UNIVERSITY
OF OSLO

Master thesis

A Large-Scale Measurement Study of the IPv6 Flow Label

Erlend Hapnes

Programming and System Architecture
60

Department of Informatics
Faculty of Mathematics and Natural Sciences

A Large-Scale Measurement Study of the IPv6 Flow Label

Erlend Hapnes

Master's Thesis
Department of Informatics
University of Oslo

May 14, 2023

Abstract

Congestion control coupling methods rely on knowing if multiple flows between the same host-pair share the same bottleneck. However, IPv4 load balancing hash algorithms traditionally use data from the transport layer header for flow identification, making such a guarantee impossible. We propose a novel method of controlling packet path behaviour using only IPv6 network-layer header data, and perform measurement studies from vantage points stationed around the globe to verify our proposed methodology. We identify challenges with traceroute measurement studies in the face of Equal-Cost Multi-Path (ECMP) and Link Aggregation Group (LAG) load balancing. Finally, we devise and execute an extensive measurement study to investigate whether the IPv6 flow label is completely carried to its destination and whether it can influence the behaviour of load balancing algorithms using a Paris traceroute-based implementation.

Contents

List of Figures

Listings

Chapter 1

Introduction

1.1 Problem Statement and Motivation

It is common that multiple TCP connections are initiated between the same source-destination pair. Different shortest paths can be taken by these TCP connections between the same two hosts because of Equal-Cost Multi-Path (ECMP) or Link Aggregation Group (LAG) load balancing algorithms in routers. However, such parallel connections may also traverse a common path. This may result in competition since each connection uses its own congestion control instance and tries to maximize its sending rate. Such competition can cause undesirable spikes in queuing delay and packet losses. Such competition can be eliminated by using a coupled congestion control mechanism which combines the congestion control mechanisms of all the flows sharing a common path. Authors in [51] have shown that a TCP congestion control coupling mechanism (ctrlTCP) can significantly improve the overall performance by reducing overall delay and loss and exert precise allocation of the available bandwidth. If we can deduce that flows can share a common network path, we can always apply such a congestion control coupling mechanism.

1.2 Contribution

This thesis presents a comprehensive measurement study investigating how far along an Internet path the IPv6 flow label header field survives. Motivated by [51], we also investigate whether we can get a consistent path for multiple unique TCP connections between the same host-pair by choice of IPv6 flow label using a Paris traceroute-based [21] implementation and DigitalOcean vantage points to IPv6 destinations provided by the IPv6 Hitlist Service [43].

1.3 Research questions

The main research questions that this thesis will attempt to answer is:

- Does the IPv6 flow label get changed in-transit, and if yes, how often?
- Is the 3-tuple consisting of (source IP, destination IP, flow label) a valid flow identifier in-use in today's internet?

- Can we control a TCP path as a result of setting the flow identifier?

We address these by running Internet-wide measurements to figure out if we can enforce this situation by choice of IPv6 flow label.

1.4 Thesis outline

This thesis is split into the following sections: In Chapter 2, we introduce IPv6, TCP Congestion Control, network flow identification and load balancing before finally diving into measurement studies with traceroute. In Chapter 3, we introduce a design for a comprehensive measurement study investigating the IPv6 flow label, whether it is safely carried to its destination and if it is used as a flow identifier today without the need to parse upper-layer information. In Chapter 4, we detail the implementation of our measurement study on top of Paris traceroute, while in Chapter 5, we discuss and analyze the results of our measurement study. Finally, Chapter 6 concludes the thesis and introduces future work.

Chapter 2

Background

This chapter aims to provide an overview of the most relevant concepts and related works to our thesis. In section 2.1 we provide an overview of IPv6; including its history, functionality and its family of related protocols. Section 2.2 provides a short introduction to TCP congestion control and congestion control coupling, the impetus for this thesis. In section 2.3 we delve deeper into multi-path routing and finally, in section 2.4, we discuss internet measurement-studies with traceroute and related works.

2.1 The Internet Protocol: IPv4 and IPv6

The Internet Protocol is the primary protocol in-use today for internetwork packet switching. The Internet Protocol is split into two major versions: IP version 4 (IPv4) and IP version 6 (IPv6). The primary purpose of the Internet Protocol is provide end-to-end addresses which identify a particular host. We call such an address an *IP-address*. Every device connected to the Internet requires at least one IP-address. When the Internet Protocol was first standardized, the creators decided on a 32-bit address field yielding an address space of 2^{32} possible addresses, thought to be sufficient at the time. However, as the Internet experienced exponential growth in the late 1980s and early 1990s, researchers quickly realized that the available IP-address space would soon become depleted [18] of allocateable addresses, and that more addresses were needed to cover the coming influx of devices (a problem commonly referred to as *IPv4 Address Exhaustion*). As such, it was decided that a new version of the Internet Protocol would be created.

IPv6 [RFC8200] is the next-generation Internet Protocol designed as a successor to IPv4 [RFC2460]. The primary purpose of IPv6 is to offer more routable addresses by extending the address space to 2^{128} possible addresses. In addition, it simplifies certain aspects of the protocol and implements new features not present in IPv4. An example of this simplification is the removal of the header checksum field, as it is considered redundant (checksums are instead implemented by lower and upper layer protocols such as Ethernet and TCP). It also simplifies the processing of packets in routers by placing the responsibility for packet fragmentation squarely at the end points, leading to closer adherence with a well-known principle in system design called the *end-to-end principle* [81]. Figures 2.1 and 2.2 illustrate the difference between the IPv4- and IPv6-headers.

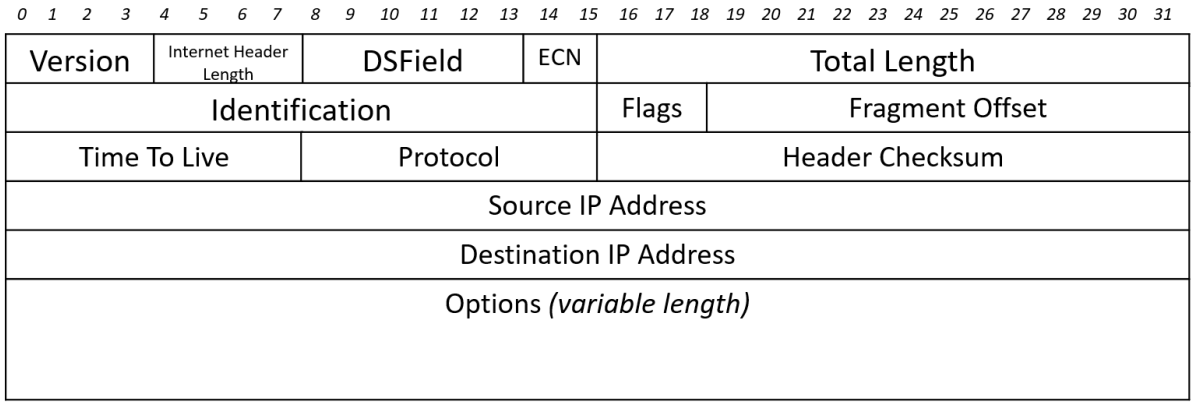


Figure 2.1: The IPv4 header

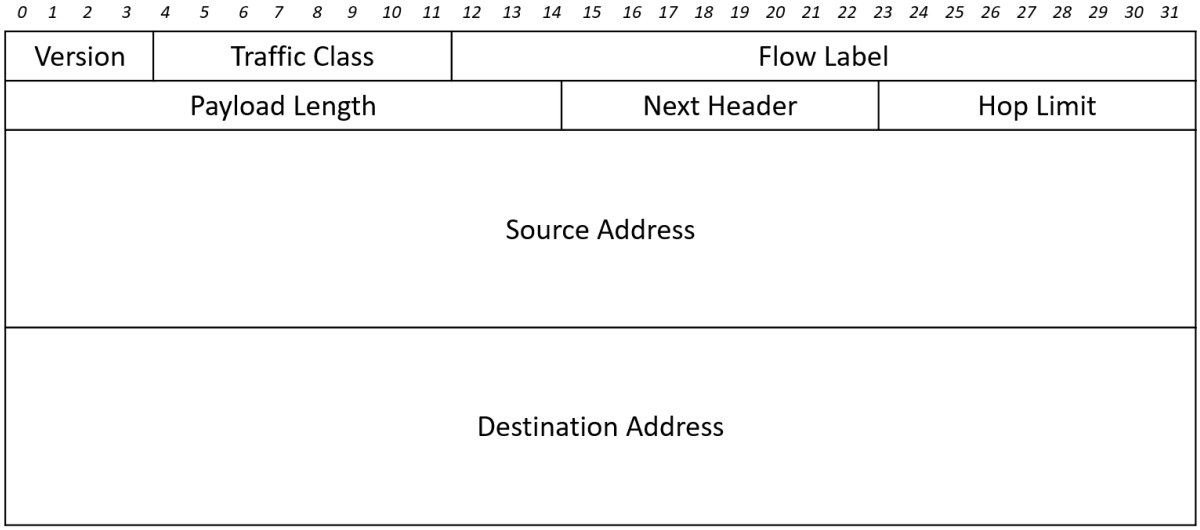


Figure 2.2: The IPv6 header excluding any extension headers

As seen in the figures, the headers alone contain a number of changes. Most obvious is the increase in source- and destination-address field size from the 32-bit IPv4-field to the 128-bit IPv6-field. The IPv4 Options-field, previously a part of the main IPv4-header, has now been separated into a number of *extension-headers* which sit outside the main IPv6-header. We will discuss the IPv6 extension headers in more detail shortly. The Time-To-Live field has been renamed to the Hop Limit field in light of the deprecation of using clock timers to control packet lifetime, but stays the same 8-bit (255 hop) length. The Internet Header Length (IHL) field has been removed as it is no longer needed since the base IPv6-header is always a fixed size. In addition, fields that are used for packet fragmentation and reassembly such as the Flags, Identification and Fragment Offset fields have now been separated from the main header and moved into a distinct extension header. The IPv4 Protocol field, which indicates the type of the next-layer protocol contained in the IP payload has been renamed to the Next Header-field. IPv6 adds a 20-bit Flow Label field which can be used for layer 3 flow identification. We will be discussing the IPv6 Flow Label at length later in section 2.3.7. Finally, fields that deal with Type of Service (ToS) and congestion notification, such as the previously named Type of Service-field (now split into DSField [RFC2474] and ECN [RFC3168]) have been replaced by the Traffic Class-field.

2.1.1 Advances Over IPv4

In addition to simplifications and changes to the IP-header, IPv6 includes a number of advances over IPv4.

Addressing Model The IP Version 6 Addressing Architecture [RFC4291] describes IPv6's addressing model in detail. IPv6 addresses use a hexadecimal text representation, as opposed to the decimal representation of IPv4-addresses. In addition, the group separator is changed from a single dot "." to one or more colons ":". Due to its length, an IPv6-address may be represented in a compressed form where leading or consecutive zeroes are omitted. An IPv6-address can hold an embedded IPv4 address, where the IPv4-address is represented as an IPv6-address consisting of exclusively leading zeroes up to the last 4 octets which contain the IPv4-address. IPv6, unlike base IPv4, supports anycast addressing by default, but removes support for broadcast addressing, instead opting to replace it with multicast-addressing. This approach helps eliminate unnecessary broadcast traffic to uninterested nodes or hosts. Multicast is also handled differently, as IPv4 uses a separate protocol, the Internet Group Management Protocol (IGMP), to establish multicast group memberships, while IPv6 uses Multicast Listener Discovery (MLD), a baked-in part of ICMPv6.

Security IPv6 nodes are *required* to implement IPSec [RFC2401].

Neighbour Discovery The Neighbor Discovery [RFC4861] Protocol (NDP) implemented in IPv6 improves a number of features related to address discovery were previously implemented by a number of different protocols and mechanisms in IPv4. For example, it replaces IPv4's usage of the Address Resolution Protocol (ARP) to resolve link-layer addresses with a multicast-based implementation and defines a mechanism for Neighbor Unreachability Detection, for which there is no established protocol in IPv4. To perform its operations, the NDP builds on top of ICMPv6, defining five new ICMPv6 packet types used by NDP-speaking nodes to redirect, advertise and solicit addresses.

NAT elimination The usage of NAT brings with it a number of well-documented architectural implications [RFC2993], some positive while others largely disruptive such as going against the End-to-End model of the Internet. IPv6 was designed to make NAT unnecessary [RFC4864]. Its large address space eliminates the need for NAT in most situations, and other situations where NAT still has some applications within IPv6, such as obfuscating or hiding internal network topology, there have been proposed standards such as [RFC4941] for generating temporary IPv6 addresses.

2.1.2 IPv6 Extension Headers

IPv6 does not have support for options embedded in its header like IPv4. Instead, it includes extension headers which offer similar capabilities. The advantage of this design is that the IPv6 header will always be of a fixed size of 40 bytes, adding extension headers only when needed. This simplifies packet processing and can lead to increases in network performance. The presence of any Extension Headers will be indicated by the value of Next Header field, and several extension headers may be chained together to add a number of options. Typically we distinguish IPv6 extensions between Destination Options and Hop-by-Hop Options, where the Destination Options are only meant to be processed by the destination node while the Hop-by-Hop Options can be processed by each router (or hop) in the path to the destination.

The Hop-by-Hop Options Header This header contains information that must be processed by every node on the path to a destination. As of this thesis, only the Pad1 and PadN options, which are used to add padding, are defined by the IPv6 specification.

The Routing Header The Routing Header [RFC2460] specifies a list of one or more intermediate hops that need to be visited on the path to a packet's destination. This can be used to, partially or fully, control the path a packet takes through the network[RFC2460].

The Fragment Header The Fragment Header contains information used to fragment and reassemble packets larger than the path Maximum Transmission Unit (MTU). In IPv6, only the source host is allowed to perform packet fragmentation, differing from IPv4 where each individual node along a path may fragment packets if the MTU to the next hop is not sufficiently large. Since the packet size specified by the host must not be larger than the minimum MTU across the entire path to the destination, a path MTU discovery mechanism is needed. IPv6 path MTU discovery works in conjunction with ICMPv6, where a packet is sent with an initial size, and if it is deemed too large to be forwarded by any of the nodes along the path, the packet is dropped and an ICMPv6 *Packet Too Big*-message is generated in response, indicating to the source host that a packet size reduction is needed.

The Destination Options Header The Destination Options header contains optional information only needed to be processed by the destination host. While the IPv6-specification allows the definition of new extension headers, it recommends that the destination options header is used instead of defining new extension headers unless explicitly required.

Authentication Header This is one of two extension headers providing additional security mechanisms to IPv6. As defined by [RFC2402], this header provides integrity and data origin authentication for IP packets, and additionally, provides optional protection against replay attacks provided a Security Association is established, which we won't delve into here.

Encapsulation Security Payload Header (ESPH) This is the second of two extension headers providing additional security mechanisms to IPv6. Often used in conjunction with the Authentication Header, any information following the ESPH is encrypted. Since the information following the header is encrypted, it is typically placed before the Destination Options header in the header chain, and after the other extension headers that may be processed by intermediary routers.

Mobility Header The Mobility Header as defined in [RFC6275] is used for the implementation of the Mobile IPv6 protocol which facilitates the movement of a node from one link to another while keeping a constant IPv6-address.

2.1.3 Transition To IPv6

The transition to IPv6 is, as of this thesis, still ongoing, however a full transition from IPv6 has been slow and may never completed. There are a number of reasons for the slow transition. For one, since there is no forced transition, each company or controlling interest will have to do work on their own time updating their currently functional IPv4 networking infrastructure to be IPv6 capable. Firmware updates or new hardware purchases may be needed, slowing the transition. In addition, a number of new IPv4 addresses were "found" by releasing or selling

IPv4-addresses from organizations whose address space was previously over-allocated. The widespread use of NAT has also played a major role, helping to alleviate the need for externally routable addresses. While the transition to IPv6 has been slow, it is gaining traction in recent years. According to a June 2018 report published by the Internet Society [11], about 25 percent of all Internet connected networks advertise IPv6 connectivity, and 28 percent of the Alexa top 1,000 ranked websites are currently functional with IPv6.

2.1.4 ICMP

The Internet Protocol by itself provides no way for a host or end-user to learn the outcome of a sent packet that fails to reach its destination. To support such services, a separate protocol sitting on top of IP was created: the Internet Control Message Protocol [RFC792] (ICMP) and its IPv6-equivalent; ICMPv6 [RFC2463]. ICMP is a special protocol used together with IP to provide diagnostics information, error reporting and round-trip time estimation, among others. It is the basis of the popular *ping*- and *traceroute*-utilities used to measure network responsiveness and discover network topology. ICMP technically sits on top of IP in the protocol stack, however it is typically considered a part of IP itself and not a higher level protocol like TCP or UDP, instead sitting somewhere in-between layer 3 and layer 4. Indeed, ICMP is so deeply coupled with the implementation of IP that there is a requirement that ICMP *must* be included in every IP implementation [RFC792]. An ICMP-packet is encapsulated in an IP-packet (either IPv4 or IPv6) and is identified by protocol-number 1 and 58 for ICMP and ICMPv6 respectively.

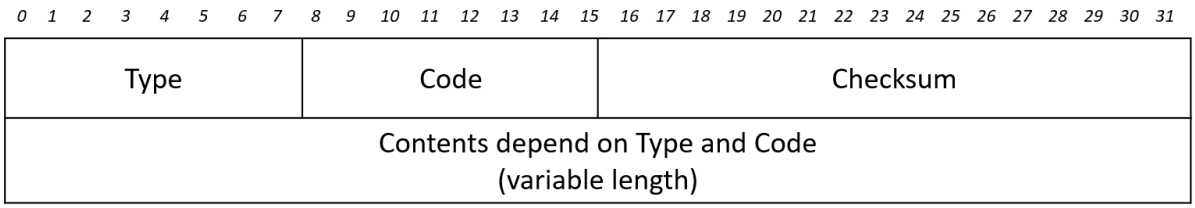


Figure 2.3: The ICMPv6 header

ICMP messages are split into two categories: Error Messages and Informational Messages. Informational Messages include the well-known Echo Request- and Echo Reply-messages, and mechanisms such as the previously discussed Neighbor Discovery (ND) and Multicast Listener Discovery (MLD) (ICMPv6 only). ICMP Error Messages are further divided into four types: Destination Unreachable, Packet Too Big, Time Exceeded and Parameter Problem, where each type corresponds to its own class of errors. The types themselves are subdivided into more explicit errors specified by the 8-bit value in the *code*-field in the header. Common to all ICMP Error Messages is that they include part of the invoking packet which triggered the error in their payload, as illustrated by figure 2.4.

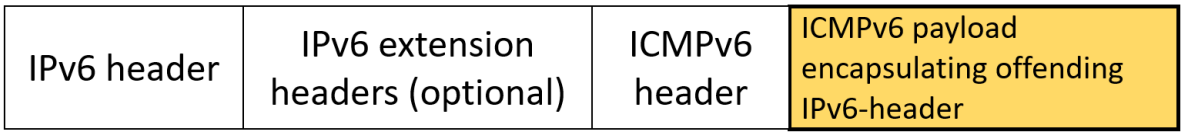


Figure 2.4: The response ICMPv6 payload includes a copy of as much of the invoking IPv6 packet as possible without exceeding the minimum path MTU

2.2 TCP Congestion Control

The Transmission Control Protocol [RFC793] (TCP) is a connection-oriented, stateful host-to-host protocol that provides reliable, ordered, and error-checked delivery of a stream of bytes between applications running on hosts that communicate via an IP network. In the early days of the Internet, as computers were becoming increasingly connected and hardware was improving, a problem was quickly becoming apparent: the outgoing transmission rate of connected Internet hosts was overwhelming the available bandwidth in the network. Not only that, due to faults in the TCP packet loss detection and retransmission algorithm, if a packet was as lost it would sometimes be retransmitted more than once, further congesting the network[52]. Starting in October 1986 this lead to a series of events known as congestion collapses, where the transmission rate between two nodes could be reduced by more than a factor of a thousand compared to the normal rate. As a result, a number of measures primarily developed by Van Jacobson were implemented in the TCP protocol, widely credited with "saving the internet".

2.2.1 Congestion Control Algorithms

TCP has several congestion control algorithms that work in conjunction to collectively form TCP's congestion control mechanism:

Slow-start

The slow-start algorithm [52] is designed to gradually increase the amount of data in-transit by a sender. The algorithm adds a new variable, the congestion window *cwnd*, to the per-connection state. On initial startup, or when restarting after packet loss, the *cwnd* is set to equal the size of one packet. After this, the *cwnd* is increased by the size of one packet on each ACK received. When sending, the minimum of the receivers advertised window and cwnd will be sent. While the algorithm is called **slow-start**, the name is deceptive in that the slow-start algorithm is actually an exponential increase algorithm, meaning that the slow-start may not be so slow after all.

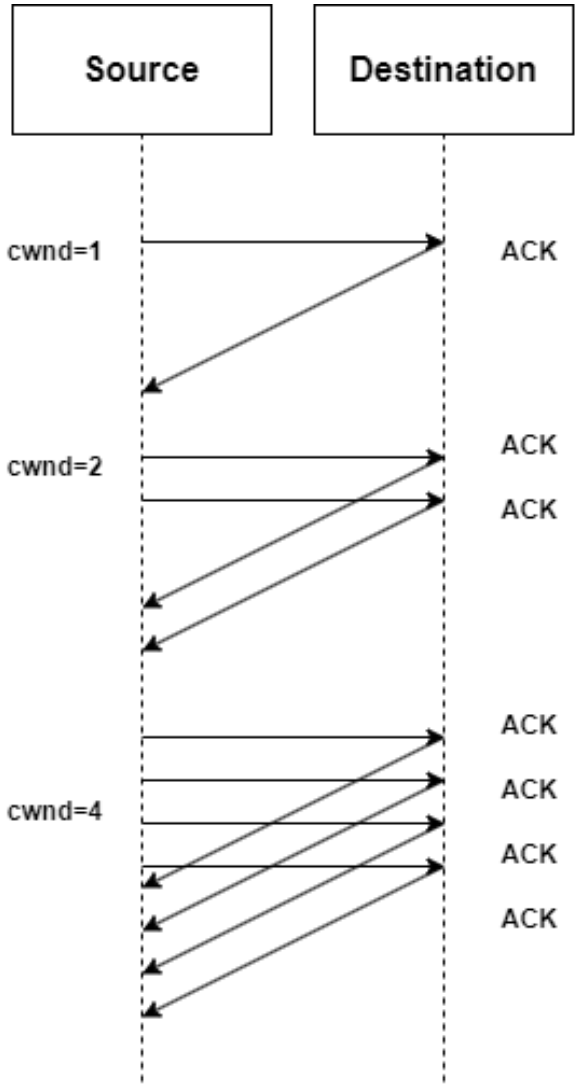


Figure 2.5: Demonstration of the TCP Slow-start algorithm. As each acknowledgement is received by the source host, the cwnd is incremented by one leading to an exponential increase.

The slow-start phase will execute as long as the *cwnd* is less than the slow-start threshold *ssthresh*. Once the *cwnd* catches up to it, the slow-start algorithm will reach equilibrium and TCP will enter the Congestion Avoidance phase.

Congestion Avoidance

In Congestion Avoidance (CA), the *cwnd* increases linearly instead of exponentially by incrementing the *cwnd* by one for each window of acknowledged segments, or in other words: $cwnd += 1/cwnd$. TCP uses a packet timeout as an indicator that the network is congested. If congestion is detected, CA will set the *cwnd* to half of its current value and continue with the linear incrementation. However, if CA is used in conjunction with slow-start, it will instead set the *ssthresh* to half the current *cwnd*-value and reduce the *cwnd* to one, thereby re-initiating slow-start.

Fast Retransmit

Fast retransmit [RFC5681] is a strategy that uses duplicate acknowledgements, instead of waiting for a timeout, as an indicator for packet loss. This reduces time spent waiting until a packet is retransmitted. As an example, if a sender sends four packets, and packet number two is lost, the receiver will continuously acknowledge only the last known value before the out-of-order segment was detected, in effect sending duplicate ACKs upon receiving packet three and four. If the sender receives three identical duplicate ACKs in a row, it will assume that a packet was lost and instantly retransmit the missing segment without waiting for the retransmission timer to expire.

Fast Recovery

If the sender receives a duplicate ACK message from the receiver, it indicates that a segment was received out-of-order. The sender can then assume that a segment was lost in-transit, but also that the congestion is clearing up since, as opposed to receiving no message at all (a timeout), a duplicate ACK message was instead successfully transmitted and received through the network. This means that the sender can continue sending segments, albeit at a reduced rate, as opposed to going back to slow-start.

