OST
Eastern Switzerland
University of Applied Sciences

Network insights in OpenTelemetry

Bachelor's Thesis

Department of Computer Science
OST – Eastern Switzerland University of Applied Sciences
Campus Rapperswil-Jona

Autumn Term 2023

Authors

Michael Brändli & Leandro Ceriani

Supervisor / Co-Supervisor

Urs Baumann / Yannick Zwicker

Expert / Proofreader

Philip Schmid / Markus Stolze

12.01.2024

Abstract

The complexity of modern applications demands comprehensive observability to pinpoint performance bottlenecks and optimize application performance. OpenTelemetry, a vendor-neutral observability framework, excels at capturing application-level insights, but the inclusion of network visibility is crucial. This thesis explores the integration of network telemetry data into OpenTelemetry, aiming to provide a holistic understanding of application performance and detect network-related issues.

The project successfully establishes the groundwork for incorporating network visibility into application monitoring using OpenTelemetry. The developed system is positioned for submission as an open-source project. Focusing on the network connecting a 3-tier application, the project ensures that the network path and associated latency are visible during the evaluation of application performance.

The achieved objectives include displaying individual network components, capturing ingress and egress timestamps, and seamlessly integrating network monitoring data into existing traces in OpenTelemetry. These enhancements empower the system to offer insights into latency, network component processing time, and their correlation with requests and existing traces.

Despite these accomplishments, the project falls short of fully identifying the reasons for latency. The selected data retrieval method, Netflow / IPFIX, lacks detailed information about the device's status, limiting the ability to precisely determine the causes of latency. Nevertheless, the project significantly advances observability in distributed applications, providing valuable troubleshooting and performance optimization insights.

Future enhancements could focus on capturing more detailed device status information to identify the root causes of latency better, thereby continuing the trajectory of improvement in observability within distributed applications.

Keywords: OpenTelemetry, Network insights, Traffic flow, Observability, Jaeger, Netflow, IPFIX, PT in SRv6, tcpdump, IOAM.

Executive Summary

Initial situation

Distributed applications are increasingly becoming state-of-the-art in software development in the age of containers and various cloud providers. This naturally offers excellent flexibility for application developers. However, distributing applications has some downsides. The more systems are involved in a distributed application, the more challenging it becomes to troubleshoot in case of an error or slow application behavior. OpenTelemetry, a vendor-neutral observability framework, excels at capturing application-level insights. In OpenTelemetry, an essential component is still missing. Specifically, there is currently no information about the network devices between the different tiers of an application. Telemetry of Network components is especially useful when the application is hosted across different data centers or clouds.

Approach

The aim was to evaluate different variants and test how telemetry data can be collected from network devices. Four different variants were considered and evaluated. The next step was to collect this data in a central location and integrate it into a trace in OpenTelemetry.

Results

Network components like routers and firewalls sent Netflow / IPFIX data to a central ElasticSearch cluster. A newly developed processor in the OpenTelemetry collector was created. This ipfix_lookup processor searches the ElasticSearch cluster for network telemetry data as soon as the application telemetry data arrives. This processor then integrates the network data into existing application traces.

The developed solution successfully provides a comprehensive view of latency and performance across the distributed system by integrating network component data into the existing traces. Latency at the network layer could be analyzed and correlated with application-level requests, enabling more effective troubleshooting and optimization. This advantage

minimizes the effort required for troubleshooting, which indirectly leads to a reduction in resources and costs. The enhanced visibility provided by the ipfix_lookup processor will invariably contribute to an extended mean time between failures.

Further work

While the developed solution achieved the desired objectives, there are opportunities for further improvement. Future work could focus on enhancing the analysis capabilities by including information about the reason for a delay. This could be archived by incorporating additional data from the network components, such as the device status and error information. Additionally, the solution could be extended to support other observability backends and protocols for collecting network component data.

Acknowledgment

We would like to thank the following people for helping with this Bachelor's thesis:

- **Urs Baumann** for the guidance and supervision during the course of this Bachelor's thesis.
- **Yannick Zwicker** for the guidance and supervision during the course of this Bachelor's thesis.
- **Philip Schmid** took on the role of the expert and reviewed the Bachelor's thesis.
- **Markus Stolze** for proofreading the Bachelor's thesis.
- **Severin Dellsperger** for answering questions about path tracing in SRv6.

Contents

1 Introduction **1**

1.1 Assignment . . . 1
1.2 Motivation . . . 1
1.3 User target group . . . 2
1.4 Functional requirements . . . 2
1.5 General Conditions . . . 3
1.6 Preliminary work . . . 3
1.7 Structure of the thesis . . . 4

2 Problem Analysis **5**

2.1 System context . . . 5
2.2 Troubleshooting challenges in daily business . . . 5
2.3 Use cases . . . 6

3 Research **7**

3.1 Concepts . . . 7
3.1.1 What is observability? . . . 7
3.1.2 What are telemetry data? . . . 7
3.1.3 Categories (signals) of telemetry . . . 8
3.1.4 What is OpenTelemetry? . . . 9
3.1.5 What is an observability backend? . . . 9
3.1.6 How does distributed tracing work ? . . . 9
3.2 Displayed information in the observability backend . . . 11
3.2.1 Option 1: OTLP events . . . 13
3.2.2 Option 2: Span per network device . . . 14
3.2.3 Comparison of the options . . . 14
3.2.4 Supporting multiple events . . . 14
3.3 Informations required for a complete trace . . . 15
3.4 Variations of information gathering and their requirements . . . 17

3.4.1 Variation 1: Netflow/IPFIX . . . 18
3.4.2 Variation 2: Path Tracing in SRv6 . . . 20
3.4.3 Variation 3: TCPDump . . . 22
3.4.4 Variation 4: IOAM . . . 23
3.4.5 Evaluation of the variants in relation to the requirements . . . 24
3.5 How to correlate the data? . . . 26
3.5.1 OTEL collector . . . 26
3.5.2 Receivers . . . 27
3.5.3 Processors . . . 27
3.5.4 Exporters . . . 27
3.5.5 Connectors . . . 27

4 Solution **28**
4.1 Choice of technologies . . . 28
4.1.1 Information display . . . 28
4.1.2 Way of information gathering . . . 28
4.1.3 Data correlation . . . 29
4.2 Definition of MVP . . . 36
4.3 Implementation of the MVP . . . 36
4.3.1 Lab environment . . . 38
4.3.2 Network . . . 40
4.3.3 Application . . . 42
4.3.4 Instrumenting of the application . . . 43
4.4 Result of MVP . . . 48
4.5 Contributing to the OTel community . . . 49
4.6 Challenges of MVP . . . 49

5 Results **50**
5.1 Revisiting the use cases . . . 50
5.2 Limitations . . . 51
5.2.1 TCP/IP quartet extraction . . . 51
5.2.2 Timing . . . 51
5.2.3 Same ephemeral port used by several connections . . . 53

6 Conclusion **55**
6.1 Conclusion . . . 55
6.2 Open pull requests / contribution . . . 56
6.3 Possible future improvements . . . 56
6.3.1 Test within Kubernetes . . . 56

6.3.2 Add device metrics for possible reason for latency 56

Glossary **57**

List of Figures **60**

List of Tables **61**

Bibliography **62**

Chapter 1

Introduction

In the dynamic landscape of modern applications, where intricate systems span diverse tiers and intricate network infrastructures, the demand for advanced monitoring tools has surged. This thesis is motivated by the crucial need to bridge the gap between application monitoring and network visibility, with OpenTelemetry serving as a promising catalyst for this integration.

Traditional monitoring solutions often fall short in offering a holistic view, particularly in the context of 3-tier architectures prevalent in today's software systems. Recognizing this gap, this project aims to establish a robust foundation for seamlessly integrating network visibility into application monitoring, leveraging the capabilities of OpenTelemetry.

1.1 Assignment

This Bachelor's thesis aims to establish the groundwork for incorporating network visibility into application monitoring through the utilization of OpenTelemetry. The resulting project will be presented as an open-source initiative. Specifically, the project concentrates on the network connecting a 3-tier application. Gaining insight into the network path and associated latency is crucial for evaluating the overall performance of the application.

1.2 Motivation

This thesis is motivated by the growing complexity of modern applications and the need for enhanced monitoring tools. With the rise of OpenTelemetry, an open-source observability framework, there is a unique opportunity to address the limitations in current monitoring solutions. The focus here is on integrating network visibility into application monitoring, particularly in the context of 3-tier architectures.

The motivation behind this project is to fill the gap in existing monitoring tools that often overlook the vital role of the underlying network in overall application performance. By seamlessly incorporating network visibility through OpenTelemetry, the aim is to provide a more holistic perspective.

The decision to contribute this project as an open-source initiative reflects the commitment to fostering collaboration and knowledge-sharing within the community. Open-source projects encourage diverse perspectives and collective problem-solving, contributing to the development of more robust solutions.

In essence, this thesis aims to bridge the gap between application monitoring and network visibility, leveraging OpenTelemetry to enhance the understanding of application performance in today's dynamic software landscape.

1.3 User target group

The target users are network administrators, application developers, and application owners. If users complain that an application is slow, application owners can check which part is slow and escalate an issue to the network or application team accordingly. Discussions about whose fault it is between the application developers and the network engineers should be put to an end.

1.4 Functional requirements

The result of the Bachelor's thesis should show the latency of the individual network components between the application tiers for a 3-tier application that is hosted on-premise. This information is integrated into an existing trace in OpenTelemetry.

1.5 General Conditions

This work was conducted as part of a Bachelor's thesis (Bachelorarbeit). A time budget of 720 hours (2x 360h) is reserved for the work on this assignment. It will be rewarded with twelve ECTS credits.

1.6 Preliminary work

This is a Bachelor's thesis that has no preliminary work from Leandro Ceriani and Michael Brändli or other students. All work is done during the given amount of working hours described in the general conditions.

1.7 Structure of the thesis

The Bachelor's thesis is organized as follows:

Chapter 2 – Problem Analysis The problem analysis chapter briefly describes the daily challenges of finding out when an application is experiencing delays and long load times. In particular, the discussion between the application developers and the network engineers is always a complex topic if you don't have more precise analysis tools at your disposal.

Chapter 3 – Research The research chapter briefly introduces observability and explains the individual terms used in OpenTelemetry. It then examines how the telemetry data could be displayed in an observability backend. Finally, four variants for obtaining telemetry data from network devices are compared and evaluated, and how this information can be added to a trace in OpenTelemetry is examined.

Chapter 4 – Solution The solution chapter first describes why the decision was made to implement which technology in the individual areas. It then explains how the custom-written processor works and how the MVP was achieved. In addition, the goals set for the Bachelor's thesis and the challenges encountered are described.

Chapter 5 – Results In the results chapter, the use cases that were previously defined are reviewed and checked to see whether they have been fulfilled. In addition, the general limitations and the ones from the implementation described in the solution chapter are discussed.

Chapter 6 – Conclusion In the conclusion chapter, the entire Bachelor's thesis is reviewed, and what went well and what did not is concluded. Open points were addressed, and future improvements were listed.

Chapter 2

Problem Analysis

2.1 System context

OpenTelemetry is an open-source observability framework that provides standardized instrumentation, APIs, libraries, and agents to enable seamless collection of distributed traces and metrics from applications. It helps developers gain insights into the performance and behavior of their systems by facilitating the monitoring and analysis of data across various components in a unified manner. It is not designed out of the box to integrate with telemetry data from network devices. In order to integrate network telemetry data, the code base of an OpenTelemetry collector must be extended.

2.2 Troubleshooting challenges in daily business

In the daily working world of IT, it happens that a user of an application reports a problem and usually opens a ticket. The service desk reports the problem to the relevant application support team. It often happens that if there is no apparent cause for a problem, the ticket is passed back and forth between the network and the development teams, so the users ultimately suffer. The reason for this is usually inadequate monitoring of the application or the monitoring tools not interacting to the extent that it is possible to say that it is either a network problem or a problem with the application itself. Even OpenTelemetry does not solve this problem out of the box, as the OpenTelemetry framework only provides SDKs for programming languages. This is exactly where this Bachelor's thesis should start, and the code base of OpenTelemetry should be extended to answer whether it is a network problem or a problem of the application itself.

2.3 Use cases

The following table lists the use cases that should be fulfilled in this Bachelor's thesis from a vision perspective. These use cases will be revisited later to see how they have or have not been achieved and why not.

Nr.	Title	Description
1	Latency	From the observability backend, the latency of a request can be differentiated between network components and the application.
2	Network components processing time	From the observability backend, a trace's ingress and egress timestamps through a network device are visible.
3	Reason for latency	From the observability backend, metrics of the network devices indicate a possible reason for the latency. (CPU load, RAM load, packet drops, NIC flapping, retransmissions...)
4	Correlate with requests and existing traces	The telemetry data of the network components, which are fetched via a corresponding channel, are integrated into existing traces.

Table 2.1: Descriptions of the use cases

Chapter 3

Research

In this chapter, the core questions of the Bachelor's thesis will be researched, and a foundation should be built to answer them. This chapter begins with the basics and concepts of observability and telemetry.

3.1 Concepts

3.1.1 What is observability?

Observability, within computer systems and software engineering, denotes the capacity to comprehend a system's internal state by analyzing its outputs. It quantifies the extent to which one can deduce the internal mechanisms of a system. In the domains of DevOps and site reliability engineering (SRE), observability holds immense significance. It empowers teams to proficiently monitor, diagnose, and enhance intricate systems and applications.

3.1.2 What are telemetry data?

Telemetry data encompasses information gathered from system sources integral to the concept of observability. When analyzed collectively, this data offers valuable insights into the interconnections and interdependencies within a distributed system. Typically categorized as the 'three pillars of observability,' this data is divided into logs, metrics, and traces, providing a comprehensive view of system behavior.

3.1.3 Categories (signals) of telemetry

Traces

A trace documents the entire journey of a request as it traverses a distributed system. During this journey, numerous operations are executed. Each operation is marked with vital information, known as a "span," which includes details like trace ID, span ID, operation name, start and end timestamps, logs, events, and other indexed data. Through a technique called distributed tracing, examining a trace enables you to map out the entire execution path. This process helps identify problematic areas within the code, such as errors, latencies, or resource bottlenecks, providing valuable insights for troubleshooting and optimization. [1]

Metrics

A metric represents a numerical value measured within a specific time frame. It includes essential attributes like timestamp, event name, and its corresponding value. Metrics are inherently structured, making them easier to query compared to logs. This structured format not only facilitates efficient querying but also optimizes storage, enabling the retention of more metrics over extended periods. This extended data retention offers a comprehensive historical perspective on a system's developments.

