilized several open-source packages made available on *npm*. The following is a list of the most important packages we used in development:

- **ReactFlow** is an open-source React-based flowchart library. This library was used as the base of the path visualization as it had the capability to represent nodes and edges, provide easy customization, and allow for automatic path layouts utilizing external automatic layout algorithms.
- **Dagre** is an open-source JavaScript-based automatic graph layout algorithm. This library helped position all the nodes and edges of the received traceroute data, as ReactFlow does not have built-in automatic layout functionality.
- **Material UI Core (MUI Core)** is an open-source React-based library that provides several prebuilt and stylized React components. MUI Core’s main functionality in our application was to provide pop-ups with node information when clicking on any individual node in a graph.
- **React Bootstrap** is a form of the popularly used front-end framework Bootstrap, designed specifically for React. It makes use of React components to allow for easy use of Bootstrap implementations like Navigation bars and Popovers. We used this Library to create the top navigation bar in PATHly.

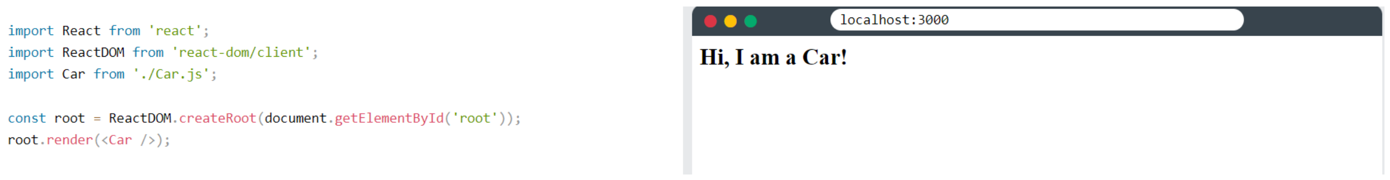
Furthermore, React, being a Javascript framework, has several built-in upgrades to vanilla Javascript that we utilized throughout development:

- **Components** are portions of code created to be reused. For example, say that we want more than one of the same form on a page. Instead of copying and pasting the form twice in the code, we can create a React component with the form's code, import it into our page, and only have to use one line of code. An example of a functional react component being displayed as code can be seen in Figure 15, and it is seen being used in a separate file in Figure 16.



```
function Car() {
  return <h2>Hi, I am a Car!</h2>;
}

export default Car;
```

Figure 15: Functional React Component.[22]



```
import React from 'react';
import ReactDOM from 'react-dom/client';
import Car from './Car.js';

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Car />);
```

Hi, I am a Car!

Figure 16: Rendering React Component on Page.[23]

- **Hooks** were created to be used with Functional React components. There are two types of components: Function and Class. Class components have access to helpful React features like accessing state. Prior to React v. 16.8, Functional components were unable to make use of these features. Version 16.8 added Hooks, which provide Functional components with these features. Our project's components are Functional. Because of this, we made use of the following hooks:

    - The **useState** hook gives Functional components the ability to track state. State is a React-specific feature, essentially being the metadata of a component. This data can be changed, which causes a re-render of the component on the client side.

[22] From *React Components*. (n.d.). W3Schools. Retrieved December 5, 2022, from https://www.w3schools.com/react/react_components.asp
[23] From *React Components*. (n.d.). W3Schools. Retrieved December 5, 2022, from https://www.w3schools.com/react/react_components.asp

- The **useCallback** hook caches specified functions in order to avoid resource intensive re-renders of specific parts of code. This hook can be provided with props, and the code within this hook will only be called when the specified props are updated.
- The **useMemo** hook is extremely similar to useCallback. The main difference is that useCallback returns functions while useMemo returns values.

Data Representation

Data is collected from the back-end utilizing custom-built *REST API Routes*. The front-end sends requests through these routes using the *Fetch API*. The front-end then parses the received response and is able to represent it in several meaningful ways:

- The **full traceroute** route collects all recorded traceroute data with no cleaning. This means that all timeouts are represented in the data. These timeouts, as shown in Figure 17, are labeled with the * character because their IP addresses are unknown.

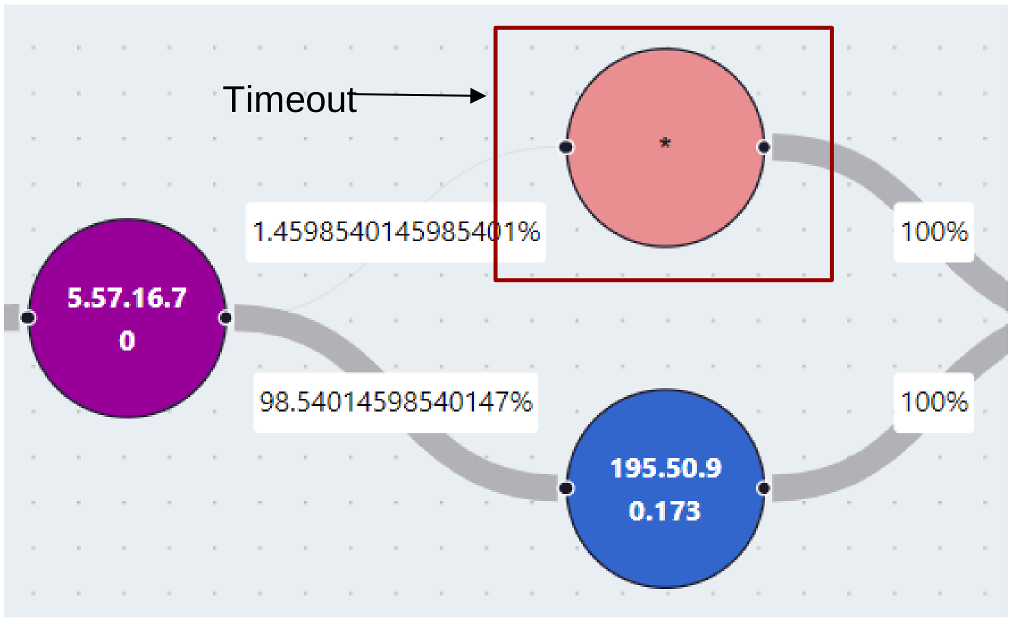
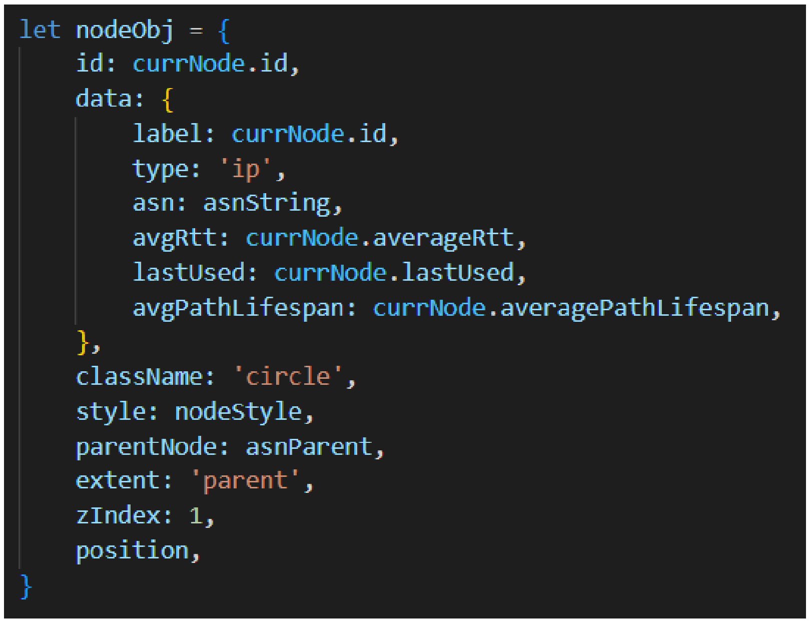


Figure 17: Full Traceroute Data Representation with Timeout.