2.2.2 TCP Variants

TCP has several variants developed over the years adding improvements to the TCP implementation. Some of the most prominent variants are *TCP Tahoe*, *TCP Reno* and *TCP NewReno*.

TCP Tahoe

TCP Tahoe, named for Lake Tahoe around which it was designed, is the first variant of TCP that includes Congestion Control.

TCP Reno

TCP Reno is an extension of TCP Tahoe that includes Fast Recovery.

TCP NewReno

TCP NewReno, defined in [RFC6582], is an extension to TCP Reno that adds improvements to the fast recovery phase that occurs after three duplicate ACKs have been received.

2.2.3 TCP Congestion Control Coupling

TCP Congestion Control Coupling (TCP-CCC) is the concept of combining the congestion control mechanisms of multiple TCP connections between the same pair of hosts. Throughout the years, there have been several proposals for a shared congestion control mechanism as it could have many benefits such as more efficient multiplexing and increased performance.

Ensemble-TCP

Ensemble-TCP [40] (E-TCP), is a variant of the TCP stack that utilizes shared TCP state variables to achieve improved performance. Most TCP implementations save TCP state information such as the *cwnd*, *ssthresh* and round-trip time (rtt) in a data structure called the TCP Control Block (TCB). The TCB is typically unique per TCP connection, with no shared state information even between connections between the same host-pairs. This can be highly inefficient for new connections to the same host, which will have to initate from a slow-start position with a *cwnd* of one, despite the congestion information already being known by the previous connections. To address this, E-TCP performs destination matching when creating a new connection and checks if the connection can be associated with one of the cached or existing connections. If there is a match, the congestion state variables will be shared between the two connections forming an aggregate variable. On a successful ACK, the aggregate variable is increased, and if there is packet loss, the aggregate variable is decreased.

Congestion Manager

Congestion Manager (CM), as defined in [RFC3124], is a framework consisting of an API, a scheduler and a Congestion Controller that allows applications to share congestion information, learn about network characteristics and schedule data transmissions. The end goal for CM is to increase efficiency, particularly among heterogeneous data-streams originating between the same host-pairs.

Streams between host-pairs often end up competing for resources because they lack a way of sharing information amongst each other. In addition, many applications built on top of UDP often have their own transmission mechanisms that do not react well to congestion control. In contrast, by registering themselves with the CM API, applications are able to communicate with the central controller and request bandwidth- and congestion-information, removing the need to keep this information on a per-application basis. The CM-scheduler is then responsible for transmitting the messages in a controlled manner. CM groups applications on a per-destination basis, combining them into a *macroflow*, meaning a flow whose congestion state is shared.

ctrlTCP

CtrlTCP [51] is a mechanism devised by Islam et. al. for combining the congestion controls of multiple TCP connections. The mechanism differentiates itself from other coupled congestion control (CCC) implementations, such as Congestion Manager [RFC3124], in that it is able to couple *all* outgoing connections that traverse a shared bottleneck, including connections destined to different destinations.

In [51], the authors postulate that there are three approaches to identifying a shared bottleneck:

- Multiplexing multiple application-layer flows onto a single transport-layer connection.
- Configuration (requires network admin privileges)
- Measurement methods such as [46], which use network metrics such as packet loss, variability, oscillation and skewness.

In [51] and [50], the authors show that a coupled congestion control mechanism comes with significant advantages, albeit with some restrictions. Most notably, CtrlTCP relies heavily on

knowing that there is only a single shared bottleneck. In the case of multiple bottlenecks, the behavior quickly falls apart.

2.2.4 Multi-Path TCP

Multi-path TCP [RFC6824] (MPTCP) is a variant of TCP that allows TCP to utilize multiple simultaneous paths, if they are available and both the source and the destination are multi-path capable. TCP-flows naturally prefer to take a single path in order to prevent packet-reordering and head-of-line blocking. In addition, while both the source and destination will have a TCP state established during the three-way handshake, middleboxes on a secondary path which do not already have an established TCP state will reject TCP packets which do not have a corresponding TCP connection.

To address this, MPTCP uses multiple subflows, each sent over a single path, that individually look like and act as a regular TCP flow, but together comprise a single MPTCP connection. It is then up to the end host to "glue" the subflows together.

To establish a connection, MPTCP uses an Option in the TCP header during the initial SYN to indicate to the destination host that it is multi-path capable. If the destination is also MP-capable, it will respond by adding a similar Option to the returning SYN-ACK. Once the connection is established, the source is now free to establish a secondary subflow to the destination. When establishing a seconadry subflow, a new Option is set during its three-way handshake, indicating that it is a new subflow. If the subflow is accepted by the destination, it will add its confirmation to an Option in the subflow's returning SYN-ACK. The source will then acknowledge the SYN-ACK, finishing the subflow establishment. This process may be repeated for any number of subflows.

In order to distinguish between subflows, the source may vary any one of the following header fields:

- The source port
- The destination port
- The source IP address
- The destination IP address

However, since NAT may change addresses and port numbers, they cannot be used by the destination host as a reliable subflow identifier. In order to identify flows belonging to a single MPTCP-connection, MPTCP instead uses a unique integer identifier called a "token". The token is included as part of every transmission in the TCP header Option field.

Because stateful middleboxes reject packets with gaps in the sequence numbers, MPTCP uses two levels of sequence numbers: One which spans the entire MPTCP-connection, and one for each individial subflow which is used in the header. If a subflow fails completely, MPTCP re-transmits all unacknowledged data over the other working subflows. The data is then reordered by using the MPTCP-connection sequence number. If there are packet losses, MPTCP uses timeouts and the fast retransmit-algorithm as in regular TCP. For Congestion Control, MPTCP uses coupled congestion control mechanism with a single window (cwnd) shared by all subflows that is designed to not be unfair to hosts using a single TCP-connection.

From an application's point of view, the usage of MPTCP is entirely similar to regular single-path TCP. The MPTCP controller in the kernel is responsible for managing subflows.

2.3 Network Flow Identification and Load Balancing

In the real world, a packet will typically have more than just one single path to its destination, some of which will be of equal cost. Making use of these additional paths can have many advantages, such as higher bandwidth, reduced congestion and lower latency. However, it also brings with it its own set of challenges. In this section we will provide an overview of some the most popular multi-path routing techniques and how they relate to our study.

2.3.1 Middleboxes

On the Internet, there exist a multitude of devices that can affect or modify packet flows that we collectively refer as *middleboxes*. Middleboxes come in a variety of forms, such as load balancers, firewalls, NAT gateways, reverse proxies, and more:

Load Balancers Load balancing is the act of directing traffic to different endpoints to maintain an even load. A load balancer is the middlebox responsible for routing the individual packet. There are a number of different algorithms that a load balancer may use in its decision-making, and a number of different types of load balancers such as per-destination and per-flow load balancers. We will discuss load balancers in more detail shortly.

Performance Enhancing Proxies As detailed in [RFC3135], a Performance Enhancing Proxy (PEP) is a middlebox designed to improve network performance on links where the characteristics of the link, such as high latency and high loss rate, cause standard network protocols to suffer. PEPs may work at many different layers throughout the network stack, with various behavior depending on the layer it's at. For example, a layer7-aware PEP is able to modify the transmission-behavior on a per-application basis by removing unnecessary header information, thereby increasing performance.

Firewalls and Security Appliances Firewalls are ubiquitous in today's Internet, and with good reason. With ever-increasing threats from both state- and private-actors, it is more important than ever to securely protect internal infrastructure and the data-transmissions between internal and external sites. Firewalls come in many shapes, and may be used for things such as Transport Layer Security (TLS) termination, packet header inspection (and potentially modification), IPSec-tunneling and more.

NAT Gateways A Network Address Translation (NAT) gateway is a middlebox that performs NAT (Network Address Translation), allowing internal services such as a webserver to reachable from an external network. NAT is the act of replacing either the source address field, or the destination address field, or both, in a packet's header. NAT gateways are popular on cloud services such as Microsoft's Azure or Amazon's AWS, and are particularly often used in conjunction with IPv4 due to its limited address-space.

VPN Concentrators A Virtual Private Network (VPN) concentrator is an accumulator and endpoint for VPN-connections which allow a device to connect securely, appearing as if it is connected to the internal network. VPNs often differ by nature, behavior and their underlying protocols, varying from the usage of layer 3 protocols (IPsec) to higher-layer protocols such as SSL-VPN.

Voice Gateways A voice gateway is a network gateway for Voice over IP (VoIP) traffic, acting as an interface between traditional telephone traffic and traffic sent over the Internet.

These are just some of the multitudes of available middleboxes, but what they all have in common is the behaviour that packet data is either inspected or modified on the path to its destination, serving as an important reminder that the packet a sender transmits is often not the same packet received on the other end.

2.3.2 Load Balancing Algorithms

We will now go into load balancers in detail. We typically distinguish between three types of load balancing: *per-flow* load balancing, *per-packet* load balancing and *per-destination* load-balancing.

Per-flow load balancing In per-flow load balancing, the information in the packet header is used to assign each individual packet to a flow, often using a hash-based implementation. The router or load balancer will forward all packets belonging to a same flow out of the same interface, while also ensuring that packets from the same flow are delivered in order. The header fields used to determine which flow a packet belongs to vary depending on configuration and protocol.

Per-packet load balancing Per-packet load balancing is a technique that load balances on the individual packet-level, focusing only on maintaining an even load. In contrast to more sophisticated methods, such as per-flow load balancing, it makes no attempt to keep packets of the same flow together. A typical per-packet load balancing algorithm is the round-robin algorithm, which simply loops through a list of servers, delivering each arriving packet to the next server in the list. The consequence of this type of load balancing is that a sequence of packets belonging to an individual flow may be routed wildly differently, leading to fluctuations in latency, differing link MTU, and packet reordering (a well-known cause of performance issues in protocols such as TCP [RFC2992].)

Per-destination load balancing Per-destination load balancing will load balance packets based only on their destination IP-address, similar to standard routing. However, there is no concept of a *flow* since the source IP-address is disregarded. Per-destination load balancing is the most widely used type of load balancing, with a 2007 study [22] finding that 70% of paths traversed contain a per-destination load balancer, compared to 39% for per-flow per-flow load balancers and between 0 and 2% for per-packet load balancers.

2.3.3 Equal-Cost Multi-Path Routing

Equal-cost multi-path routing [RFC2992] (ECMP) is a load-balancing technique for forwarding packets along multiple paths of equal cost. Its main purpose is to increase bandwidth by utilizing otherwise unused bandwidth on links to the same destination. Historically, splitting flows unto multiple paths has been seen as bad due to variances in latency and packet-reordering. However, by routing all packets belonging to a single flow to the same path, one is able to increase link bandwidth while maintaining connection integrity.

Unequal-Cost Multi-Path routing

Unequal-cost multi-path [3] (UCMP), as the name suggests, is a load balancing technique for forwarding packets along multiple paths of unequal cost. While similar to ECMP is concept, it is not widely used due to variances in bandwidth and latency (hence the difference in cost) between each path, causing issues such as Head-of-Line (HoL) blocking and packet reordering, ultimately reducing bandwidth and creating an inconsistent experience for applications and the end-user.

2.3.4 Link Aggregation

Network Link Aggregation is a technique of combining multiple network links in order to increase the available bandwidth beyond what a single link can offer, while also providing redundancy in case one of the links should fail. A Link Aggregation Group [10] (LAG) is a collection of physical ports logically combined together. The ports may be combined at either layer 1 (the physical layer), the link layer (layer 2) or the network layer (layer 3). Similar to ECMP, LAGs will attempt to send all frames associated with a particular session across the same link in order to avoid frame reordering, typically by way of hashing header fields. It is important to highlight the differences between Link Aggregation and ECMP. While both LAG and ECMP balance load and increase resource utilization, they go about it in very different ways that have distinct consequences for downstream network devices. A LAG is a way of directly connecting to an adjacent router or switch, and while increasing the amount of data sent to the adjacent device, it will typically not have an effect on the network at large [RFC7424]. ECMP is different: by routing traffic to one path over the other, it may increase congestion on downstream routers several hops away from the offending load balancer.

2.3.5 MPLS Load Balancing

Multi-Protocol Label Switching [RFC3031] (MPLS) is a mechanism for forwarding traffic based on 20-bit labels placed between the link layer header and the layer 3 network header. Inside an MPLS network, forwarding decisions are made by examining only the MPLS label without the need to parse the layer-3 network layer header, speeding up forwarding processing. MPLS networks support load balancing in a manner similar to traditional IP networking. To maintain even distribution, MPLS-capable load balancers create a hash based on one or more MPLS labels combined with information in the outer layer of the payload (such as the IP header). Some vendors also support the inclusion of inner header data in cases where there are multiple chained protocols, such as Ethernet over MPLS. MPLS networks typically support load balancing over both ECMP and aggregated interfaces (LAG), similar to traditional networking. Since forwarding decisions inside an MPLS network are made exclusively by examining the MPLS label, it is also possible to perform a manually configured type of load balancing by choice of MPLS label assignment.

2.3.6 Network Flow Identification

A network flow, as defined by [RFC6437], is a sequence of packets originating from a single source to a particular unicast, anycast, or multicast destination. To identify the flow an individual packet belongs to, we inspect a set of values from the packet's header fields and match it to a particular flow. While values at either layer 2, 3 or 4 can be used, an IPv4-flow is typically

identified by either the (source IP-address, destination IP-address) 2-tuple or, more commonly, the 5-tuple consisting of the (source- and destination IP-addresses, source- and destination-ports, and the transport layer protocol identifier). IPv6 differs by introducing a new header field, the flow label, which can be used alongside the other fields as an additional identifier.

Figures 2.6 to 2.9 illustrate the layer-3 and -4 header fields traditionally used for flow identification by load balancers:

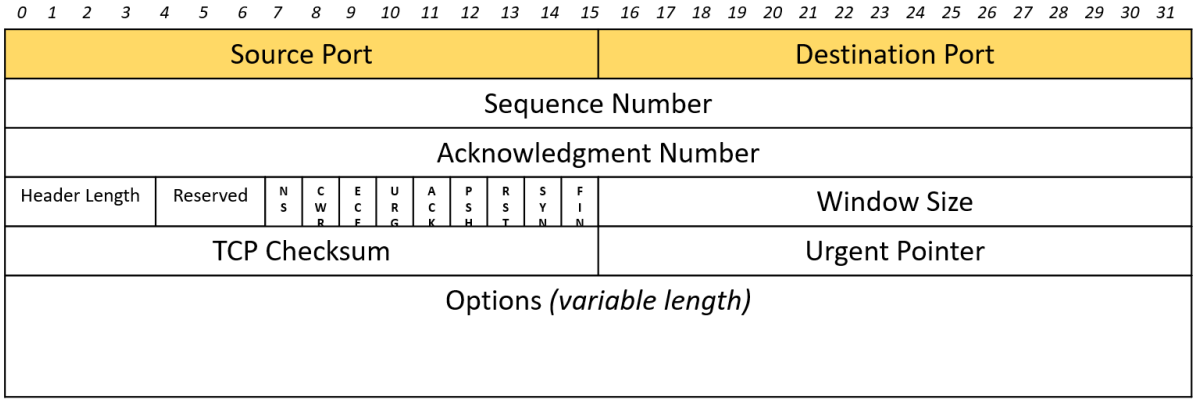


Figure 2.6: TCP header fields traditionally used for load balancing.

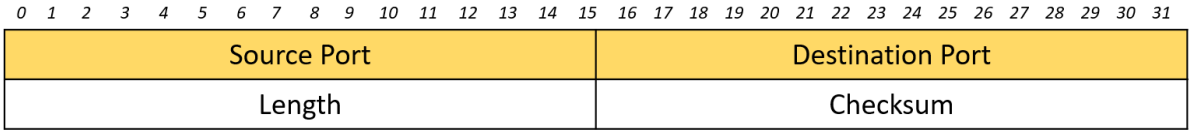


Figure 2.7: UDP header fields traditionally used for load balancing.

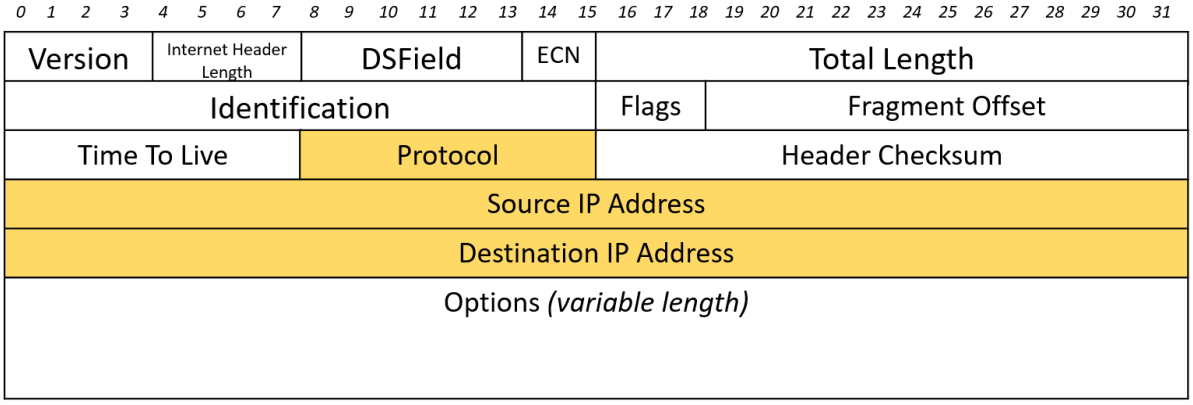


Figure 2.8: IPv4 header fields traditionally used for load balancing.

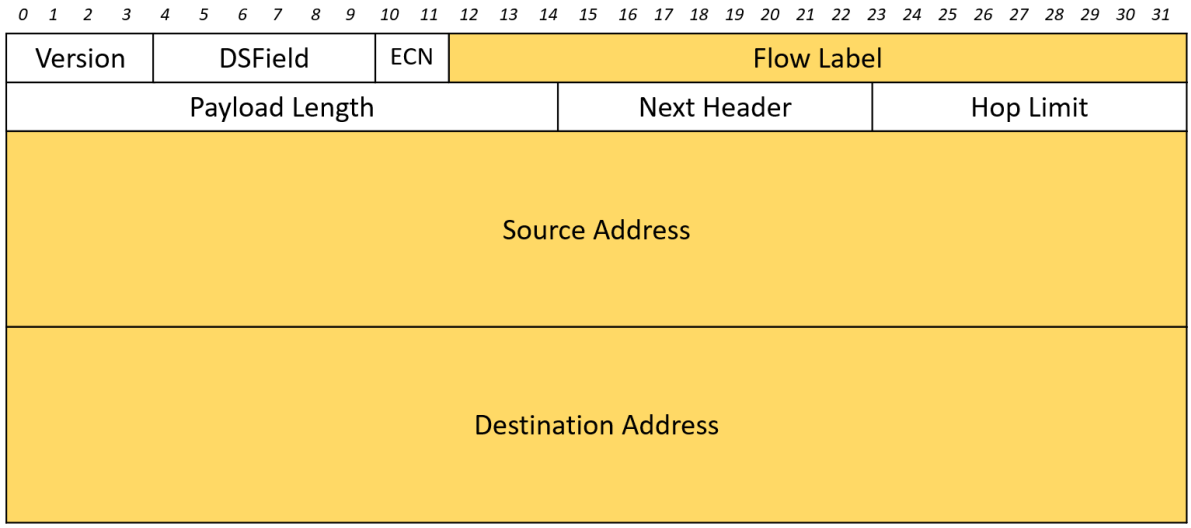


Figure 2.9: IPv6 header fields traditionally used for load balancing.

2.3.7 The IPv6 Flow Label

The traditional 5-tuple comes with several drawbacks. Since identifying flows by the 5-tuple requires inspecting both the layer-3 header and the layer-4 header, if one of the headers are unavailable or modified due to encryption or packet fragmentation, the flow identification will fail. For IPv6 this also holds true, but the scenario is worse: due to the potential chain of IPv6 extension-headers one would have to traverse to locate the requisite transport-layer header fields, it is highly inefficient compared to IPv4. Another drawback is that by including both layer-3 and layer-4 information in flow identification algorithms, the implementation locks itself to the current transport-layer protocols. This bars the way for any future protocols, barring any software update.

It is for this reason that the designers of IPv6 sought to improve the situation with the introduction of the flow label. The flow label field is a value located in the main IPv6-header designed to simplify flow identification while at the same time making it more efficient. The intent behind the flow label is for it to be used to classify and identify a specific flow of packets between a source and destination without the need for a router or load balancer to inspect layer 4 and above header fields [RFC3697].

The flow label itself is a 20-bit unsigned integer where a flow label of zero means that the packet does not belong to any flow. Source nodes can assign each unrelated transport connection and application data stream with a new flow label value, which will enable Flow-Label-based classification.

The flow label is designed to be used in both stateless and stateful scenarios. In the stateless scenario, any node processing the flow label does not need to store any information about the flow, either before or after a packet has been processed. In a stateful scenario, a node that processes the flow label needs to store information about the flow, including the flow label value.

Flow Label Mutability

The IPv6 Flow Label Specification [RFC3697] states that routers or other middleboxes should *not* modify the flow label value in-transit. However, important to highlight that the Flow Label is not a protected value, meaning it is not encrypted or protected via other means such as

checksums. In practice, this means that setting the flow label is done on a "best effort"-basis and it is possible that the flow label may be accidentally or intentionally changed en-route. The end result is that typically, routers and load balancers do not use the flow label alone to classify a flow, but the 3-tuple of consisting of the source and destination addresses and the flow label is instead used.

Flow Label Restrictions

The flow label has some restrictions on use and, in particular, reuse. We quote the Flow Label Specification [RFC3697]: “A source node MUST ensure that it does not unintentionally reuse Flow Label values it is currently using or has recently used when creating new flows.” It goes on: “Flow Label values previously used with a specific pair of source and destination addresses MUST NOT be assigned to new flows with the same address pair within 120 seconds of the termination of the previous flow.” These restrictions are in place to ensure that two unique flows originating from the same destination get correctly classified as distinct.

The authors in [RFC6438] further describe restrictions when using the flow label for ECMP-routing and link-aggregation hash calculations. We identify the following recommendations from the RFC:

- The hash algorithm used to determine the outgoing link for a particular packet must minimally include the 3-tuple (dest addr, source addr, flow label).
- Because the flow label may be set to zero, the hash algorithm should also include the input keys (protocol, dest port, source port) to provide additional entropy.

IP-in-IP Tunneling And Flow Identifier Preservation

An IP-in-IP tunnel is a logical link between two routers that makes them appear as if they are directly connected[RFC2473]. Tunneling can be used to carry IPv6-traffic over IPv4-networks, or vice versa, by wrapping traffic forwarded through the tunnel in an outer IPv4- or IPv6-packet. If we have a tunnel between two endpoints, A and B, the same outer address-pair and port-pair may be used for all traffic between the endpoints, giving them the same flow-identifier[RFC6438]. The default flow label value assigned to outer IPv6-headers is zero according to the Generic Packet Tunneling in IPv6 specification[RFC2473], however some implementations[5] include features such as flow label inheritance, where the outermost flow labeled will mirror the inner flow label. Most hardware can only identify flows based on information in the outermost header[RFC6438], making inheritance an absolute necessity for flow identifier preservation in tunneled traffic. A tunnel itself may encapsulate another tunneled packet, up to a limit set by maximum IPv6 packet size.

2.4 Internet Measurement Studies With Traceroute

2.4.1 Traceroute Overview

Traceroute is a tool which uses ICMP *Time Exceeded* messages to deduce the route a message takes between two internet hosts. Invented by Van Jacobson in the late 1980’s, there now exist several varieties and derivatives of traceroute, some of which will be discussed later. However, we will first give an overview of its general probing mechanism as first performed by Jacobson’s original traceroute [53].

A typical invocation of traceroute involves three parameters: destination address (either IPv4 or IPv6), protocol (UDP, ICMP or TCP) and the number of probes sent per hop (normally between 1 and 3). The destination address may be any valid IP-address. The default protocol is typically UDP, though ICMP is also common. Due to its prevalence in today's Internet, TCP has been seeing more usage in recent years with variants such as Toren's tcptraceroute [83], which is able to more easily bypass firewalls by using TCP SYN packets, as opposed to UDP or ICMP probes which are often blocked by network administrators to mitigate against DDoS attacks. Traceroute works by cleverly utilizing the Time-to-Live header field in IPv4, or the Hop Limit header field in IPv6. The Time-to-Live field was originally conceived to hold the maximum number of seconds an IP-datagram may be allowed to remain active in the network before being discarded in an effort deal with undeliverable packets stuck in a forwarding loop. As routers were required by the standard [RFC1812] to decrement the TTL field by one on each hop and datagram forwarding times were often less than a fraction of a second, in practice the field became an upper bound on the number of hops a datagram could traverse before being discarded by the network [41]. Later, the IPv6 Hop Limit field was formalized with this precise definition [RFC2460].