Metrics are particularly suitable for handling large datasets or data collected at regular intervals to address specific inquiries. They are often aggregated over time, a crucial practice in modern systems. Timely aggregation allows for in-depth analysis and rapid responses to issues. Metrics serve as the foundation for generating alerts, whether in the form of aggregate values or individual data points. Consequently, they frequently serve as the initial indicators of problems within a system. [2]

Logs

A log serves as a written account of a specific event transpiring at a particular time. Whenever a block of code is executed, a log entry is generated, capturing the event's timing and providing significant contextual information. Log data is available in three formats: unformatted text (plain text), structured, and unstructured. Unformatted text is the most prevalent, but structured logs, enriched with additional metadata for easier querying, are gaining popularity. In contrast, unstructured logs are more challenging to interpret. Logs primarily act as the primary source of information about the application's activities. Developers turn to logs to diagnose problems when issues arise within a system. Logs help developers identify errors in their code and scrutinize code execution. In complex distributed systems,

errors typically have multiple underlying causes, often termed "core causes." Logging in these scenarios offers intricate insights into the execution of specific code segments. [3]

Baggage

Baggage refers to contextual information exchanged between spans in OpenTelemetry. It operates as a key-value store located alongside the span context within a trace. This arrangement ensures that the stored values are accessible to any subsequent span generated within the same trace. [4]

3.1.4 What is OpenTelemetry?

OpenTelemetry is a means to achieve observability. Organizations need to add instrumentation to their infrastructure to achieve end-to-end visibility. While this is difficult to achieve with a single commercial solution, OpenTelemetry standardizes the instrumentation needed to collect telemetry data, enabling observability.

3.1.5 What is an observability backend?

The observability backend receives the data from OpenTelemetry and displays it visually in a GUI. Well-known observability backends are Jaeger or Prometheus.

3.1.6 How does distributed tracing work ?

To allow distributed tracing to work, an HTTP header field called `traceparent` can be attached to every HTTP request in the system. This technique propagates a single trace-id from the initial call to every subsequent service and function.

```
traceparent = 00-a2229a794449a549258a50bcfc519505-abb56380f565ddd4-01
```

The `traceparent` HTTP header field identifies the incoming request in a tracing system. It has four fields:

- version (always 00)
- trace-id (16 byte array, hex-encoded)
- parent-id (8 byte array, hex-encoded)
- trace-flags (An 8-bit field that controls tracing flags such as sampling, trace level, etc.)

[5]

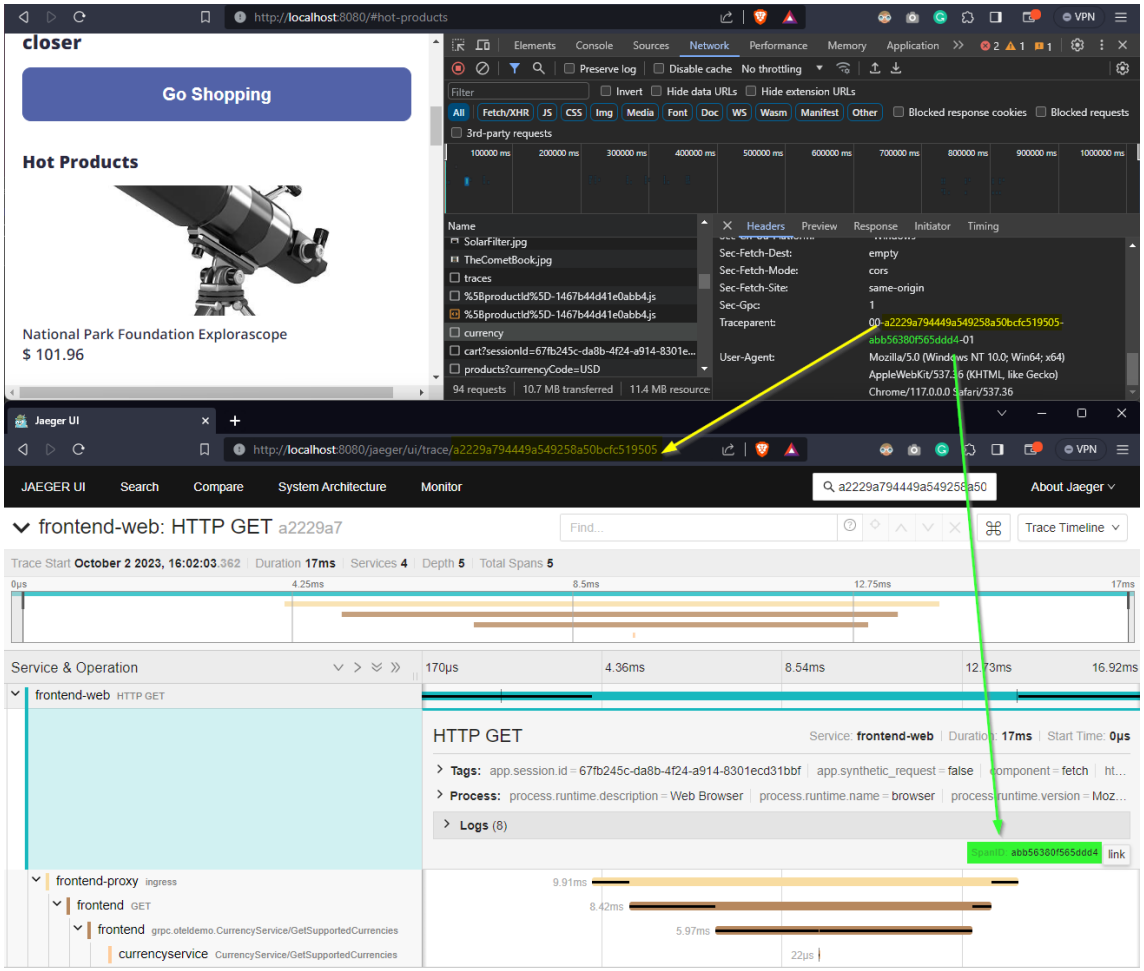

Figure 3.1: HTTP traceparent header

The trace-id can be seen from the client side and searched in the Jaeger UI. This can be seen in the figure 3.1 using the OpenTelemetry demo application.

3.2 Displayed information in the observability backend

One of the primary objectives is to answer whether latency comes from the network connecting two services or the called service. Mocks of OpenTelemetry data are created in order to better understand and evaluate possible solutions for displaying the network latency in an OpenTelemetry trace.

To get started, an easy-to-understand example is analyzed. It is a classic 3-tier application that contains a frontend, a backend, and a database. It is seen in the figure. 3.2

The 3-tier app consists of three separate network flows. They are depicted in the figure 3.2 as A, B, and C. In this example, the focus is set on trace B, where the frontend (Frontend01) connects to the backend (Backend01) through a firewall (FW01).

The timestamps of the packets in the trace must be measured four times. B1 and B2 are measuring the request to the backend. While B3 and B4 are tracking the response from the backend. How exactly all these timestamps can be collected will be explained in chapter 3.5

- B1: At the ingress interface of the firewall from the frontend (request)
- B2: At the egress interface of the firewall to the backend (request)
- B3: At the egress interface of the firewall to the frontend (response)
- B4: At the ingress interface of the firewall from backend (response)

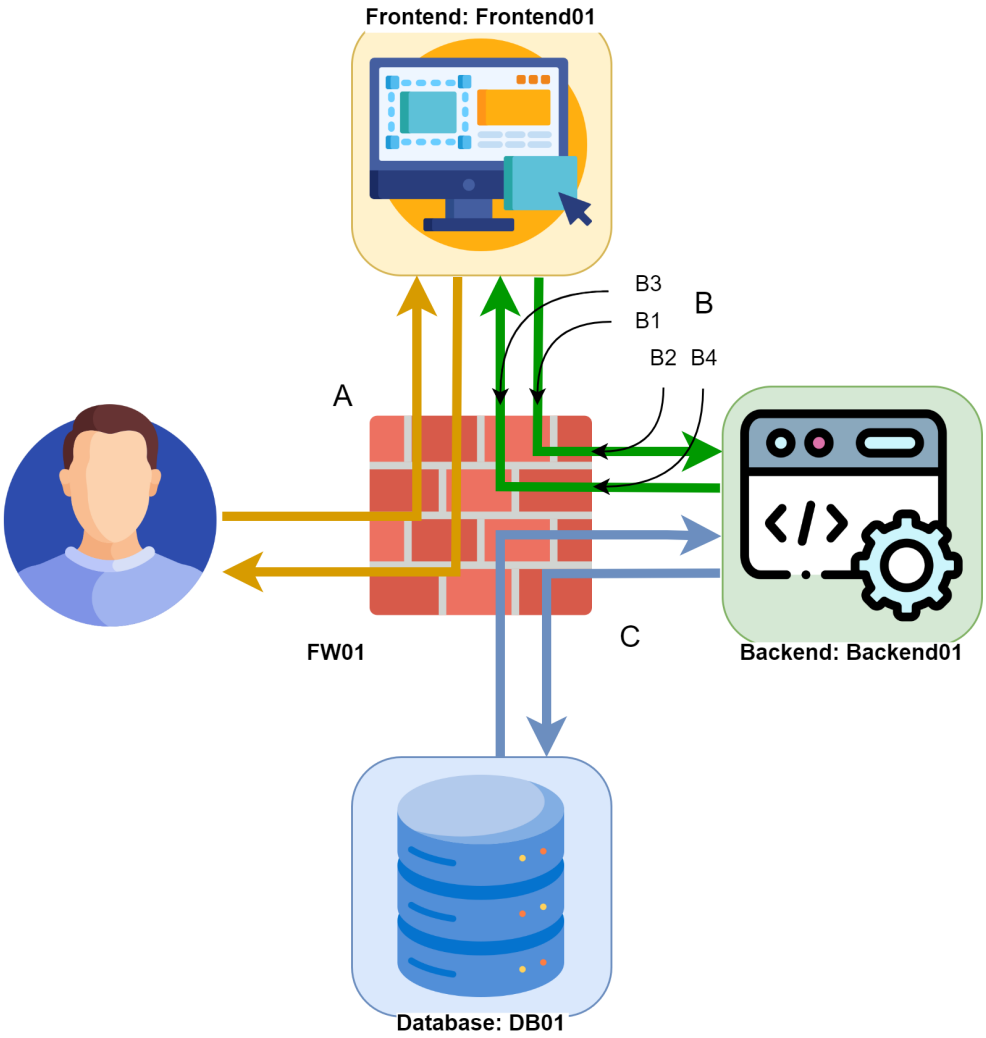

Figure 3.2: 3-tier-application

There are two options for displaying the network latency in an OpenTelemetry trace.

3.2.1 Option 1: OTLP events

The first option uses the OpenTelemetry protocol (OTLP) to add events to a span. In this case, a single span would cover the entire network path. The events could be used for ingress and egress of a device and any other point in time the packet can be identified in the network. For example, when traveling through a proxy or a switch. Displaying longer routes or routes with missing events would be possible.

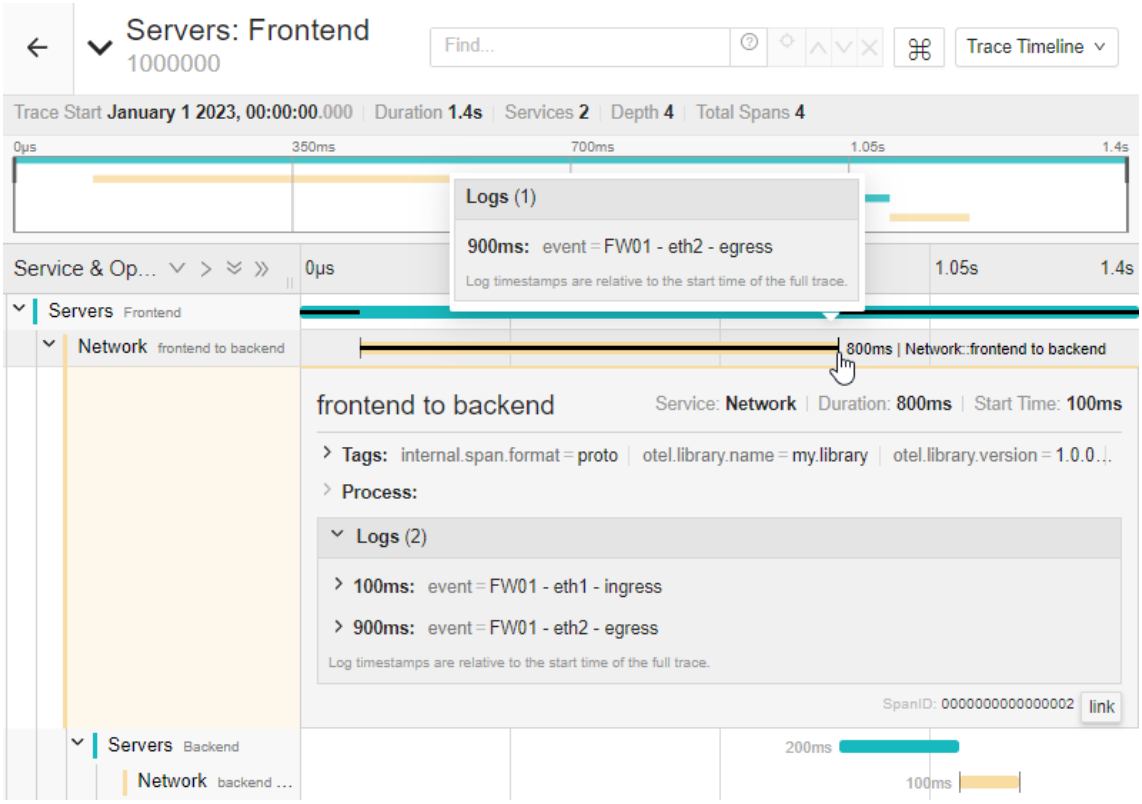

Figure 3.3: Frontend to backend connection in Jaeger using OTLP events

3.2.2 Option 2: Span per network device

In the second option, a separate span is used for every network device that handles the packet. Each span starts at the ingress of a device and ends when egressing the same device. This option adds the challenge of collecting two timestamps from every device. The span will be incomplete or wrong if one timestamp cannot be retrieved.