- The **clean traceroute** route collects cleaned traceroute data. This means that timeouts and redundant paths are cleaned from the data. No timeouts are rendered when utilizing this route.

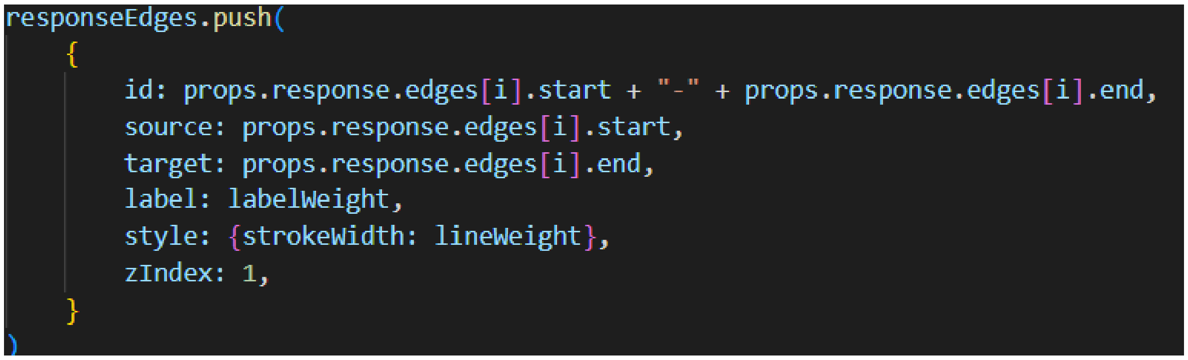
- The **raw traceroute** route collects all traceroute data from the specified input probe and destination IP in the form of an attachment. A *blob* is then created to put this data into a JSON file. If the file is unable to be created, a new window will open with the appropriate loaded data.

The traceroute path data is represented using the ReactFlow library. The retrieved data from the back-end is formatted so the nodes and edges objects include the required ReactFlow object elements. Nodes must include an ID, position, and data. The data included in a node only requires a label, any other amount of data may be included aside from that. Figure 18 shows how an average node in our data is formatted for ReactFlow. Edges must include an ID, source, and target. Figure 19 shows how an average edge in our data is formatted for ReactFlow.



```
let nodeObj = {
    id: currNode.id,
    data: {
        label: currNode.id,
        type: 'ip',
        asn: asnString,
        avgRtt: currNode.averageRtt,
        lastUsed: currNode.lastUsed,
        avgPathLifespan: currNode.averagePathLifespan,
    },
    className: 'circle',
    style: nodeStyle,
    parentNode: asnParent,
    extent: 'parent',
    zIndex: 1,
    position,
}
```

Figure 18: Node Formatting for ReactFlow Implementation.



```
responseEdges.push(
    {
        id: props.response.edges[i].start + "-" + props.response.edges[i].end,
        source: props.response.edges[i].start,
        target: props.response.edges[i].end,
        label: labelWeight,
        style: {strokeWidth: lineWeight},
        zIndex: 1,
    }
)
```

Figure 19: Edge Formatting for ReactFlow Implementation.

Several of these nodes collected by the back-end belong to an AS and will be passed to the front-end with a specified ASN. This data is important in understanding BGP and Anycast forwarding, so we needed to be able to represent which ASNs the nodes belong to. To do this, we began by looking at ReactFlow's sub flow functionality. ReactFlow sub flows are defined as a flow inside of a node. This logic can be seen in Figure 20 below.

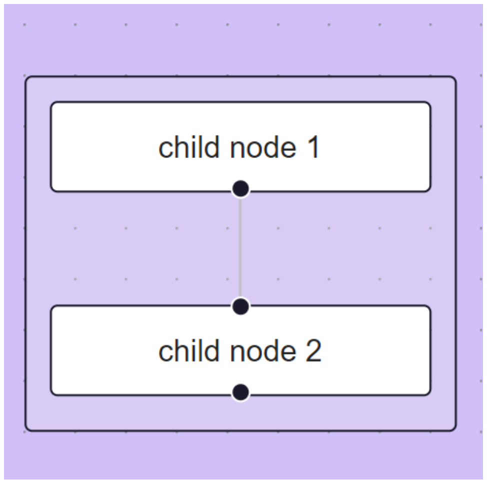


Figure 20: ReactFlow Sub Flow Example.[24]

However, difficulty sprang up when we were using sub flows as the size of the sub flow must be specified beforehand. This meant we needed to devise a way to calculate how large a sub flow must be to cover each node that belonged to it. Furthermore, sub flow size caused issues with the automatic layout algorithm, Dagre. Sizing needed to occur after every node was placed, meaning the sub flows would often cover nodes that did not belong to it as they were in the way of another node that did. While this did not always present issues, it would in larger datasets. To combat this, we decided to implement an option to represent node ASNs through their colors.

The colors decided on needed to be accessible to those who are color blind. To verify their safety, we used the Adobe Color Contrast Analyzer.[25] This tool has built in Color Blind Safety functionality which we were able to use to verify the colors we were using. Figure 21 shows the color palette we ended up using. If there are ever more ASNs than colors, the color coordination restarts from the beginning. An example of ASNs being represented by boxes can

[24] From *Sub Flows Guide*. (n.d.). React Flow. Retrieved December 7, 2022, from https://reactflow.dev/docs/guides/sub-flows/
[25] From *Color contrast checker analyzer tool*. (n.d.). Adobe Color. Retrieved December 7, 2022, from https://color.adobe.com/create/color-contrast-analyzer

be seen in Figure 22. The same path with ASNs being represented by colors can be seen in Figure 23.

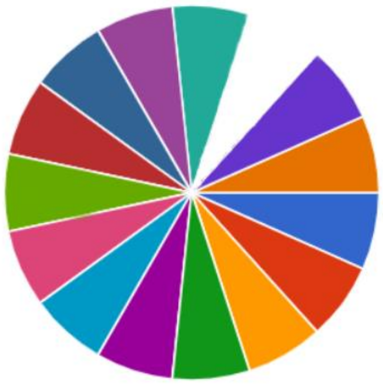


Figure 21: ASN Color Palette.

