Network Fault Localization for the InterEdge

Matthew Fogel

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2025-54
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2025/EECS-2025-54.html

May 14, 2025

Network Fault Localization for the InterEdge

Matthew Fogel

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science**.

Approval for the Report and Comprehensive Examination:

5/9/2025

Professor Scott Shenker, Research Advisor — Date

5/9/2025

Professor Sylvia Ratnasamy, Second Reader — Date

Abstract

Two challenges the modern Internet faces are the architectural stagnation that has widened the performance gap between private and public networks, and the increasing difficulty of diagnosing failures across distributed systems. The InterEdge architecture addresses the first challenge by enabling standardized in-network services without compromising Internet compatibility, while the "Where's the Fault?" (WTF) methodology tackles the second by designing cross-domain fault localization methods. This paper implements the WTF methodology within the InterEdge project to enable cross-domain and cross-layer fault identification. Lightweight scope requests traverse network paths based on historical routing states, which provides users and applications with a standardized mechanism to determine where faults occur without requiring extensive instrumentation or compromising proprietary information. The effectiveness of this approach is demonstrated through targeted test cases that successfully identify network path failures, node crashes, and service-level issues. This can help to reduce the time and complexity involved in troubleshooting distributed applications while maintaining compatibility with the InterEdge's existing service-oriented architecture. While further work and challenges remain, this work is progress toward addressing the growing complexity of diagnosing faults in modern Internet services. This work provides a guide for InterEdge service developers as well as a foundation for future enhancements.

Contents

1 Background **4**
- 1.1 Modern Internet Architecture . . . 4
- 1.2 Localizing Faults . . . 5

2 Related Work **6**
- 2.1 The InterEdge Project (formerly The Extensible Internet Project) . . . 6
  - 2.1.1 Architectural Stagnation and the Need for an Extensible Internet 6
  - 2.1.2 Core Contributions of the InterEdge Project . . . 6
  - 2.1.3 Technical Architecture . . . 7
  - 2.1.4 Comparison with Existing Approaches . . . 9
- 2.2 The "Where's the Fault?" Project . . . 9
  - 2.2.1 Cross-Domain and Cross-Layer Fault Localization . . . 9
  - 2.2.2 Comparison with Existing Fault Localization Techniques . . . 10
  - 2.2.3 Advantages and Novel Contributions . . . 11

3 Overview **12**
- 3.1 Goal . . . 12
- 3.2 User Flow . . . 12

4 Methodology **13**
- 4.1 Existing Development Structure . . . 13
- 4.2 Enabling Scope Requests . . . 13
  - 4.2.1 Background . . . 13
  - 4.2.2 Application Changes . . . 13
  - 4.2.3 Host Stack Changes . . . 14
  - 4.2.4 Service Module Data-Plane Changes . . . 14
  - 4.2.5 Service Module Control-Plane Changes . . . 15
  - 4.2.6 More Service Module Data-Plane Changes . . . 15
- 4.3 Handling Historical Routing State . . . 16

5 Results **17**
- 5.1 Testing . . . 17
- 5.2 Demonstrated Use Cases . . . 17
- 5.3 Limitations . . . 18

6 Future Work **19**

7 Conclusion **20**

References **21**

List of Figures

2.1 Core Components of the InterEdge Architecture [1] 8
2.2 ILP header and processing at an SN [1] 9

4.1 Example Scope Request Flow for HWMS 14

Acknowledgments

I would like to thank Professor Scott Shenker for advising me during my Master's studies. I would also like to thank Emily Marx for her help and guidance throughout this project. Finally, I would like to thank the other members of the Berkeley NetSys Lab for fostering a welcoming research community.

1. Background

1.1 Modern Internet Architecture

The Internet has undergone remarkable growth since its inception over thirty years ago, evolving from a small research-focused network to a global infrastructure that currently supports billions of daily users [2]. This expansion has been driven not only by an increase in the number of users, but also by the growing diversity and complexity of applications. These have imposed different demands and constraints. Modern Internet applications, ranging from real-time video streaming to large-scale cloud computing and Internet of Things (IoT) devices, require low-latency, high-bandwidth, and reliable connectivity. Meeting these demands has required continuous innovation in networking technologies that are part of the layered Internet architecture.