As described in section 2.1.4, a router will generate an ICMP *Time Exceeded*-message when a datagram gets discarded due to the TTL-field or Hop Limit being reduced to zero. Upon returning to the source, the ICMP *Time Exceeded* message parsed to extract the IP-address of the originating router. By continually incrementing the TTL of outgoing packets by one and recording the address in the returning *Time Exceeded*-messages, one may effectively build a graph of hops encountered on the path to a destination. Some routers optionally provide DNS info as well, which can be used to identify network and AS boundaries.

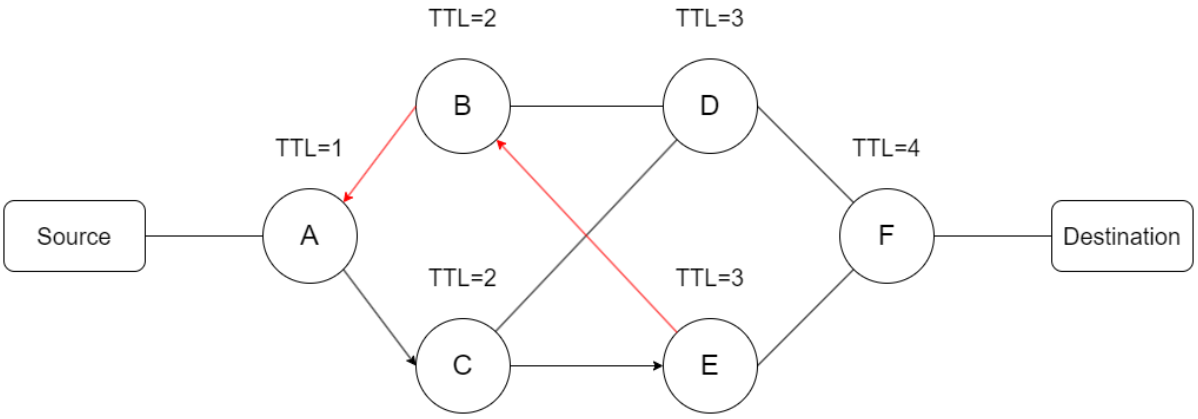


Figure 2.10: In this example, an ICMP Echo is sent with TTL=3 and the responding ICMP Time Exceeded-message, here marked in red, is sent back to Source via an asymmetric return path. However, the source IP-address in the ICMP Time Exceeded-message will still be that of the ingress interface of router E.

Latency calculation

Typical traceroute applications also include the latency measurement (RTT) of each successful probe. In [82], the author describes the latency measurement as the sum of three components:

1. The time it takes to forward a packet to the individual hop.
2. The time it takes for the hop to generate an ICMP *Time Exceeded*-response.
3. The time it takes for the ICMP-packet to reach the sender.

Since the return path may be asymmetric and ICMP generation time can vary greatly depending on load, latency measurements are often inconsistent and sudden spikes in latency are common.

Matching return packets

Due to variations in network latency and asynchronous parallel probing, a hop with a higher TTL may respond before the hop with a lower TTL. In order for the traceroute application to match outgoing probes with the returning ICMP *Time Exceeded*-packets, it needs to designate a unique probe identifier per outgoing probe. Traceroute traditionally does this by modifying the first 8 octets of the transport-layer header. For TCP- and UDP-probes, it varies the source port number. For ICMP *Echo*-packets, it increments the Sequence Number. If a router silently drops the traceroute probe, or the ICMP *Time Exceeded*-packet was lost somewhere along the path back to the traceroute originator, traceroute will output a star after a certain amount of time has elapsed, indicating that the probe received no response.

2.4.2 Traceroute and MPLS

Many of today's networks have an MPLS core. The nature of label switched networks present some unique challenges and side effects for traceroute which we will go into more detail shortly, but first we need to introduce the MPLS header.

The MPLS header

The MPLS header consists of a 20-bit label, followed by a Traffic Class field used for QoS. This is then followed by a single bit indicating whether this MPLS header is at the bottom of the stack (several MPLS labels may be chained in a stack-like-structure). Finally, there is a TTL-field which works in identical ways to the equivalent fields in the IPv4- or IPv6-header, meaning it is decremented by one on encountering a new hop.

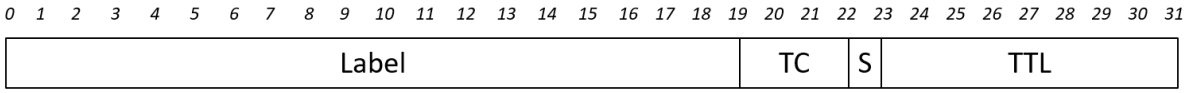


Figure 2.11: The MPLS header.

TTL Inheritance

MPLS supports two ways of setting the MPLS header TTL value: it can assign a new value (such as 255) or it can inherit the value from the inner IP-header. Traceroute traces performed over a non-inherited TTL MPLS network may look strange, as displayed in figure 2.12. In this example, the MPLS label TTL is set to 255 by the label edge router (LER) A. Since the TTL on router B is still above zero, the packet is forwarded as normal until reaching router B on the other edge of the MPLS core. Upon reaching B the IP TTL is decremented to zero and an ICMP *Time-Exceeded* is generated. In effect, a non-inherited TTL means that the core LSRs are hidden from view of the traceroute user.

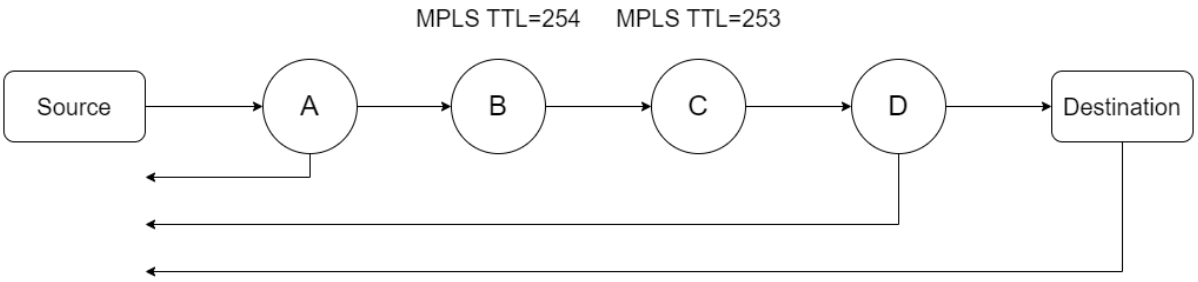


Figure 2.12: Label edge router A assigns a TTL value of 255 to forwarded MPLS messages.

ICMP Tunneling

MPLS devices inside the core, typically referred to as label switched routers (LSRs), often do not have an IP routing table, instead relying entirely on the label forwarding table. A question thus arises: How does an LSR convey ICMP control information back to source hosts when it receives an undeliverable message? The procedure, as demonstrated in figure 2.13, is as follows: The LSRs B and C will, upon encountering an undeliverable message, strip the MPLS stack and submit the exposed datagram to an error processing module [RFC4950]. The error module will then, depending on the error type, generate an appropriate ICMP-message and add to it the previously stripped MPLS stack from the original message. The message will then be forwarded to the appropriate LER. Since label switched networks are unidirectional, LER D is the only possible destination. Upon receiving the message, D will replace the MPLS label in the datagram with a new MPLS label associated with router A and send it back through the MPLS core. Once the datagram reaches router A, it will strip all MPLS labels and forward it to the appropriate destination based on information in the IP header (the red arrow in the figure). This process is called *ICMP Tunneling*.

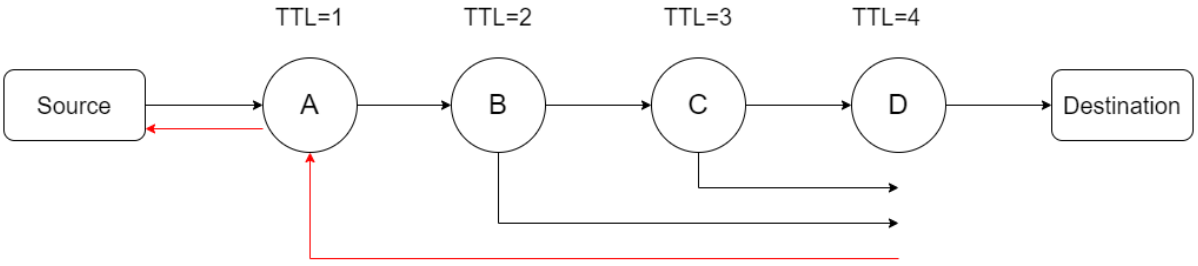


Figure 2.13: ICMP-messages from router B and C get forwarded to router D before being returned to Source via router A.

2.4.3 Traceroute Variants

prtraceroute

PRtraceroute [14] is a variant of Van Jacobson's original traceroute that includes the ability to perform ASN lookups on hops encountered along the path. It does this by using data provided by the Internet Routing Registery (IRR). It also records each hop encountered along the path to optionally perform routing policy analysis on the taken path by comparing it against information in the IRR routing database (RDB). The IRR RDB is a registry of routing policies used

for peering between ASes, and by comparing the information using PRtraceroute one is able to verify the routes registered in the RDB against the actual, in-use policies.

traceroute-nanog

Gavron's traceroute-nanog [44] is, like PRtraceroute, based on Van Jacobson's original traceroute, but includes additional features such as the ability to change the IPv4 Type-of-Service (ToS) field, parallel probing, DNS lookup, ASN lookup using WHOIS data, path MTU discovery using the process described in [RFC1191], and microsecond time-stamps.

Paris Traceroute

When probing with the traditional version of traceroute, a packet may take one of several available paths to each destination due to ECMP, per-packet load-balancing and/or transient routing changes. Figures 2.14 and 2.15 demonstrate the problem:

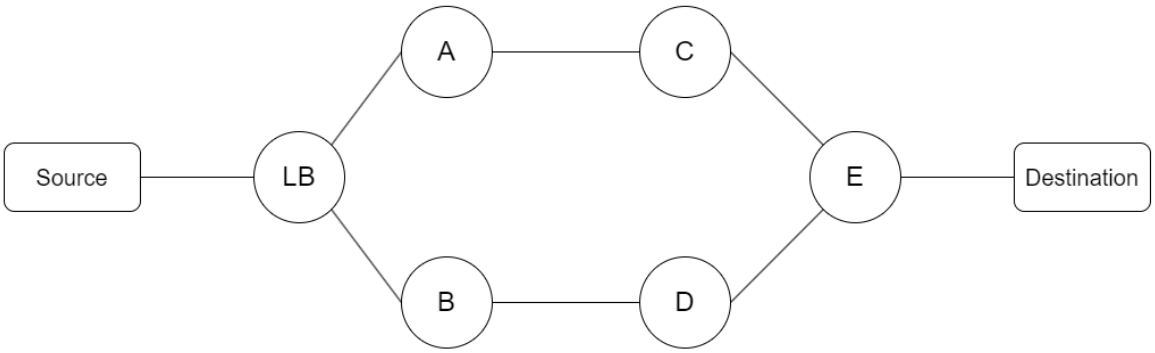


Figure 2.14: Example network topology between a Source and Destination, consisting of a Load Balancer LB and intermediary nodes A, B, C, D and E.

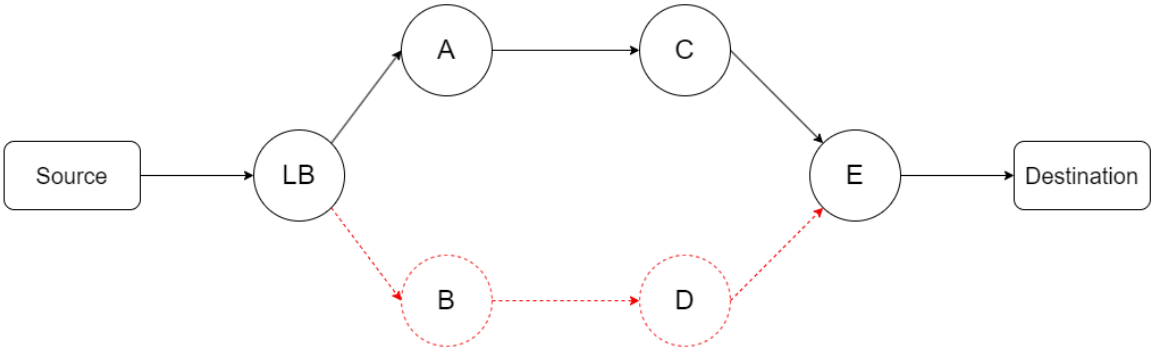


Figure 2.15: In this example, connections LB->B, B->D and D->E are left undiscovered by traditional traceroute.

In addition, if the load balancers favor one path over another it could lead to missed nodes or false links. Figures 2.16 and 2.17 demonstrate an example topology and false links reported by traceroute.

Paris traceroute [21], so named after its place of conception, is an open-source, path-aware variant of traceroute that is able to keep a constant flow identifier for all probes, allowing them

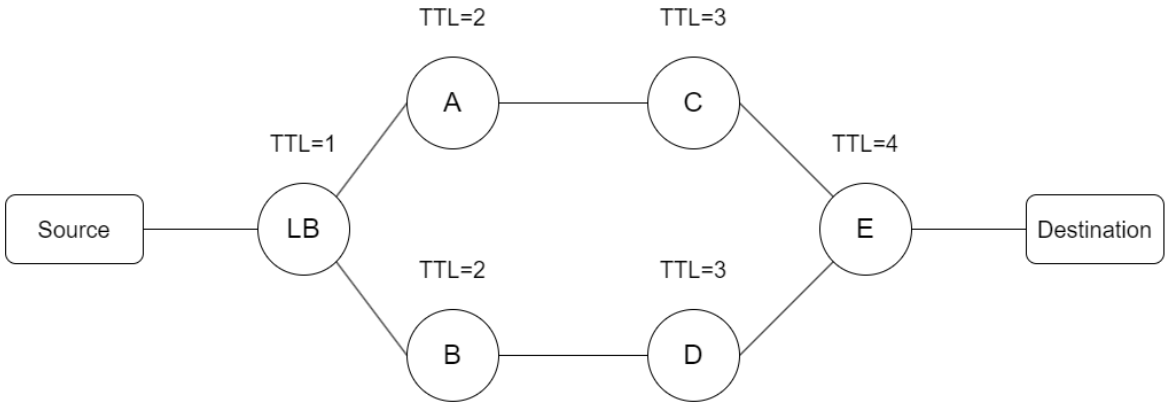


Figure 2.16: Example network topology between a Source and Destination, consisting of a Load Balancer LB and intermediary nodes A, B, C, D and E.

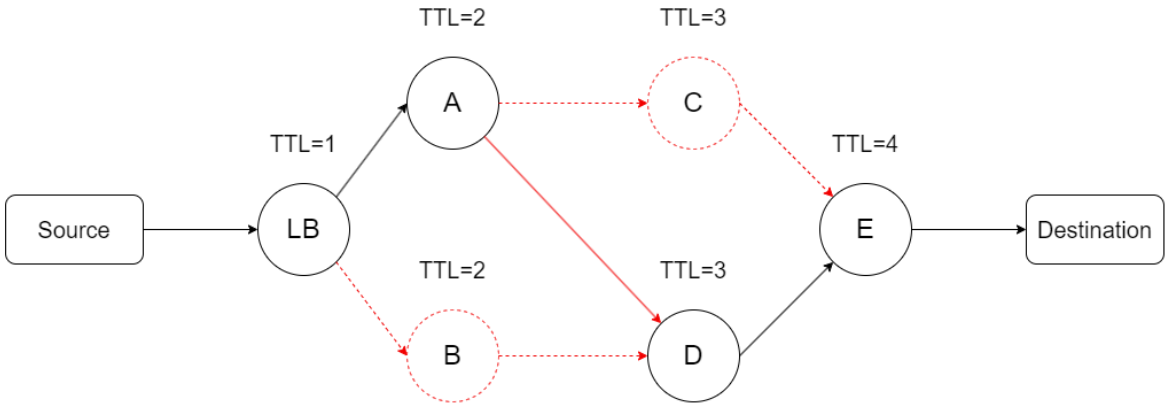


Figure 2.17: In this example, the false link A->D is reported by traceroute.

to take the same path even in the face of per-flow load balancing. It does this by keeping constant the header fields that are typically used for load balancing, such as the TCP source and destination port fields. In addition, it implements a multipath detection algorithm (MDA) that is able to enumerate all paths between a source and a destination and report whether that path is behind a per-packet or per-flow load balancer.

Header field manipulation

The major innovation of Paris traceroute is to manipulate the first eight octets of the transport-layer header fields in such a manner that the flow identifier remains constant over several probe packets. As mentioned in section 2.3.6, a typical flow identifier for IPv4 is the 5-tuple consisting of (Source IP, Destination IP, Source Port, Destination Port and Protocol). However, the authors of [21] have found that certain additional fields are being used for load balancing:

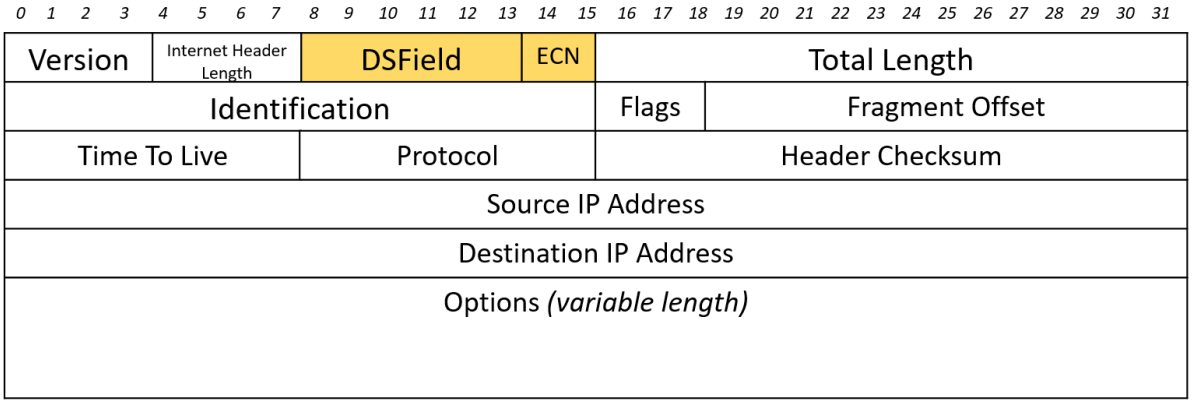


Figure 2.18: Additional IPv4-header fields used for load balancing.

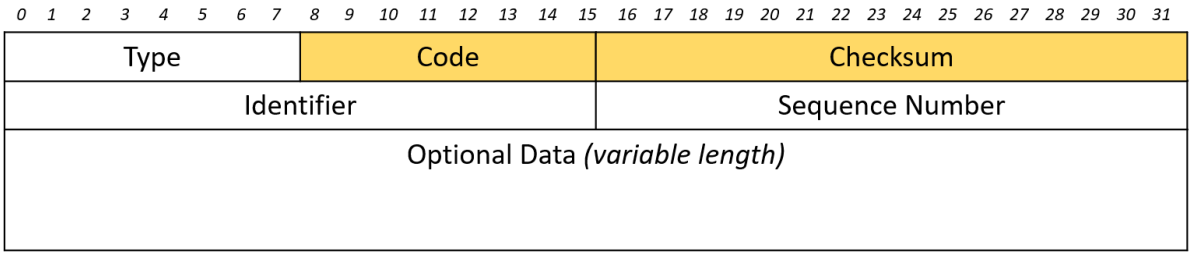


Figure 2.19: Additional ICMP Echo header fields used for load balancing.

As illustrated in figures 2.19 and 2.18, in addition to the classic 5-tuple used for flow identification, three more fields are used by load balancers to distinguish network flows. These are the IPv4 Type-of-Service (DSField/ECN) fields and the ICMPv4 Code and Checksum fields [21]. The Paris traceroute application has been carefully designed with these in mind and is able to avoid doing any modification these header fields during its execution.

Matching returning probe packets

Since it is stateless, traceroute needs a way to match its outgoing probes with the correct returning ICMP *Time Exceeded*-messages. As mentioned in section 2.4.1, traceroute traditionally does this by varying the source port-number. However, since varying port-numbers will change the flow identifier, Paris traceroute instead carefully modifies different values that do not affect the flow identifier. For ICMP Echo probes, it modifies the Sequence Number and the Identifier fields in such a way that the Checksum field remains constant. For TCP, it modifies the sequence number, and for UDP, it modifies the Checksum indirectly by manipulating the packet payload. This is necessary to ensure that the UDP-packet is not discarded by the receiver due to a mismatched checksum.

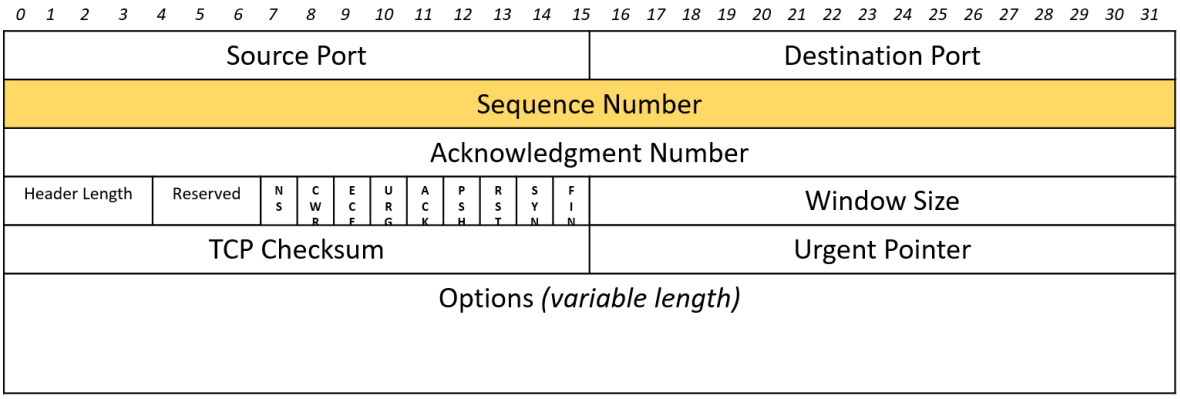


Figure 2.20: TCP header fields modified by Paris traceroute to match returning probe packets.

Dublin Traceroute

Dublin traceroute [25], developed by Andrea Barberio and named for the city where he lives and works, is a NAT-aware multipath variant of traceroute that builds on Paris traceroute. It introduces a new method for NAT detection that uses the IPv4 *Identification* header field in outgoing probes to deduce the presence of NAT. The method works by setting the *don't-fragment* flag in the IP header and giving each outgoing probe a unique IPv4 *Identification* value which is then stored and tracked. By inspecting the *Identification*-fields of the embedded packets in the returning ICMP *Time-Exceeded*-messages, and comparing them against the values set in the outgoing probes, one can matching the outgoing probes and verify whether the IP-address has changed en-route, and is thus behind a NAT. The presence of the *don't-fragment* flag ensures that the *Identification* field remains unchanged throughout the network path. In addition, Dublin traceroute builds on the techniques invented by the authors of Paris traceroute to implement ECMP flow-based path enumeration.

Scamper

Scamper [62] is an open-source traceroute tool and packet-prober developed by Matthew Luckie. Developed as a successor to Skitter (a tool used by CAIDA and other researchers for Internet-wide probing and topology- and performance-analysis), it supports IPv4 and IPv6 packet-probing and includes useful features such as ping-probing, traceroute data collection, path comparison, the ability to set the packet-per-second rate, and support for Paris traceroute's multipath detection algorithm. It can execute probes in parallel and supports both simple and more complex measurements that are implemented with the help of custom driver addons that can be installed separately to handle the more complex logic. Scamper is widely used in Internet-wide Measurement projects. For example, it is currently deployed in CAIDA's Macroscopic Topology Project [17], an ongoing project since 1994 that collects connectivity and latency data across the Internet to derive maps of the Internet at the IP, router or AS granularity.

2.4.4 Related Work

Internet measurement studies using traceroute have been performed for more than two decades, and is to this day an active area of research where organizations such as CAIDA operate a vast set of measurement nodes publicly available for use by interested researchers. We will now

dive into some previous measurement studies and provide a brief summary of their methodology.

On the Utility of Unregulated IP DiffServ Code Point (DSCP) Usage by End Systems

The Differentiated Services Code Point (DSCP) is a 6-bit field in the IPv4- and IPv6-headers designed to provide Quality of Service (QoS) capabilities to the Internet. Originally defined in [RFC791] as a Type of Service (ToS) identifier, it was later superseded by [RFC2474] which introduces a number of *code points* used by DSCP-capable routers to determine forwarding behaviour.