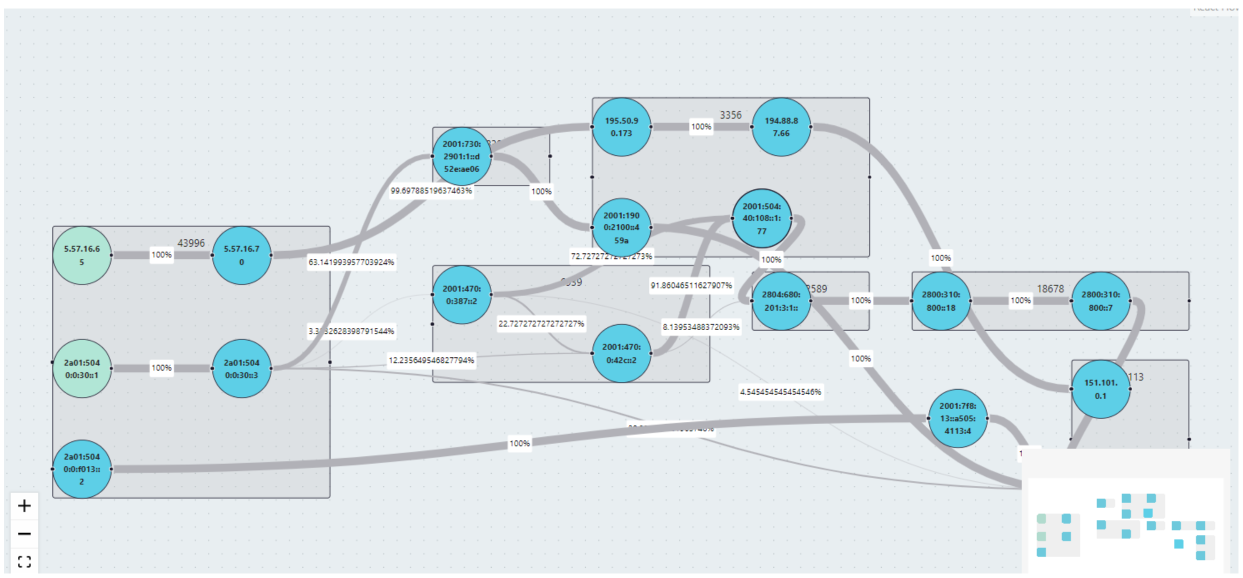


Figure 22: Clean Traceroute Data with ASN Boxes.

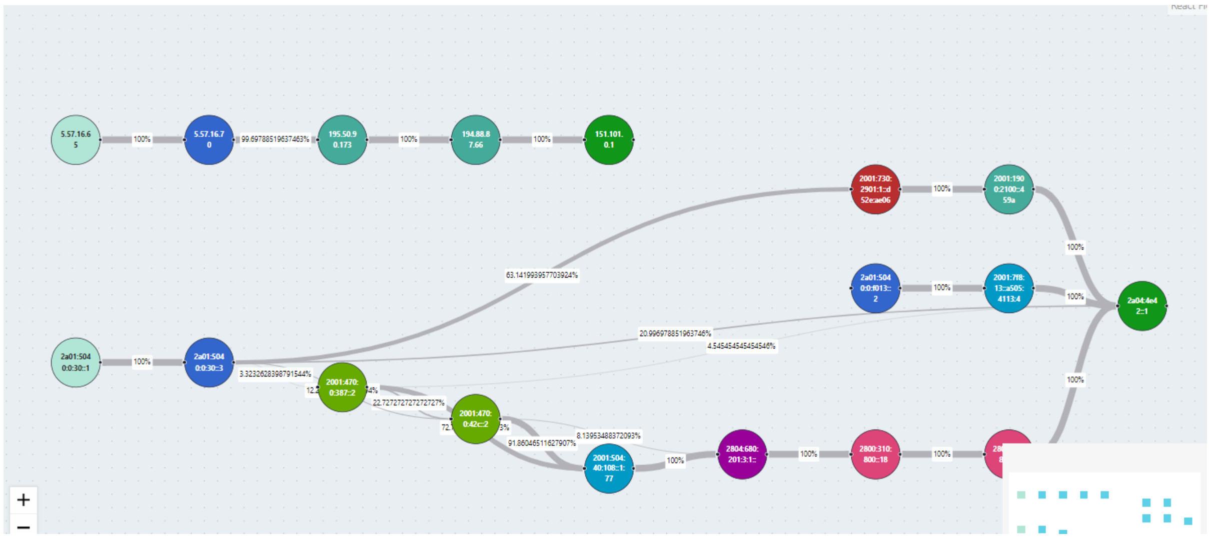
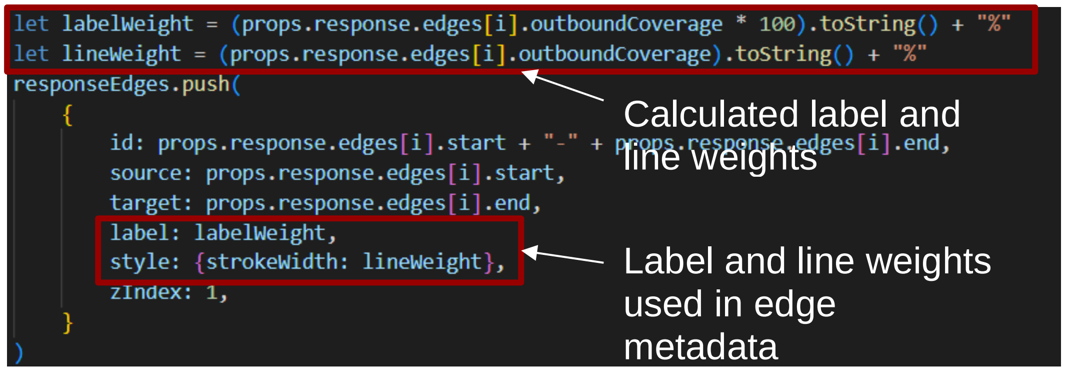


Figure 23: Clean Traceroute Data with ASN Colors.

Additional data, outbound and total traffic coverage, is provided to the edges to specify how often they are used. This data is important to understanding the frequency of path changes for certain nodes. To visualize this data, we made use of ReactFlows customization. ReactFlow edges are treated as Scalable Vector Graphics, SVG, objects. SVG objects can be customized using basic CSS properties. As shown below in Figure 24, we modified the edges' stroke width with its "outboundCoverage" element. This means the lines will be thinner when they are less traversed on. Also shown in Figure 24, we made use of the ReactFlow edge property "label" in order to label each edge with its percent usage of all packets that have traversed it.



```
let labelWeight = (props.response.edges[i].outboundCoverage * 100).toString() + "%"
let lineWeight = (props.response.edges[i].outboundCoverage).toString() + "%"
responseEdges.push(
    {
        id: props.response.edges[i].start + "-" + props.response.edges[i].end,
        source: props.response.edges[i].start,
        target: props.response.edges[i].end,
        label: labelWeight,
        style: {strokeWidth: lineWeight},
        zIndex: 1,
    }
)
```

Calculated label and line weights
Label and line weights used in edge metadata

Figure 24: Using outboundCoverage to Determine ReactFlow Edge Weight and Label.

These visual representations, like shown in Figure 25, provide insight on which nodes frequently change outbound paths.

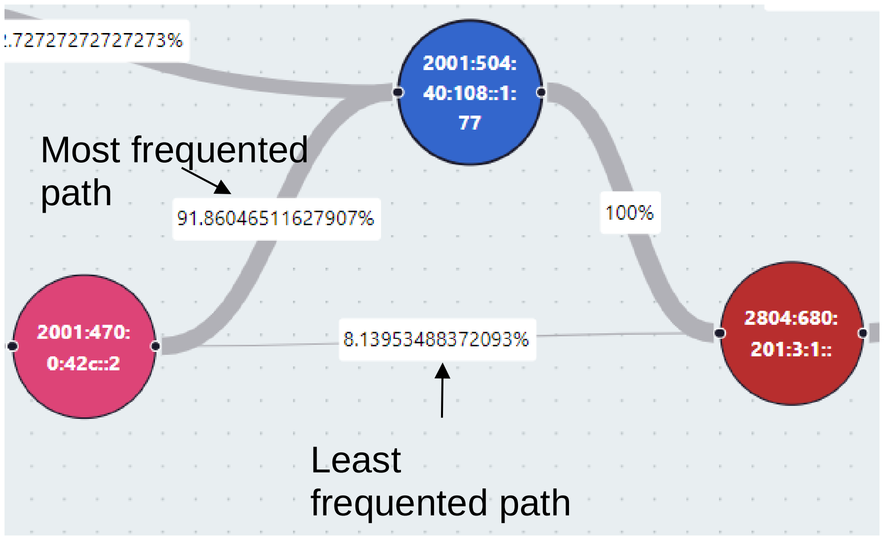


Figure 25: Visual Representation of Outbound Node Traffic.