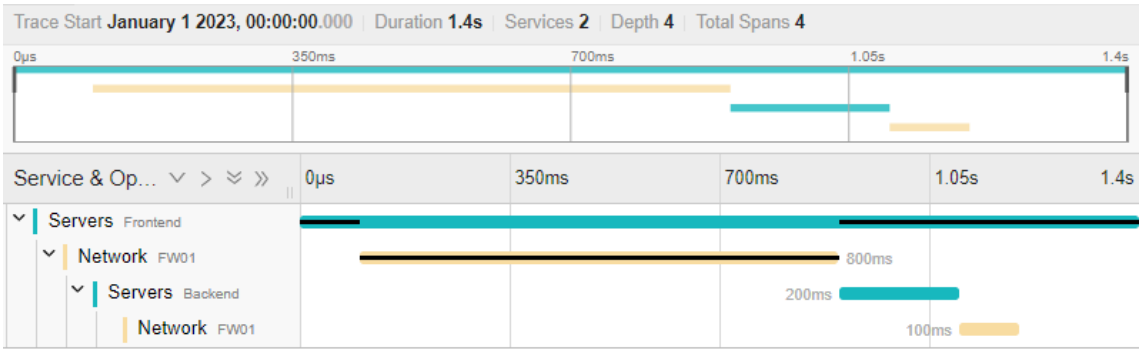

Figure 3.4: Frontend to backend connection in Jaeger using OTLP spans

3.2.3 Comparison of the options

	Option 1: OTLP events	Option 2: Span per network device
Pros	+ Easily expandable with additional events	+ Clearly displays network devices + Useful in "Graph view"
Cons	- Events are not visible in the "Graph view" - A lot of information in one span	- In bigger networks, many additional spans will be added - Missing timestamps will lead to incomplete spans

Table 3.1: Comparison of the options

3.2.4 Supporting multiple events

Often, a network connection does not only traverse a single router or firewall but multiple. Additionally, every connection will have at least two spans, one for the request and another for the response. Therefore, the solution must transparently support multiple hops. It can be very challenging to order the spans correctly. Using a summary span, it was deceived to group all spans from a single connection together. This keeps the traces clean and makes them more readable. Every event under the summary span still has its own span, as described in Option 2. A summary span can be seen in Figure 4.3.

3.3 Informations required for a complete trace

OpenTelemetry provides the data for a trace for the application part. The idea is that the network components involved between the individual tiers of the application should also be considered and included in a trace.

OpenTelemetry does not offer an SDK or plug-in for network components with which this can be solved. So another way has to be found.

Of particular interest is the timestamp of a packed arriving at an ingress interfaces and leaving an egress interface of a network device. With this information, it can be determined how long a packet takes to cross the network device.

Once you have this data, this information must be entered into a trace from OpenTelemetry. OpenTelemetry works with trace-ids and parent-ids. This information is not available for network components. It must, therefore, be possible to make a mapping between the connections via a network component and a trace.

Multi-tier applications use the TCP/IP stack so that, for example, the frontend can connect to the backend and load or process the data.

A TCP/IP connection contains always the following information:

- **`source.ip`**
- **`source.port`**
- **`destination.ip`**
- **`destination.port`**

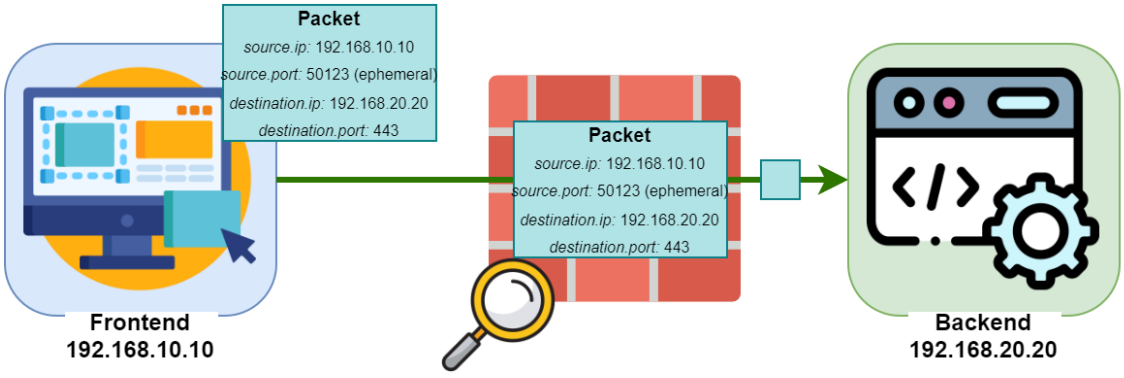

Figure 3.5: TCP/IP information for trace mapping

The source port is usually an ephemeral port chosen by the source system. Using this quartet of information, it should be possible to make a mapping between the traces and

the packages on the network components. It is important that this information is obtained from the packets on the network components and the software stack, e.g., on the frontend and the backend, so that a solution can then be used to map this information accordingly and combine it in the trace.

3.4 Variations of information gathering and their requirements

Different variants are considered for obtaining the required information from the network components. Basically, there are two approaches. The first one obtains the logged information from each network device individually. The second one uses a network technology that forwards the information on the selected route from hop to hop and carries this through to the end.

Requirements

The requirements for the different variants are described in the table 3.2.

Nr.	Title	Description
1	Provides TCP/IP quartet	The data collection approach provides the source.ip, source.port, destination.ip and destination.port of the packet.
2	Full flow capturing	The data collection approach provides a sampling rate of 1 to 1. Meaning every packet is analyzed.
3	Scalable	The data collection approach scales to the size of an entire network of many routers and interfaces
4	Kubernetes	The data collection approach works inside of a Kubernetes cluster.
5	Extract traceparent header	The value of the HTTP header field "traceparent" can be extracted

Table 3.2: Description of requirements for network data collection

3.4.1 Variation 1: Netflow/IPFIX

NetFlow and IPFIX (Internet Protocol Flow Information Export) are network protocols used for monitoring and collecting network traffic data. They provide valuable insights into network behavior, aiding in security, performance analysis, and capacity planning.

NetFlow

NetFlow, developed by Cisco, is a network protocol that enables network devices to collect and monitor IP traffic information. It records details about IP flows, including source and destination IP addresses, ports, protocols, and the amount of data transferred. NetFlow technology allows network administrators to analyze network usage patterns, identify anomalies, and optimize network resources. It plays a crucial role in network security by detecting and mitigating various types of attacks, such as DDoS attacks and port scans.

IPFIX

IPFIX, standardized by the IETF (Internet Engineering Task Force), is an extension of NetFlow that offers a more flexible and scalable approach to flow data export. IPFIX provides a standardized format for exporting flow information from routers, switches, and other network devices to external collectors or analyzers. It supports various types of data, allowing organizations to define their own information elements for specific monitoring needs. IPFIX is vendor-neutral, ensuring interoperability between different network devices and analysis tools.

NetFlow vs. IPFIX

In summary, both NetFlow and IPFIX are powerful tools for network monitoring and analysis. NetFlow, with its origins in Cisco technology, provides essential traffic visibility. On the other hand, IPFIX, as an open standard, offers enhanced flexibility and scalability, making it a preferred choice for organizations aiming for interoperability and customizability in their network flow monitoring solutions.

The protocols allow sampling to reduce computational overhead and enhance scalability in high-traffic environments, making real-time analysis more efficient. However, this approach sacrifices granularity, potentially leading to inaccuracies in representing overall network traffic patterns and application monitoring insights. The decision to employ sampling should consider the trade-offs between resource conservation and the need for precise, comprehensive data in network monitoring and analysis. Collecting more samples increases the resource usage on the network devices and requires more storage and

performance on the storage backend. This can be slightly mitigated by keeping the Netflow / IPFIX data for only a short period of time.

For the Bachelor's thesis, both network protocols would be a suitable choice, as the information source.ip, source.port, destination.ip, and destination.port can be obtained from both. In terms of overhead, however, IPFIX would be the preferred choice, as you can determine the information elements yourself and thus reduce the amount of data in larger environments.

[6]

3.4.2 Variation 2: Path Tracing in SRv6

SRv6

Segment Routing with IPv6 (SRv6) is a networking technology that simplifies and enhances packet forwarding in computer networks. It extends the IPv6 data plane to provide source routing, allowing network operators to specify paths that packets should take through the network. SRv6 achieves this by encoding routing instructions directly into the IPv6 header, eliminating the need for additional protocols or headers.

This approach offers several advantages, including improved network scalability, reduced complexity, and enhanced traffic engineering capabilities. SRv6 enables efficient and flexible packet forwarding, making it particularly valuable for large-scale networks, data centers, and service provider environments. By leveraging SRv6, organizations can optimize their network resources, enhance network reliability, and streamline innovative services and application deployment. [7]

PT (Path Tracing)

Path Tracing in SRv6 is a method that records the packet's path using interface IDs, including details like end-to-end delay, per-hop delay, and load on egress interfaces. This technique enables the tracing of 14 hops using a compact 40-byte IPv6 Hop-by-Hop extension header. Additionally, it supports precise timestamps and is specifically designed for efficient implementation in hardware pipelines, ensuring linear performance.

Path tracing uses the IPv6 header extension number 0 (hop-by-hop options) and is currently in the process of being standardized by the IETF (Internet Engineering Task Force). The draft is available under: https://datatracker.ietf.org/doc/draft-filsfils-spring-path-tracing/

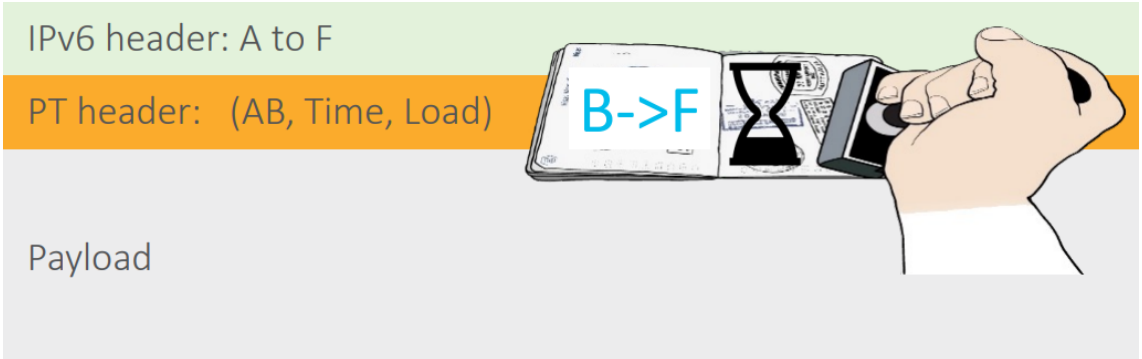

Figure 3.6: Path tracing extension header [8]

With segment routing over IPv6 data plane, the route for a data packet is determined in advance. A "stamp" with the information outgoing interface ID, timestamp, and load is now

added at each hop the packet goes through effectively. The router at the end of the path (often called edge router) then sends the path tracing information to a central collector.

3.4.3 Variation 3: TCPDump

TCPDump is a powerful command-line packet analyzer widely used by network administrators and security professionals for network troubleshooting and analysis. It operates on Unix-based systems, capturing and displaying network packets in real-time. This tool plays a vital role in diagnosing network issues, monitoring network traffic, and detecting security threats.

At its core, TCPDump captures packets traversing a network interface and decodes them into human-readable format. It supports a plethora of protocols, allowing users to inspect traffic at various layers of the OSI model. From ethernet frames to IP packets and application-layer data, TCPDump provides a detailed view of network activities.

One of TCPDump's key features is its flexibility. Users can apply filters based on protocol, source or destination IP addresses, port numbers, and even specific packet content. This capability allows for precise, targeted analysis, enabling network professionals to focus on specific aspects of network traffic. Filters can be combined, creating complex expressions, ensuring that the captured data is highly relevant to the analysis at hand.

Additionally, TCPDump is scriptable and integrates seamlessly with other tools and scripts. Its output can be redirected to files for later analysis or processed in real time by other programs, enhancing its usability in various network monitoring setups. Moreover, it supports the reading of captured packet files from other packet capture tools, making it an essential component of the network analyst's toolkit.

One of TCPDump's most significant advantages is its ability to capture packets without putting additional load on the network. By sniffing packets directly from the network interface, it provides an unobtrusive method of monitoring, making it invaluable for diagnosing intermittent network issues.

[9]

3.4.4 Variation 4: IOAM

In Situ Operations, Administration, and Maintenance (IOAM) collects operational and telemetry data in the packet header while the packet traverses a path between two hosts in a network. IOAM is currently a proposed IETF standard. The draft is available under RFC 9197. [10]

An IOAM network system consists of the following components:

- **The data collection module**
- **The data analysis module**

The data collection module consists of three types of nodes: encapsulation, transit, and decapsulation. The encapsulation node (entry node of the packet) samples service packets, adds IOAM information to the IOAM header, and forwards the packet to the next node. The transit node adds IOAM information to the header and forwards the packet to the next node. The decapsulation node (exit node) adds its own IOAM information to the header, then removes the header and forwards the information from the header to the analysis module in NetStream V9 format and the original packet to the destination host.

The data analysis module's main role is to analyze the collected data that it receives from the data collection module. [11]

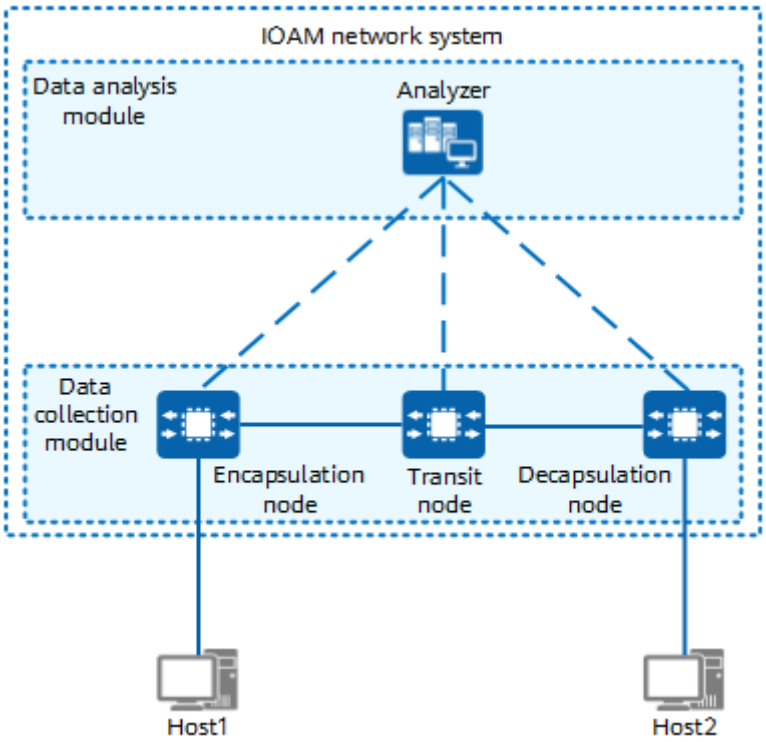

Figure 3.7: IOAM network system [11]

3.4.5 Evaluation of the variants in relation to the requirements

In the following table, the requirements for the different variations described above are evaluated and justified in the table below.