In [26], the authors use the *fling* measurement tool to investigate whether the DSCP code point is changed or zeroed out along the path to its destination, and in particular whether it remains intact across AS boundaries. The motivation behind this paper is a proposed method for WebRTC [38] using the DSCP to specify packet forwarding behaviour. In addition to measuring survivability, the authors run a performance measurement study in an artificially congested environment to evaluate the potential benefits of using a non-zero DSCP value.

The authors utilize *fling's* built-in capabilities to set the DSCP value and send outgoing probes with varying levels of TTL from a combination of IPv4- and 1Pv6-capable vantage points hosted on CAIDA's Ark platform, PlanetLab, NorNet Core and Amazon Cloud. By parsing the returning IP-packets embedded in the ICMP *Time Exceeded*-payload, they confirm whether the DSCP has changed en-route. DSCP code points *CS1*, *AF42*, and *EF*, were selected as the outgoing values as they are seen as the most important to WebRTC [38].

In the end, the resulting measurement campaign found that there is a small, but measureable reduction in latency when using a non-zero DSCP. As for the DSCP value itself, the findings suggest that the value is often interfered with by middleboxes, where its value only managed to survive the initial egress AS hop 70% and 80% of the time for IPv4 and IPv6, respectively.

Inferring Persistent Interdomain Congestion

In Inferring Persistent Interdomain Congestion [37], the authors devise a method to measure congestion on interdomain links without direct access. Luckie et al.'s bdrmap tool [65] was used to get an overview of which links to probe, and Time Series Latency Probing [64] (TSLP) was utilized to identify links with evidence of recurring congestion by analyzing the time series on a week-by-week basis and highlighting episodes of elevated congestion. In addition, by investigating repeating patterns separated by 24-hour intervals, they are able to see if congestion is repeatedly induced by periodic demand.

Figure 2.21 illustrates the principle behind TLSP. Because control-plane packets sent directly to routers are treated differently than data-plane traffic, the probes are instead sent with increasing TTL-values to a destination *behind* the border routers.

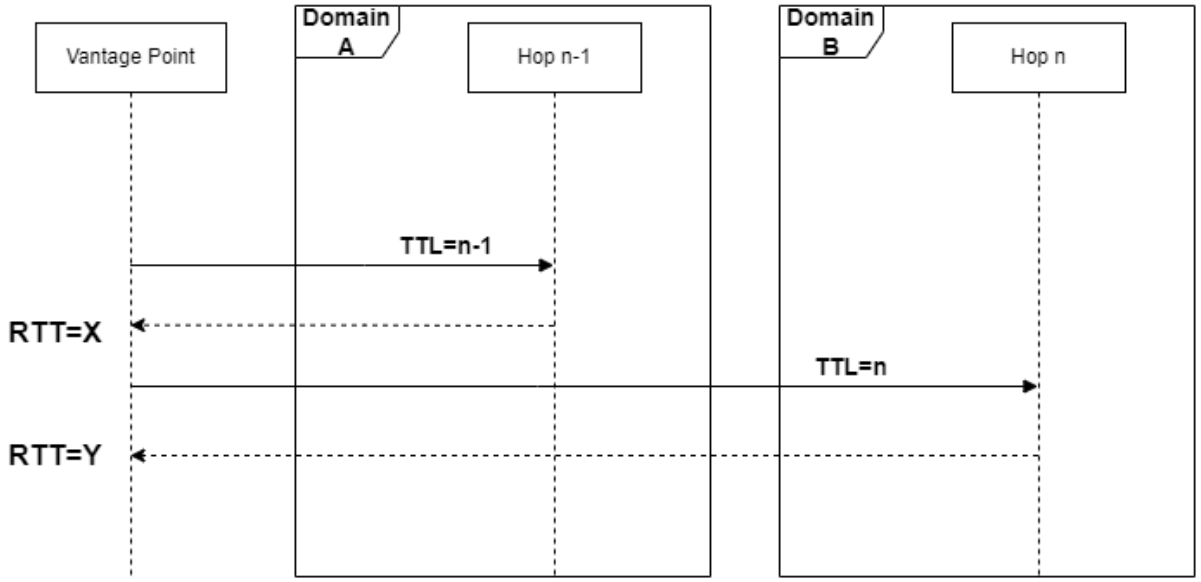


Figure 2.21: Time series latency probing illustrated. By sending TTL-limited packets from the VP that expire at network border routers A and B and then comparing the difference in latency between each response, one can measure the propagation delay between two nodes.

In addition, the authors perform packet loss and TCP throughput measurements by using the Network Diagnostic Tool (NDT) running on an M-lab server and collect measurements from 86 vantage points worldwide, using 47 ISPs in 24 countries. In the end, the resulting measurement study did not produce evidence of widespread congestion.

MAP-IT: Multipass Accurate Passive Inferences from Traceroute

Multipass Accurate Passive Inferences from Traceroute (MAP-IT) [66] describes a new graph refinement algorithm for inferring the IP addresses used for directly connected inter-AS links. The primary problem with IP-to-AS mapping of inter-AS link interfaces occurs due to address-space assignment. As link-interfaces are assigned IP-addresses from the same network prefix (typically the `/30` or `/31` prefix for IPv4), one of the interfaces will map to the incorrect AS, since the prefix is designated to only one AS. As a result, inferring inter-AS links from a single traceroute trace is difficult.

Instead, the authors of [66] present a new algorithm using data from multiple traceroutes obtained via Caida's IPv4 Routed Topology Dataset [2]. The algorithm works by utilizing IP-to-AS mappings of hop-addresses observed *before* and *after* a given hop. By creating a graph from traceroute data and executing multiple passes through the graph, the confidence level of each inference being correct is improved. Since traceroute data may contain false adjacencies due to ECMP load balancing or buggy routers forwarding packets with TTL=1, the data is sanitized before analysis.

The results of the algorithm are promising: when comparing against topology data published by Internet2 [70], the inferences from MAP-IT achieved 100% correctness. When comparing against topology data derived from DNS-hostnames in ISPs Level 3's and TeliaSonera's address space, they achieved a correctness of 95%.

Chapter 3

Experiment Design

In this chapter we introduce the experiment design for the main contribution of this thesis: an extensive measurement study to show conclusively if a) the flow label is carried completely to its destination and b) if load balancers and ECMP routers consistently use the IPv6 3-tuple for flow identification. While Augustin et al. [21] found that the IPv4 flow identifier used by conventional load balancers is a combination of both layer-3 and layer-4 header fields, we are not aware of a similar measurement study performed for IPv6. As discussed in section 2.3.7, the IPv6 flow label header field brings with it major changes to packet flow identifier designation. And, while not the intended usage for the flow identifier, if we can find that the IPv6 flow identifier is exclusively layer-3 3-tuple based, it can be used to assume equal the paths of two separate IP connections between the same source and destination, ultimately leading to advances for Islam et. al.'s coupled TCP congestion control mechanism [51].

3.1 Measuring Flow Label Persistence Across A Network Path

Is the flow label completely preserved over a path to a destination? The specification in [RFC6438] indicates yes. We quote the RFC: "The Flow Label value set by the source MUST be delivered unchanged to the destination node(s)". However, the answer might not be so straight-forward. As discussed in chapter 2.3.6, the flow label is an unprotected field that may be modified at any time, either accidentally or intentionally, en-route. In addition, the IPv6 Flow Label Specification [RFC6437] *does* allow for the flow label to be changed in specific circumstances as a defense against covert channel attacks. We quote the specification:

"In situations where the covert channel risk is considered significant, the only certain defense is for a firewall to rewrite non-zero flow labels. This would be an exceptional violation of the rule that the flow label, once set to a non-zero value, must not be changed. To preserve load distribution capability, such a firewall SHOULD rewrite labels by following the method described for a forwarding node, as if the incoming label value were zero, and MUST NOT set non-zero flow labels to zero."

In the light of this, we identify the following rules for flow label value persistence:

- A non-zero flow label must not be changed, with only a security exception related to using it for a covert channel. The flow label CAN be changed to a different value, but NOT to zero.
- A zero flow label is OK to be changed to a non-zero value.

- A non-zero flow label may not be changed to zero.

The questions thus arise: is the flow label preserved along a network path? How far along a path does the source flow label value persist before it is potentially modified? And if it is modified, what are the characteristics of the middlebox or host performing the modification?

3.1.1 Methodology

Using traceroute, we visit every node along a path and inspect the invoking packet embedded in the returning ICMPv6 *Time Exceeded*-message. We compare the flow label in the embedded packet against the flow label assigned to the invoking packet to detect if a flow label change has occurred. To deduce which router along the path caused the flow label change, we utilize the TTL. We make the following assumptions:

- Flow label changes occur only on data-plane traffic forwarding.
- If a packet with a TTL of one reaches a router that would have changed the flow label, the TTL is decremented and the packet dropped before any flow label change occurs. In this instance, no flow label change will be detected.
- If a packet is forwarded by a router that causes flow label changes, the flow label change will be first detected by inspecting the returning ICMPv6 *Time Exceeded*-message from the next hop.

Figure 3.1 provides a visualization of the above assumptions.

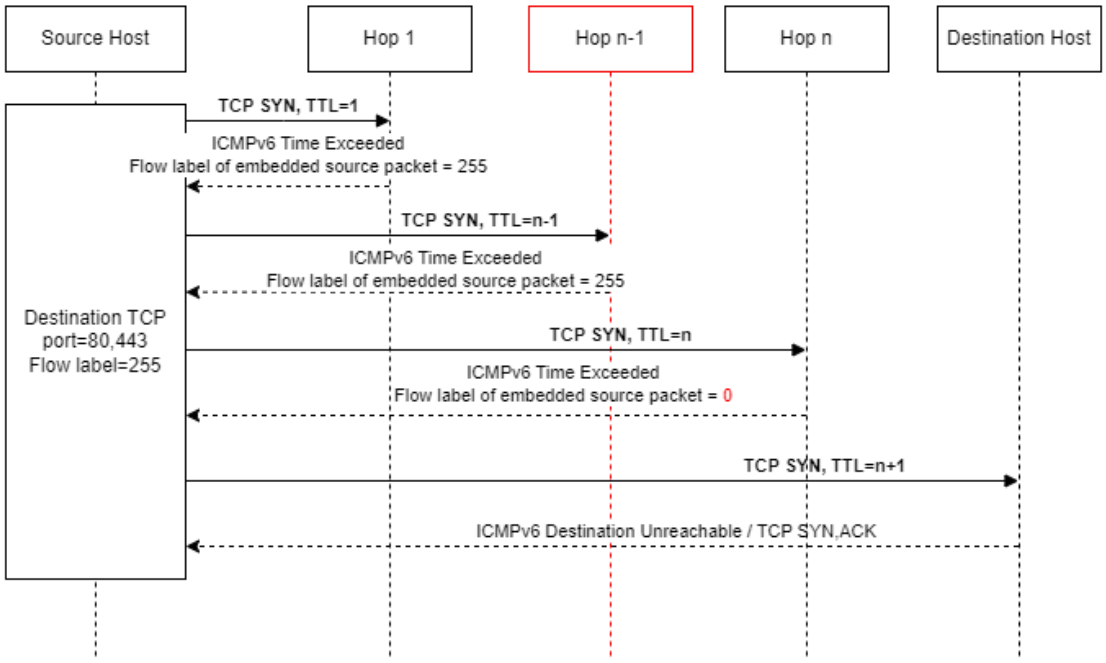


Figure 3.1: In this example, hop n-1 sets the flow label on data plane traffic to zero. The results are observed in the embedded invoking packet received from hop n.

A consequence of these assumptions is that if a flow label change occurred at hop N in the path, and the previous router at hop N-1 is non-responsive (either due to the return ICMPv6

Message getting lost along the path or due to a deliberate choice of configuration), we will not be able to identify the router causing the flow label change as it is ambiguous; the offender could be either at hop N-1 or hop N-2.

3.1.2 Choice of Flow Label Values

We took great care in selecting the outgoing flow label values. From a feasibility standpoint it is neither practical nor time efficient to do a brute-force test of *all* possible 2^{20} flow label values. We must also not choose too few, as this could bear the risk of missing out on edge-cases and losing potential crucial data. As shown in section 2.1, the flow label field spans 3 bytes of the header: two in full, and half of the highest-order one. We theorize that some systems, either at the hardware or software level, may parse only part of the flow label for use in their hashing algorithms which are used to assign a flow to a particular bucket.

Table 3.1 illustrates our chosen flow label values.

Table 3.1: Flow label values in decimal, hex, and binary formats.

Flow Label Value (decimal)	Flow Label Value (hexadecimal)	Flow Label Value (binary)
0	0x0	0000 00000000 00000000
255	0xFF	0000 00000000 11111111
65280	0xFF00	0000 11111111 00000000
983040	0xF0000	1111 00000000 00000000
1048575	0xFFFFF	1111 11111111 11111111

The rationale for each flow label value is as follows: Flow label 0 was chosen as the base case, representative of a packet belonging to no flow. This will be the primary measurement result that we will be comparing the results of measurements with other flow labels against. Flow label 0xFF was chosen to fully cover the least significant byte of the header field. In the event that some hardware considers nibbles (4-bit blocks), the lowest nibble will be covered as well. Flow label 0xFF00 covers the next full byte in the header, while leaving the least significant byte zeroed. An argument could be made to OR this with our next chosen header, 0xF0000, as they together partly constitute the most significant 8 bits of the header. However, to increase our test coverage and to properly cover the nibble-case, we include this as a separate value. Finally, the all-ones flow label, 0xFFFFF, is useful to rule out any type of partial ignorance that could arise from our previous measurements.

3.1.3 Target IP Address List

We execute traceroute against select IPv6 target addresses, each located in a unique Autonomous System (AS). The target addresses are grouped together in a list, commonly known as a hitlist. In creating the target hitlist we drafted the following requirements:

ASN variety The aim of our measurement campaign is to get a diverse set of data, visiting as many unique routers/middleboxes as possible. In a scenario where one randomly targets a range of addresses, there is a risk that all targets are a part of the same network, meaning that all packets would be routed along the same path, repeatedly sampling the same routers. This would effectively yield the same results as a target hitlist size of one.

Reachability As noted in [21] and [85], traces towards unused IP addresses may artificially inflate traceroute anomalies such as loops and cycles. As such, we impose the requirement that the hitlist must consist of only reachable, "alive", IP addresses available on the public Internet.

Number of target addresses The IPv6 address-space is massive, consisting of 2^{128} addresses. It is clearly not feasible to execute probing against the entire address-space, not only due to the time constraints, but such a strategy would also lead to considerable overlap as discussed earlier. At the same time, the hitlist can also not be too short as the sample size would be too low to draw a valid conclusion. We keep the hitlist size limited in order to keep the execution time of our measurement campaign at a reasonable length.

Vantage Point variety In order to reduce traceroute sampling bias[61], we require a variety of vantage point locations.

Using BGP Route Data For Internet Measurement Studies

The Border Gateway Protocol [RFC4271] (BGP) is the de-facto standard routing protocol in use today for exchanging network reachability data between Autonomous Systems (ASes). While it is also used to exchange network reachability data between BGP-speakers inside an AS (called iBGP, or internal-BGP), its most prevalent function is to exchange network reachability information between BGP nodes located in different ASes. In order to facilitate this, BGP-speaking systems connect to each other with long-lasting TCP sessions and exchange *BGP UPDATE* messages that contain network reachability information along with a list of ASes that were traversed between the source node and the receiving BGP-speaking node.

The BGP Routing Information Base

Each node's BGP system stores all of its routing data in a data structure called the Routing Information Base (RIB). The RIB is divided into three sections: the Adj-RIBs-In, the Loc-RIB, and the Adj-RIBs-Out. The Adj-RIBs-In-section stores BGP routing information learned from inbound UPDATE messages received from other BGP peers [RFC4271], the Loc-RIB-section stores the valid local routes used by the BGP speaker, and the Adj-RIBs-Out-section stores the routing information that will be advertised to fellow BGP peers.

Sources of Aggregated Routing Data

There are several publicly available sources of aggregated routing data, such as the RIPE Routing Information Service [13], CAIDA's Archipelago (Ark) Measurement Infrastructure IPv6 Topology Dataset [12], or the University of Oregon's RouteViews Project [15]. Among these, RouteViews is one of the oldest and best known, and its data has been used in several previous measurement studies such as [66]. Full details on how the RouteViews project collects their data can be obtained by going to their website, but this thesis will give a short introduction to the project and how the dataset was produced.

The RouteViews Project

The University of Oregon's RouteViews project is a route data collection project first conceived as a tool for Internet operators to obtain real-time information about the global routing

system [1]. The project's goal is to provide real-time access to BGP route data, and the project currently includes archived data going as far back as 1997. The heart of the project consists of a number of RouteViews Collectors stationed around the globe. A RouteViews Collector [7] is a router configured using BGP multihop or singlehop to peer with Internet backbones and other ASes at interesting locations that continuously collects and stores BGP route data on the form of BGP RIBs and BGP UPDATEs. An individual researcher may freely log on to a RouteViews Collector via Telnet and view the BGP RIB in real-time. A snapshot is also taken of each Collector's RIB every two hours and published freely available to the public as a downloadable Multi-Threaded Routing Toolkit (MRT) file.

Prefix-to-AS mapping And ASN lookups

There are many ways to perform network prefix to AS mapping. The earliest version of traceroute to include AS mapping of each gateway encountered along a path is the previously discussed PRtraceroute [39]. This version of traceroute uses data from the Internet Routing Registry (IRR) to perform an analysis of each hop, outputting a list of the ASes traversed by each gateway encountered. Another option is to use BGP AS Origin data, which is seen by many as more accurate and up-to-date. Studies such as [66] is one among many that have used BGP prefix announcements collected from RouteViews monitors for IP-to-AS mappings. There are also a plethora of online lookup services such as Team Cymru's IP to ASN Mapping Service [4], which have been used in previous measurement studies [66] and can be seen as trustworthy. The downside of such services is that there is often a limit on the number of requests one may perform in a given time span.

The IPv6 Hitlist Service

As mentioned in section 3.1.3, it is not feasible to execute probing against *all* available IPv6 addresses. In addition, each target address needs to meet the requirements set regarding responsiveness and reachability. How then, to best choose a suitable list of target addresses? The IPv6 hitlist service [43] is an online service providing a weekly generated list of responsive IPv6 addresses designed for use in IPv6 measurement studies. Their methodology consists of collecting IPv6-addresses from a variety of sources, including forward DNS lookup lists, RIPE Atlas, the Bitnodes API which allows one to get a list of peers connected to the Bitcoin network, and from collected router IP addresses obtained from traceroutes using Scamper. In addition, they perform aliased prefix detection, removing aliased prefixes from the list. An aliased prefix is defined as a network prefix where a single host responds to all addresses within the entire prefix. This is possible, for example by using the *IP_FREEBIND*-option in Linux.

Each listed address is probed for responsiveness daily where any non-responsive addresses are succinctly removed. The measurement service probes the addresses on the most commonly used services; ICMP, TCP port 80 and 443, UDP port 53 (DNS), and UDP port 443, however an address is considered reachable if it responds to at least one of these probes. As of May 2023, their target address list contains more than 6,7 million addresses, where 1,5 million are TCP port 80 responsive, and 1,3 million are TCP port 443 responsive.

Constructing the target IP address list

To construct our target IP address list, we perform network prefix matching using RouteViews AS Origin data obtained from the *routeviews6-prefix2as* dataset [8], matching each listed ASN

with a responsive IPv6 address obtained from the IPv6 Hitlist Service.

3.2 Inferring Path Consistency by choice of IPv6 Flow Label

In this section, we will build on the design from the previous section to describe our methodology for the main contribution of this thesis: is the 3-tuple (src ip, dst ip, flow label) a valid flow identifier in-use in today's Internet, and by extension: is it possible to consistently get the same path by setting the flow label?

3.2.1 Methodology

We execute two parallel traceroute traces, each with a distinct transport-layer flow identifier, to every destination within a hitlist. We then compare each parallel trace against the other for *path equality*. We define path equality as two traces where the responses from each probe with the same originating TTL match. A consequence of this definition is that two traces will be deemed not equal if an ICMP response was not generated or lost along the return path from its source.

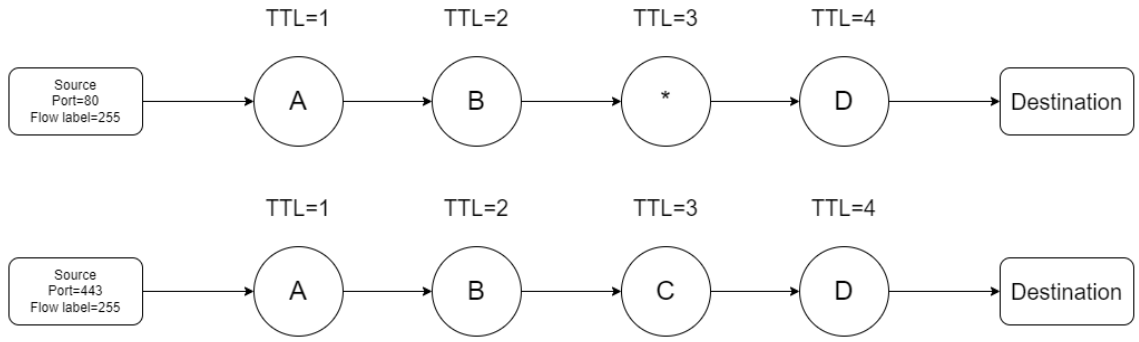


Figure 3.2: In this scenario, node C does not generate a response ICMP, or the response ICMP is lost along the return path. Since not all the responses match, the paths are deemed not equal.

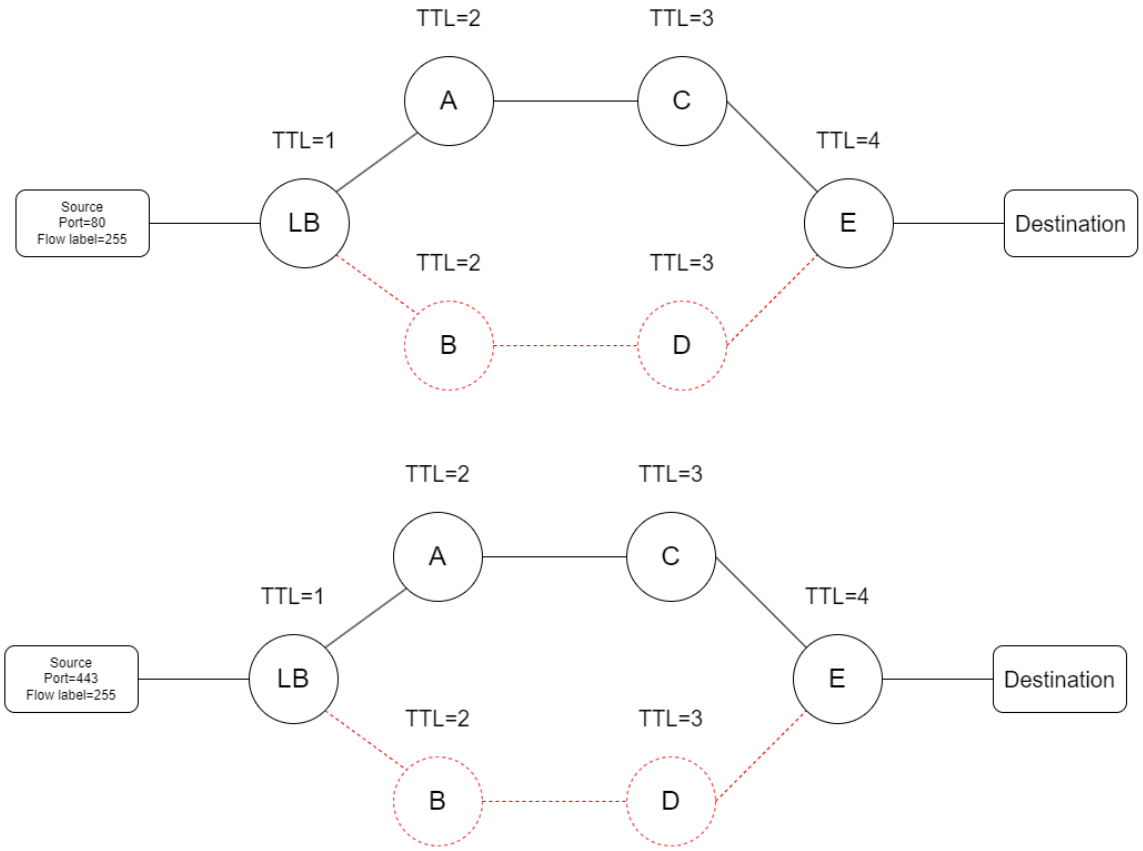


Figure 3.3: In this scenario, packets are forwarded along identical paths by load balancer LB.