Editing Measurements

A separate page was created to make use of the RIPE Atlas measurement changing API routes discussed previously. This page allowed users to start and stop measurements being performed on the back-end, as well as list what measurements are being performed. Figure 26 below shows what the page looks like.

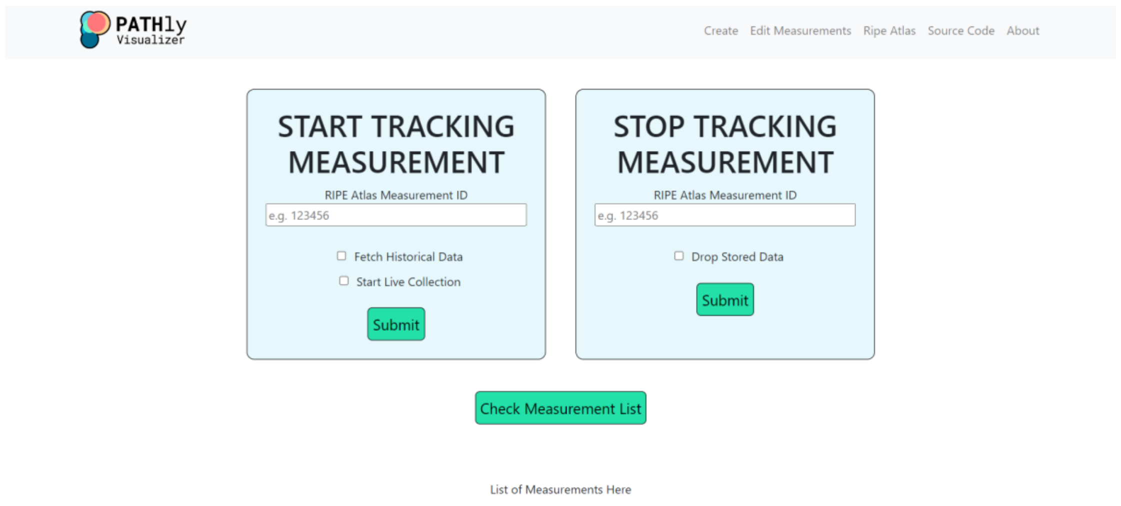
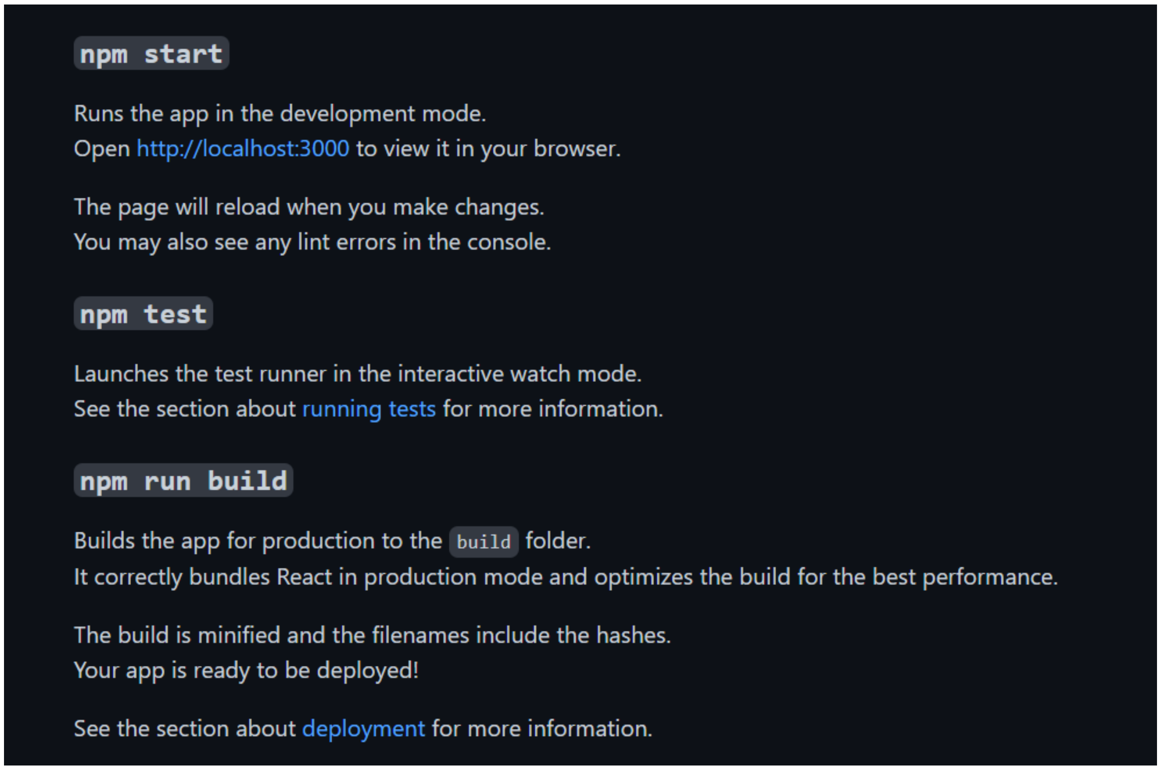


Figure 26: PATHly Edit Measurements Page.

Hosting

Because React applications run on a Node.js server, utilizing an NPM package called react-scripts, we can start the application and server with a command line prompt. Listed below in Figure 27 are react-scripts command line prompts that come with the prebuilt package.json file in the React boilerplate application.



`npm start`

Runs the app in the development mode.
Open http://localhost:3000 to view it in your browser.

The page will reload when you make changes.
You may also see any lint errors in the console.

`npm test`

Launches the test runner in the interactive watch mode.
See the section about running tests for more information.

`npm run build`

Builds the app for production to the `build` folder.
It correctly bundles React in production mode and optimizes the build for the best performance.

The build is minified and the filenames include the hashes.
Your app is ready to be deployed!

See the section about deployment for more information.

Figure 27: React Boilerplate React-scripts Command Line Prompts.

The front-end is hosted on one of the provided GCP *Virtual Machines* (VMs). We first installed all the necessary dependencies and node modules using the command "npm install --legacy-peer-deps". We wanted to use "npm start" to boot up the application. However, "react-scripts start" defaults to hosting on port 3000, which is restricted access on the VM. We modified the command to "PORT=8000 react-scripts start" so that "npm start" boots up on port 8000, a public port, instead of 3000.