Nr.	Title	IPFIX	PT in SRv6	TCPDump	IOAM
1	Provides TCP/IP quartet	Yes	No [1]	Yes	Yes
2	Full flow capturing	Yes	No [1]	Yes	No
3	Scalable	Yes	(Yes) [1]	No	Yes
4	Kubernetes	Yes	No [1]	Yes	No
5	Extract traceparent header	No	No [1]	Yes	No

Table 3.3: Evaluation of requirements for network data collection

[1] These requirements were evaluated together with Severin Dellsperger. Severin Dellsperger is Network & Research Engineer at INS Institute for Network and Security and is experienced with PT in SRv6.

Variation	Req-Nr.	Justification
IPFIX	1	IPFIX brings a lot of entities to use including the TCP/IP quartet. [12]
	2	Full flow capturing depends on the hardware of the network components but is supported by the most commonly known devices like a Cisco Nexus 9000 Series [13]
	3	IPFIX, in combination with distributed collector systems like ElasticSearch, can handle large-scale networks. [14]
	4	There are Kubernetes CNIs which support IPFIX, for example the OVN-CNI. [15]
	5	IPFIX provides monitoring and valuable insights into network behavior but does not include the payload of the traffic. Therefore, the traceparent header can not be extracted from it. [16]
PT in SRv6	1	No TCP connections are tracked, but the path of the packet is recorded with per-hop interface id, latency, and load. [1]
	2	Path tracing (currently) uses its own probe packages. It, therefore, does not work with a sampling rate (such as IPFIX) [1]
	3	It is certainly designed to be used in large networks. It is currently in the test phase with some large ISPs, including Swisscom and Bell Canada. [1]
	4	There is currently no CNI with PT in SRv6 implemented. [1]
	5	PT in SRv6 is not intended to extract layer 7 information. [1]
TCPDump	1	TCPDump allows to inspect traffic at various layers of the OSI model, including layer 2 and layer 3. Therefore, the TCP/IP quartet is provided by TCPDump. [9]
	2	TCPDump captures every packet (TCP and UDP). Full flow capturing is therefore supported by TCPDump. [9]
	3	TCPDump is generally more of an analysis and troubleshooting tool used in the event of a network malfunction and not for general network monitoring. Therefore, TCPDump does not scale for the use case of this Bachelor's thesis. [9]
	4	TCPDump is a commonly used tool and is available either on the host of a Kubernetes environment or within containers. [9]
	5	TCPDump is able to extract the traceparent header as it inspects all layers of the OSI model. [9]
IOAM	1	The IOAM decapsulation node exports its data to the analyzer in the NetStream V9 format which contains a lot of information, including the TCP/IP quartet. [11] [17]
	2	Full flow capturing is not possible. There is a given sampling ratio by NetStream which can be edited, but not set to full flow. [18]
	3	As IOAM is still new and in a proposed state, no studies of high scalability have been found.
	4	There is currently no CNI which has IOAM implemented. There is a first open source IOAM implementation for IPv6 in the Linux kernel, which is namespace oriented and should work with containers like Docker and Kubernetes. [19]
	5	The traceparent header is on layer 7 and therefore in the payload of the packet. The IOAM header is between the TCP header and the payload. There is no functionality from IOAM to export headers from the payload. [11]

Table 3.4: Justification of whether a variation meets the requirement

3.5 How to correlate the data?

Determining which network flow belongs to which trace is very challenging. As described in section 3.1.6 OpenTelemetry uses an HTTP header field called `traceparent` to propagate the trace-id when requests over HTTP are executed. Unfortunately, this information is unavailable on the network layer without deep-packet inspection or similar techniques. However, each HTTP request should be uniquely identifiable by the quartet of `source.ip`, `source.port`, `destination.ip` and `destination.port`. Using these four fields, a network flow captured on OSI model layer 4 can be associated with a trace.

Thus, a piece of software is needed, which inserts the received data from the network into a trace of OpenTelemetry. For the purposes of this thesis, this piece of software will be referred to as the "correlation unit".

3.5.1 OTEL collector

OpenTelemetry collector is a versatile observability tool designed to collect, process, and export telemetry data from various sources in modern software applications. It acts as a middleware, intercepting and aggregating data like traces, metrics, and logs from different services and applications. This unified approach allows for seamless integration and standardization of telemetry data, enabling better monitoring and troubleshooting of complex, distributed systems. The collector can perform transformations and enrichments on the data, ensuring its consistency and relevance before forwarding it to backend systems like observability platforms or storage solutions. By providing a centralized and configurable solution, OpenTelemetry collector simplifies observability, making it easier for developers and operators to gain insights into their applications' performance and behavior.

The OTEL collector consists of the following four components, which can be configured and coded by yourself:

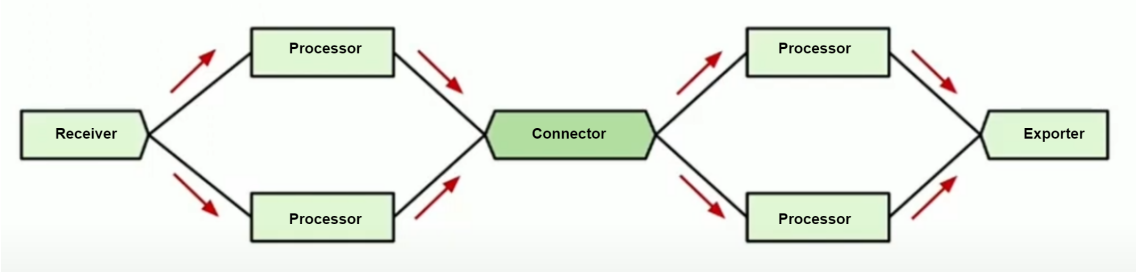

Figure 3.8: OTEL collector components

3.5.2 Receivers

A receiver, which can be push or pull-based, is how data gets into the collector. Receivers may support one or more data sources.

3.5.3 Processors

Processors are run on data between being received and being exported. Processors are optional, though some are recommended.

3.5.4 Exporters

An exporter, which can be push or pull-based, is how you send data to one or more backends/destinations. Exporters may support one or more data sources.

3.5.5 Connectors

A connector is both an exporter and a receiver. As the name suggests, a connector connects two pipelines: It consumes data as an exporter at the end of one pipeline and emits data as a receiver at the start of another pipeline. It may consume and emit data of the same data type or of different data types. A connector may generate and emit data to summarize the consumed data or simply replicate or route data.

[20]

Chapter 4

Solution

4.1 Choice of technologies

This chapter lists and justifies the choice of technologies described in the research chapter.

4.1.1 Information display

Referencing to section: 3.2

For the display of the network telemetry data in the observability backend, option 2 (span per network device) is the better choice. It clearly displays the known network devices which makes it easier for the user to understand where the latency is coming from. In addition, option 2 is used in combination with a summary span, as described in chapter 3.2.4.

4.1.2 Way of information gathering

Referencing to section: 3.4

Path tracing with SRv6 looks like an inspiring approach and will continue to develop and spread over the next few years. This technology is still relatively new and is supplied by Cisco, for example, from IOS XR version 7.8.1 [8]. IOS XR 7.8.1 was released on the Cisco ASR 9000 Series on November 14, 2022 [21] and on the Cisco 8000 Series on November 29, 2022 [22]. This is currently only about 1 year ago.

Companies will need to evaluate and roll out this version before they can even start to use path tracing with SRv6. Furthermore, only a few other manufacturers supply or offer an operating system update that includes the feature path tracing. The idea that the last

hop (the edge router) sends the information to a central collector is a good approach. However, it brings the risk that if this edge router has a problem, all the information from a trace is lost. Netflow / IPFIX is a standard that has been around for over 15 years [23] and is already widely used and integrated [16]. Path tracing with SRv6, in comparison, is only about one year old. During the initial setup, this variant requires more effort, as the configuration for Netflow / IPFIX must be carried out on each network device. However, this approach can reduce the risk of a possible loss of all tracing data, as this variant does not have a single point of failure for the full trace information, which on path tracing with SRv6 would be the edge router. For the reasons mentioned above, Netflow / IPFIX is therefore chosen for the further procedure in the Bachelor's thesis.

4.1.3 Data correlation

Referencing to section: 3.5

The correlation unit is responsible for looking through the spans in each trace. If the TCP/IP quartet is found in a span, the corresponding flow is looked up in ElasticSearch. When flows are found, a new span is added to the trace at the correct location.

Initially, an OpenTelemetry collector connector was thought to be the best way to add the correlation functionality to OpenTelemetry. A connector supports both receiving and exporting events. It is possible to connect multiple different pipelines together using a connector. After a discussion on #28692 with Daniel Jaglowski (@djaglowski), a member of the "OpenTelemetry - CNCF" Team, it was noticed that an OpenTelemetry collector processor would, in this case, serve the same functionality as the connector while requiring less configuration for the end user.

The design in figure 4.1 depicts the final flow diagram for the correlation unit.

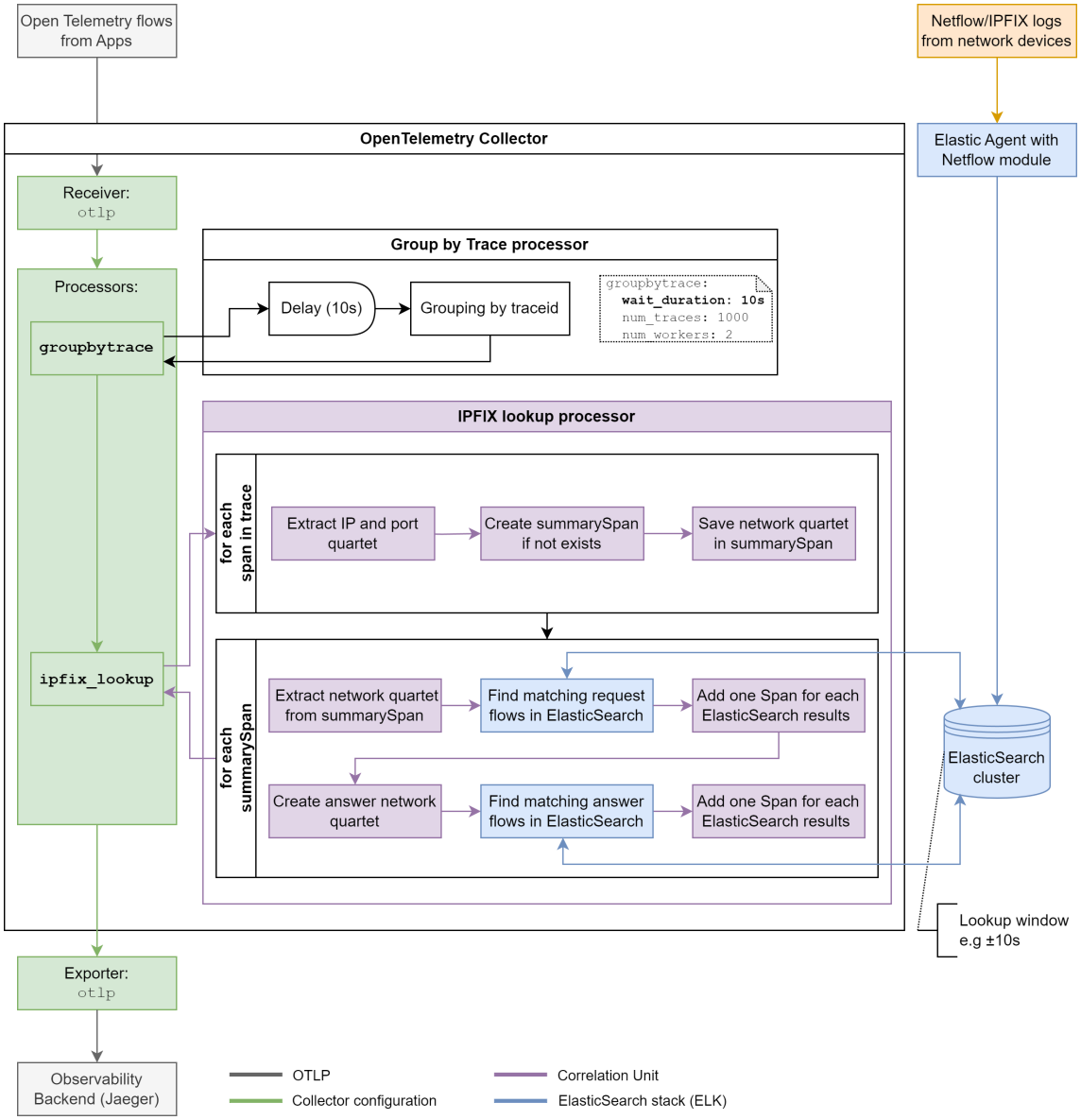

Figure 4.1: Correlation unit data flow

OpenTelemetry collector pipelines

Inside the OpenTelemetry pipeline, a new processor called `ipfix_lookup` can be configured. Before the IPFIX lookup is performed, all the traces are grouped together, and a delay is added by the `groupbytrace` processor. The `groupbytrace` will group all the incoming spans by trace and wait for the `wait_duration` until forwarding it to the `ipfix_lookup` processor.

```
processors:
 groupbytrace:
  wait_duration: 10s
  num_traces: 1000
  num_workers: 2
 ipfix_lookup:
  elastic_search:
   connection:
    addresses:
     - https://<ElasticSearch IP here>:30200/
    username: elastic
    password: <password here>
    certificate_fingerprint: <certifiacte fingerprint here>
  timing:
   lookup_window: 10
  query_parameters: #optional parameters
   base_query:
    field_name: input.type
    field_value: netflow
   device_identifier: "fields.observer\\.ip.0"
   lookup_fields:
    source_ip: source.ip
    source_port: source.port
    destination_ip: destination.ip
    destination_port: destination.port
  span_attribute_fields: #optional parameters
   - "@this"
   - "fields.event\\.duration.0"
   - "fields.observer\\.ip.0"
   - "fields.source\\.ip.0"
   - "fields.source\\.port.0"
```

```
    - "fields.destination\\.ip.0"
    - "fields.destination\\.port.0"
    - "fields.netflow\\.ip_next_hop_ipv4_address"
  spans: #optional parameters
    span_fields:
      source_ips:
        - net.peer.ip
        - src.ip
      source_ports:
        - net.peer.port
        - src.port
      destination_ip_and_port:
        - http.host
      destination_ips:
        - dst.ip
        - net.peer.name
      destination_ports:
        - dst.port
```

Every trace passing through the pipeline will first be grouped and then checked for potential Netflow / IPFIX flows inside the ElasticSearch cluster.

```
service:
  pipelines:
    traces:
      receivers: [otlp]
      processors: [groupbytrace, ipfix_lookup]
      exporters: [otlp/jaeger, debug]
  telemetry:
    logs:
      level: debug
```

This implementation allows the `ipfix_lookup` processor to seamlessly integrate into existing OpenTelemetry collectors.