As the Internet has scaled, its original design principles – particularly interconnection and the end-to-end principle – have been increasingly supplemented by additional in-network functionality in order to enhance performance, security, and privacy [1]. A key development in this evolution has been the rise of edge networking, where computation and data processing are shifted closer to users to reduce latency and improve efficiency. While these edge-based solutions provide significant benefits, they also introduce new architectural shifts that diverge from the original decentralized, neutral model of the Internet. The end-to-end principle originally dictated that the network should provide only best-effort packet delivery, leaving all additional functionality to the communicating endpoints. However, modern network infrastructure now incorporates various in-network enhancements, such as content delivery networks (CDNs) and software-defined wide area networks (SD-WANs). These introduce application-layer functionality between endpoints. Similarly, interconnection, which enables a global network by linking disparate systems through standardized protocols and peering agreements, has not extended to these new edge services. Different edge service providers (ESPs) do not interconnect their services in the same way that ISPs do for packet delivery, which leads to fragmentation and a reliance on proprietary solutions. This departure from the Internet's foundational tenets has resulted in reduced neutrality and increased reliance on private, non-interoperable infrastructure.

One of the most notable consequences of this shift is the growing disparity between the capabilities of private networks, which are often operated by large cloud and content providers, and the public Internet, which still primarily adheres to the best-effort packet delivery model. Private edge networks can implement advanced optimizations, such as intelligent traffic routing, distributed caching, and real-time data processing. This gives them significant performance and reliability advantages over the broader public Internet.

Meanwhile, public Internet infrastructure has remained largely unchanged in its core architecture, leading to an increasing reliance on proprietary edge services for critical application performance.

1.2 Localizing Faults

Modern applications depend on complex ecosystems of interconnected components, spanning cloud providers, edge service providers, ISPs, home and enterprise networks, and client devices. The distributed nature of these systems introduces significant challenges in diagnosing and resolving performance and correctness issues [3]. When a failure occurs, whether due to network congestion, misconfigurations, software bugs, or hardware faults, users often experience degraded performance or complete service outages. However, determining the root cause of these issues is particularly difficult because failures can occur at any point along the chain of dependencies, and different stakeholders operate each component with limited visibility into others. End users, lacking diagnostic tools and insight into the underlying infrastructure, are often left unsure whether the problem lies with their local network, an ISP, or an upstream service provider. Even technical operators may struggle to localize faults efficiently due to the lack of standardized cross-domain monitoring and reporting mechanisms. This gap in fault localization leads to increased downtime, frustrated users, and operational inefficiencies, which highlights the need for better techniques to identify and isolate faults across distributed systems.

Traditional fault localization methods typically operate within a single domain, such as a cloud provider's infrastructure or an enterprise network, and they rely on domain-specific logs, monitoring tools, and tracing mechanisms. However, as modern applications integrate multiple external services and networks, the challenge of diagnosing faults has expanded beyond the boundaries of any single administrative entity.

2. Related Work

2.1 The InterEdge Project (formerly The Extensible Internet Project)

2.1.1 Architectural Stagnation and the Need for an Extensible Internet

The InterEdge (IE) project [1] addresses the fragmentation of Internet services by proposing a framework that introduces an architectural change to the Internet in order to support in-network services without compromising interoperability, neutrality, or openness. As previously noted in Section 1.1, the Internet's original design principles – particularly the end-to-end principle and interconnection neutrality – have been increasingly bypassed by private optimizations. These developments have led to growing architectural stagnation, where public Internet infrastructure has remained largely unchanged while private networks have introduced significant performance enhancements. The InterEdge project argues that this stagnation is not due to a lack of innovation, but rather the inability to integrate these enhancements into the public Internet in a structured, extensible manner. Its proposal aims to bridge the gap by allowing in-networking processing within a standard, universally accessible framework, rather than relying on proprietary, closed solutions.

A key consequence of this architectural stagnation, as outlined previously, is that large private networks are effectively replacing the public Internet's role as the primary transport medium for global traffic. The InterEdge project highlights that companies such as Google, Amazon, Facebook, and Microsoft have built vast private backbone networks that bypass traditional ISPs and directly interconnect with end-user networks via points of presence (PoPs) and off-network caches. These networks optimize performance, security, and reliability in ways that the public Internet cannot match. The InterEdge framework aims to counteract this trend by enabling public Internet infrastructure to offer similar, extensible services, ensuring that the advantages of in-network processing are not confined to private networks. Specifically, it proposes introducing service nodes, which are network elements that allow packet processing beyond basic forwarding while maintaining full compatibility with existing Internet protocols.

2.1.2 Core Contributions of the InterEdge Project

1. **Enabling In-Network Services Without Breaking Compatibility**

   Unlike traditional efforts to modify the Internet's core protocols, it preserves exist-

ing IP-layer functionality while introducing a new "service layer" (Layer 3.5). This layered approach ensures that applications can incrementally adopt new services while remaining compatible with the traditional Internet model. Service Nodes (SNs) are introduced, which are deployed at network edges, such as ISP central offices, cloud PoPs, or datacenters. They provide caching, flow termination, load balancing, DDoS mitigation, and other services. Since service nodes operate above the IP layer, no fundamental changes to routing or packet forwarding are required, making it incrementally deployable.