During testing and development, to be able to access backend Rest API routes, all route paths in the front-end were changed to directly access the VM with its IP address:

```
http://<ip_addr_here>:8080/…
```

4.3 SUMMARY

Our implementation of PATHly Visualizer utilized a service model in Go. Our services included retrieving traceroute data from RIPE Atlas, parsing the traceroute data for internal storage, managing probe information, and cleaning up old data. Our services are linked to our front-end, developed in React JS, through REST API routes. PATHly Visualizer has a "Create

Visualization” page where users can interactively display traceroute data. This can allow them to better understand and evaluate Anycast forwarding in IPv4 and IPv6. There is also a page to edit ongoing measurements, where users can view, add, and stop tracking RIPE Atlas measurements. PATHly Visualizer is hosted on a Google Cloud Platform VM.

5. FUTURE WORK

While the team successfully implemented a minimum viable product to suit Fastly's needs, there is more work to be done to improve the reach and application of PATHly. This section will discuss what we believe to be the most important key features to prioritize in future project iterations.

5.1 Full Route Implementation

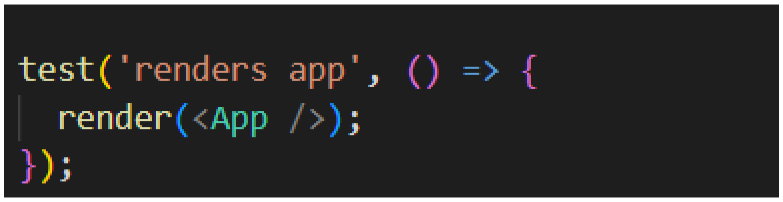
One integral implementation would be the full implementation of missing routes specified in documentation. Currently, the work is mostly complete for the "Get Probes" API route. However, it was not fully working by the time we needed to complete active development, so it was left undone. This API route would work by checking when a user selects a destination IP address on the front-end, and sending a request with the destination address as the body to get all probes linked with it. These probes would then be used to populate a selection menu, rather than having the user input a probe manually. This would eliminate user input error, as any probe that a user could choose should both exist and be connected to the corresponding destination address. The "Get Destinations" API route would also be an important addition. Currently, users are able to start new RIPE Atlas measurement tracking from the front-end, but are unable to visually represent these measurements. If the "Get Destinations" API was implemented, the front-end would create a GET request to fetch the array of all possible current destinations every time PATHly's home page is opened. These destinations would then populate the destination selection menu.

5.2 More Continuous Integration

While we did thoroughly test this project, further methods of testing can be implemented. Firstly, there could have been more *Continuous Integration (CI)* using GitHub Actions. The tests we have are necessary and important for ensuring that newly pushed code does not break any functionality. However, more tests related to key features of the program and not just dependency installation or build tests would be an important addition. Furthermore, UI testing is minimal within our project and can be vital to verify its functionality. A potential testing option is Jest, a JavaScript testing framework that works with React projects as well.[26] Tests can be

[26] From (n.d.). Jest · Delightful JavaScript Testing. Retrieved December 9, 2022, from https://jestjs.io/

made using the React testing library in the App.test.js file. Figure 28 shows an example of a test that verifies the application renders correctly.



```
test('renders app', () => {
  render(<App />);
});
```

Figure 28: React Testing Library with Application Rendering Test.

5.3 Additional Public Data Sources

Another future improvement would be including publicly available BGP data. Earlier in the project, we performed experiments collecting *RIS (Routing Information Service) Live* data from RIPE NCC[27] using a golang websocket library.[28] This data could be used to gain a high level view of global routing. This could then inform our decision on which routes we measure and how frequently. We believe that this data would better enable network engineers to analyze Anycast routing on PATHly and is a vital feature to implement in future work.

5.4 Better Visualize Difference Between IPv4 and IPv6

PATHly's goal as an application is to allow for network engineers to ascertain differences between IPv4 and IPv6 Anycast forwarding. As PATHly is now, engineers are left to do that at an eye level as IPv4 and IPv6 paths are rendered separately on the visualized graphs. At the start of the project, we created UI mockups trying to determine the best way to visualize the traceroute data. We ended up splitting up the IPv4 and IPv6 paths as we were unsure how often the paths truly lined up with each other. To make determining path differences easier, we believe it would be important to create a better method of visualizing the two paths. This would require work on figuring out how to determine if an IPv4 and IPv6 address belong to the same probe. Figure 29 shows a mockup of one potential graphical visualization, how the graphs could have looked where the different protocols are represented by shapes. IPv4 nodes are squares. IPv6 nodes are hexagons. Their intersections are then represented by circles.

[27] From (n.d.). RIS Live — RIPE Network Coordination Centre. Retrieved December 9, 2022, from https://ris-live.ripe.net/

[28] From *gorilla/websocket: A fast, well-tested and widely used WebSocket implementation for Go.* (n.d.). GitHub. Retrieved December 9, 2022, from https://github.com/gorilla/websocket

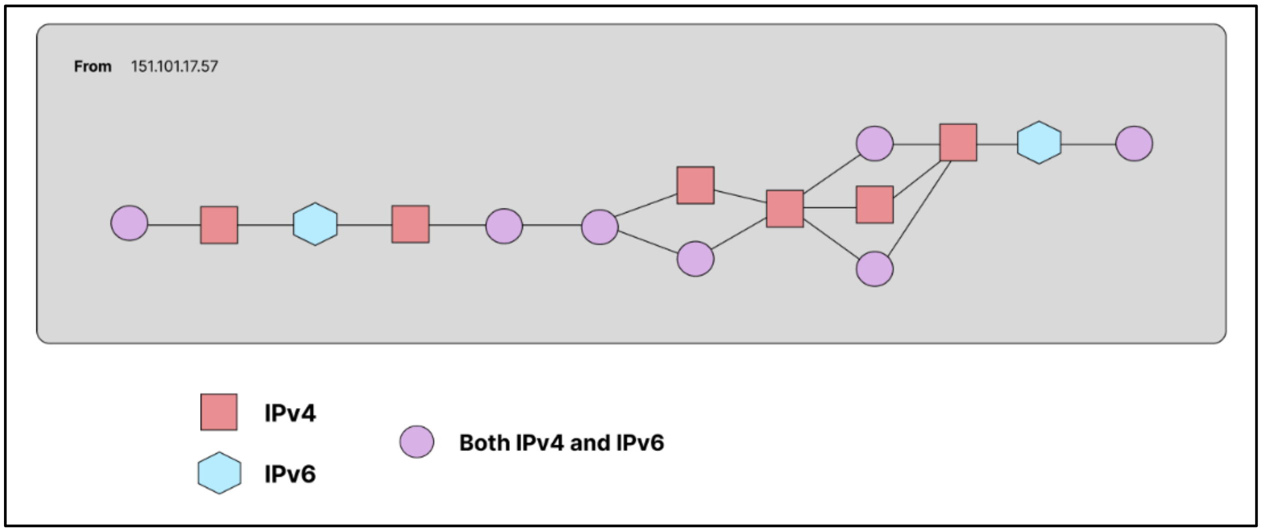


Figure 29: UI Mockup of Combined IPv4 and IPv6 Paths.

5.5 Better Visualize Most Frequently Changing Paths

PATHly would benefit from features that make it easier to identify what graphs have more frequently changing paths. Currently, network engineers using PATHly must identify what graphs have frequently changing paths by eye. Furthermore, there's no current visual representation for how long ago a path was used by a packet. We had previously discussed a feature that would change an edge's color based on its "lastUsed" Unix Timestamp. These colors could be on a scale where a lighter shade represents a path that hasn't been used recently, while a darker shade represents a path used not that long ago. A mockup of this feature can be seen in Figure 30. Furthermore, users representing multiple graphs may want to compare graphs based on how often their paths change. This could be done by creating a sorting method that leaves graphs that change often towards the top of the page, and graphs that don't change often towards the bottom.