ElasticSearch and Netflow / IPFIX

The ElasticSearch cluster is needed to store the Netflow / IPFIX information. Short-term storage of the Netflow / IPFIX information is required as the time it takes for the OpenTelemetry traces to reach the collector is different than for the Netflow / IPFIX information. Technically, only short-term storage is required. The minimum retention period can be calculated using `processors.ipfix_lookup.timing.lookupWindow + processors.groupbytrace.wait_duration`. However, most companies will already have a Netflow / IPFIX collection and storage that will fulfill and even exceed this requirement.

The Elastic Agent has an existing integration for Netflow / IPFIX records. This Integration can be enabled on an Elastic Agent connected to an ElasticSearch and Kibana instance. Thanks to this integration, Netflow / IPFIX data is properly parsed and imported into ElasticSearch. This means that standard fields like `@timestamp`, `event.duration`, `source.ip`, `source.port`, `destination.ip`, `destination.port`, `observer.ip` are populated and can be used. This will simplify adding new protocols and ensuring compatibility in the future.

- **`@timestamp`**: Date/time when the event originated. (RFC3339: 1985-04-12T23:20:50.52Z)
- **`event.duration`**: Duration of the event in nanoseconds ($1ns = 10^{-9}s$)
- **`source.ip`**: IP address of the source (IPv4 or IPv6).
- **`source.port`**: Port of the source.
- **`destination.ip`**: IP address of the destination (IPv4 or IPv6).
- **`destination.port`**: Port of the destination.
- **`observer.ip`**: IP addresses of the observer. (e.g. the router that exportet the flow)

[24]

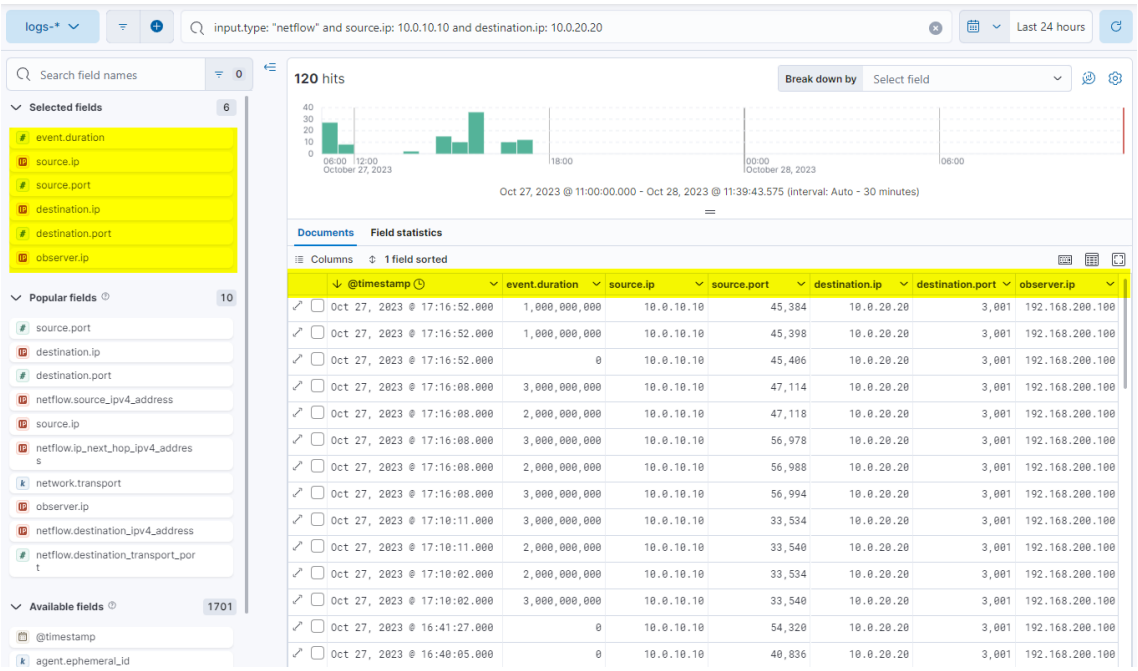

Figure 4.2: Netflow / IPFIX data in ElasticSearch

ipfix_lookup processor

The processor brings everything together. First, the traces from the OpenTelemetry collector pipelines are received. Each trace span is then checked to see if the TCP/IP quartet can be extracted. The corresponding flow is searched in ElasticSearch when the four values are found. For the time frame of the search, two considerations must be made.

Firstly, there is an ingest delay in any large distributed search engine. Because of this, the spans need to be pre-processed by the `groupbytrace` processor. The delay can be defined in the `processors.groupbytrace.wait_duration` value. Afterwards, the search can be started. The time window that will be searched can be configured in the `processors.ipfix_lookup.timing.lookupWindow`. To keep the processor simple, the `lookupWindow` is added before the start timestamp and after the end timestamp. This way, the chance that the Netflow / IPFIX records leading or being caused by this span is found is maximized.

Summary span in Jaeger

A summary span was added to simplify the display of the spans in Jaeger, under which all Netflow / IPFIX spans are placed. As depicted in 4.3 the summary span is highlighted yellow and contains the TCP/IP quartet in the name. Both request and response are grouped under the same summary span.

The summary span improves the `ipfix_lookup` processor as it can be split into two separate actions. First, the trace will be checked for the TCP/IP quartet, and summary spans will be created. In the second step, the processor iterates through the summary spans and looks up the data in ElasticSearch.

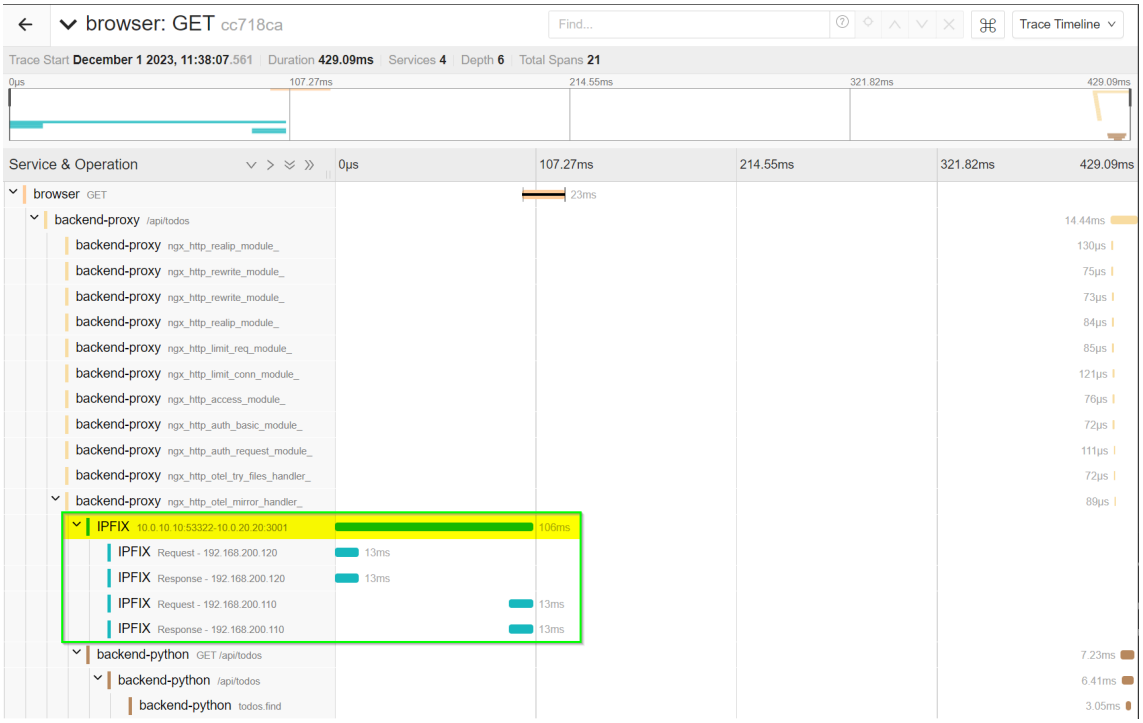

Figure 4.3: Summary span in Jaeger

4.2 Definition of MVP

In the first step, the correlation unit, part of an OpenTelemetry collector, is developed with a simple application. This is a simple 3-tier application, as usually known, with the following components:

- Frontend
- Backend
- Database

From a network perspective, each component will be placed into a different network subnet so that the requests between the components go via the default gateway, and the information from the router can be collected via IPFIX. For the MVP, it is enough if two components are OpenTelemetry instrumented, and the network connection between two components gets injected via the correlation unit into the trace in the observability backend.

4.3 Implementation of the MVP

The MVP will consist of a 3-tier application where all components are instrumented and reporting to either OpenTelemetry or ElasticSearch. By the end of the MVP only the flow from the proxy to the backend will be injected automatically into OpenTelemetry. This flow is marked green in the figure 4.4

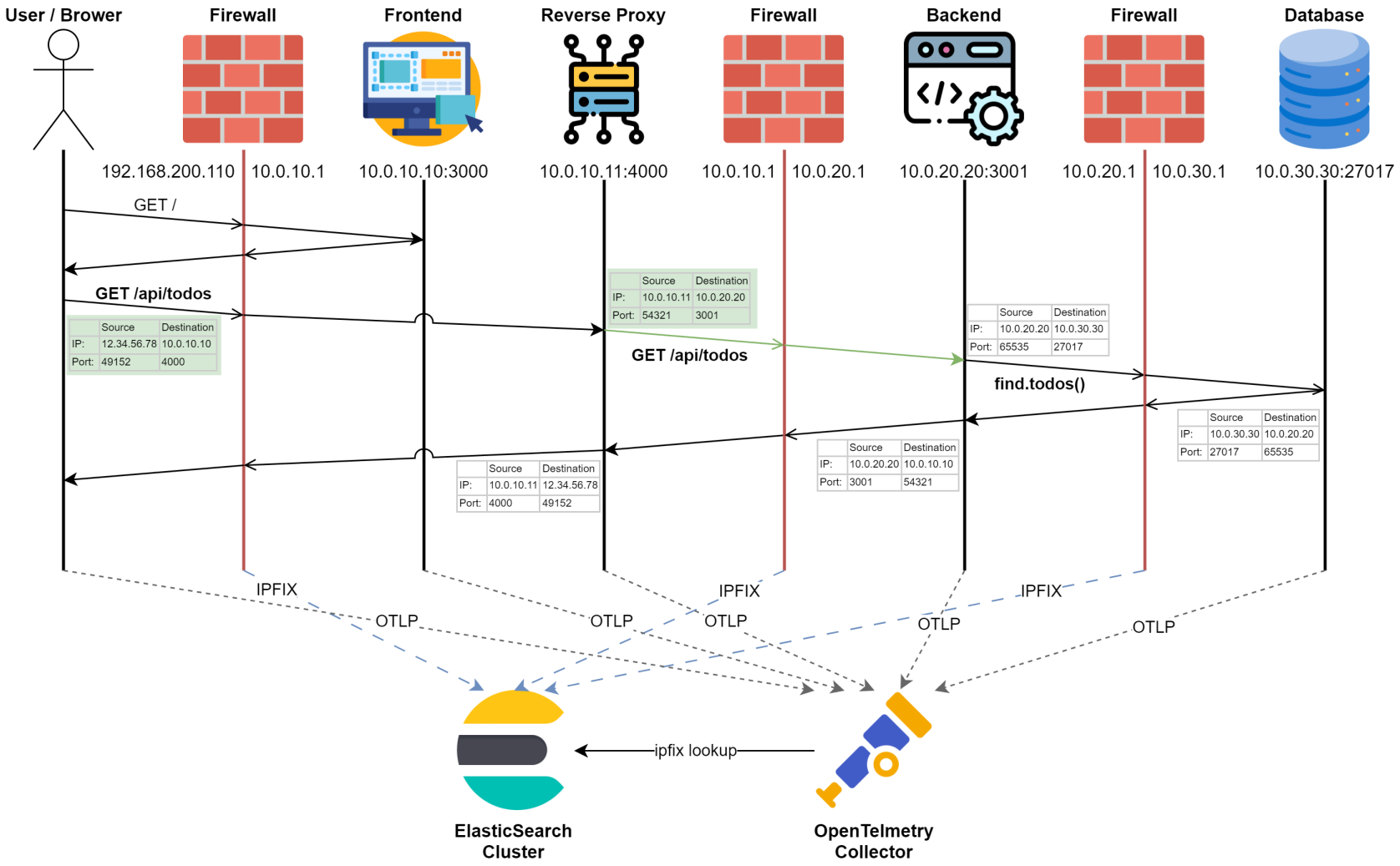

Figure 4.4: MVP packet flow diagram

4.3.1 Lab environment

A lab environment is used for the implementation of the MVP. The lab environment consists of a physical server (HPE DL360 G7) running Proxmox 7.4. The MVP is hosted on virtual machines running on this hypervisor.

Specifications:

CPU	2x Intel(R) Xeon(R) CPU X5675 / 24 Cores
RAM	110 GB
Disk space	1.6 TB SSD

Table 4.1: Lab environment hardware specifications

VMs & hosts

The following table lists the virtual machines and describes which part of the application runs on them.