2. **Addressing the Fragmentation of Internet Services**

   One of the major challenges identified previously is that different edge service providers do not interconnect their services the way ISPs do for packet delivery, leading to fragmentation and proprietary lock-in. The InterEdge project directly tackles this issue by establishing a common interface for invoking in-network services, allowing applications to explicitly request services such as enhanced security, congestion control, or application-aware routing. The InterEdge project allows applications to explicitly request in-network services via a standardized mechanism, as opposed to CDNs or SD-WANs, which transparently optimize traffic for their own customers. Additionally, by distinguishing between service invocation and packet forwarding, the InterEdge project ensures that applications can choose how traffic is handled without requiring network-wide protocol changes. This all reduces reliance on proprietary infrastructure while restoring interoperability between public and private networks.

3. **A Market-based Approach to Innovation**

   A major reason for the Internet's architectural stagnation has been the difficulty of deploying and standardizing new features. Traditional IETF standardization processes are slow and require broad consensus before deployment, leading to long delays in adopting new technologies (e.g., IPv6 took decades to gain adoption). The InterEdge proposes a market-driven model where new services can be deployed as open-source modules on service nodes. This allows rapid experimentation and deployment, since new services can be introduced without requiring full protocol standardization. It also allows for competitive innovation, as ISPs, cloud providers, and third parties can develop and offer new services without breaking existing applications.

2.1.3 Technical Architecture

The InterEdge is built on key service nodes (SNs) provided by InterEdge Service Providers (IESPs), and they handle application-layer functionality using commodity compute clusters rather than traditional networking hardware such as ASICs. The architecture assumes a larger and more diverse set of IESPs compared to today's ESPs, enabling better geographic coverage and competition.

The main components of the InterEdge architecture (see Figure 2.1) include:

1. **Interposition-Layer Protocol (ILP)**

   This is a new tunneling protocol that facilitates communication between SNs and between hosts and SNs.

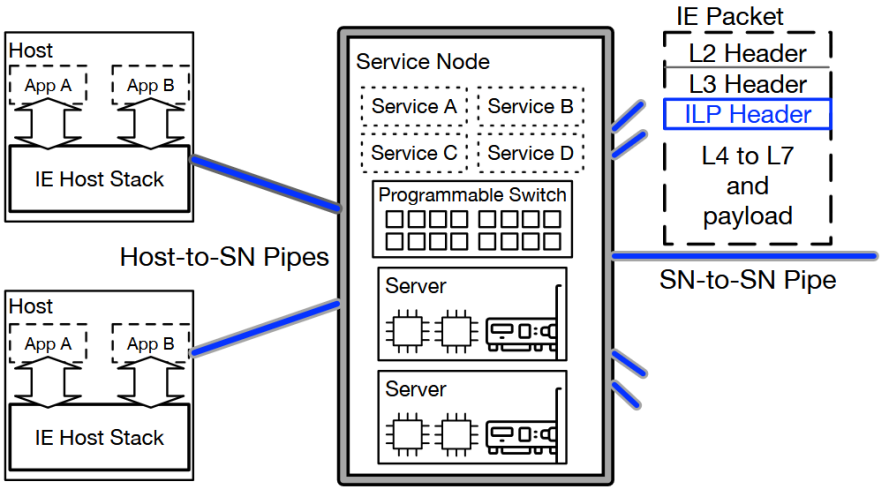

Figure 2.1: Core Components of the InterEdge Architecture [1]

2. **Service Nodes (SNs)**

   These are compute clusters that process edge services, serving as the backbone of the architecture.

3. **InterEdge Host Stack**

   This is a software stack implemented on end-hosts to facilitate interaction with the InterEdge framework.

4. **Decision Cache and Pipe-Terminus**

   These are mechanisms within SNs that optimize routing and forwarding by caching match-action pairs to improve performance.

InterEdge services can be explitly invoked by hosts, when applications request specific services through the host OS, signaling service preferences via ILP metadata. They can also be invoked with out-of-band invocation, in which hosts can use a control protocol to apply services selecctively to portions of traffic. Another method of invoking services is with third-party imposed services. Network operators (such as ISPs or enterprises) can impose services on all traffic within their domain, similar to how firewalls or SD-WANs are deployed today.

Figure 2.2 provides a detailed breakdown of the ILP header structure and the packet processing pipeline within an SN. When a packet arrives at an SN, the ILP header is first decrypted using a shared key associated with the sender (either a host or another SN). The pipe-terminus queries its decision cache using the L3 header and service/connection IDs to determine the next forwarding action. If a match is found, the packet is forwarded to its designated next hop, and if not, it is sent to the appropriate service module for further processing. The service will process the request and may update the decision cache with new forwarding rules. Before leaving the SN, the ILP header is re-encrypted using the key for the next hop, and the appropriate network headers are added for transmission.