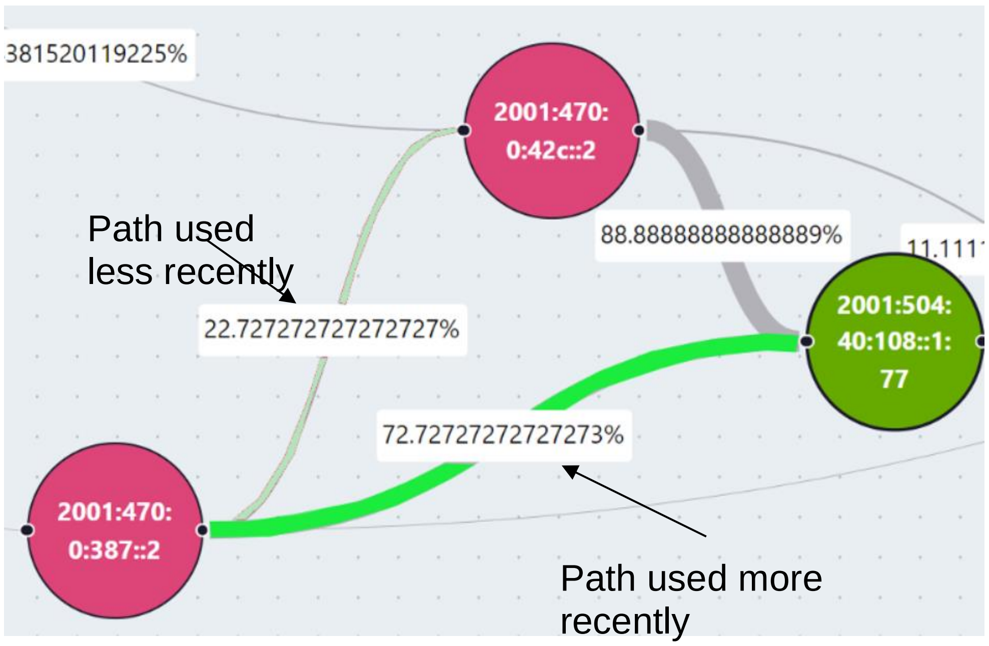


Figure 30: Mockup of Representing Edge Usage with Color Scale.

6. CONCLUSION

IP Anycast is a routing delivery scheme where multiple locations are identified by the same IP range, and the network distributes traffic across those locations. If one location fails to connect, traffic can be redirected to another location within the IP range. Modern Internet services make significant use of IP Anycast as it allows CDNs like Fastly to process and cache network data closer to the end users. Anycast usage bolsters system performance, security, and load balancing. However, Anycast traffic management is difficult. This is especially evident in IPv6 Anycast variations due to their lack of maturity in comparison to their IPv4 counterparts. Packet routing is variable, meaning packets can take several path variations going from a source to a destination.

To address this, this project's aim was to create a tool that utilizes public measurement data in order to identify and analyze differences in IPv4 and IPv6 Anycast routing. To fulfill this project's vision, we developed a traceroute visualization tool called PATHly Visualizer, using RIPE Atlas data being forwarded towards Fastly's Anycast servers. PATHly provides clean and customized visualizations of RIPE Atlas' traceroute data. PATHly users are also able to modify what RIPE Atlas measurements are being tracked on the backend. Network engineers can utilize PATHly to discern where path changes often occur and identify differences between both the IPv4 and IPv6 forwarding.

We evaluated our code's efficiency and functionality through GitHub Actions, rigorous code reviews, and application testing. GitHub Actions were written to verify specific functionality whenever new code was pushed to our codebase. Code reviews were done often by several different reviewers to make sure all code was written efficiently and documented well. Finally, the application was run dozens of times to verify its functionality and to make any minor tweaks to its appearance. PATHly is hosted on Fastly distributed VMs on GCP, which allowed us to host a prototype of the application and receive feedback from our sponsor. Given all the testing performed throughout the project, we validated PATHly's functionality and efficiency.

PATHly enables users to visualize the path a packet takes from a source probe to its destination in an accessible and low-cost manner. This method of analyzing Anycast path visibility provides for a more user-friendly experience and can open more doors in the world of Anycast stability research. There are existing tools that do similar work, but they are often only available at the enterprise-level and at high costs. PATHly provides Fastly engineers with a way to visually explore packet routes. As technology continues to grow and the world becomes more interconnected, it is crucial to be able to better understand the systems and networks in place.

PATHly is one step to gaining more insight into Anycast stability, which can lead to a wealth of technological development.

REFERENCES

(n.d.). RIS Live — RIPE Network Coordination Centre. Retrieved December 9, 2022, from https://ris-live.ripe.net/

(n.d.). Jest · Delightful JavaScript Testing. Retrieved December 9, 2022, from https://jestjs.io/

(n.d.). DNS-OARC: Home. Retrieved December 12, 2022, from https://www.dns-oarc.net/

Cloudflare. (n.d.). *What is an autonomous system? | What are ASNs?* Cloudflare. Retrieved November 8, 2022, from https://www.cloudflare.com/learning/network-layer/what-is-an-autonomous-system/

Cloudflare. (n.d.). *What is the network layer? | Network vs. Internet layer*. Cloudflare. Retrieved November 8, 2022, from https://www.cloudflare.com/learning/network-layer/what-is-the-network-layer/

codstar. (2019, May 27). *traceroute command in Linux with Examples*. GeeksforGeeks. Retrieved December 6, 2022, from https://www.geeksforgeeks.org/traceroute-command-in-linux-with-examples/

Color contrast checker analyzer tool. (n.d.). Adobe Color. Retrieved December 7, 2022, from https://color.adobe.com/create/color-contrast-analyzer

Differences between IPv4 and IPv6. (2022, June 28). GeeksforGeeks. Retrieved November 8, 2022, from https://www.geeksforgeeks.org/differences-between-ipv4-and-ipv6/

DNS-OARC/ripeatlas: Go bindings for RIPE Atlas API. (n.d.). GitHub. Retrieved December 12, 2022, from https://github.com/DNS-OARC/ripeatlas

Fisher, T. (2022, March 8). *What Is a Hostname? (Host Name Definition)*. Lifewire. Retrieved December 6, 2022, from https://www.lifewire.com/what-is-a-hostname-2625906

Garn, D. M. (2022, April 4). *The Ping Command | Computer Networking*. CompTIA. Retrieved November 8, 2022, from https://www.comptia.org/blog/understanding-ping-command-results

GeeksforGeeks. (2022, July 1). *Border Gateway Protocol (BGP)*. GeeksforGeeks. Retrieved November 8, 2022, from https://www.geeksforgeeks.org/border-gateway-protocol-bgp/

gin-gonic. (n.d.). *gin-gonic/gin*. GitHub. Retrieved December 15, 2022, from https://github.com/gin-gonic/gin

gorilla/websocket: A fast, well-tested and widely used WebSocket implementation for Go. (n.d.). GitHub. Retrieved December 9, 2022, from https://github.com/gorilla/websocket

Gridelli, S. (2019, November 27). *Traceroute Benefits and Limits*. NetBeez. Retrieved December 3, 2022, from https://netbeez.net/blog/traceroute/

Grimmick, R. (2022, June 25). *What is Traceroute? How It Works and How to Read Results*. Varonis. Retrieved December 6, 2022, from https://www.varonis.com/blog/what-is-traceroute