VM name	Interface - IP	Application / Services
ba-vm-1	eth0 - 192.168.200.100/24	Running an ElasticSearch stack within Kubernetes
frontend	eth0 - 10.0.10.10/24	Running the frontend & backend-proxy as docker container
backend	eth0 - 10.0.20.20/24	Running the backend as docker container
database	eth0 - 10.0.30.30/24	Running the database as docker container
FW01	Frontend (vlan010) 10.0.10.1/24 - WAN 192.168.200.110/24	FortiGate firewall with Netflow / IPFIX service running
FW02	Backend (vlan020) 10.0.20.1/24 - WAN 192.168.200.120/24	FortiGate firewall with Netflow / IPFIX service running
FW03	Database (vlan030) 10.0.30.1/24 - WAN 192.168.200.130/24	FortiGate firewall with Netflow / IPFIX service running

Table 4.2: VMs & hosts

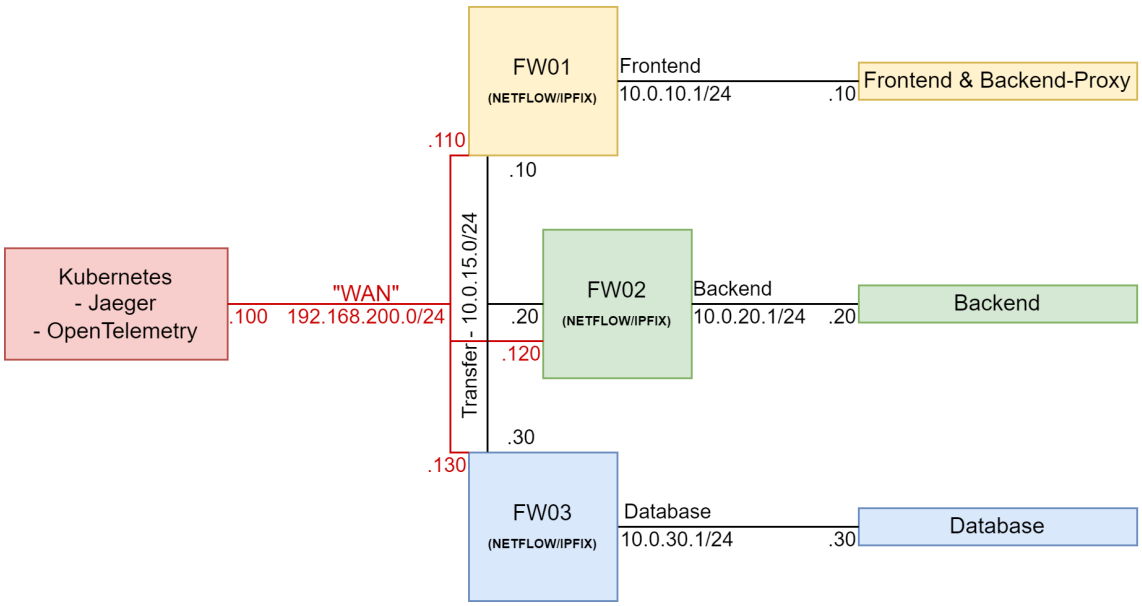

Figure 4.5: MVP Lab environment networking

4.3.2 Network

Subnets

The components of the 3-tier application will be placed into the following different subnets:

Name	Subnet/Mask
Frontend	10.0.10.0 / 255.255.255.0
Backend	10.0.20.0 / 255.255.255.0
Database	10.0.30.0 / 255.255.255.0

Table 4.3: Subnets for the 3-tier application

Firewall / Router

Three virtual instances of a Fortinet FortiGate firewall are in place. Fortinet's FortiGate firewalls are robust network security appliances designed to protect organizations from a wide range of cyber threats. These devices offer a comprehensive security solution, integrating firewall, antivirus, intrusion prevention, and virtual private network (VPN) capabilities in a single platform. FortiGate firewalls employ advanced threat intelligence and machine learning to detect and mitigate evolving threats effectively.

[25]

Netflow / IPFIX configuration

The configuration of Netflow / IPFIX on a FortiGate firewall is quite simple but has to be done on the CLI and not via GUI.

The following is an example configuration from the FW01. First of all, the general Netflow settings have to be configured. This includes setting the IP and the port of the collector, the source IP from which the packets are sent, and the active-flow-timeout plus the inactive-flow-timeout value.

```
FW-FortiGate-1 # show system netflow
config system netflow
    set collector-ip "192.168.200.100"
    set collector-port 32055
    set source-ip "192.168.200.110"
    set active-flow-timeout 60
    set inactive-flow-timeout 10
end
```

After the general Netflow settings have been made, some configuration on the respective interface that Netflow packets should be sent for this interface (set netflow-sampler) needs to be done. The value "both" specifies that Netflow packets will be sent for incoming and outgoing traffic on that particular interface.

```
FW-FortiGate-1 # show system interface
config system interface
    edit "port1"
        set vdom "root"
        set ip 192.168.200.110 255.255.255.0
        set allowaccess ping https ssh http
        set type physical
        set netflow-sampler both
        set alias "WAN"
        set lldp-reception enable
        set role wan
        set snmp-index 1
    next
```

[26]

4.3.3 Application

3-tier application code basis

For the simple 3-tier application, a to-do list application is used. The docker-frontend-backend-db application repository from knaopel is used as the basis for this.

Initially, the application is defined in one docker-compose. This is split up into three different docker-compose deployments that can be deployed on three different servers (frontend, backend, database).

Frontend

The frontend is a simple React / JavaScript application that connects to the backend API and shows all the to-do entries received.

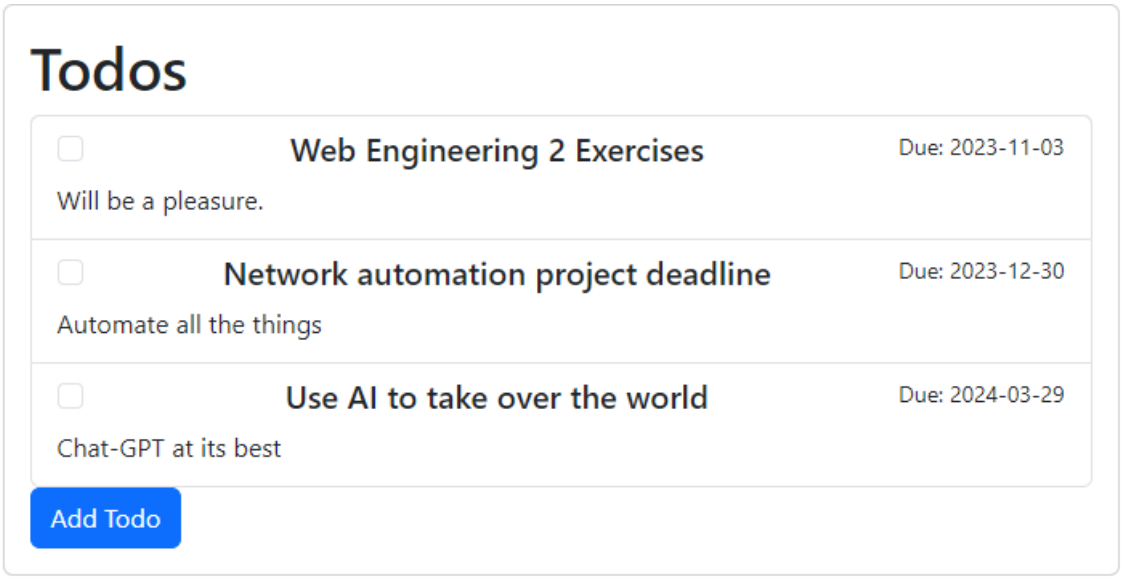

Figure 4.6: 3-tier application - Frontend

Backend

The backend is a JavaScript web server that provides an API interface on /api. It takes the API requests (GET, POST, PATCH, DELETE) and sends the corresponding database commands to the database on the database server.

Database

The database is a MongoDB database version 4.4.18. The version is essential for the lab environment as the newer MongoDB version 5.X requires host CPUs that are able to handle AVX instructions, which is not supported by the lab environment host CPU.

[27]

Adding a reverse proxy

When testing the 3-tier application, it became apparent that when a call is made to the backend on the frontend, the user's browser makes the API call to the backend and not the frontend itself. Therefore, a reverse proxy is set up before the backend so that the call from the frontend or from the user can be instrumented and tracked correctly. For this purpose, a NGINX proxy is used, which receives the calls on /api and forwards them one-to-one to the backend. NGINX was chosen because it can be instrumented with OpenTelemetry.

4.3.4 Instrumenting of the application

To effectively test the new ipfix collector, each component of the 3-tier application needs to be instrumented with OpenTelemetry.

Frontend instrumentation

Instrumenting the frontend is a fascinating topic, as the react-based framework creates all the requests from the user's browser itself. Therefore, two instrumentations are needed in the frontend. One sits on the frontend server, and the other is inside the user's browser. The instrumentation of the browser is described in the next chapter.

The frontend server instrumentation uses the following `instrumentation.js`. It uses the `getNodeAutoInstrumentations()` function, which loads all the instrumentation modules automatically.

```
1  const opentelemetry = require("@opentelemetry/sdk-node");
2  const {
3      getNodeAutoInstrumentations,
4  } = require("@opentelemetry/auto-instrumentations-node");
5  const {
6      OTLPTraceExporter,
7  } = require("@opentelemetry/exporter-trace-otlp-proto");
8  const {
9      diag, DiagConsoleLogger, DiagLogLevel,
10 } = require('@opentelemetry/api');
11
12 diag.setLogger(new DiagConsoleLogger(), DiagLogLevel.DEBUG);
13
14 const sdk = new opentelemetry.NodeSDK({
15   traceExporter: new OTLPTraceExporter({
16     headers: {},
17   }),
```

```
18
19   instrumentations: [
20     getNodeAutoInstrumentations({
21       "@opentelemetry/instrumentation-fs": {
22         enabled: false,
23       },
24     }),
25   ],
26 });
27
28 sdk.start();
```

Next, the following environment variables are added to configure the instrumentation. Especially important is the `NODE_OPTIONS`, which loads the `instrumentation.js` before the app starts.

```
1 ENV OTEL_TRACES_EXPORTER="otlp"
2 ENV OTEL_EXPORTER_OTLP_PROTOCOL="http/protobuf"
3 ENV OTEL_EXPORTER_OTLP_COMPRESSION="gzip"
4 ENV OTEL_EXPORTER_OTLP_ENDPOINT="http://192.168.200.100:30168"
5 ENV OTEL_RESOURCE_ATTRIBUTES="service.namespace=3-tier-app"
6 ENV OTEL_SERVICE_NAME="frontend"
7 ENV NODE_OPTIONS='--require ./instrumentation.js'
```

[28]

Browser instrumentation

The browser instrumentation works a bit differently. The code needs to be injected into the user's browser. Firstly, a file called `tracing.js` is created where the SDK is configured. This time, all the configuration needs to be coded into this file as well because this tracer will run on the user's browser.

```
1 ...redacted...
2 registerInstrumentations({
3   instrumentations: [
4     fetchInstrumentation,
5     // new UserInteractionInstrumentation(),
6     new XMLHttpRequestInstrumentation({
7         propagateTraceHeaderCorsUrls: [new RegExp('.*')]
8       }),
9   ],
10   tracerProvider: provider,
11 });
12
13 export default function TraceProvider({ children }) {
14   return <>{children}</>;
```

```
}
```

Using the `<TraceProvider>` HTML tag, the trace provider from the `tracing.js` can be wrapped around the react app.

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import './index.css';
import App from './App';
import TraceProvider from './tracing';
import './index.css';
import 'bootstrap/dist/css/bootstrap.min.css';

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

[29]

Using this, the browser sends traces to the OpenTelemetry collector when an HTTP request is executed.

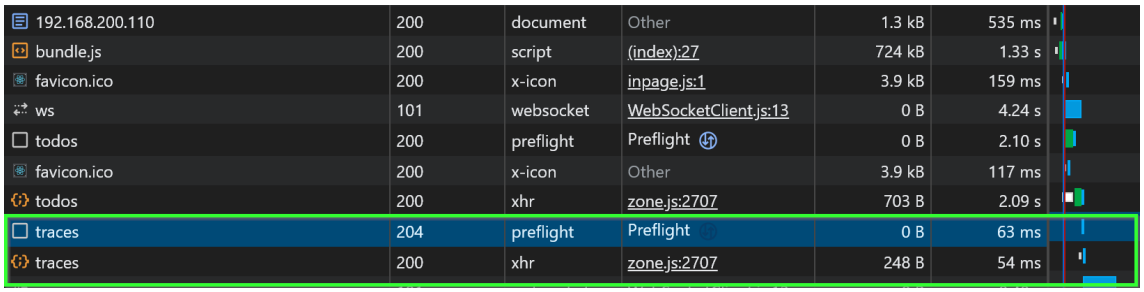

Figure 4.7: Browser sending traces

Backend-Proxy instrumentation

OpenTelemetry has a blog that describes how to implement monitoring for the NGINX proxy. Initially, the versions in the guide had a major problem. The `traceparent` HTTP header would not propagate appropriately to the backend. After updating to the latest versions, everything worked as expected.

Final Dockerfile for the NGINX with OpenTelemetry monitoring:

```
FROM nginx:1.23.1
RUN apt-get update && apt-get install unzip -y

# Add OpenTelemetry
ADD https://github.com/open-telemetry/opentelemetry-cpp-contrib/releases/ ↵
download/webserver%2Fv1.0.3/opentelemetry-webserver-sdk-x64-linux.tgz /opt
RUN cd /opt ; unzip opentelemetry-webserver-sdk-x64-linux.tgz.zip; tar xvfz  ↵
opentelemetry-webserver-sdk-x64-linux.tgz
RUN cd /opt/opentelemetry-webserver-sdk; ./install.sh
ENV LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/opt/opentelemetry-webserver-sdk/ ↵
sdk_lib/lib
RUN echo "load_module /opt/opentelemetry-webserver-sdk/WebServerModule/Nginx ↵
/1.23.1/ngx_http_opentelemetry_module.so;\n$(cat /etc/nginx/nginx.conf)" > / ↵
etc/nginx/nginx.conf

# Add Configs
COPY opentelemetry_module.conf /etc/nginx/conf.d
COPY default.conf /etc/nginx/conf.d/default.conf
```

[30]

A pull request was created to update the blog post: Update NGINX instrumentation versions. #3484

The `opentelemetry_module.con` file needed to connect to the OpenTelemetry collector is straightforward to configure.

```
NginxModuleEnabled ON;
NginxModuleOtelSpanExporter otlp;
NginxModuleOtelExporterEndpoint 192.168.200.100:30167;
NginxModuleServiceName backend-proxy;
NginxModuleServiceNamespace 3-tier-app;
NginxModuleServiceInstanceId BackendProxyId;
NginxModuleResolveBackends ON;
NginxModuleTraceAsError ON;
```

In the later implementation, this NGINX instrumentation was replaced by nginx-aio from dburianov. Which provides the crucial TCP/IP quartet.