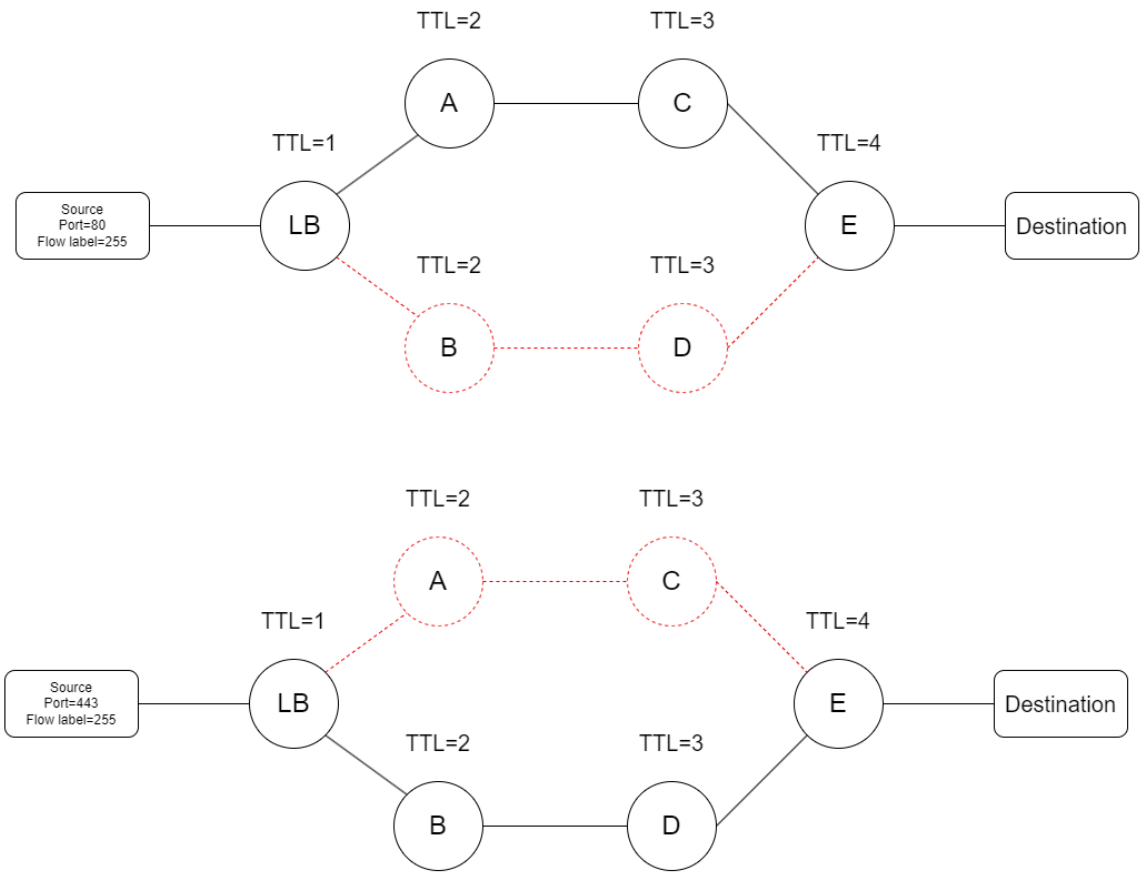


Figure 3.4: In this scenario, packets are forwarded along different paths by load balancer LB, indicating that it is either a per-packet load balancer or the flow identifiers were unequal.

We use TCP as the outbound protocol for traceroute execution. To rule out that the 5-tuple (src ip, dst ip, src port, dst port, protocol) is being used for flow identification, we vary the source and destination port numbers.

3.2.2 Choice of Port Numbers

To more easily slip through firewall filters, we use TCP destination ports 80 and 443, emulating web-traffic. The source port cycles between the range starting with 33456 (the default traceroute source port) and ending with 33465.

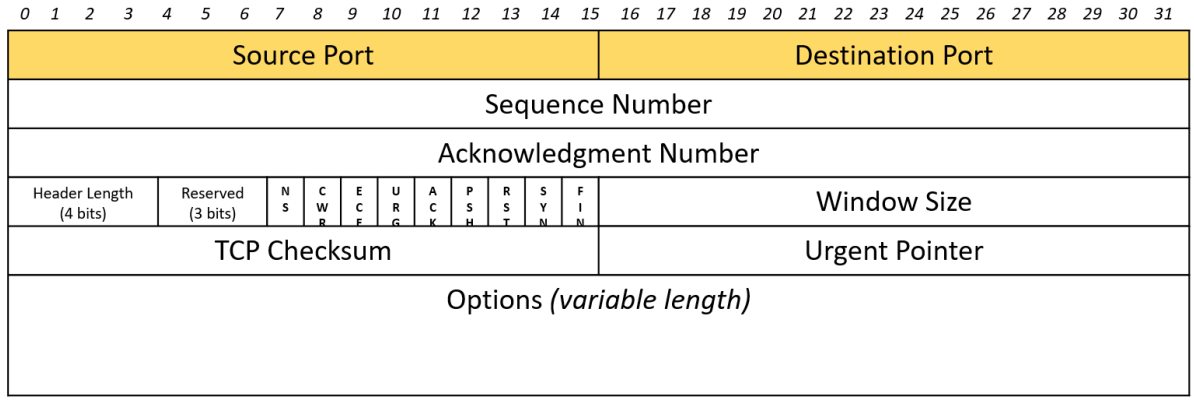


Figure 3.5: We alternate the destination port in the TCP header between 80 and 443 for two parallel traceroute traces, giving them a different transport-layer flow identifier.

To keep a constant network-layer flow identifier we carefully avoid modifying the header fields used in flow identifier hashing algorithms as found by the authors of Paris traceroute [84].

3.2.3 Number of Probes Per Hop

Most Traceroute-applications include an option to set the number of probes executed per hop. For the average user, executing multiple probes per hop is useful as it leads to a higher likelihood of response from each node. However, in the presence of ECMP routing, executing more than one probe per hop can lead to some undesired results. Consider the topology in figure 3.6:

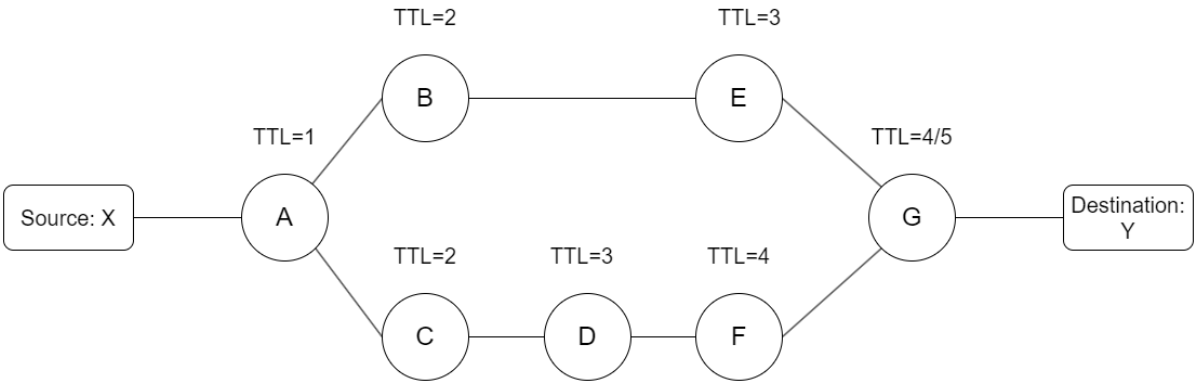


Figure 3.6: Example topology where A->B->E->G and A->C->D->F->G are equal-cost paths.

Table 3.2 illustrates what a traceroute with three probes may end up looking like if the topology consists of an ECMP load balancer at hop A:

Table 3.2: Possible traceroute result when using three probes per hop on an ECMP load balanced path.

TTL	Probe 1	Probe 2	Probe 3
1	A	A	A
2	C	B	B
3	D	D	E
4	F	G	F
5	Y	G	G

As seen in table 3.2, the resulting probe responses with a similar TTL appear to fluctuate between addresses giving an inconsistent and hard to interpret result. To avoid this, the solution is to use a single probe per TTL. As a helpful side effect, this will also speed up execution time of each individual traceroute.

3.3 Methodological Challenges

In this section we discuss the limitations and challenges with our methodology. Some of the challenges are specific to our methodology, while others are shared among all measurement campaigns using traceroute.

Per-packet load balancers and per-destination load balancers

Per-packet load balancers and per-destination load balancers, as discussed in section 2.3.2, create significant challenges to our testing methodology, since we don't know whether an equal or unequal path result is due to the randomness of per-packet load balancing, or if it is a valid result from per-flow load balancing. While Paris traceroute implements an algorithm for detecting per-packet load balancers, it is likely impossible to design an experiment around avoiding them.

Transient route changes

Let's say we launch two simultaneous packets X and Y to a destination D. If a transient route change occurs (perhaps due to link failure) on an ECMP load balancer A on the path to D after it has finished processing and sending packet X, but before it has sent packet Y, this will lead to packet Y being sent to D via a different path. In our resulting data, the resulting traces will not have path equality even though the load balancer could potentially have found the flow identifier of both packets to be the same. We call this result a false-negative.

ICMP source interface IP address

A router may have hundreds, or thousands, of virtual IP addresses. While [RFC1812] states that routers must respond to ICMP *Time Exceeded*-messages using the egress interface's IP-address, in practice, routers fortunately generate responses with the ingress interface's IP-address as the source IP. If this wasn't the case, traceroute would not function properly [82].

IP-in-IP tunneling

Since our methodology relies on inspecting the packet information in the returning packet, there are some properties of IP-in-IP tunneling that make things difficult. The first is flow label inheritance or flow label zeroing. Some devices allow an outer IPv6-packet to inherit the inner IPv6 packet's flow label, but this can be an optional setting that may be disabled. Other devices may not support inheritance at all and simple use a flow label of zero in the outermost header. IP-in-IP tunneling makes parsing the returning innermost packet more difficult as one would have to first unwrap the outermost header. In addition, IP-in-IP may be chained, leading to several outer headers.

ICMP responsiveness

While traceroute provides us with path data to our destination, it needs to be emphasized that it is based on a 30-year old hack. The unfortunate reality is that today's network protocols have no inherent mechanism for providing topology data and were never designed with this in mind. In fact, due to security considerations, network administrators are often actively encouraged to hide or obfuscate this data, making life for researchers difficult. Some routers do not respond with ICMP *Time Exceeded*-messages, instead choosing to silently discard the packets. As detailed in [RFC1435], this may be done deliberately to protect against buggy or outdated implementations that immediately drop connections if an ICMP message is received. Firewalls and other middleboxes are often configured to silently drop packets whose TTL reaches zero, or may suppress all outgoing ICMP-messages entirely [RFC2923]. In addition, it is important to highlight that ICMP messages are a low priority for routers. Modern routers typically have several distinct levels of processing depending on the destination of the packets and the contents within. Packets destined to the router itself are said to be on the "control plane", while packets forwarded *through* the router reside within the "data plane". The data plane itself is split between a "fast lane" and a "slow lane", depending on whether a packet can be processed entirely within hardware (the fast lane) or if it may require extra processing on the software side (the slow lane). Data plane traffic is typically prioritized, and if a router is congested, ICMP generation is *not* prioritized and may be discarded entirely. Time-exceeded is just one of many ICMP-messages that a router may be generating at a time, and many routers will rate-limit ICMP generation to protect against DDoS-attacks and routing loops. Data plane traffic will typically have specialized ASICs which handle the forwarding operations, while ICMP generation is left to a slower general-purpose CPU.

Transient Route Changes

Transient route changes may occur at any time due to power outages, faulty links, planned maintenance or a host of other reasons. When executing several traceroutes to a single destination, a transient route change may change the reported path independent of any ECMP- or LAG- hash-based forwarding. Two traces being judges as unequal due to transient route changes would be a false-negative.

NAT

Source address replacement done by NAT can make several distinct routers appear as a single host. While traceroute variants such as Dublin-traceroute allow a user to identify whether address translation has occurred, there is no way to identify the original IP-address of the ICMP-originating host behind a NAT.

MPLS

As discussed in section 2.4.2, label switched networks will hide topology information when opting not to enable TTL-inheritance. The resulting traceroute outputs on such networks may include a number of repeating IP addresses (loops).

Flow Handling Lifetime

Specification [RFC2460] details the challenges of flow-handling lifetime. In particular, the specification states: "A source must not re-use a flow label for a new flow within the maximum lifetime of any flow-handling state that might have been established for the prior use of that flow label" and "A flow is uniquely identified by the combination of a source address and a non-zero flow label". In other words, if a source host reuses a flow label while the flow-handling state of a previous flow with the same flow label is still active, there is a risk that the new flow could be misclassified as the old flow. However, for our purposes this is not as much of a concern. The reason for this is that we are testing whether two messages sent in parallel with different port numbers, but the same flow label value, are identified as belonging to the same flow. If the flow-handling state on a particular router already exists for that flow label value, it's fine as the parallel packets will still be identified as belonging to the same flow. A bigger concern here is the corner case where a flow-handling state lifetime expires in-between traceoutes to the same destination with the same flow label.

Consider the following example: We send traceroute A to destination D with flow label F and destination port P. We then send traceroute B to destination D with the same parameters except for a different destination port Q. In the time between sending traceroute A and B, the lifetime of the flow-handling state on the first load-balancer in the traceroute's path expires. Traceroute B would then be treated as a new flow. The end result is that, two traces which would be otherwise classified as belonging to the same flow could instead be treated as different flows, routed differently and ultimately leading to a false-negative in our dataset. Note that this example assumes that only layer-3 information is used for flow identification.

Sampling topology

A downside of traceroute is that it repeatedly samples close to the vantage point [61], leading to results where the total number of unique routers visited may be low.

Chapter 4

Implementation

In this section we describe the implementation of our work. Section 4.1 describes the choice of traceroute application to use as a starting point. In section 4.2, we describe the Paris traceroute implementation and considerations taken to implement the flow label value change. Finally, section 4.3 details how we quickly and efficiently deploy and collect data from any number of hosts.

4.1 Application Requirements

In choosing the traceroute platform on which to base our implementation, we identified the following application requirements:

TCP probing The focus of this measurement study is TCP. As such, the traceroute application needs to be TCP-capable.

Flow label modification In order to test flow label persistence, we require a traceroute application capable of setting a custom flow label value in its outbound probes and matching the incoming ICMP packets to the correct probe.

ICMP payload inspection As detailed in chapter 2.1.4, the ICMP payload will contain the first 8 octets of the response packet which includes the flow label header field as received it was when by each individual router. By inspecting the received flow label value and comparing it against the source flow label value, we can identify whether the flow label was changed by the previous hop.

Parallel probing A requirement for our path-equality measurement study is to launch probes in parallel.

Concurrency To reduce test execution time we will execute measurement probes to several destinations concurrently.

4.1.1 Choice of Application

As our starting point, we use the Paris traceroute application. When referring to Paris traceroute, it is important to distinguish between the Paris traceroute *application*, and the Paris traceroute *library*. The library is the core implementation of Paris traceroute algorithm, freely

available to be included in any application. The application with the same name, also developed by the authors of the library, provides a working implementation of the library along with a full traceroute application.

We decided upon the Paris traceroute application for several reasons. First and foremost, it is a well-tested program that has been used in numerous measurement studies, such as [21]. It comes prebuilt with a variety of probing strategies and options, such as being able to set the number of probes per hop, that are useful for tailoring our experiment parameters. Its simplicity makes it quicker and easier to modify compared to more extensive applications such as Scamper [62]. And finally, it is fully open source and freely available for use and modification under the GNU Lesser General Public License.

4.2 Paris Traceroute Implementation

We extend the Paris traceroute application by adding an addition file, *ext.c* and its associated header file. This file contains most of the logic for parsing and extracting the returned flow label. In addition, we modify existing files *paris-traceroute.c*, *ipv6.c* and *network.c* to add code responsible for getting and setting the outbound flow label value, perform ASN lookup and writing the results to a SQLite-database.

4.2.1 Flow Label Modification

Enabling flow label modification required only a few extra lines of code. First, we get the flow label from the argument list.

```
// We assume that the flow-label is always the second-to-last argument
flow_label = atoi(argv[argc - 2]);
set_flow_label(flow_label);
```

Listing 4.1: paris-traceroute.c: Getting the flow label from the input argument list

We then set the flow label using a simple setter-function. This overrides the default flow label value of zero set by Paris traceroute for outbound flows.

```
void set_flow_label(int flow_label)
{
    if (flow_label >= 0 && flow_label <= 0xFFFFF)
    {
        non_default_flow_label = flow_label;
    }
    else
    {
        fprintf(stderr, "Flow label must be a value between %x and %x",
    0x0, 0xFFFFF);
        exit(1);
    }
}
```

Listing 4.2: ipv6.c: Modifying the flow label. non_default_flow_label is a global variable controlled by set_flow_label.

```
size_t ipv6_write_default_header(uint8_t *ipv6_header)
{
    size_t size = sizeof(struct ip6_hdr);
    uint32_t ipv6_flow;
```

```
5
6       if (ipv6_header)
7       {
8           // Write the default IPv6 header.
9           // The tuple "flow" = {version, traffic class, flow label} will
        be overwritten.
10          memcpy(ipv6_header, &ipv6_default, size);
11
12          // Prepare the tuple {version, traffic class, flow label}.
13          ipv6_flow = ipv6_make_flow(
14              IPV6_DEFAULT_VERSION,
15              IPV6_DEFAULT_TRAFFIC_CLASS,
16              non_default_flow_label);
17
18          // Write this tuple in the header.
19          memcpy(
20              &((struct ip6_hdr *)ipv6_header)->ip6_flow,
21              &ipv6_flow,
22              sizeof(uint32_t));
23      }
24      return size;
25  }
```

Listing 4.3: ipv6.c: Writing the modified flow label to the IPv6-header.

4.2.2 IPv6 Packet Parsing and Data Extraction

We implement a parser to extract the returning flow label from the ICMP payload. The parser is passed a pointer to the first byte of the incoming packet from the underlying network layer and uses this a starting point. Since the IPv6 header has a constant size, we only need to increment the pointer by the IPv6 header length to reach the next header.

```
1  ipv6_header *parse_ipv6(const uint8_t *first_byte)
2  {
3      ipv6_header *h = calloc(1, sizeof(ipv6_header));
4
5      /* Fill IPv6 struct */
6      h->version = (*first_byte >> 4);
7      h->traffic_class = ((uint16_t)(*first_byte & 0x0F) << 8) | (*(
       first_byte + 1) >> 4);
8      h->flow_label = ((uint32_t)(*(first_byte + 1) & 0x0F) << 16) | ((
       uint32_t) * (first_byte + 2) << 8) | *(first_byte + 3);
9      h->payload_length = (((uint16_t) * (first_byte + 4)) << 8) | *(
       first_byte + 5);
10     h->next_header = *(first_byte + 6);
11     h->hop_limit = *(first_byte + 7);
12
13     /* Set source */
14     memcpy(h->source.__in6_u.__u6_addr8, (first_byte + 8), 16);
15     /* Set destination */
16     memcpy(h->destination.__in6_u.__u6_addr8, (first_byte + 24), 16);
17
18     return h;
19 }
```

Listing 4.4: ext.c: Parsing the IPv6 Header.

```
icmp6_header *parse_icmp6(const uint8_t *icmp_first_byte)
{
    icmp6_header *h = calloc(1, sizeof(icmp6_header));
    h->type = *icmp_first_byte;
    h->code = *(icmp_first_byte + 1);
    h->checksum = ((uint16_t) * (icmp_first_byte + 2) << 8) | *(
   icmp_first_byte + 3);
    // Depending on the type there can be a value between bytes 5-9 as
   well,
    // though this value is not used in our project.
    return h;
}
```

Listing 4.5: ext.c: Parsing the ICMPv6 Header.

We do not need to be concerned about the packet being fragmented as ICMPv6 will only include enough of the invoking packet without exceeding the path MTU [RFC4443]. Security headers like the Authentication Header and Encapsulation Security Payload Header are not set by our egress probes, and will be ignored by the parser if included in the response. Any Hop-by-hop options, Destination Options or Routing Headers set by a router generating ICMP response messages are most likely indicators of an erroneous or misconfigured router, and thus these messages are ignored as well.

```
ipv6_header *get_inner_ipv6_header(uint8_t *first_byte)
{
    const int IPV6_HEADER_LENGTH = 40;
    const int ICMPV6_HEADER_LENGTH = 8;
    icmp6_header *icmp6;
    ipv6_header *inner_ipv6;

    if ((*first_byte >> 4) == 6) // If IPv6
    {
        ipv6_header *ip6h = parse_ipv6(first_byte);
        uint8_t nh = ip6h->next_header;
        switch (nh)
        {
        case NH_ICMPv6:
            icmp6 = parse_icmp6(first_byte + IPV6_HEADER_LENGTH);
            switch (icmp6->type)
            {
            case ICMP_TIME_EXCEEDED:
                inner_ipv6 = parse_ipv6(first_byte + IPV6_HEADER_LENGTH
    + ICMPV6_HEADER_LENGTH);
                return inner_ipv6;
            case ICMP_DESTINATION_UNREACHABLE:
                if (icmp6->code == 4)
                {
                    inner_ipv6 = parse_ipv6(first_byte +
   IPV6_HEADER_LENGTH + ICMPV6_HEADER_LENGTH);
                    return inner_ipv6;
                }
                else
                {
                    return NULL;
                }
            default:
                return NULL;
            }
```

```
34          default:
35              return NULL;
36          }
37      }
38  #ifdef EXT_DEBUG
39      fprintf(stderr, "get_inner_ipv6_header: Error: packet is not an
        IPv6-packet.");
40  #endif
41      return NULL;
42  }
```

Listing 4.6: ext.c: Exctracting the inner IPv6 Header from the ICMP-payload. An ICMP type TIME EXCEEDED is generated by routers upon the Hop Limit reaching zero. An ICMP DESTINATION UNREACHABLE with error code 4 is generated if the targeted transport-layer port is unreachable. Since we are using destination TCP port 80 and 443, these messages will typically be generated by end-hosts not responsive to HTTP or HTTPS. If the TCP SYN request to the end-host is successful, no ICMP will be generated and the response is simply ignored.

Upon reaching the embedded IPv6-header, we parse the inner packet and create a hop structure. To the hop structure we assign the extracted inner flow label, the matching egress probe's hop-number, the hop's IP-address (i.e. source address from the "outer" IPv6-packet) and the hop's ASN obtained by performing ASN lookup on the hop IP. We limit the total number of hops to 35, as any hop beyond this point will likely be due to a routing loop.

4.2.3 ASN Lookup

While the basic Paris traceroute application includes an ASN-lookup feature, it proved non-functional in our testing. We tested standalone tools performing WHOIS lookups with varying degrees of success. One problem we encountered is that ASN information is often not included by regional Internet registries in WHOIS lookup-data. In addition, repeatedly performing WHOIS lookups toward a remote server for thousands of IP-addresses is often rate-limited and something we would like to avoid. Instead, we provide our own implementation using Routeviews ASN Origin data [8] as described in section 3.1.3. The implementation uses a variant of a RADIX-trie called a PATRICIA-trie to perform prefix matching. The algorithm traverses each node in the trie starting from the root, and sequentially follows each branch by matching the next character in a given IP-address. At the end, the result is either a match in which case the data contained at the leaf node is returned, or the prefix is not found, in which case we return the string "NULL". Since implementing a PATRICIA-trie from scratch in C is both time-consuming and could be a possible source of errors, we use an open-source version developed by Dave Plonka at the University of Michigan.

```
1  int asnLookupInit(char *filename)
2  {
3      struct in6_addr *my_addr = calloc(1, sizeof(struct in6_addr));
4      while ((read = getline(&line, &len, f)) != -1)
5      {
6          token = strtok(line, " ");
7          int nmb = 1;
8          while (token)
9          {
10             switch (nmb)
11             {
12             case 1:
13                 address = token;
```

```
                inet_pton(AF_INET6, address, my_addr);
                break;
            case 2:
                mask = atoi(token);
                break;
            case 3:
                nmb = 1;
                asn = malloc(sizeof(char) * 200);
                strcpy(asn, token);
                /* Strip trailing newline */
                asn[strcspn(asn, "\n")] = 0;
                /*  Insert into patricia-tree */
                insert(AF_INET6, *my_addr, mask, asn);
                break;
            default:
                puts("Error: default");
                break;
            }
            nmb++;
            token = strtok(NULL, " ");
        }
    }
```

Listing 4.7: ext.c: Initializing the PATRICIA-trie.