How Anycast Works - An Introduction to Networking. (2018, October 4). KeyCDN. Retrieved November 8, 2022, from https://www.keycdn.com/support/anycast

ICMP - Echo / Echo Reply (Ping) Message. (n.d.). Firewall.cx. Retrieved December 6, 2022, from https://www.firewall.cx/networking-topics/protocols/icmp-protocol/152-icmp-echo-ping.html

Kesavan, A. (2016, May 19). *Using Anycast for Internet Services*. ThousandEyes. Retrieved December 5, 2022, from https://www.thousandeyes.com/blog/using-anycast-for-internet-services

React Components. (n.d.). W3Schools. Retrieved December 5, 2022, from https://www.w3schools.com/react/react_components.asp

Sub Flows Guide. (n.d.). React Flow. Retrieved December 7, 2022, from https://reactflow.dev/docs/guides/sub-flows/

Taylor, R. (2021, July 22). *IPv4 vs IPv6: What's the difference? – BlueCat Networks*. BlueCat Networks. Retrieved November 8, 2022, from https://bluecatnetworks.com/blog/ipv4-vs-ipv6-whats-the-difference/

What is RIPE Atlas? (n.d.). RIPE Atlas. Retrieved December 8, 2022, from https://atlas.ripe.net/about/

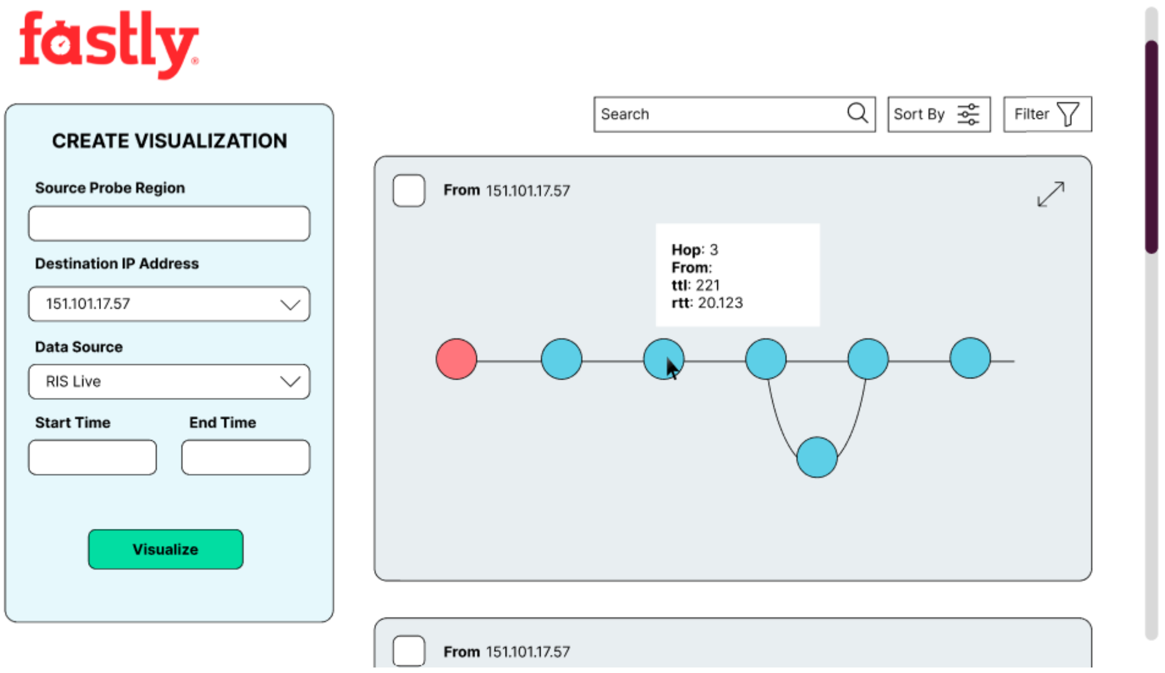
Wikimedia. (n.d.). *Routing*. Wikipedia. Retrieved November 8, 2022, from https://en.wikipedia.org/wiki/Routing

APPENDIX A - USER STORIES

As a network operator, I want to be able to:

- view IPv4 + IPv6 paths from a particular probe to a target (Fastly, or other) to show the inbound paths taken
- view IPv4 + IPv6 paths from a particular ASN to a target to show the inbound paths taken
- view IPv4 + IPv6 paths from a particular country to a target to show the inbound paths taken
- view IPv4 + IPv6 paths *via* a particular ASN to identify paths that traverse a problematic network
- view all paths taken from a probe recently and show recent changes to help identify network instability
- view all paths taken from a probe recently and show which paths are taken most frequently, to indicate typical vs. atypical paths
- visualize side-by-side IPv4 and IPv6 paths at the IP or ASN level, to identify differences by eye
- prioritize and view paths that change frequently, as a means to identify routing issues
- identify AS paths where IPv4 and IPv6 paths do not match, as a sign of mismatched network policy

APPENDIX B - USER INTERFACE MOCKUPS


CREATE VISUALIZATION

Source Probe Region:

Destination IP Address: 151.101.17.57

Data Source: RIS Live

Start Time:

End Time:

Figure 31: Main Dashboard Mockup.

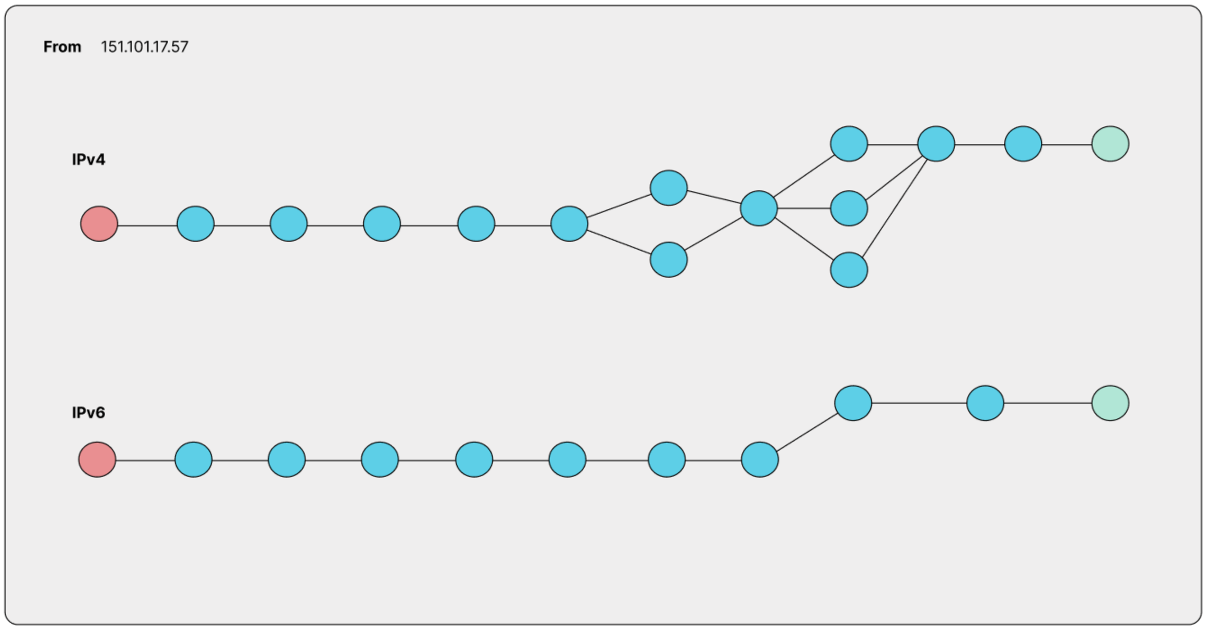


Figure 32: Data Visualization Mockup 1 - IPv4 and IPv6 Side-By-Side.