Backend instrumentation

Initially, the original backend was using expressjs. However, there was an inexplicable issue where some spans did not get sent to the collector. After extensive troubleshooting, the root cause could not be found. Therefore, it was decided to rewrite the backend in Python. This was completed quickly thanks to some help from ChatGPT.

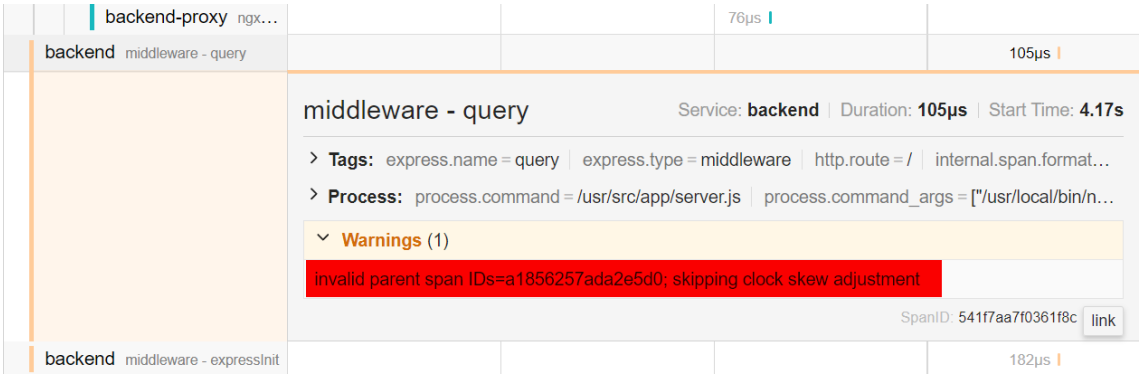

Figure 4.8: Express.js backend - issue with invalid parent span id

Instrumenting the Python application required multiple steps. First, add the following environment variables to the Dockerfile and additionally run the `opentelemetry-bootstrap -a install` command to read through the list of packages installed in the site-packages folder and install the corresponding instrumentation libraries for these packages. Finally, the app can be started using the `opentelemetry-instrument` command

```
1  # Set environment variables for OpenTelemetry
2  ENV OTEL_METRICS_EXPORTER="otlp"
3  ENV OTEL_TRACES_EXPORTER="console,otlp"
4  ENV OTEL_METRICS_EXPORTER="console"
5  ENV OTEL_EXPORTER_OTLP_TRACES_ENDPOINT="http://192.168.200.100:30067"
6  ENV OTEL_RESOURCE_ATTRIBUTES="service.namespace=3-tier-app"
7  ENV OTEL_SERVICE_NAME="backend-python"
8
9  # install opentelemetry in the python packages
10 RUN opentelemetry-bootstrap -a install
11
12 # Start the application with OpenTelemetry instrumentation
13 CMD opentelemetry-instrument gunicorn app:app -b 0.0.0.0:3001
```

Afterward, the HTTP requests to the backend did show up properly in Jaeger. To get the connection to the database reported as well, line number 6 in the Python script itself needed to be added, which manually instruments WSGI. [31]

```python
# FLASK
app = Flask(__name__)
CORS(app)

# If WSGI is not manually instrumented the mongodb spans are not found
app.wsgi_app = OpenTelemetryMiddleware(app.wsgi_app)

# DB
app.config['MONGO_URI'] = 'mongodb://10.0.30.30:27017/todos'
mongo = PyMongo(app)
```

4.4 Result of MVP

The MVP is completed, and the Netflow / IPFIX packets are injected between the proxy and the backend. In figure 4.9, a complete trace from the browser to the database call is represented. As this is only an MVP, the solution still has room for improvement.

There are two IPFIX injections in figure 4.9. They contain the exact same event. Both are injected because both spans contain the TCP/IP quartet. Additionally, the injected span should be the parent of the "backend-python GET /api/todos" span and not its child.

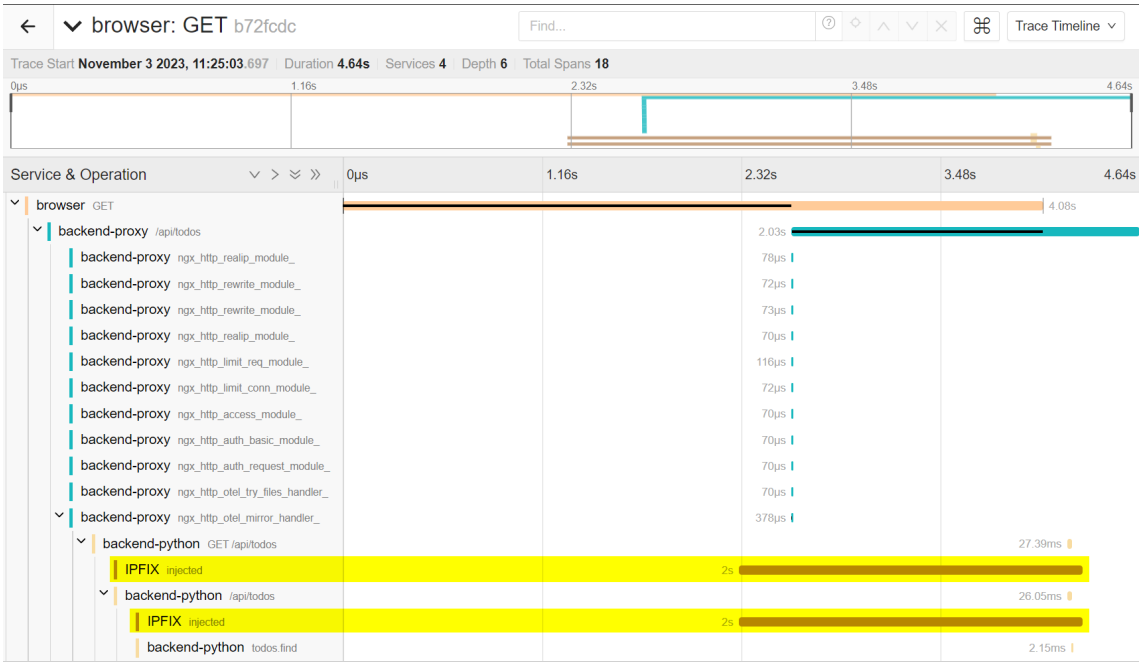

Figure 4.9: Full trace at the end of MVP

4.5 Contributing to the OTel community

The result of the MVP implementation was reviewed and discussed with the supervisors.

In the GitHub repository OpenTelemetry Collector Contrib, there is an issue where the community asks for exactly such an implementation of this Bachelor's thesis. It has been decided that the MVP will not be deployed in a larger environment in a further step. The aim of this Bachelor's thesis is to improve this MVP as far as possible and also to increase the code quality.

In the folder processor in the GitHub repository, all custom processors that have been created by the community can be found. The goal is that the processor of this Bachelor's thesis will be contributed to this collection of custom processors at least in the status of "Last PR" regarding the Contributing Guide.

The various statuses are briefly summarised as follows:

- **First PR** - includes the overall structure of the new component
- **Second PR** - includes the concrete implementation of the component.
- **Last PR** - should mark the new component as *Alpha* stability

4.6 Challenges of MVP

There were a lot of challenges with getting the timing right. The problems where mainly related to network monitoring. To have a good baseline for the time all network devices and servers were configured with the same Network Time Protocol (NTP) server, to reduce the amount of clock drift between them. Next, the behavior of the different Netflow versions was explored.

- **IPFIX/Netflow v10** was not fully supported by ElasticSearch, which caused the `event.duration` field to not automatically compute.
- **Netflow v9** worked well, but the timestamps where consistently 0.5s-1s off. This a known behavior of the softflowd. [32]
- **Netflow v5** worked well and set the time correct, no vital information was lost, downgrading to Netflow v5.

In the demo's final version, the softflowd was set to Netflow v5. However, some timing variations and instability of the softflowd where still present. It was therefore decided to install a Fortinet firewall with which Netflow worked a lot better than with the softflowd.

Chapter 5

Results

5.1 Revisiting the use cases

In this chapter, the use cases defined in 2.3 will be revisited, and it will be determined whether these have been fulfilled or not.

Nr.	Title	Fulfilled?	Justification
1	Latency	Yes	The individual network components between connections are displayed separately, and the latency caused in the connection can be considered individually for the application and the network part.
2	Network components processing time	Yes	The ingress and egress timestamps are visible to the extent that the entire trace has a start time, and the span from the network device has a relative start to this start time. The span has a duration, so the start and end times of ingress and egress are known.
3	Reason for latency	No	In the observability backend, no data can be found for a possible reason for the latency. The selected information retrieval method, Netflow / IPFIX, does not provide any further data about the device's status.
4	Correlate with requests and existing traces	Yes	The spans representing the data from the network components are seamlessly inserted into existing traces in OpenTelemetry.

Table 5.1: Revisiting the use cases

5.2 Limitations

5.2.1 TCP/IP quartet extraction

The approach presented in this Bachelor's thesis to inject Netflow / IPFIX spans into OpenTelemetry traces is based on getting the full TCP/IP quartet values. It was noticed that getting the following three values is relatively common: `source.ip`, `destination.ip` and `destination.port`. The most challenging information to get is the ephemeral `source.port`. This port value is set by the sender when opening the connection. The programming language is not always aware of this port, as the lower-level implementation of the network stack chooses it. The best way to get the `source.port` is paradoxically at the receiver side, which will see the port in the packet. If the receiving side is not instrumented with OpenTelemetry traces, as is often the case with databases, getting information for the Netflow / IPFIX lookup will be nearly impossible. The whole endeavor is further complicated when the components are installed inside containered environments. These environments often use internal NATed networks, making it difficult for the component to know its own IP on the actual network.

5.2.2 Timing

The timestamps displayed in the Jaeger UI will not always be 100% correct. The network layer operates very fast and with very few delays. Packets only take a few milliseconds to pass from one server to another. Therefore, even when all clocks are synced with NTP, the timing will not always line up correctly. In addition to these physical limitations, the Netflow / IPFIX export is not built to be accurate down to the millisecond.

To demonstrate this inaccuracy, a network packet capture was started on all three firewalls and all three virtual machines. Next, the demo app was opened once. In figure 5.2, the trace for this request can be seen, and in figure 5.1, the actual HTTP packets are displayed. The packet captures reveal an interesting truth, as the measured timestamps do not match the reality of the packet flow. The timing of all packets is very close. The first and last packages are only 0.044555s (44.555ms) apart. This is not a lot and will lead to inaccuracy as shown by *Peeling Away Timing Error in NetFlow Data*: This limits the accuracy to about 70ms, reinforcing the point that implementation matters when conducting research on network measurement data. [33]. This same effect leads to the red marked section in 5.1 where it seems that the Firewalls have 120ms of delay between each other, while in reality, there is none.