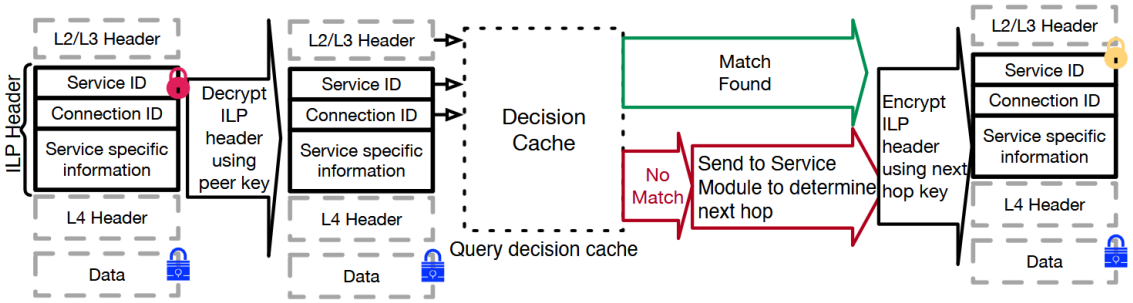

Figure 2.2: ILP header and processing at an SN [1]

2.1.4 Comparison with Existing Approaches

1. **Traditional CDN and Edge Services**

   Content delivery networks such as Akamai, Cloudflare, and Google Edge Network have long provided in-network optimizations to improve performance. However, they operate outside of the Internet's core architecture, requiring direct contracts with content providers. The InterEdge differs in that it integrates in-network processing into the internet itself, allowing any application to leverage standardized services rather than relying on proprietary CDN solutions.

2. **Software-Defined Networking (SDN) and Network Virtualization**

   Software-defined networking and network function virtualization (NFV) have introduced programmable network control within datacenters and enterprise environments. However, SDN remains domain-specific, often restricted to private cloud and enterprise networks. Additionally, NFV requires deep network modifications, making it difficult to deploy at scale. The InterEdge, in contrast, preserves existing Internet functionality while enabling service-layer programmability, offering a more practical and deployable approach to enhancing the Internet.

3. **Clean-Slate Internet Architecture**

   Several clean-slate Internet redesigns have been proposed, such as Named Data Networking (NDN) and DONA, but these require fundamental protocol changes, making them difficult to deploy incrementally. The InterEdge avoids this issue by ensuring backward compatibility, allowing for gradual adoption without disrupting existing networks.

2.2 The "Where's the Fault?" Project

2.2.1 Cross-Domain and Cross-Layer Fault Localization

The "Where's the Fault?" (WTF) project [3] aims to address the issue of localizing faults by providing a primitive that is cross-layer, cross-domain, and cross-application. WTF provides a standardized, scalable, and efficient mechanism for identifying faults across disparate networked components. Its fault localization method is lightweight and domain-agnostic. It allows different network elements to participate in a distributed diagnostic

process without exposing proprietary or sensitive telemetry data. This is achieved with the following key mechanisms and concepts:

1. **Scoping Queries**

   After a fault is detected for a specific request, an initiator triggers a scoping query containing details about the faulty request. This query is sent shortly after the fault occurs.

2. **Collectors**

   WTF assumes there is a deployment of collectors within organizations. Each collector is associated with one or more computation elements (physical or logical, e.g., routers, servers, microservices, network stacks). Collectors handle scoping queries on behalf of their elements using existing interfaces.

3. **Query Handling and Forwarding**

   Upon receiving a scoping query, a collector first determines if any of its associated elements, potentially involved with the faulty request, behaved anomalously based on element-specific logic, and existing monitoring data. These anomalous elements are flagged. It then forwards the scoping query to the next collectors.

4. **Result Aggregation**

   The response generated by individual collectors includes a set of identifier for local elements that it flagged as potentially anomalous. They are sent back to the collector that originally forwarded the query to it, and the responses are combined. The original initiator of the entire scoping query eventually receives an aggregated response, which contains a set of potentially faulty elements located within its own organization. Each involved organization ultimately obtains its own set of flagged elements. This allows each organization to investigate internally without exposing its data to others.

2.2.2 Comparison with Existing Fault Localization Techniques

1. **Single-Domain Fault Localization Methods**

   Many existing fault localization approaches focus on identifying failures within a specific administrative domain. For example, NetPoirot focuses on fault diagnosis within a datacenter network by correlating network telemetry data to infer root causes. Another example is FChain, which localizes problems in cloud environments but lacks a network-layer perspective, making it unsuitable for diagnosing end-to-end faults. While these methods are effective in their respective environments, they do not generalize well to multi-domain architectures, where application components span multiple service providers and infrastructure layers.