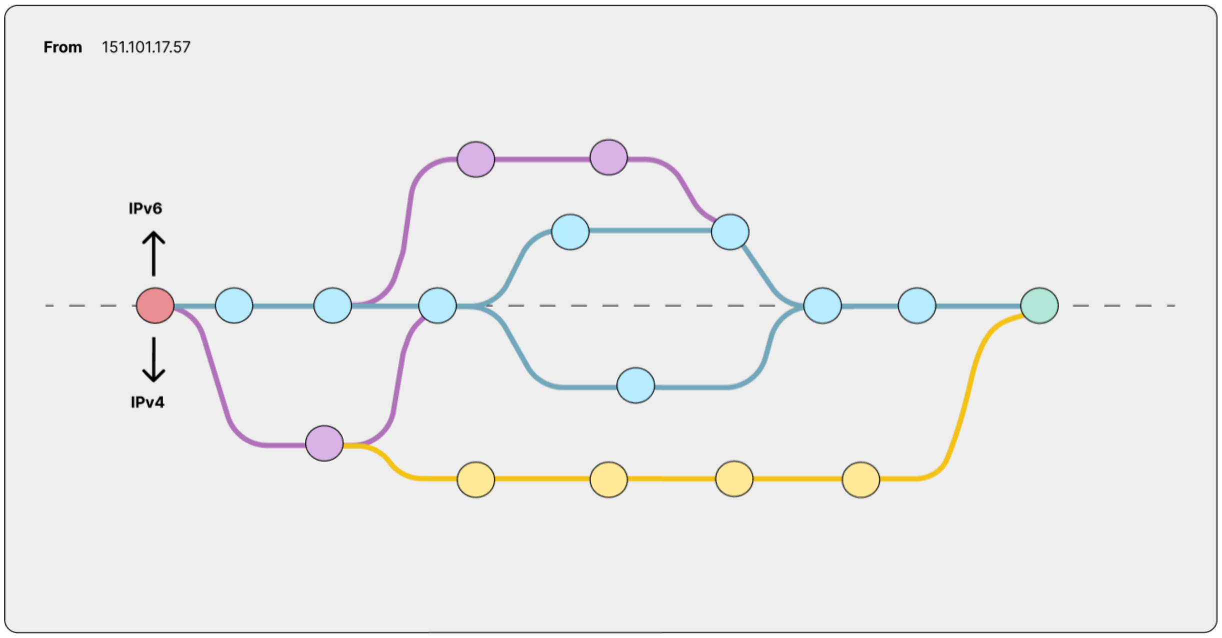


Figure 33: Data Visualization Mockup 2 - IPv4 and IPv6 Branching.

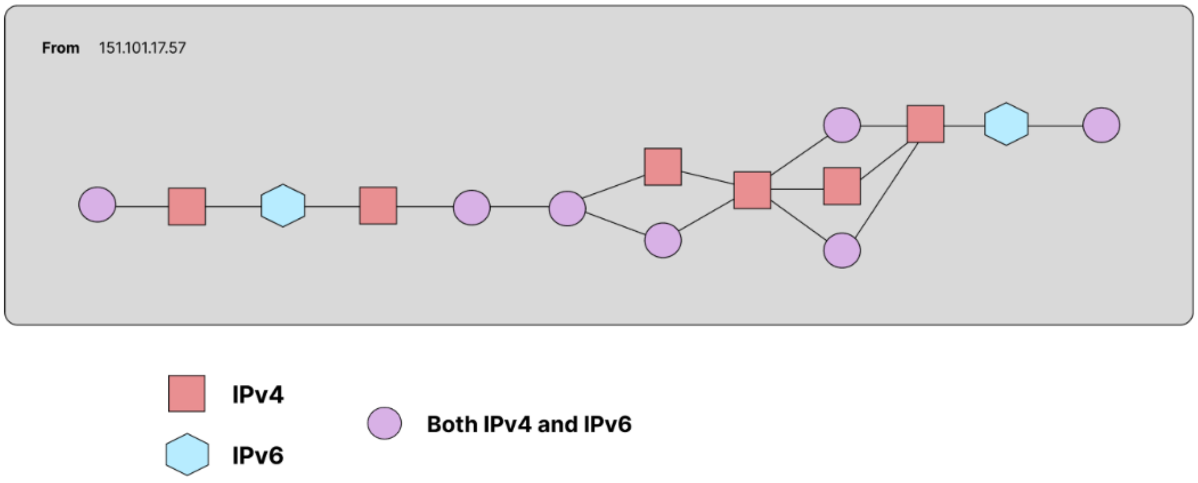


Figure 34: Data Visualization Mockup 3 - IPv4 and IPv6 Together.

APPENDIX C - GOLANG ISSUES

Our team had little experience with Go, but with it being one of Fastly's primary languages and a huge learning opportunity, we decided to take on the challenge of learning and utilizing Go.

After beginning to learn Go, we ran into a number of issues with the language. These issues may have been mitigated had we known which tools and frameworks we would be using earlier in the project preparation term. This would have allowed us to front-load gaining more familiarity with the language. Despite having decided to break into two teams for the frontend and backend, all project team members spent roughly the first two weeks learning and working with Go on the backend. If given the opportunity to do the project again, the team should split into sub teams earlier on to make more effective use of time.

Of the challenges we faced with Go, many were derived from how the language is set up. Some of these stumbling blocks were relatively minor since they could be overcome by gaining more experience and familiarity with the language. However, some complications that arose from Go included:

- **Go's insistence on no unused variables.** This could be frustrating at times since it inhibited learning through trial-and-error. If an unused variable gave a linter warning rather than a full compilation error, it would have made for better error handling.
- **Lack of support for iterators.** This can greatly complicate the process of abstraction. You can choose to follow one convention, but it starts to break down when dealing with other libraries and value/error pairs. For example, the steps involved in processing traceroute data could be easily described by an iterator chain. However, this would require that we implement our own iterator type and functionality that would normally be provided by the standard library (map, filter, fold, chain, flatten, etc.).
- **Lack of first class support for tuples.** Go tries to get around this by allowing functions to return tuples that must be immediately unwrapped. However since these are not types so they don't play well with the new generics system.
- **Poor error handling.** While treating errors as values can have some benefits, Go lacks an easy way to do propagation. It is frustrating to manually propagate errors and checking errors before using the function result is not enforced in any way. This issue could have been mitigated if the Go language had support for a try operator or builtin macro support.

- **Using absolute dependency paths is okay, but Go enforces their use everywhere.** If you want to fork a project then you need to change every import to your repository instead.
- **Lack of operator overloading.** This is a minor point, but it would make a lot of code more maintainable. This was not too much of an issue for our project though, as there were not many areas that would have benefitted from operator overloading.
- **Poor standard library collections.** They are not inefficient, but do have limited options. Slices are difficult to mutate as they are exclusively passed by value. You also do not get many standard library options other than maps and slices. If you want anything even slightly more complex like a dequeue, min heap, or binary tree map you will need to implement it yourself or find a library.