We use a tokenizer to separate each individual network prefix and ASN. After separating each network prefix into a token, we use the Berkeley sockets library to create an IPv6 address-struct. The struct is inserted into the PATRICIA-trie along with its mapped data (the ASN).

```
char *asnLookup(struct in6_addr *ipv6_address)
{
    char *lookup_result = lookup_addr(AF_INET6, *ipv6_address);
    return lookup_result;
}
```

Listing 4.8: ext.c: Performing ASN lookup.

We use the lookup_addr-function provided as part of the PATRICIA-trie implementation to perform longest-prefix matching and return the ASN lookup data.

```
typedef struct hop
{
    uint8_t hopnumber;
    uint32_t returned_flowlabel;
    char hop_asn[200];
    struct in6_addr hop_address;
} hop;
```

Listing 4.9: ext.h: The Hop-structure.

The data associated with each hop is saved to a *Hop*-structure, which contains the hop number, the returned inner flow label of the contained in the ICMP payload, the IPv6-address of the hop and the ASN-lookup result of the hop address.

```
char *asnlookup_result = asnLookup(&h->hop_address);
if (asnlookup_result != NULL)
{
  memcpy(h->hop_asn, asnlookup_result, strlen(asnlookup_result) + 1);
}
else
{
```

```
8   strcpy(h->hop_asn, "NULL");
9 }
```

Listing 4.10: network.c: Writing ASN to the hop struct.

If the network prefix is not found in the dataset, we return the string *"NULL"*.

4.2.4 Calculating the Path Hash

To compare path equality, we create a hash of each path using each (hop IP-address, hop number)-tuple visited along the path. This ensures that two paths will only have an equal hash (and by extension; an equal path) if-and-only-if both the IP-address and hop numbers match. This is necessary to protect against loops, where one IP-address is repeatedly returned for increasing Hop Limit-values. Since we are only comparing paths to the same destination, hash-collisions are unlikely.

```
1     /* Create address tuples */
2     addr_tuple *address_tuples = malloc(sizeof(addr_tuple) * t->
      hop_count);
3     for (int i = 0; i < t->hop_count; i++)
4     {
5         address_tuples[i].hop_address = t->hops[i].hop_address;
6         address_tuples[i].hopnumber = t->hops[i].hopnumber;
7     }
8     /* Create path hash */
9     uint8_t *path_hash = hashPathTuple(address_tuples, t->hop_count);
10    free(address_tuples);
11    char output_buffer[21];
12
13    /* Create string-representation of the hash digest */
14    for (int i = 0; i < SHA_DIGEST_LENGTH; i++)
15    {
16        sprintf(output_buffer + (i * 2), "%02x", path_hash[i]);
17    }
18    output_buffer[20] = '\0';
```

Listing 4.11: paris-traceroute.c: Calculating the 20-byte path hash using the standard openSSL SHA1 implementation. The path hash is calculated using both the hop address and the hop number.

```
1  uint8_t *hashPathTuple(addr_tuple arr[], int arraySize)
2  {
3      unsigned char *obuf = malloc(sizeof(uint8_t) * 20);
4      SHA_CTX shactx;
5      SHA1_Init(&shactx);
6      for (int i = 0; i < arraySize; i++)
7      {
8          SHA1_Update(&shactx, &arr[i].hop_address, sizeof(struct
       in6_addr));
9          SHA1_Update(&shactx, &arr[i].hopnumber, sizeof(uint8_t));
10     }
11     SHA1_Final(obuf, &shactx); // digest now contains the 20-byte SHA-1
       hash
12     return obuf;
13 }
```

Listing 4.12: ext.c: A deeper look at the hashPathTuple-function. By iterating through the list of all hops

The string representation of the digest is then written to the SQLite database.

4.2.5 Output Format

We were not able find a standardized format for serializing raw traceroute output. Some applications, such as the previously discussed Scamper [62], offer several output formats, raw text (CSV) and JSON. Scamper in particular has its own native file format *warts* [63]. Warts is a custom format designed specifically for traceroute measurement studies which contains metadata such as the source- and destination-address, the number of attempts, the timeout length for each probe, ToS information, and more.

However, the Warts format as it stands, is unsuitable for our measurement study due to a few reasons. Being an application-specific custom crafted format makes it difficult to translate and import into tools such as Python's Pandas library without the use of external tools. It is not suited for recording egress flow label values as it lacks the data fields to do so, nor does it have fields for recording the received flow label from each hop. It also includes a number of parameters unnecessary to our methodology which increases complexity and could lead to larger file sizes and increased processing times. Instead, we provide our own implementation based on writing to an SQLite database. Table 4.1 details the SQLite database schema used in our implementation.

Table 4.1: SQLite database schema.

Name	Type	Description
START TIME	INTEGER	Start time in Unix time format
SOURCE TCP PORT	INTEGER	Source TCP port
SOURCE FLOW LABEL	INTEGER	Start time in Unix time format
SOURCE IP	TEXT	Start time in Unix time format
SOURCE ASN	TEXT	ASN lookup result of the source IP address
DESTINATION IP	TEXT	Destination TCP port
DESTINATION ASN	TEXT	ASN lookup result of the destination IP address
PATH HASH	TEXT	Cumulative 20-byte hash of every (hop IP-address, hop-number)-tuple
HOP COUNT	INTEGER	The number of recorded hops
HOP IP ADDRESSES	TEXT	Whitespace-separated, ordered list of every hop IP address
HOP NUMBERS	TEXT	Whitespace-separated, ordered list of every hop TTL
HOP RETURNED FLOW LABELS	TEXT	Whitespace-separated, ordered list of the returned inner flow label from the ICMP payload
HOP ASNS	TEXT	ASN lookup result of the hop IP addresses

Since the number of hops is variable per traceroute trace, we record the hop information in an ordered, whitespace-separated list. These fields can later be separated and combined in a row-by-row basis for analysis.

4.2.6 Writing To Database

We use the official SQLite C library to connect and write the resulting traceroute output to a SQLite database-file. The SQLite implementation ensures by default that only one write is executed at a time, alleviating concerns regarding parallelism and data race conditions.

```
int db_insert(sqlite3 *db, traceroute *t, char *src_ip_in, char *
    dst_ip_in)
{
    char *error_message;
    int result_code;
    char sql[4096];
    size_t s_len;

    char src_ip[INET6_ADDRSTRLEN + 2];
    char dst_ip[INET6_ADDRSTRLEN + 2];
    src_ip[0] = '\"';
    dst_ip[0] = '\"';
    strcpy((src_ip + 1), src_ip_in);
    strcpy((dst_ip + 1), dst_ip_in);
    s_len = strlen(src_ip);
    src_ip[s_len] = '\"';
    src_ip[s_len + 1] = '\0';
    s_len = strlen(dst_ip);
    dst_ip[s_len] = '\"';
    dst_ip[s_len + 1] = '\0';

    char *hiats = hop_ip_addresses_to_string(t);
    char *hnts = hop_numbers_to_string(t);
    char *hrfts = hop_returned_flowlabels_to_string(t);
    char *hats = hop_asns_to_string(t);
    char *hash = path_id_to_string(t->path_id);

    sprintf(sql,
        "INSERT INTO TRACEROUTE_DATA (START_TIME,\
    SOURCE_TCP_PORT,\
    SOURCE_FLOW_LABEL,\
    SOURCE_IP,\
    SOURCE_ASN,\
    DESTINATION_IP,\
    DESTINATION_ASN,\
    PATH_HASH,\
    HOP_COUNT,\
    HOP_IP_ADDRESSES,\
    HOP_NUMBERS,\
    HOP_RETURNED_FLOW_LABELS,\
    HOP_ASNS) \
    VALUES (%ld,%d,%d,%s,%s,%s,%s,%s,%d,%s,%s,%s,%s);",
            t->start_time,
            t->outgoing_tcp_port,
            t->outgoing_flow_label,
            src_ip,
            t->source_asn,
            dst_ip,
            t->destination_asn,
            hash,
            t->hop_count,
            hiats,
```

```
            hnts,
            hrfts,
            hats);
    /* Insert into table */
    if ((result_code = sqlite3_exec(db, sql, &db_callback, NULL, &
    error_message)) != SQLITE_OK)
    {
        fprintf(stderr, "Debug: DB insert failed: %s\n", error_message)
    ;
        return result_code;
    }
    fprintf(stderr, "Debug: DB insert completed successfully\n");
    return result_code;
```

Listing 4.13: ext.c: Writing to the SQLite database.

4.3 Scripting and Testbed Implementation

To create our target IP-address list, execute testing, and transfer data to a central server we use a variety of tools, including shell-scripting, Python-scripting, and Terraform. While it would have been possible to implement some of these steps directly into the Paris traceroute application, we opted to use a microservice-esque approach to separate the concerns of each step.

4.3.1 Creating the Target Address List

As discussed in section 3.1.3, we perform network prefix matching using RouteViews AS Origin data to create the target address list, matching each listed ASN with a responsive IPv6 address obtained from the IPv6 Hitlist Service. Our Python implementation makes use of the SubnetTree-module [6] to create a tree structure of network prefixes and performs lookups using longest-prefix matching.

```
# gets one responsive IP per AS using RouteViews and https://
    ipv6hitlist.github.io/ data
# example usage: python3 generate-hitlist.py -a=routeviews_prefixes.txt
     -i=responsive-addresses.txt -r=routeviews-rv6-20220411-1200.pfx2as.
    txt > hitlist.txt

def get_asn(prefix, my_hashmap):
    asn = 0
    try:
        return my_hashmap[prefix]
    except KeyError as e:
        print(f"KeyError: prefix {prefix} not found in hashmap", file=
    sys.stderr)
    return asn

def fill_tree(tree, fh):
    for line in fh:
        line = line.strip()
        try:
            tree[line] = line
        except ValueError as e:
            print("Skipped line '" + line + "'", file=sys.stderr)
```

```
    return tree

def main():
    parser = argparse.ArgumentParser()
    parser.add_argument("-a", "--aliased-file", required=True, type=argparse.FileType('r'), help="File containing RouteViews prefixes")
    parser.add_argument("-i", "--ip-address-file", required=True, type=argparse.FileType('r'), help="File containing IP addresses to be matched against RouteViews prefixes")
    parser.add_argument("-r", "--routeviews-file", required=True, help="File containing full RouteViews data")
    args = parser.parse_args()

    as_numbers = set()

    # Store aliased and non-aliased prefixes in a single subnet tree
    tree = SubnetTree.SubnetTree()

    # Read aliased and non-aliased prefixes
    tree = fill_tree(tree, args.aliased_file)

    #my_hashmap = create_hashmap(args.routeviews_file)
    my_hashmap = {}
    with open(args.routeviews_file, "r") as file:
        data = file.readlines()
        # Create hashmap (python dictonary)
        for line in data:
            line = line.strip()
            line = line.split()
            key = line[0] + "/" + line[1]
            my_hashmap[key] = line[2]

    # Read IP address file, match each address to longest prefix and print output
    for line in args.ip_address_file:
        line = line.strip()
        try:
            prefix = tree[line]
            tmp_asn = get_asn(prefix, my_hashmap)
            if tmp_asn not in as_numbers:
                as_numbers.add(tmp_asn)
                #print(f"{get_asn(prefix, my_hashmap)},{line},{prefix}") # if you want to print the corresponding AS-number and prefix as well
                print(f"{line}")
        except KeyError as e:
            print("Skipped line '" + line + "'", file=sys.stderr)

if __name__ == "__main__":
    main()
```

Listing 4.14: create-hitlist.py: Creating the target address list

We iterate over every address obtained from the IPv6 Hitlist Service. To ensure that we only get one target address per ASN, we keep a list of all ASNs that have been added. Only if the next IP-address is not prefix-matched to an already-listed ASN does it get added to our final target address list.

4.3.2 Testbed Setup Using TerraForm and Salt

We use a tool developed by HashCorp called TerraForm [9] to execute API calls against our chosen cloud provider. Terraform allows us to define our infrastructure as code which is saved by the TerraForm as a *state*. If we add or remove resource definitions, Terraform will recognize that the state is updated and perform the changes against the remote API as necessary. This way, we can easily scale up to a potentially infinite number of hosts restricted only by cost and availability. Once a testbed is deployed, we use a shell script to install the Paris traceroute-application and its dependencies. We also install SaltStack, which is another host management tool that allows us simultaneously send commands to any number of connected hosts. This allows us to concurrently initiate the measurement study on all test sites.

```
resource "digitalocean_droplet" "www-1"
{
  count = 1

  image  = "ubuntu-20-04-x64"
  name   = "ubuntu-lon1-${count.index}"
  region = "lon1"
  size   = "s-2vcpu-2gb"
  ipv6   = true
  ssh_keys = [
    data.digitalocean_ssh_key.new-key.id
  ]

  provisioner "remote-exec" {
    inline = [
      "export PATH=$PATH:/usr/bin",
      "sudo apt update -y",
      "sudo apt upgrade -y",
      "apt install autoconf build-essential git libtool -y",
      "curl -L https://bootstrap.saltstack.com -o install_salt.sh",
      "sudo sh install_salt.sh -A 209.97.138.74"
    ]
  }
}
```

Listing 4.15: Snippet from the TerraForm config demonstrating deployment of a host to the London datacenter.

4.3.3 Concurrent Probing

To reduce test completion time, we launch sixteen parallel Paris traceroute processes that each probe a unique destination address from the hitlist. This sequence loops until all destinations have been probed. The number of simultaneous parallel processes is a limitation of our vantage point system specs, where we discovered that an increase in the number of parallel processes beyond sixteen would lead to processes getting killed by the operating system due to insufficient RAM. We envision that with improved VP system specs the number of concurrent processes could increase beyond sixteen, further reducing testing time.

Execution Algorithm

We execute traceroute traces to all destinations in the target address list. Once the inner while-loop has finished iterating through the address list, we immediately repeat the experiment for

the next flow label value in the FLOW_LABELS-list. The FLOW_LABELS-list is a list of our chosen flow labels starting from the lowest value 0 and ending with the highest value 1048575. We use the control operator "&" to run processes in parallel. This operator instructs the shell to execute the preceding command in a separate subshell in the background, and then immediately proceed to the next instruction instead of waiting for the command to finish. Due to variations in process startup time and the sequential nature of ethernet datagram transfer, it is impossible to achieve true parallelism. However, this approximation should be well within the flow handling lifetime window discussed in section 3.3.

The bash *wait* statement acts as a barrier, halting further execution until all spawned processes are finished. The start time is equal for all measurement probes in the inner loop, however we have found that moving the start time expression to the inner loop makes no difference for the values produced in the measurement output due to the speed at which new processes are launched.

```
SRC_PORT=33456
HITLIST_LENGTH=$(wc -l <$HITLIST)
for FLOW_LABEL in "${FLOW_LABELS[@]}"; do
    N=1
    M=$N_PARALLEL
    while [ $N -lt $HITLIST_LENGTH ]; do
        readarray -t ip_array < <(sed -n "${N},${M}p" $HITLIST)
        START_TIME=$(date '+%s')
        for ADDRESS in ${ip_array[@]}; do
            for DESTINATION_PORT in "${DESTINATION_PORTS[@]}"; do
                pt_run "$ADDRESS" "$DESTINATION_PORT" "$FLOW_LABEL" "
    $SRC_PORT" &
                ((SRC_PORT=SRC_PORT+1))
            done
        done
        wait
        SRC_PORT=33456
        let N=$N+$N_PARALLEL
        let M=$M+$N_PARALLEL
    done
done
```

Listing 4.16: pt-run-src-port.sh: Execution algorithm implementation.

Rate Limits and DoS

While VP processing power is a limiting factor, we need to be wary of executing too many probes in parallel to avoid effectively performing a Denial-of-Service (DoS) attack against the gateways closest to the VP. As a preventative measure, many routers rate-limit ICMP probes from a single host, leading to a higher rate of dropped packets when probing in parallel. For this reason, some applications, such as the previously mentioned Scamper, allow the user to rate-limit the number of outgoing probes. It is also possible to apply a crude form of rate-limitation at the Linux OS-level by using the traffic control (tc) tool [59]. The Paris traceroute application has no rate limiter built-in by default, and we opt not to implement one or enable it at the OS-level due to the relatively low number of probes simultaneously in flight. We imagine that with increased probing rates, rate limits should be enforced to avoid the aforementioned problem.

4.3.4 Transferring Data To A Central Server

Once Paris traceroute has finished iterating through the list of all flow labels and all target IP addresses, we compress the resulting database-file into a tarball, send it to a central server using Secure Copy (SCP) and perform cleanup. Listing 4.17 illustrates the shell functions responsible for this.

```
create_tarball() {
    cd $TAR_DIR
    echo "Creating tarball..."
    tar -czvf $TAR_FILENAME -C $DB_FILEPATH $DB_FILENAME
    echo "Tarball saved to $TAR_DIR$TAR_FILENAME."
    echo "Transferring tarball to remote host..."
    scp -o UserKnownHostsFile=/dev/null -o StrictHostKeyChecking=no -i
   /root/.ssh/scp-key $TAR_DIR$TAR_FILENAME 209.97.138.74:/root/db-
   storage/$TAR_FILENAME
    if [ $? -eq 0 ]; then
        echo "Transfer completed successfully. Deleting local tarball
   ..."
        rm $TAR_DIR$TAR_FILENAME
        echo "Tarball deleted."
        echo "Cleaning up raw data..."
        find /root/db/ -maxdepth 1 -name "*.db" -print0 | xargs -0 rm
    else
        echo "Transfer to remote host failed."
    fi
}

transfer_db() {
    cd $TAR_DIR
    echo "Creating tarball..."
    tar -czvf $TAR_FILENAME -C $DB_FILEPATH $DB_FILENAME
    echo "Tarball saved to $TAR_DIR$TAR_FILENAME."
    echo "Transferring tarball to remote host..."
    scp -o UserKnownHostsFile=/dev/null -o StrictHostKeyChecking=no -i
   /root/.ssh/scp-key $TAR_DIR$TAR_FILENAME 209.97.138.74:/root/db-
   storage/$TAR_FILENAME
    if [ $? -eq 0 ]; then
        echo "Tarball transfer to remote host completed successfully."
    else
        echo "Tarball transfer to remote host failed."
    fi
}
```

Listing 4.17: Tarball creation and transfer to remote host.

Chapter 5

Measurement Results and Analysis

In this chapter we evaluate the results of our study. Section 5.1 provides an overview of our measurement setup and vantage point configuration. In section 5.2 we detail the results of our flow label persistence study. In section 5.3, we discuss traceroute artifact sanitation before diving into the results of our network path consistency study in section 5.4.

5.1 Measurement Setup

5.1.1 Choice of Vantage Point Provider

We considered several vantage point (VP) providers. A popular option among researchers is the PlanetLab-project [30], which has been used in numerous of previous measurement studies. Unfortunately this project is now shut down [73] in the US, leading us to look for other options. Some measurement studies have been conducted using the Internet2 [70] research network which has the advantage of providing ground-truth data for topology. There are, in addition, a plethora of private cloud providers available such as Amazon's AWS, Microsoft's Azure, and others that offer varying levels of compute for rent at various price points.

In choosing our provider we identified a number of criteria:

Connected to the Internet at large The intent of our study is to study packet behavior within the Internet as it is today. A research network, while useful, may not be sufficiently representative of the internet in-practice in order to yield valid results.

Cost Cost was a major limiting factor. While deploying a single or even multiple VMs via a cloud provider is largely inexpensive, the cost can quickly balloon to unreasonable levels if we were to scale up to a hundred or more VMs, particularly if these need to exist for an extended period of time.

IPv6 capability Due to the nature of our experiment, the VPs have to be IPv6 capable.

Firewall Paris traceroute sets a unique process identifier in the packet payload. It also modifies specific header values, making it critical that the outgoing- or returning ICMP-packets are not blocked by our provider's firewall.

Availability While it is technically possible to restart the experiment from a specific point, any downtime that could adversely affect our the final results of our experiment mid-run is not acceptable.

Bandwidth and latency Due to the low bandwidth requirements of our experiment, this is not a critical issue. High latency could affect the total execution time execution of our experiment.

Geographical diversity In order to reduce traceroute sampling bias [61], we require a variety of vantage point locations.

Network diversity Ideally, the VPs would all be part of a separate ASN and, if possible, be connected via numerious unique ISPs.

In the end, we chose VMs provided by DigitalOcean as our platform. Several factors lead to this decision, with the most important one being application functionality. Through testing we found that our Paris traceroute probes were not receiving any response when executed from VMs provided by Amazons Lightsail and Microsofts Azure. We speculate that the non-standard header field values set by Paris traceroute may have caused the middleboxes at the edge of the network to treat the traffic as malicious. On the DigitalOcean platform, we experienced no such problems. DigitalOcean in addition has a wide range of data centers positioned around the globe.

5.1.2 Cloud Provider Limitations

While DigitalOcean offers VMs in several regions, the variety is limited with Oceania and Africa notably being omitted. DigitalOcean offers no way to limit outgoing or incoming bandwidth for a particular VM at the network level, instead you have to configure the VM or application manually to use a set bandwidth. Another downside of DigitalOcean is a lack of ASN-variety: by performing ASN lookups on our VP's IP-addresses and their default gateways, we found that they all belong to DigitalOcean's ASN 14061.

5.1.3 Vantage Point Configuration

For our test bed, we used eight VPs, one in each datacenter provided by DigitalOcean. This might seem low compared to other studies such as [26], which used 111 vantage points deployed on CAIDAs ARK platform. However, it is unlikely that using several vantage points in the same data center would yield more useful data, provided that the configuration and routing is the same for each VP. Each VP was uniquely located in one of the data centers listed in table 5.1. Figure 5.1 provides a geographic overview of the aforementioned VP sites.

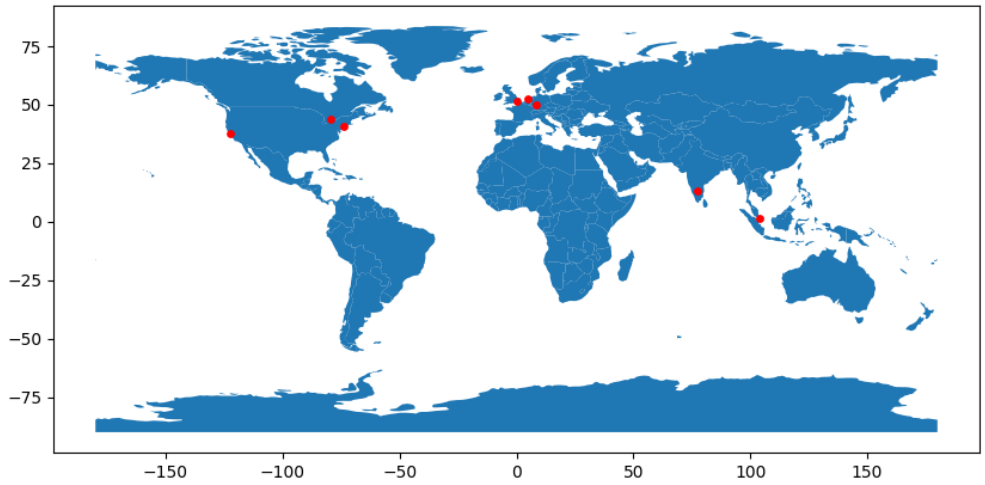


Figure 5.1: Vantage Point locations around the world.

Table 5.1: Vantage Point names and their respective datacenter locations.

Vantage Point	Datacenter Location
nyc	New York, USA
lon	London, The United Kingdom
ams	Amsterdam, The Netherlands
blr	Bangalore, India
sfo	San Francisco, USA
sgp	Singapore, Singapore
fra	Frankfurt, Germany
tor	Toronto, Canada

5.1.4 Test Parameters

Paris traceroute configuration

We set the number of probes per hop=1. We use the -T flag to enable TCP, and vary the source- and destination-ports as described in section 3.2.2. In addition, we set the flow label to one of the values described in section 3.1.2. The rest of the options are kept as per the Paris traceroute defaults.

Test period

Testing occurred in April 2023 from April 12 to April 15.

Target hitlist

Our hitlist contains N=15757 targets, each belonging to a unique AS. This is 13.9% of the 113,221 allocated ASes as of April 2023 [16].