2. **Distributed Tracing and Telemetry-Based Methods**

   Techniques like X-Trace, Dapper, Zipkin, Jaeger, and Canopy provide distributed tracing mechanisms that instrument requests as they traverse a system. However, these approaches have limitations, such as needing to modify application code in order to insert tracing metadata, and they struggle with cross-domain visibility

because logs and trace data are typically isolated within a single provider. Additionally, many tracing systems rely on sampling strategies, which may fail to capture failures that have a low frequency of occurring. In contrast, WTF does not require code-level modifications and operates independently of specific communication protocols.

3. **Network-Centric Fault Localization Approaches**

   There are several network fault localization systems that exist, such as Packet Obituaries, which track packet losses across ISP networks; FaultPrints, which analyzes network telemetry to detect service degradations; and Tulip, which enables end-users to diagnose network path failures. These methods are effective for detecting network-layer failures but are not well-suited for application-layer fault localization, where failures may originate from distributed services rather than network infrastructures. WTF extends the fault localization beyond the network layer by incorporated application-layer and service-level elements.

2.2.3 Advantages and Novel Contributions

WTF distinguishes itself from prior work by addressing cross-domain fault localization without requiring deep instrumentation or revealing detailed internal logs. Unlike traditional telemetry-based approaches that generate large volumes of logs, WTF's compact flags drastically reduce the overhead of cross-domain fault reporting. WTF also applies to diverse domains, including microservices architectures, cloud-hosting applications, and content delivery networks (CDNs), as demonstrated in its case studies. By only exposing flags within each specific organization, WTF avoids privacy concerns associated with full-stack logging and deep packet inspection techniques, as well.

The project also acknowledges limitations and open challenges. The authors note that the fault localization of WTF is inherently coarse-grained, meaning it may identify multiple potential fault sources rather than pinpointing the exact root cause. WTF also assumes that components honestly report their health status, but in adversarial settings, misleading reports could complicate localization efforts. WTF is optimized for real-time fault location but does not focus on forensic analysis after the fact. This limits its effectiveness for diagnosing intermittent, long-term failures. Additionally, the scope of WTF is limited to detectable anomalies. There is a dependency on collector implementations to recognize deviations from normal operation, and WTF may not effectively localize faults that do not produce detectable signals. For example, subtle semantic errors in application logic or certain types of misconfigurations may not be detectable, unless they indirectly cause performance or error anomalies.

3. Overview

3.1 Goal

This project implements the design and ideas behind the WTF project into the InterEdge in order to support fault localization. In its current form, the InterEdge project has a working implementation of its core architecture described above, including InterEdge-enabled hosts and multiple services. This project will extend that functionality and allow users and applications to determine where faults might be occurring. It will also provide documentation for service developers to enable this support.

3.2 User Flow

The following user flow is designed to allow end users to quickly identify faults, reducing downtime and improving service reliability.

1. **Issue Detection**

   Users typically become aware of a problem when they experience degraded performance, increased latency, service unavailability, or outright failures. This could manifest in various ways, such as a website failing to load, streaming services buffering excessively, or API requests timing out. Upon noticing an issue, the user wants to determine whether the problem originates from their local environment, an ISP, or an upstream service provider.

2. **Initializing and Propagation of a Scope Request**

   To begin fault localization, the user issues a scope request. This scope request is the same as the packet that would have experienced the issues, but with an additional flag set. This request is then propagated across the network path associated with the affected service. It traverses the various elements in the network based on historical routing states so that it can replicate where the original packet reached. Each node it reaches sends a response back to the user. These responses can then be aggregated to provide a snapshot of how the network behaved over time.

3. **Debugging, Resolution, and Verification**

   Users can analyze the aggregated responses and based on the insights gained, they can take corrective actions.

4. Methodology

4.1 Existing Development Structure

The InterEdge currently has a working prototype built on POX [4], a networking software platform written in Python and commonly used for Software-Defined Networking (SDN) research and prototyping. POX enables dynamic network control, allowing the InterEdge to implement its service-based routing, interposition-layer protocol (ILP), and distributed service discovery mechanisms within a programmable framework.

Each logical node in the InterEdge is a service node (SN), which is made of processes. These processes include an SN manager (SNM) which runs the SN control plane, a pipe terminus, which is the entry and exit point for all of the packets entering and leaving the SN, and various service modules (SMs), which provide service-specific functionality. These processes are running at all times, and the SNM regularly polls the other processes for their health. Each of these processes have a control-plane component and a data-plane component, with the exception of the SNM, which is only in the control-plane. The control-plane code is written in Python and the data-plane code is written in C++. Hosts are currently similar to SNs, but they also run a host stack component. SNs are connected to each other by pipes, which are implemented as UDP sockets.