5.1.5 Performance

By running several tests in parallel, we reduce the overall time it takes for an experiment run to complete. We spawn twenty Paris traceroute instances on each VM, running on 8 VMs concurrently. The total time it takes each VM to finish probing all target addresses varies due to differing hop lengths to the destinations, however the time it takes for all VPs to complete a typical run is 4-5 real-world days. The number of parallel instances was carefully chosen in order to not overwhelm the network and VP. Due to the CPU load and memory usage demanded by the Paris traceroute on application startup, mostly as a result of having to load and initialize the ASN lookup tables, the performance demands become a limiting factor on the number of parallel instances one can run on a single VM. To combat this, we decided to scale up our VP configuration from the lowest tier (1 vCPU, 1 GB RAM) to a higher tier with 2 vCPUs and 2 GB RAM. One can imagine that with a more powerful VP even more parallel instances of Paris traceroute would be possible. Eventually, the network bandwidth could become a limiting factor and in these instances reducing the packet rate at either at an application- or a system-level would be necessary in order to avoid overwhelming the network (effectively causing a Denial-of-Service attack).

5.2 Test: Flow Label Persistence

As a reminder, we will quickly summarize what routers are allowed to do with the flow label and what not according to the specification [20]:

- A non-zero flow label must not be changed, with only a security exception related to using it for a covert channel. There, the flow label CAN be changed to a different value, but NOT to zero.
- A zero flow label is OK to be changed to a non-zero value.
- A non-zero flow label may not be changed to zero.

5.2.1 Results

Throughout our our entire measurement campaign consisting of 1,246,156 traceroute traces, encountering 1329 measured load balancers, to 15757 destinations, we did not observe a single instance of the flow label value being changed en-route. This result holds for all vantage points. While highly discouraged by the specification, it is surprising that the resulting changes are none considering the specification does allow for it in some circumstances as discussed in section 3.1. We theorize that with an increased number of destinations and traces, a flow label change may eventually occur, particularly when crossing barriers such as the Great Firewall of China where zeroing out the flow label to protect against covert-channel attacks may seem more likely.

5.3 Traceroute Artifact Sanitation

Inspired by [84] and [66], we sanitize our results by removing invalid traceroute traces.

We define an invalid traceroute trace as a trace where:

- The flow label was not completely carried to its destination.
- A hop IP address was repeated twice or more in a row, known as a loop.
- A hop IP address was repeated, separated by at least one other address, known as a cycle.

5.3.1 Loops

A loop is an IP-address occurring more than once in succession. Loops are typically seen due to NAT, MPLS with no TTL-inheritance, or routers decrementing the TTL (Hop Limit) by more than one. Other explanations exist as well, such as a documented bug in some versions of the BSD-kernel where routers erroneously perform forwarding of packets with a zero-TTL [54].

All in all, the number of loops observed was high: Out of 1,246,156 traces, 245,414 (19.69%) contained a loop. Sometimes more than one was observed in a single trace, leading to a total number of 516,509 loops in the dataset.

Table 5.2: Number of loops observed per vantage point.

Vantage Point	Total number of traceroute traces	Total number of loops observed	Total number of traces containing at least one loop
nyc	155753	39298	18470 [12%]
lon	155754	69821	30333 [19%]
ams	155779	85001	41436 [27%]
blr	155780	76243	38161 [24%]
sfo	155776	48093	24575 [16%]
sgp	155759	71786	35598 [23%]
fra	155777	78009	36073 [23%]
tor	155778	48258	20768 [13%]

5.3.2 Cycles

A cycle, as defined by [84], is a traceroute trace in which the same IP address appears twice, separated by at least one other IP address. In our dataset we observed a total number of 221,951 traces that contain at least one cycle (17.81%). As with loops, a single traceroute trace may contain more than one cycle, meaning that in total 448,817 individual cycles were observed. Cycles can occur as a result of per-packet load balancing or transient route changes which make interfaces on disconnected routers appear adjacent in a trace [84].

Table 5.3: Number of cycles observed per vantage point.

Vantage Point	Total number of traceroute traces	Total number of cycles observed	Total number of traces containing at least one cycle
nyc	155753	42947	23139 [15%]
lon	155754	58332	28597 [18%]
ams	155779	67899	31750 [20%]
blr	155780	71336	34209 [22%]
sfo	155776	52834	28529 [18%]
sgp	155759	46566	22299 [14%]
fra	155777	66287	31097 [20%]
tor	155778	42616	22331 [14%]

5.3.3 Dropping Invalid Traces

All cycles, loops and traces where the flow label was not completely carried to its destination are considered invalid and dropped from the dataset before proceeding to the next step of analysing path consistency with a particular flow label value. Because comparing paths requires at least two traces to compare against each other, we drop all traces to the same path with the same flow label if at least one trace was considered invalid. Fortunately, this made very little difference to the total size of the dataset as most invalid traces mirror each other.

5.4 Test: Network Path consistency

In this section we investigate the results of performing two parallel traceroutes to the same destination with a similar network-layer flow-identifier. By comparing each path, we can draw a conclusion on whether a particular flow label has a measureable effect on path consistency.

5.4.1 Path Consistency Results

Flow Label	Number of Consistent Destinations	Number of Divergent Destinations
0	620 [5%]	11722 [95%]
255	596 [5%]	11493 [95%]
65280	606 [5%]	11389 [95%]
983040	605 [5%]	11044 [95%]
1048575	551 [5%]	11181 [95%]

(a) VP nyc

Flow Label	Number of Consistent Destinations	Number of Divergent Destinations
0	358 [4%]	9024 [96%]
255	378 [4%]	8994 [96%]
65280	337 [4%]	8953 [96%]
983040	362 [4%]	8762 [96%]
1048575	395 [4%]	8933 [96%]

(b) VP ams

Flow Label	Number of Consistent Destinations	Number of Divergent Destinations
0	298 [3%]	9332 [97%]
255	327 [3%]	9257 [97%]
65280	337 [4%]	9091 [96%]
983040	325 [3%]	9371 [97%]
1048575	323 [3%]	9274 [96%]

(c) VP blr

Flow Label	Number of Consistent Destinations	Number of Divergent Destinations
0	293 [3%]	9494 [97%]
255	295 [3%]	9486 [97%]
65280	291 [3%]	9248 [97%]
983040	295 [3%]	9275 [97%]
1048575	268 [3%]	9388 [97%]

(d) VP fra

Flow Label	Number of Consistent Destinations	Number of Divergent Destinations
0	443 [4%]	10047 [96%]
255	556 [5%]	9731 [95%]
65280	528 [5%]	9836 [95%]
983040	506 [5%]	9636 [95%]
1048575	471 [5%]	9699 [95%]

(e) VP lon

Flow Label	Number of Consistent Destinations	Number of Divergent Destinations
0	842 [7%]	10411 [93%]
255	827 [8%]	9996 [92%]
65280	820 [8%]	9955 [92%]
983040	760 [7%]	9873 [93%]
1048575	798 [8%]	9714 [92%]

(f) VP sfo

Flow Label	Number of Consistent Destinations	Number of Divergent Destinations
0	493 [4%]	10614 [96%]
255	479 [4%]	10171 [96%]
65280	488 [4%]	10541 [96%]
983040	451 [4%]	10389 [96%]
1048575	451 [5%]	9313 [95%]

(g) VP sgp

Flow Label	Number of Consistent Destinations	Number of Divergent Destinations
0	523 [4%]	11414 [96%]
255	923 [8%]	10694 [92%]
65280	1010 [9%]	10637 [91%]
983040	898 [8%]	10727 [92%]
1048575	913 [8%]	10609 [92%]

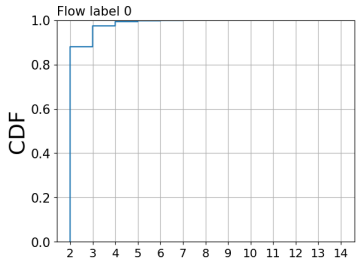
(h) VP tor

Table 5.4: Number of consistent destinations for each flow label value. The percentage of destinations compared to the total number of destinations is in brackets.

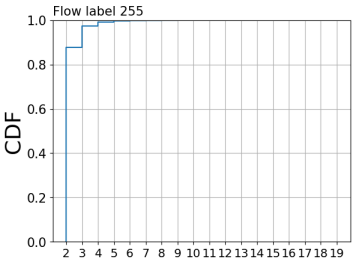
As seen in table 5.4, we are not seeing a significant change in path consistency by setting the flow label compared the base case of flow label zero (indicating a packet belonging to no flow). Based on the guidelines specified in [RFC6438] regarding which header fields should be included in flow identification hash-algorithms, this result is not unexpected, though it is disappointing due to what it means for our wanted use-case of TCP path consistency. It appears that, even with IPv6, most load balancers do not use the short 3-tuple consisting of (source IP, destination IP, flow label) to identify flows but rather most likely a tuple also containing other fields such as the TCP port numbers. This is a lot more inefficient compared to IPv4 as routers have to walk the IPv6 extension chain in order to reach the TCP header. Our hypothesis of load balancers only using part of the flow label in their hash calculations appears to not be correct, as the number of consistent and divergent destinations is fairly consistent across all VPs and flow label values. Only for VP Toronto (tor) we see a difference in the number of consistent destinations for flows with a non-zero flow label. Our measurement study shows that the number of consistent paths are doubled for all non-zero values. Further testing may be needed to confirm whether this result holds for other VPs deployed at the same site and over an extended period of time.

5.4.2 Diving Deeper Into Divergent Paths

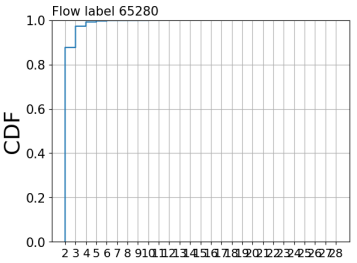
Figures 5.2 through 5.9 display the hop number at which a path divergence was first detected between two traces to the same destination and with the same flow label. As we can see, most traces that diverge(around 90%), do so already at hop number two. What we can divulge from this is that 1) A load balancer exists inside or at the edge of our VP's networks, and 2) This load balancer is either a per-packet load balancer distributing packets in a manner purely to maintain an even load, or is a per-flow load balancer that does not recognize the 3-tuple as a valid flow. Since the likelihood of it being a per-packet load balancer is very low, both due to their low prevalence in the Internet, and because it would create TCP segment reordering, the most likely explanation for this behaviour is simply that the router at hop 1 is a per-flow load balancer configured in such a way that values from the TCP segment header are included in its hash-calculations.



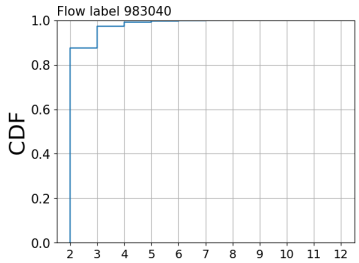
(a) Flow label 0



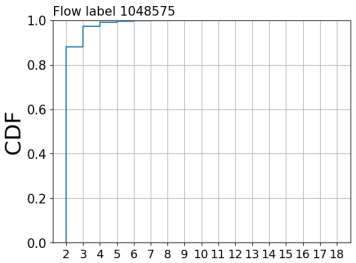
(b) Flow label 255



(c) Flow label 65280

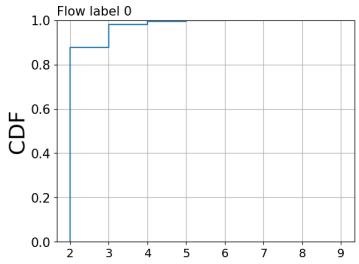


(d) Flow label 983040

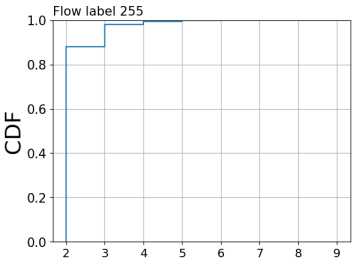


(e) Flow label 1048575

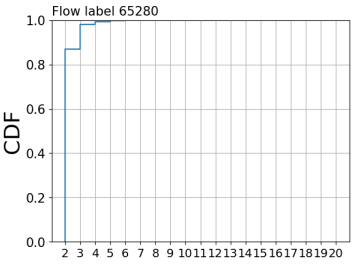
Figure 5.2: Hop number where divergence occurred for vantage point ams



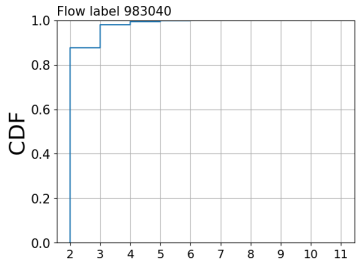
(a) Flow label 0



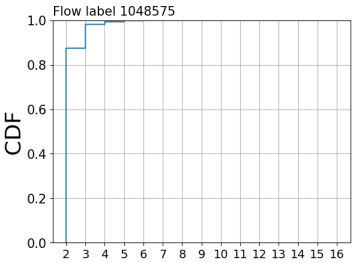
(b) Flow label 255



(c) Flow label 65280

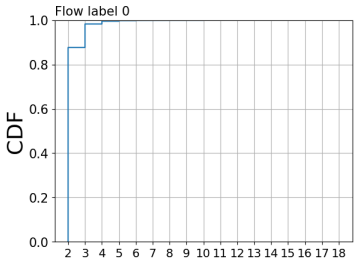


(d) Flow label 983040

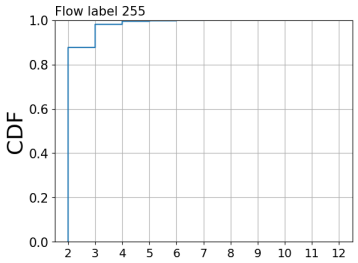


(e) Flow label 1048575

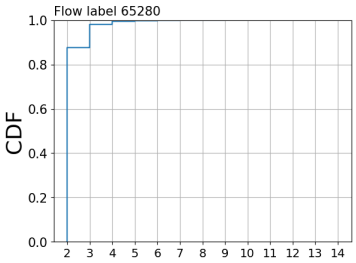
Figure 5.3: Hop number where divergence occurred for vantage point blr



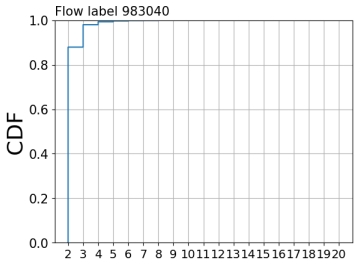
(a) Flow label 0



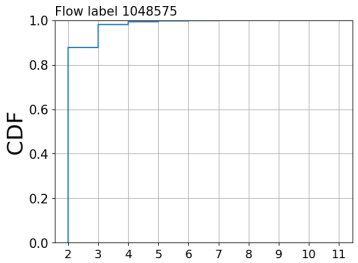
(b) Flow label 255



(c) Flow label 65280

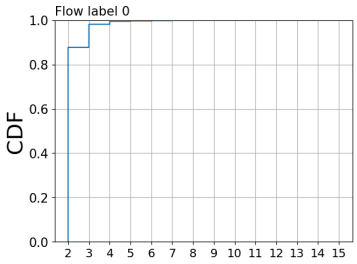


(d) Flow label 983040

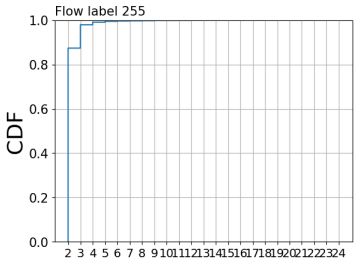


(e) Flow label 1048575

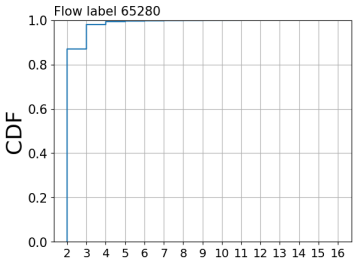
Figure 5.4: Hop number where divergence occurred for vantage point fra



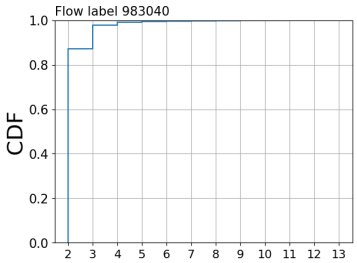
(a) Flow label 0



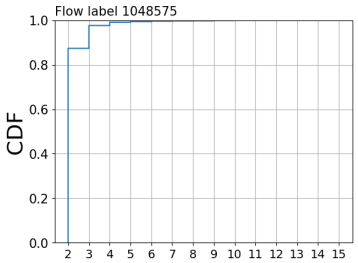
(b) Flow label 255



(c) Flow label 65280

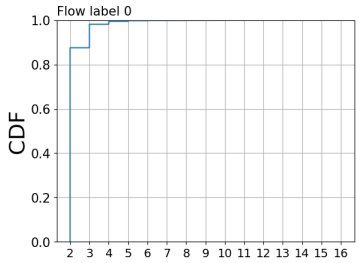


(d) Flow label 983040

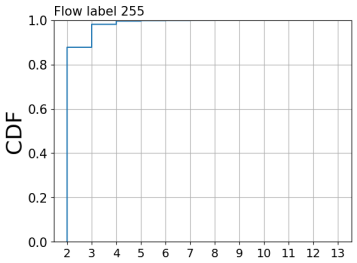


(e) Flow label 1048575

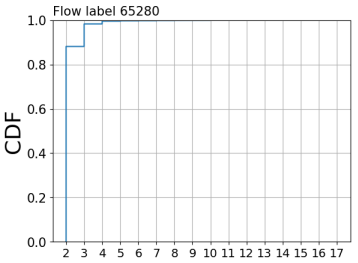
Figure 5.5: Hop number where divergence occurred for vantage point lon



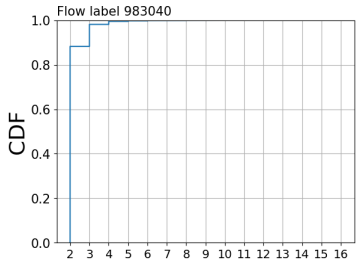
(a) Flow label 0



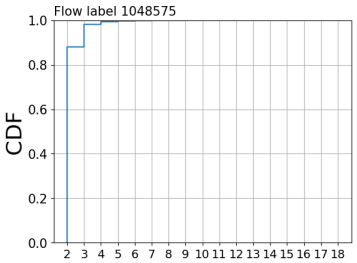
(b) Flow label 255



(c) Flow label 65280

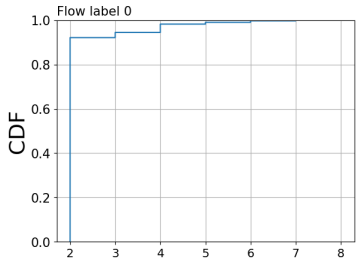


(d) Flow label 983040

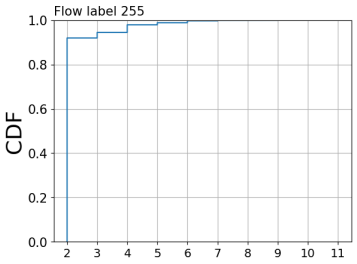


(e) Flow label 1048575

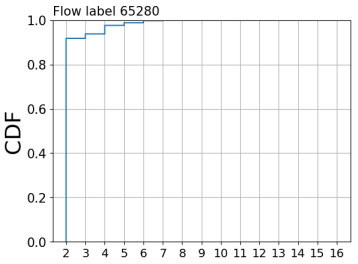
Figure 5.6: Hop number where divergence occurred for vantage point nyc



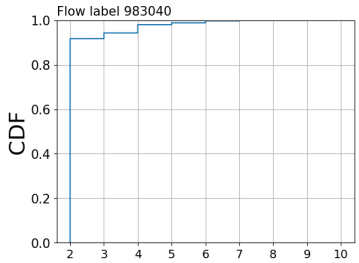
(a) Flow label 0



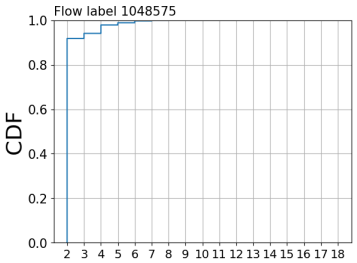
(b) Flow label 255



(c) Flow label 65280

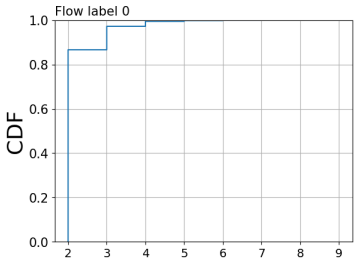


(d) Flow label 983040

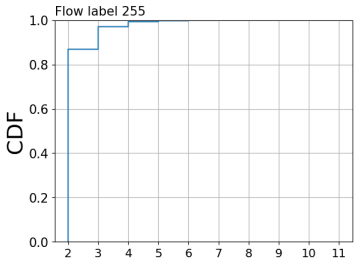


(e) Flow label 1048575

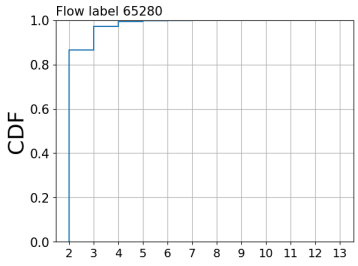
Figure 5.7: Hop number where divergence occurred for vantage point sfo



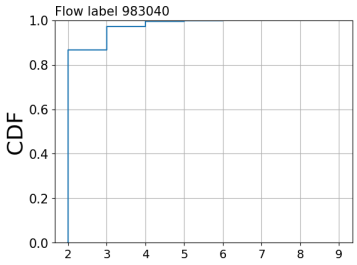
(a) Flow label 0



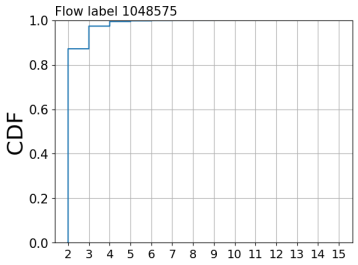
(b) Flow label 255



(c) Flow label 65280

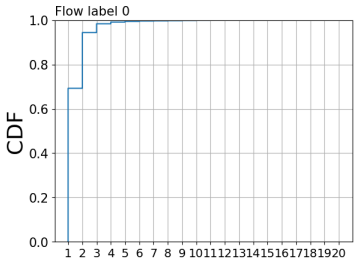


(d) Flow label 983040

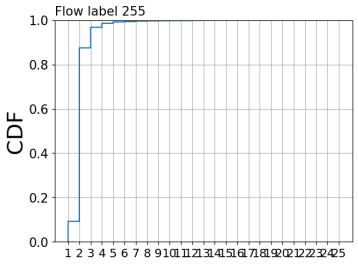


(e) Flow label 1048575

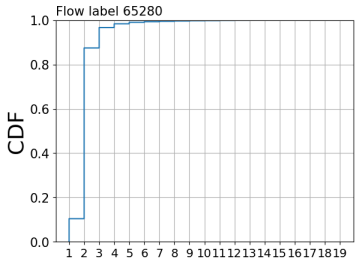
Figure 5.8: Hop number where divergence occurred for vantage point sgp



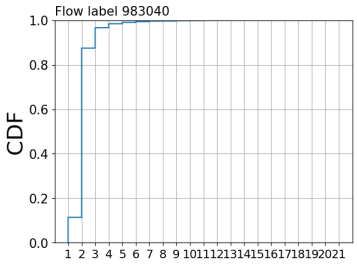
(a) Flow label 0



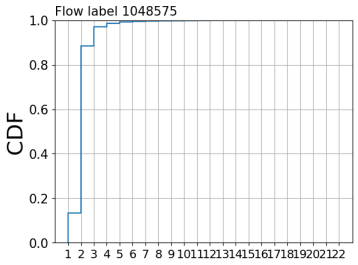
(b) Flow label 255



(c) Flow label 65280



(d) Flow label 983040



(e) Flow label 1048575

Figure 5.9: Hop number where divergence occurred for vantage point tor

5.4.3 Network Characteristics of Observed Load Balancers

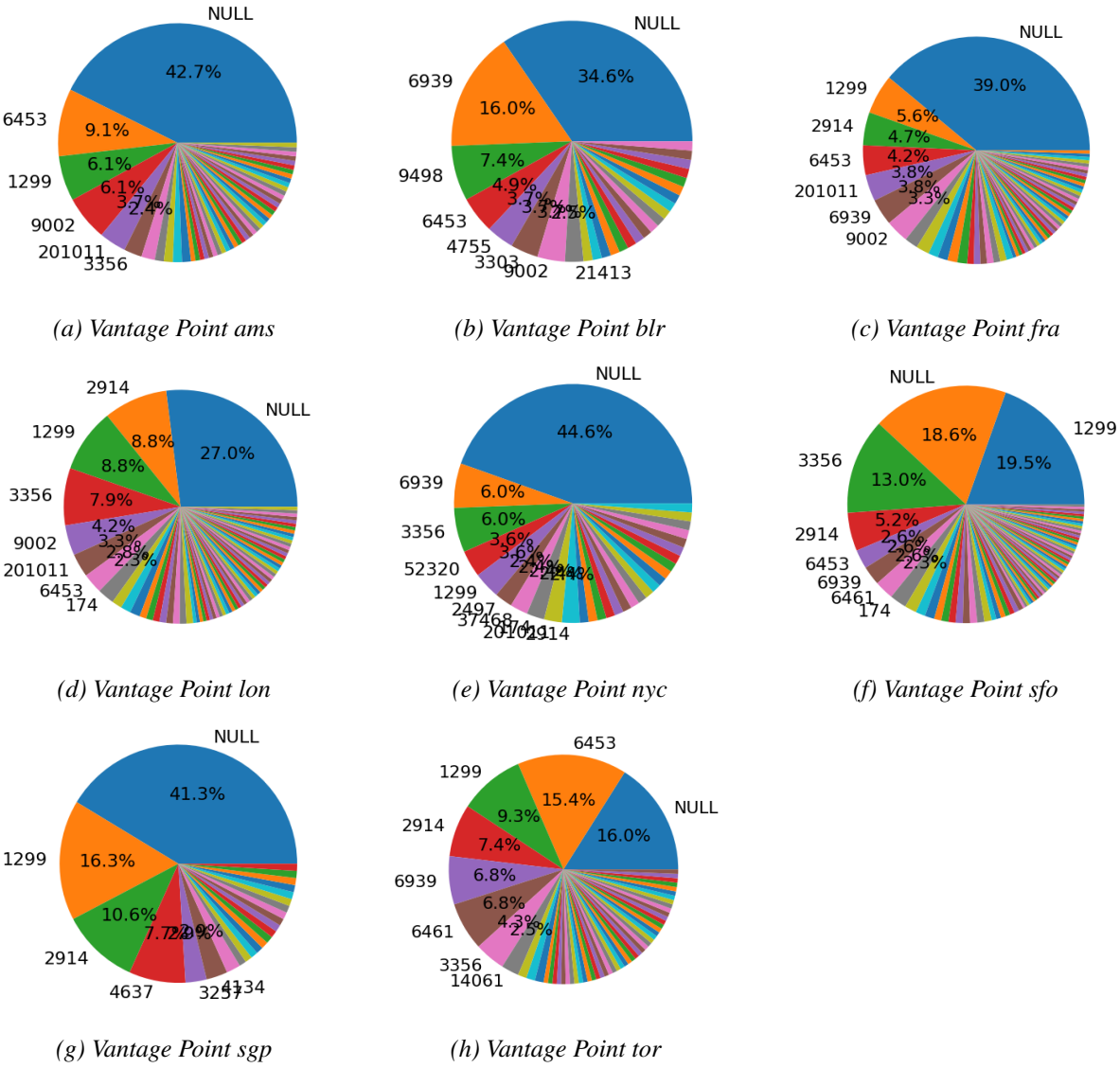
In total, we observed 1329 unique load balancers (LBs) which caused path divergence. The number of LBs observed varied greatly by VP, where the Bangalore VP had the lowest number of observed LBs, and the San Francisco VP had the highest number of observed LBs. Table 5.5 illustrates the complete number of observed LBs per VP. To deduce whether a path contained a path-changing LB, we compare the two parallel paths in a sequential, hop-by-hop manner and check for equality. If a path diverged at some hop N, we know that the offender must be hop N-1. However, in cases where we observed a path divergence at hop N and there is a gap

in responses between hop N and the last observed hop where the paths where equal, which we can call hop X, we do not know where in the path between X and N the load balancer exists. In this case, the answer is ambiguous and we do not add any device to the list of observed load balancers.

Table 5.5: Number of load balancers observed per Vantage Point.

Vantage Point	Number of unique load balancers where paths have diverged
nyc	83
lon	215
ams	164
blr	81
sfo	307
sgp	104
fra	213
tor	162
Total	1329

We perform ASN lookups of all observed load balancers. As described in section 5.4.2, we observe diverging paths already at hop number 2 or 3 in around 80 to 90% of target destination, meaning that the paths are very often load balanced by the default gateway (hop 1) or the subsequent gateway (hop 2). Unfortunately, the default gateway proved non-responsive to TTL=1 probing in our testing, which means that this IP-address is not included in our dataset. This is an important caveat when moving to our next set of figures, 5.10. This figure shows the ASN distribution of the observed load balancers where path divergence occurred. Since hop number 1 is not included in the dataset, which would have been by far the most prevalent, the DigitalOcean ASN 14061 is not as seen in the illustrated data.



(a) Vantage Point ams *(b) Vantage Point blr* *(c) Vantage Point fra*

(d) Vantage Point lon *(e) Vantage Point nyc* *(f) Vantage Point sfo*

(g) Vantage Point sgp *(h) Vantage Point tor*

Figure 5.10: ASN distribution of load balancers where a path divergence occurred. NULL is displayed for ASNs not included in the RouteViews prefix-to-AS dataset.

Chapter 6

Conclusion

In this section we conclude our findings. We provide a brief summary of the research findings and whether we have answered our research questions. We will then briefly discuss future work before finalizing with closing remarks.

6.1 Research Findings

Flow Label Persistence

In 1,246,156 traceroute traces to 15757 target addresses performed using 8 distinct vantage points positioned at differing data centers around the world, we did not detect a single occurrence of the flow label being changed en-route. While the specification discourages changing the flow label, it *does* allow changes to the flow label as protection against covert channel attacks [RFC6437]. This is promising news for router hardware designers and IPv6 network administrators who may wish to rely exclusively on the network-layer 3-tuple for faster and more efficient flow identification, as they may be secure in knowing that the flow label is unlikely to get zeroed or modified along the way to its destination.

Path Consistency

We did not find a significant change in path consistency when setting the flow label compared to the zero flow label base case (indicating a packet not belonging to any flow). While one of our vantage points demonstrated a more than a doubling in path consistency with a non-zero flow label, the overall results overwhelmingly point to the conclusion that the 3-tuple (source IP, destination IP, flow label) is not a flow identifier that is in-use in today's IPv6 Internet landscape.

6.2 Future Work

In the future, it would be interesting to repeat the tests from a variety of vantage points connected to differing Internet Service Providers (ISPs) to investigate whether these results are unique to DigitalOcean. Particularly of interest may be ISPs stationed in the great firewall of China, as its stringent security measures may lead to an increase in the flow label being modified. A repeat of the path consistency measurement study on a controlled network with

ground-truth topology may be needed to completely rule out per-packet load balancers affecting the measurement result. In addition, a measurement study on which IPv6 header fields are used for load balancing may be necessary for continued work.

6.3 Closing Remarks

In conclusion, flow label usage as it stands today appears to be in its infancy. We believe that, as the world progresses to move ever-faster toward increased IPv6 usage, using the less CPU-intensive network-layer 3-tuple for flow identification will only become more relevant. If this were to become common and operating systems and applications would begin to use the flow label more frequently, it could lead to an increase in validity for our proposed methodology of enforcing path consistency. Such a mechanism could eventually lead to ever-higher performance increases when used in conjunction with mechanisms such as Islam et al.'s ctrlTCP.

Chapter 7

Acknowledgements

I would like to thank Safiqul Islam and Michael Welzl for their great guidance in the process of making this thesis. I would also like to thank my parents and family for their unending love and support. And finally, I would like to thank my place of work Arkivverket for giving me time off for finishing this thesis.

Appendix A

Source Code

All modified Paris traceroute source code is publicly available on GitHub: https://github.com/meeps44/libparistraceroute.
Python- and shell-scripts and text files created as part of this thesis are available at: https://github.com/meeps44/scripts.
Jupyter notebooks created for data analysis are available at: https://github.com/meeps44/scripts/tree/main/stats/jupyter-notebooks
The University of Oregon's RouteViews prefix-to-AS dataset can be found here: https://publicdata.caida.org/datasets/routing/routeviews6-prefix2as/2022/05/.
The IPv6 Hitlist Service: https://ipv6hitlist.github.io/

Bibliography

[1] About routeviews. http://www.routeviews.org/routeviews/index.php/about/. [Online; accessed April 2023]. 3.1.3

[2] Caida ipv4 routed /24 topology dataset. https://www.caida.org/catalog/datasets/ipv4_routed_24_topology_dataset/. [Online; accessed April 2023]. 2.4.4

[3] Configuring unequal cost multiple path. https://support.huawei.com/enterprise/en/doc/EDOC1000154770/f19d5a22/configuring-unequal-cost-multiple-path. [Online; accessed May 2023]. 2.3.3

[4] Ip to asn mapping service. https://www.team-cymru.com/ip-asn-mapping. [Online; accessed May 2023]. 3.1.3

[5] Ipv6 tunneling feature overview and configuration guide. https://www.alliedtelesis.com/sites/default/files/documents/configuration-guides/ipv6_tunnel_feature_overview_guide.pdf. [Online; accessed April 2023]. 2.3.7

[6] Pysubnettree - a python module for cidr lookups. https://github.com/zeek/pysubnettree. [Online; accessed April 2023]. 4.3.1

[7] Routeviews collectors. http://www.routeviews.org/routeviews/index.php/collectors/. [Online; accessed April 2023]. 3.1.3

[8] Routeviews prefix-to-as mappings (pfx2as) for ipv4 and ipv6. https://publicdata.caida.org/datasets/routing/routeviews6-prefix2as/2022/05/. [Online; obtained May 5th 2022 at 12:00]. 3.1.3, 4.2.3

[9] Terraform. https://www.terraform.io/. [Online; accessed April 2023]. 4.3.2

[10] Ieee standard for local and metropolitan area networks–link aggregation. *IEEE Std 802.1AX-2008*, pages 1–163, 2008. 2.3.4

[11] State of ipv6 deployment. https://www.internetsociety.org/wp-content/uploads/2018/06/2018-ISOC-Report-IPv6-Deployment.pdf, 2018. [Online; accessed April 2023]. 2.1.3

[12] Ark ipv6 topology dataset. https://www.caida.org/catalog/datasets/ipv6_allpref_topology_dataset/, 2022. [Online; accessed May 2023]. 3.1.3

[13] Routing information service. https://www.ripe.net/analyse/internet-measurements/routing-information-service-ris, 2022. [Online; accessed April 2023]. 3.1.3

[14] prtraceroute. https://web.archive.org/web/20021212065836/http://irr.canet3.net/doc/manuals/prtraceroute.html, April 1996. [Online; accessed April 2023]. 2.4.3

[15] University of oregon routeviews project. https://http://www.routeviews.org/, April 2022. [Online; accessed April 2022]. 3.1.3

[16] World - autonomous system number statistics. https://www-public.imtbs-tsp.eu/~maigron/rir-stats/rir-delegations/world/world-asn-by-number.html, April 2023. [Online; accessed May 2023]. 5.1.4

[17] Macroscopic topology measurements. https://www.caida.org/projects/macroscopic/, August 2018. [Online; accessed May 2023]. 2.4.3

[18] Free pool of ipv4 address space depleted. https://www.nro.net/ipv4-free-pool-depleted, February 2011. [Online; accessed May 2023]. 2.1

[19] M. Allman, V. Paxson, and E. Blanton. Tcp congestion control. RFC 5681, RFC Editor, September 2009. http://www.rfc-editor.org/rfc/rfc5681.txt. 2.2.1

[20] S. Amante, B. Carpenter, S. Jiang, and J. Rajahalme. Ipv6 flow label specification. RFC 6437, RFC Editor, November 2011. 2.3.6, 3.1, 5.2, 6.1

[21] B. Augustin, X. Cuvellier, B. Orgogozo, F. Viger, T. Friedman, M. Latapy, C. Magnien, and R. Teixeira. Avoiding traceroute anomalies with paris traceroute. In *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, pages 153–158, 2006. 1.2, 2.4.3, 2.4.3, 2.4.3, 3, 3.1.3, 4.1.1

[22] B. Augustin, T. Friedman, and R. Teixeira. Measuring load-balanced paths in the internet. In *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*, pages 149–160, 2007. 2.3.2

[23] F. Baker. Requirements for ip version 4 routers. RFC 1812, RFC Editor, June 1995. 2.4.1, 3.3

[24] H. Balakrishnan and S. Seshan. The congestion manager. RFC 3124, RFC Editor, June 2001. http://www.rfc-editor.org/rfc/rfc3124.txt. 2.2.3, 2.2.3

[25] A. Barberio. Dublin traceroute. https://dublin-traceroute.net/. [Online; accessed April 2023]. 2.4.3

[26] R. Barik, M. Welzl, A. Elmokashfi, T. Dreibholz, S. Islam, and S. Gjessing. On the utility of unregulated ip diffserv code point (dscp) usage by end systems. *Performance Evaluation*, 135:102036, 2019. 2.4.4, 5.1.3

[27] R. Bonica, D. Gan, D. Tappan, and C. Pignataro. Icmp extensions for multiprotocol label switching. RFC 4950, RFC Editor, August 2007. 2.4.2

[28] J. Border, M. Kojo, J. Griner, G. Montenegro, and Z. Shelby. Performance enhancing proxies intended to mitigate link-related degradations. RFC 3135, RFC Editor, June 2001. 2.3.1

[29] B. Carpenter and S. Amante. Using the ipv6 flow label for equal cost multipath routing and link aggregation in tunnels. RFC 6438, RFC Editor, November 2011. 2.3.7, 2.3.7, 3.1, 5.4.1

[30] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman. Planetlab: an overlay testbed for broad-coverage services. *ACM SIGCOMM Computer Communication Review*, 33(3):3–12, 2003. 5.1.1

[31] A. Conta and S. Deering. Generic packet tunneling in ipv6 specification. RFC 2473, RFC Editor, December 1998. http://www.rfc-editor.org/rfc/rfc2473.txt. 2.3.7

[32] A. Conta and S. Deering. Internet control message protocol (icmpv6) for the internet protocol version 6 (ipv6) specification. RFC 2463, RFC Editor, December 1998. 2.1.4

[33] A. Conta, S. Deering, and M. Gupta. Internet control message protocol (icmpv6) for the internet protocol version 6 (ipv6) specification. RFC 4443, RFC Editor, March 2006. http://www.rfc-editor.org/rfc/rfc4443.txt. 4.2.2

[34] G. V. de Velde, T. Hain, R. Droms, B. Carpenter, and E. Klein. Local network protection for ipv6. RFC 4864, RFC Editor, May 2007. 2.1.1

[35] S. Deering and R. Hinden. Internet protocol, version 6 (ipv6) specification. STD 86, RFC Editor, July 2017. 2.1

[36] S. E. Deering and R. M. Hinden. Internet protocol, version 6 (ipv6) specification. RFC 2460, RFC Editor, December 1998. http://www.rfc-editor.org/rfc/rfc2460.txt. 2.1, 2.1.2, 2.4.1, 3.3

[37] A. Dhamdhere, D. D. Clark, A. Gamero-Garrido, M. Luckie, R. K. Mok, G. Akiwate, K. Gogia, V. Bajpai, A. C. Snoeren, and K. Claffy. Inferring persistent interdomain congestion. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 1–15, 2018. 2.4.4

[38] S. Dhesikan, D. Druta, P. Jones, and C. Jennings. Dscp packet markings for webrtc qos. *Internet Engineering Task Force, Internet-Draft draft-ietf-tsvwg-rtcweb-qos-18*, 2016. 2.4.4

[39] R. Eddy. prtraceroute, the most recent version is part of the internet systems consortiums irrtoolset, available from: http://www.isc.org/. https://web.archive.org/web/20021212065836/http://irr.canet3.net/doc/manuals/prtraceroute.html, August 1994. [Online; accessed April 2023]. 3.1.3

[40] L. Eggert, J. Heidemann, and J. Touch. Effects of ensemble-tcp. *ACM SIGCOMM Computer Communication Review*, 30(1):15–29, 2000. 2.2.3

[41] K. R. Fall and W. R. Stevens. *TCP/IP Illustrated, Vol 1. The Protocols*, chapter 8. AddisonWesley, 2 edition, 2011. 2.4.1

[42] A. Ford, C. Raiciu, M. Handley, and O. Bonaventure. Tcp extensions for multipath operation with multiple addresses. RFC 6824, RFC Editor, January 2013. http://www.rfc-editor.org/rfc/rfc6824.txt. 2.2.4

[43] O. Gasser, Q. Scheitle, P. Foremski, Q. Lone, M. Korczynski, S. D. Strowes, L. Hendriks, and G. Carle. Clusters in the expanse: Understanding and unbiasing ipv6 hitlists. In *Proceedings of the 2018 Internet Measurement Conference*, New York, NY, USA, 2018. ACM. 1.2, 3.1.3

[44] E. Gavron. Nanog traceroute. http://man.he.net/man8/traceroute-nanog, May 1995. [Online; accessed April 2023]. 2.4.3

[45] T. Hain. Architectural implications of nat. RFC 2993, RFC Editor, November 2000. 2.1.1

[46] D. Hayes, S. Ferlin, M. Welzl, and K. Hiorth. Shared bottleneck detection for coupled congestion control for rtp media. Internet-Draft draft-ietf-rmcat-sbd-09, IETF Secretariat, November 2017. 2.2.3

[47] T. Henderson, S. Floyd, A. Gurtov, and Y. Nishida. The newreno modification to tcp's fast recovery algorithm. RFC 6582, RFC Editor, April 2012. http://www.rfc-editor.org/rfc/rfc6582.txt. 2.2.2

[48] R. Hinden and S. Deering. Ip version 6 addressing architecture. RFC 4291, RFC Editor, February 2006. http://www.rfc-editor.org/rfc/rfc4291.txt. 2.1.1

[49] C. Hopps. Analysis of an equal-cost multi-path algorithm. RFC 2992, RFC Editor, November 2000. 2.3.2, 2.3.3

[50] S. Islam and M. Welzl. Start me up: Determining and sharing tcp's initial congestion window. In *Proceedings of the 2016 Applied Networking Research Workshop*, pages 52–54, 2016. 2.2.3

[51] S. Islam, M. Welzl, K. Hiorth, D. Hayes, G. Armitage, and S. Gjessing. ctrltcp: Reducing latency through coupled, heterogeneous multi-flow tcp congestion control. In *IEEE INFOCOM 2018-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 214–219. IEEE, 2018. 1.1, 1.2, 2.2.3, 3

[52] V. Jacobson. Congestion avoidance and control. *ACM SIGCOMM computer communication review*, 18(4):314–329, 1988. 2.2, 2.2.1

[53] V. Jacobson. Traceroute. *ftp://ftp. ee. lb1. gov/traceroute. tar. Z*, 1989. 2.4.1

[54] V. Jacobson. Freebsd manual pages - traceroute. https://man.freebsd.org/cgi/man.cgi?query=traceroute&manpath=FreeBSD+13.2-RELEASE+and+Ports, 2020. [Online; accessed May 2023]. 5.3.1

[55] S. Kent and R. Atkinson. Ip authentication header. RFC 2402, RFC Editor, November 1998. 2.1.2

[56] S. Kent and R. Atkinson. Security architecture for the internet protocol. RFC 2401, RFC Editor, November 1998. 2.1.1

[57] S. Knowles. Iesg advice from experience with path mtu discovery. RFC 1435, RFC Editor, March 1993. 3.3

[58] R. Krishnan, L. Yong, A. Ghanwani, N. So, and B. Khasnabish. Mechanisms for optimizing link aggregation group (lag) and equal-cost multipath (ecmp) component link utilization in networks. RFC 7424, RFC Editor, January 2015. 2.3.4

[59] A. N. Kuznetsov. tc - show / manipulate traffic control settings. https://manned.org/tc. [Online; accessed April 2023]. 4.3.3

[60] K. Lahey. Tcp problems with path mtu discovery. RFC 2923, RFC Editor, September 2000. http://www.rfc-editor.org/rfc/rfc2923.txt. 3.3

[61] A. Lakhina, J. W. Byers, M. Crovella, and P. Xie. Sampling biases in ip topology measurements. In *IEEE INFOCOM 2003. Twenty-second Annual Joint Conference of the IEEE Computer and Communications Societies (IEEE Cat. No. 03CH37428)*, volume 1, pages 332–341. IEEE, 2003. 3.1.3, 3.3, 5.1.1

[62] M. Luckie. Scamper: a scalable and extensible packet prober for active measurement of the internet. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, pages 239–245, 2010. 2.4.3, 4.1.1, 4.2.5

[63] M. Luckie. Freebsd file formats manual, warts format for scampers warts storage. https://www.caida.org/catalog/software/scamper/man/warts.5.pdf, 2011. [Online; accessed May 2023]. 4.2.5

[64] M. Luckie, A. Dhamdhere, D. Clark, B. Huffaker, and K. Claffy. Challenges in inferring internet interdomain congestion. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, pages 15–22, 2014. 2.4.4

[65] M. Luckie, A. Dhamdhere, B. Huffaker, D. Clark, and K. Claffy. Bdrmap: Inference of borders between ip networks. In *Proceedings of the 2016 Internet Measurement Conference*, pages 381–396, 2016. 2.4.4

[66] A. Marder and J. M. Smith. Map-it: Multipass accurate passive inferences from traceroute. In *Proceedings of the 2016 Internet Measurement Conference*, pages 397–411, 2016. 2.4.4, 3.1.3, 3.1.3, 5.3

[67] J. Mogul and S. Deering. Path mtu discovery. RFC 1191, RFC Editor, November 1990. http://www.rfc-editor.org/rfc/rfc1191.txt. 2.4.3

[68] T. Narten, R. Draves, and S. Krishnan. Privacy extensions for stateless address autoconfiguration in ipv6. RFC 4941, RFC Editor, September 2007. 2.1.1

[69] T. Narten, E. Nordmark, W. Simpson, and H. Soliman. Neighbor discovery for ip version 6 (ipv6). RFC 4861, RFC Editor, September 2007. http://www.rfc-editor.org/rfc/rfc4861.txt. 2.1.1

[70] A. B. Network. Internet2, 2003. 2.4.4, 5.1.1

[71] K. Nichols, S. Blake, F. Baker, and D. L. Black. Definition of the differentiated services field (ds field) in the ipv4 and ipv6 headers. RFC 2474, RFC Editor, December 1998. http://www.rfc-editor.org/rfc/rfc2474.txt. 2.1, 2.4.4

[72] C. Perkins, D. Johnson, and J. Arkko. Mobility support in ipv6. RFC 6275, RFC Editor, July 2011. http://www.rfc-editor.org/rfc/rfc6275.txt. 2.1.2

[73] L. Peterson. It's been a fun ride, March 2020. 5.1.1

[74] J. Postel. Internet Control Message Protocol. RFC 792 (Internet Standard), Sept. 1981. Updated by RFCs 950, 4884, 6633, 6918. 2.1.4

[75] J. Postel. Internet Protocol. RFC 791 (Internet Standard), Sept. 1981. Updated by RFCs 1349, 2474, 6864. 2.4.4

[76] J. Postel. Transmission Control Protocol. RFC 793 (Internet Standard), Sept. 1981. Updated by RFCs 1122, 3168, 6093, 6528. 2.2

[77] J. Rajahalme, A. Conta, B. Carpenter, and S. Deering. Ipv6 flow label specification. RFC 3697, RFC Editor, March 2004. 2.3.7, 2.3.7, 2.3.7

[78] K. Ramakrishnan, S. Floyd, and D. Black. The addition of explicit congestion notification (ecn) to ip. RFC 3168, RFC Editor, September 2001. http://www.rfc-editor.org/rfc/rfc3168.txt. 2.1

[79] Y. Rekhter, T. Li, and S. Hares. A border gateway protocol 4 (bgp-4). RFC 4271, RFC Editor, January 2006. http://www.rfc-editor.org/rfc/rfc4271.txt. 3.1.3, 3.1.3

[80] E. Rosen, A. Viswanathan, and R. Callon. Multiprotocol label switching architecture. RFC 3031, RFC Editor, January 2001. http://www.rfc-editor.org/rfc/rfc3031.txt. 2.3.5

[81] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems (TOCS)*, 2(4):277–288, 1984. 2.1

[82] R. A. Steenbergen. A practical guide to (correctly) troubleshooting with traceroute. https://archive.nanog.org/sites/default/files/traceroute-2014.pdf, 2014. [Online; accessed May 2023]. 2.4.1, 3.3

[83] M. Toren. tcptraceroute. https://web.archive.org/web/20080415084649/http://michael.toren.net/code/tcptraceroute/tcptraceroute.8.html, April 2001. [Online; accessed April 2023]. 2.4.1

[84] F. Viger, B. Augustin, X. Cuvellier, C. Magnien, M. Latapy, T. Friedman, and R. Teixeira. Detection, understanding, and prevention of traceroute measurement artifacts. *Computer networks*, 52(5):998–1018, 2008. 3.2.2, 5.3, 5.3.2

[85] J. Xia, L. Gao, and T. Fei. Flooding attacks by exploiting persistent forwarding loops. In *Proceedings of the 5th ACM SIGCOMM conference on Internet Measurement*, pages 36–36, 2005. 3.1.3