4.2 Enabling Scope Requests

4.2.1 Background

Writers of different services can add logic to enable scope requests, but this project focuses specifically on the Hello World Messaging Service (HWMS). This is the simplest service to follow, which will help other service writers copy over the logic without having to differentiate general service logic from service-specific logic. HWMS is a simple packet delivery protocol that operates via flooding, which allows senders to broadcast packets across the network. The overall handling of scope requests can be visualized in Figure 4.1, and it is elaborated on in the following sections.

4.2.2 Application Changes

The first step toward enabling scope requests is to add a flag to an inter-process communication (IPC) header so that the application can initiate a scope request. This can be implemented by modifying existing IPC message types, such as `HWMS_FLOOD` for the case of HWMS, or a new IPC message type could be created. The advantage of adding a flag to an existing type is that without any other changes, the service will follow the

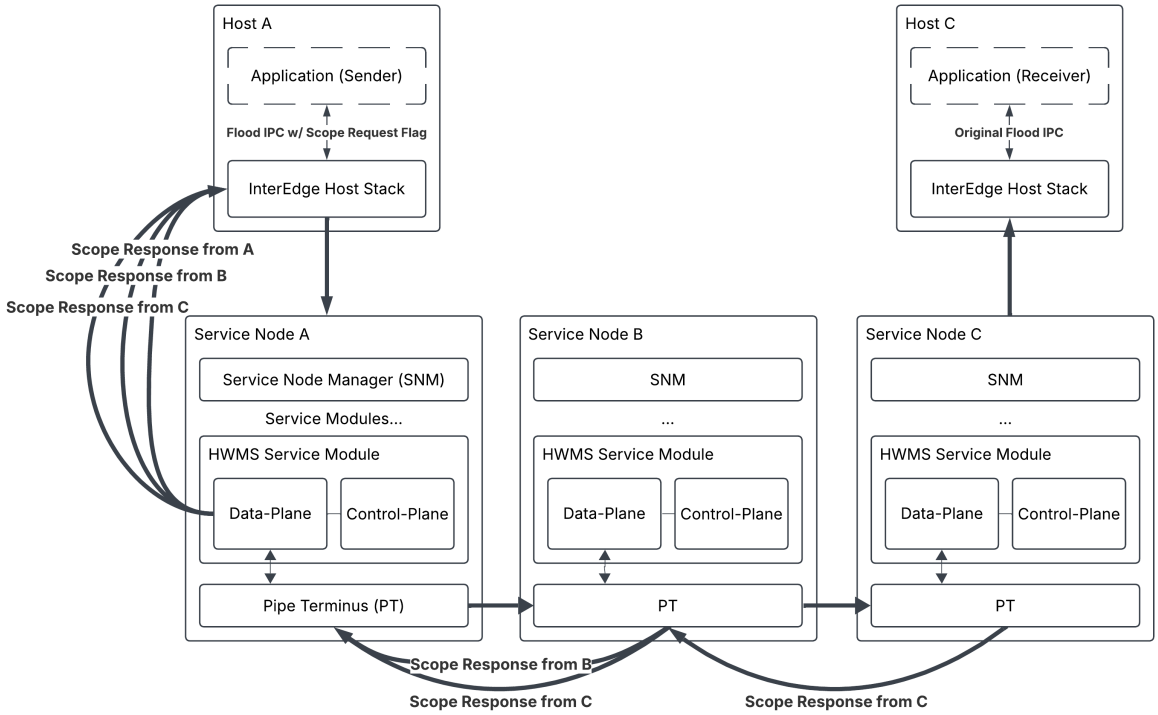

Figure 4.1: Example Scope Request Flow for HWMS

same logic that it did for the regular request. The application, which is a simple Python script, connects to the host stack and sends IPC messages via a socket.

4.2.3 Host Stack Changes

When the host stack receives an IPC message, it will invoke the service's IPC handler for that message. The host stack is implemented in C++. In many cases, the service will send a packet to the service's data-plane, although this depends on the service. For HWMS, the header of this packet stores the remaining hops as well as the destination domain / service node / pipe. In order to support scope requests, the header had to be extended to also keep track of flags representing if the packet is an outgoing scope request or returning scope response. This allows the service module to conditionally perform custom scoping logic when it receives this packet. The header also now has to store its source domain / service node / pipe / file descriptor so the nodes that receive the outgoing scope request can send information back to the node that initiated the request. Once the IPC handler in the host stack constructs this packet (comprised of the header and the message from the application), it is sent to the service's data plane over a service-specific UDP socket.

4.2.4 Service Module Data-Plane Changes

When the service's data-plane (implemented in C++) receives the packet from the host stack, the function `handle_rx` is invoked. This function has access to the SN's inter- and intra- routing tables. The intra-routing table stores information about how to route packets within the domain, while the inter-routing table contains routes to external domains. The logic of `handle_rx` is service-specific, but in many cases it sends a packet to another

SN via `tx`, which takes in a packet and outgoing pipe ID. The packet will reach the pipe terminus (PT), which sends the packet out of the indicated pipe. It is then received by the PT on the destination SN. For HWMS, `handle_rx` will first check whether the packet's destination domain is the current node's domain. If so, it checks if the packet's destination SN is the current SN, and will call `tx` on the packet's destination pipe if this is the case. If not, it will flood all SNs in the domain by calling `tx` on all of the pipes in the intra-routing table. If the destination is in another domain, it will perform the same logic over the inter-routing table's pipes in order to flood all of the SNs in a different domain. However, if this packet is a scope request, all of this logic should be deferred to the control-plane. This is because the node's historical routing tables are needed, not just the current ones (see Section 4.3). The data-plane should operate at high throughput and low latency, so this logic should be part of the control-plane. Therefore, before handling the routing logic, `handle_rx` will send the packet to the service module's control-plane along with a reason, which is a variable representing the logic to be performed in the control-plane.

4.2.5 Service Module Control-Plane Changes

The control-plane of service modules is implemented in Python. A handler for the packet from the data-plane is invoked (`_handle_DPPacket`), which performs the following two steps:

1. **Send Back a Scope Response Packet**

   The control-plane will construct a packet similar to the outgoing scope request, but with the destination domain / service node / pipe set to be the source domain / service node / pipe. The request/response flags will be modified accordingly, and a message can be attached. This will ultimately reach the application that initiated the scope request. This packet will be sent back to the data-plane, which will forward it out.

2. **Route Scope Request**

   The control-plane will also construct packets to help route the scope request, mimicking the service's original logic in `handle_rx`, but with historical routing tables (see Section 4.3). For HWMS, this means that the control-plane will copy over the flooding logic.

The control-plane doesn't send these packets to their destinations directly. Instead, it sends them to the data-plane, which will then forward them according to their attached reason.

4.2.6 More Service Module Data-Plane Changes

The function `handle_cp_packet` is invoked in the data-plane when it receives a packet from the control-plane. This function can check for which of the two cases above applies to the packet (scope response on its way back or outgoing scope request). If it is a scope response, it should be routed directly back to the sender without flooding, so a new function, `handle_rx_scope_response`, is defined. Instead of iterating over the inter- and intra- routing tables, it indexes into them based on the destination domain and SN. The original `handle_rx` also has to be modified with a check for if the packet is a scope

response. If so, a packet is on its way back, and `handle_rx_scope_response` should be called instead of performing the flooding logic.

4.3 Handling Historical Routing State

The routing tables might change in between the time that a packet is sent and the time that a scope request for that packet is actually initiated. This delay could be due to a user not noticing or taking action immediately. The routing tables might change as a result of various reasons, such as the network topology updating or dynamic routing protocols performing load balancing. If so, the scope request should attempt to take the path of the original packet. The service module's control-plane, therefore, needs to maintain a window of historical routing tables. For example, the interval can be set to 60 seconds to monitor every inter- and intra- routing table that has been set in the last 60 seconds. This allows a 60 second buffer for the user to initiate a scope request after the original packet was sent.

In order to support this functionality, routing tables are stored in a double-ended queue (deque). This allows for efficient insertion and deletion at the start and end of the deque. In the control-plane, each time the tables are updated in `set_config`, the new routing tables are appended to the deques along with a timestamp. They are then updated by continuously popping tables from the front of the deques if the table's associated timestamp exceeds the interval. Then, inside of `_handle_DPPacket`, the routing tables are updated and then iterated over to send packets along each possible path.

5. Results

5.1 Testing

The scope request implementation above was tested on a few cases, including the example in Figure 4.1. This test initiated a scope request attached to an HWMS Flood message, and made sure that the scope request along with the message reached the receiver. It also ensured that the sender received the correct amount of scope responses. The test was extended to test the historical routing state implementation, as well. The routing table change was triggered by adding a new node and pipe, and the test ensured that scope responses were sent from both of the routes that the scope request would traverse. The scope request implementation above successfully passed these tests.

5.2 Demonstrated Use Cases

The simplest use case to demonstrate the potential effectiveness of this project is the case of a possible network path going down. For example, if a pipe / link between nodes goes down, causing an increase in latency or packets being dropped, a scope request could be initiated. The user would then be able to look at the scope responses, and the failed link would be immediately apparent when responses from all nodes beyond a certain point are missing.

Nodes failing due to hardware, for example, could also be detected. If this was in a separate domain, the initiator of a scope request would receive scope responses from the local host stack, their domain's SNs, and the first SNs in another domain. However, nothing would be received from the node that crashed, which would precisely pinpoint that node as the point of failure. This is an improvement over previous methods, which might only identify that domain.

Another use case is identifying service-level failures. For example, in cases in which the network infrastructure is functioning correctly but a specific service module on an SN has failed, the scope request can identify this distinction. If the HWMS module isn't working on an SN but other services continue functioning, scope requests targeting HWMS would identify the SN as the problematic node despite overall network connectivity for other services remaining intact.

Sending scope requests over historical routing states also demonstrates an effective use case. For example, if a scope request was initiated and the routing tables have changed in the defined interval, multiple scope responses would be received. Simply looking at

all of these scope responses would reflect this fact, and this alone would be useful information. This could provide a starting point for further debugging and a reason for possible issues. Moreover, it might explain silent performance degradation. Components might not actually be failing after a routing table changes, but traffic could be rerouted over a higher-latency path. The user/application could then determine that packets are following a different path and explain a degraded performance.

5.3 Limitations

The WTF project and this implementation have some limitations and open challenges. The fault localization is inherently coarse-grained, meaning it might identify multiple potential fault sources instead of pinpointing the exact root cause. It also assumes that components honestly report their health status, but in adversarial settings, this might not be the case. WTF is optimized for real-time fault location, hence the limited time window. This creates a defined diagnostic window, which could be insufficient for users who don't immediately notice degraded performance or who need time to initiate troubleshooting procedures. Once routing information ages beyond this window, accurate fault localization becomes impossible. This limits its effectiveness for diagnosing intermittent, long-term failures.

There is also a storage / memory overhead incurred by WTF. For example, as the system scales, storing historical routing tables for every service module across all service nodes could impose significant memory requirements. This could require small historical time windows.

6. Future Work

In the future, the collecting of health statuses should be implemented. This would require that the InterEdge's existing Prometheus [5] servers are storing relevant metrics over a certain window. Then, the Prometheus servers for these metrics could be queried, with the metrics being compressed and sent back in the scope responses. This will provide more helpful health statuses. It would also be helpful to automate the process of initiating a scope request. For example, a system would be monitoring for certain events, warnings, or errors to be triggered. It would then automatically trigger a scope request so that a human does not have to worry about missing the historical routing window. The implementation should also be tested on real bugs and on a larger scale. Finally, the limitations described in Section 5.3 could be addressed with new mechanisms. For example, novel storage techniques and data structures could be used for a more scalable historical routing state tracker.

7. Conclusion

This project successfully implements much of the "Where's the Fault?" (WTF) methodology within the InterEdge architecture, enabling efficient and cross-domain fault localization. By extending the existing InterEdge implementation with scope request capabilities, a mechanism has been created for identifying where failures occur across distributed networking systems. The implementation requires modifications to multiple components of services in the InterEdge architecture, including application interfaces, host stacks, and service module data-planes and control-planes. These modifications are designed to be as minimal as possible for other service developers. Mechanisms for maintaining historical routing state have also been introduced in order to ensure that scope requests accurately reflect the network paths that packets would have traversed at the time of failure.

Testing has demonstrated the effectiveness of the implementation is multiple scenarios, including network path failures, node crashes, and service-level problems. The ability to trace failures across domain boundaries represents an improvement over traditional single-domain fault localization techniques.

While further work and challenges remain, particularly regarding the implementation of collecting detailed health statuses, this work is progress toward addressing the growing complexity of diagnosing failures in modern Internet applications. This work provides a foundation for future enhancements.

References

[1] L. Brown, E. Marx, D. Bali, E. Amaro, D. Sur, E. Kissel, I. Monga, E. Katz-Bassett, A. Krishnamurthy, J. McCauley, T. Narechania, A. Panda, and S. Shenker, "An architecture for edge networking services," in *Proceedings of the ACM SIGCOMM 2024 Conference*, ACM SIGCOMM '24, (New York, NY, USA), p. 645–660, Association for Computing Machinery, 2024.

[2] S. Kemp, "Digital around the world," *DataReportal*, 2025. Available at: `https://datareportal.com/global-digital-overview`.

[3] W. Sussman, E. Marx, V. Arun, A. Narayan, M. Alizadeh, H. Balakrishnan, A. Panda, and S. Shenker, "The case for an internet primitive for fault localization," in *Proceedings of the 21st ACM Workshop on Hot Topics in Networks*, HotNets '22, (New York, NY, USA), p. 160–166, Association for Computing Machinery, 2022.

[4] POX Developers, "POX Documentation," 2020. Available at: `https://noxrepo.github.io/pox-doc/html`.

[5] Prometheus Developers, "Prometheus Documentation," 2025. Available at: `https://prometheus.io/docs/introduction/overview/`.