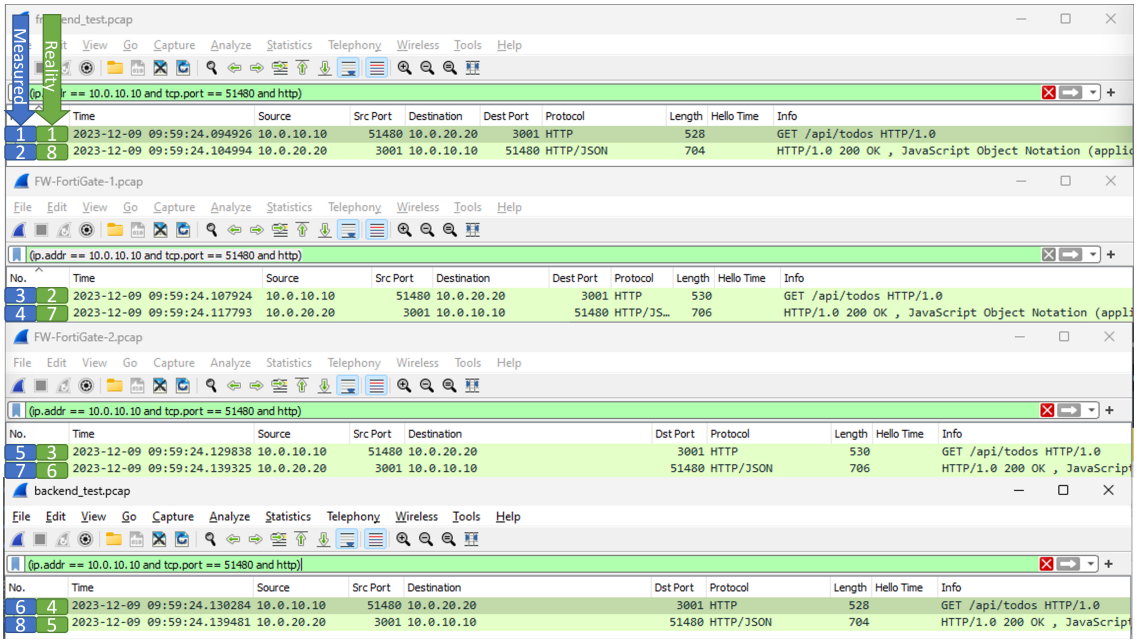

Figure 5.1: Packet capture of HTTP request

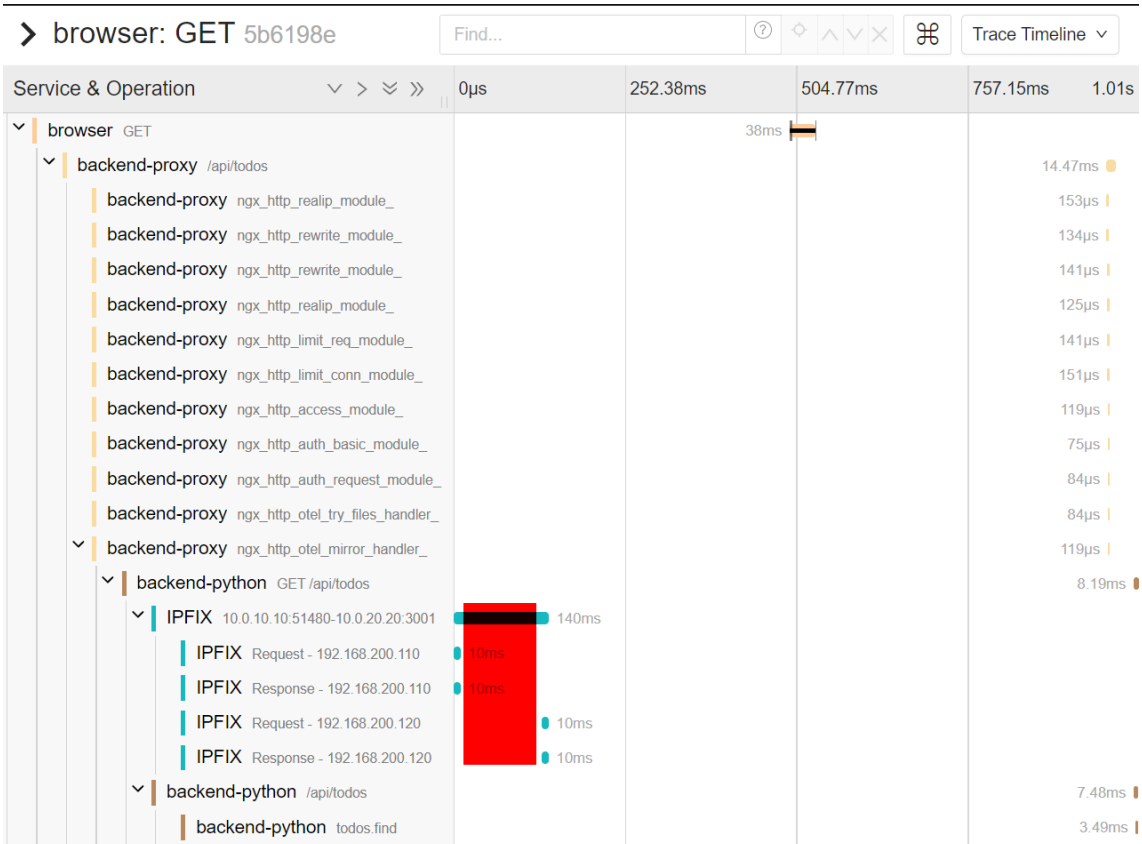

Figure 5.2: Trace in Jaeger UI of HTTP request

5.2.3 Same ephemeral port used by several connections

Because the TCP/IP quartet is used as the basis for the composition of a connection in a trace, it is possible that if several connections are made at random, or the client reuses the same range of ephemeral ports, it can happen that connections from different sessions are displayed in the same trace because the ephemeral port matches and does not differ within the `lookup_window` of the processor configuration. Therefore, the value of `lookup_window` should not be set too high.

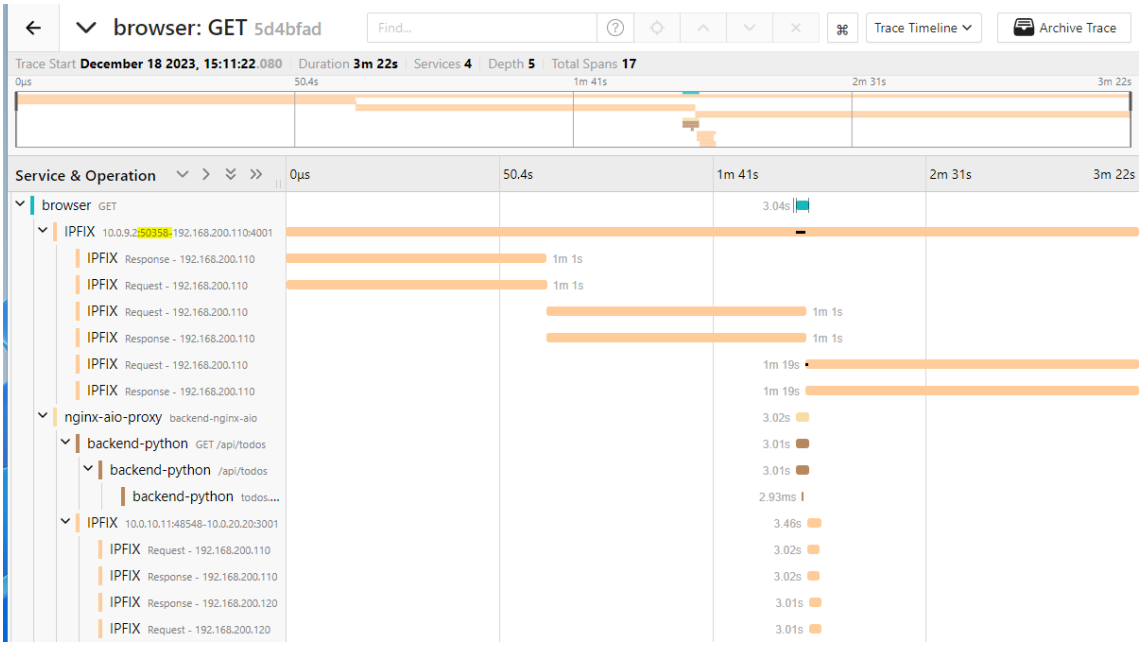

Figure 5.3: Too many spans are displayed with `lookup_window` of 200s

In ElasticSearch, it can be seen that the ephemeral port 50358 occurs several times over a short period, which is not optimal for the display in the observability backend.

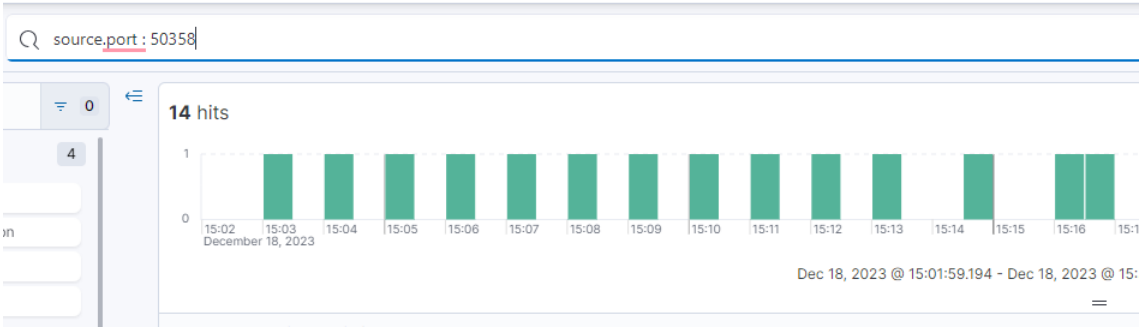

Figure 5.4: ElasticSearch ephemeral port 50358 reuse

It is, therefore, vital that you adapt the `lookup_window` for your environment accordingly. During the Lab testing a value of 25 seconds was used.

Chapter 6

Conclusion

This chapter reflects on the goals and outcomes of the Bachelor's thesis.

6.1 Conclusion

The main goal of this Bachelor's thesis was to integrate telemetry data from network devices for a connection between the different tiers of a 3-tier application into an existing trace in OpenTelemetry. Four different variants for obtaining network telemetry data were analyzed and evaluated, and a solution was developed using data via Netflow / IPFIX. The developed solution consists of providing a custom processor within an OpenTelemetry collector, which independently retrieves and integrates the network telemetry data collected in an ElasticSearch cluster for existing traces of a specific application.

The solution was developed based on an on-premise 3-tier application and fulfilled three of the four defined use cases. The fourth use case can certainly be supplemented by future work.

The code was divided into two pull requests as required by OpenTelemetry and is ready for a contribution.

6.2 Open pull requests / contribution

The implementation of the custom processor for the OpenTelemetry collector is stored in a separate fork of the original OpenTelemetry collector contrib. There are two branches (framework and implementation).

Direct link: https://github.com/fizzers123/opentelemetry-collector-contrib

The submitted pull requests for the contribution to the OpenTelemetry collector project are listed in the following table:

Description	PR ID	Direct Link
Framework	#30194	github.com/open-telemetry/opentelemetry-collector-contrib/pull/30194
Implementation	#30195	github.com/open-telemetry/opentelemetry-collector-contrib/pull/30195

Table 6.1: Pull requests and their direct links

6.3 Possible future improvements

6.3.1 Test within Kubernetes

Within the time frame of the Bachelor's thesis, it was impossible to test the custom collector's behavior with an application in Kubernetes. Exciting topics would be the NAT in Kubernetes and whether a CNI can export Netflow / IPFIX information from the Kubernetes network.

6.3.2 Add device metrics for possible reason for latency

As shown in the chapter 5.1, use case 3 was not achieved with the current implementation of a custom collector together with Netflow / IPFIX data. As a possible extension to fulfill this use case, SNMP data could also be collected in a central location, and if the lookup for the Netflow / IPFIX data is made, monitoring data could also be retrieved and displayed via the network device at the same time.

Glossary

ChatGPT ChatGPT is an AI-powered language model developed by OpenAI, capable of generating human-like text based on context and past conversations. https://chat.openai.com/ 47

CNI CNI (Container Network Interface), a Cloud Native Computing Foundation project, consists of a specification and libraries for writing plugins to configure network interfaces in Linux containers, along with a number of supported plugins. CNI concerns itself only with network connectivity of containers and removing allocated resources when the container is deleted. https://github.com/containernetworking/cni 25

Dockerfile A Dockerfile is a text document that contains all the commands a user could call on the command line to assemble an image. https://docs.docker.com/engine/reference/builder/#dockerfile-reference 46, 47

ECTS European Credit Transfer and Accumulation System https://en.wikipedia.org/wiki/European_Credit_Transfer_and_Accumulation_System 3

ElasticSearch Elasticsearch is a distributed, RESTful search and analytics engine https://www.elastic.co/elasticsearch/ iii, 25, 29, 32, 33, 34, 35, 36, 39, 49, 53, 55, 60

ephemeral port Ephemeral ports (49152-65535) are temporary communication endpoints automatically assigned by operating systems for outbound network connections, ensuring efficient and dynamic client-server interactions. https://en.wikipedia.org/wiki/Ephemeral_port 15, 53, 60

expressjs Fast, unopinionated, minimalist web framework for Node.js https://expressjs.com/ 47

IOAM In-band Operations, Administration, and Maintenance (IOAM) is a network measurement and monitoring technology

https://info.support.huawei.com/info-finder/encyclopedia/en/IOAM.html 23, 25, 60

Jaeger Open source, distributed tracing platform. Monitor and troubleshoot workflows in complex distributed systems. https://www.jaegertracing.io/ 9, 10, 13, 14, 35, 47, 51, 52, 60

Kubernetes Kubernetes is an open source container orchestration engine for automating deployment, scaling, and management of containerized applications. The open source project is hosted by the Cloud Native Computing Foundation (CNCF). [34] https://kubernetes.io/docs/home/ 17, 24, 25, 39, 56

NAT Network address translation (NAT) is a method of mapping an IP address space into another by modifying network address information in the IP header of packets while they are in transit across a traffic routing device. https://en.wikipedia.org/wiki/Network_address_translation 51, 56

NGINX Configure NGINX as a reverse proxy for HTTP https://docs.nginx.com/nginx/admin-guide/web-server/reverse-proxy/ 43, 46

NTP NTP is intended to synchronize all participating computers to within a few milliseconds of Coordinated Universal Time (UTC) https://en.wikipedia.org/wiki/Network_Time_Protocol 49, 51

OpenTelemetry OpenTelemetry is an Observability framework and toolkit designed to create and manage telemetry data such as traces, metrics, and logs. Crucially, OpenTelemetry is vendor- and tool-agnostic, meaning that it can be used with a broad variety of Observability backends, including open source tools like Jaeger and Prometheus, as well as commercial offerings. OpenTelemetry is a Cloud Native Computing Foundation (CNCF) project. [35] https://opentelemetry.io/docs/specs/otel/ i, iii, 1, 2, 4, 5, 9, 10, 11, 13, 15, 26, 29, 31, 32, 33, 34, 36, 43, 45, 46, 50, 51, 55, 56

OSI model In the Open Systems Interconnection model (OSI model) the communications between a computing system are split into seven different abstraction layers: Physical, Data Link, Network, Transport, Session, Presentation, and Application. https://en.wikipedia.org/wiki/OSI_model 22, 25, 26

OTLP The OpenTelemetry Protocol (OTLP) specification describes the encoding, transport, and delivery mechanism of telemetry data between telemetry sources, intermediate nodes such as collectors and telemetry backends. [35]
https://opentelemetry.io/docs/specs/otlp/ 13, 14, 60

Prometheus Prometheus is an open-source systems monitoring and alerting toolkit originally built at SoundCloud. Prometheus collects and stores its metrics as time series data, i.e. metrics information is stored with the timestamp at which it was recorded, alongside optional key-value pairs called labels.
https://prometheus.io/ 9

RFC3339 Date and Time on the Internet: Timestamps
https://datatracker.ietf.org/doc/html/rfc3339 33

WSGI PEP 3333 – Python Web Server Gateway Interface
https://peps.python.org/pep-3333/ 47

List of Figures

3.1 HTTP traceparent header . . . 10
3.2 3-tier-application . . . 12
3.3 Frontend to backend connection in Jaeger using OTLP events . . . 13
3.4 Frontend to backend connection in Jaeger using OTLP spans . . . 14
3.5 TCP/IP information for trace mapping . . . 15
3.6 Path tracing extension header [8] . . . 20
3.7 IOAM network system [11] . . . 23
3.8 OTEL collector components . . . 27

4.1 Correlation unit data flow . . . 30
4.2 Netflow / IPFIX data in ElasticSearch . . . 34
4.3 Summary span in Jaeger . . . 35
4.4 MVP packet flow diagram . . . 37
4.5 MVP Lab environment networking . . . 39
4.6 3-tier application - Frontend . . . 42
4.7 Browser sending traces . . . 45
4.8 Express.js backend - issue with invalid parent span id . . . 47
4.9 Full trace at the end of MVP . . . 48

5.1 Packet capture of HTTP request . . . 52
5.2 Trace in Jaeger UI of HTTP request . . . 52
5.3 Too many spans are displayed with `lookup_window` of 200s . . . 53
5.4 ElasticSearch ephemeral port 50358 reuse . . . 53

List of Tables

2.1 Descriptions of the use cases 6

3.1 Comparison of the options 14
3.2 Description of requirements for network data collection 17
3.3 Evaluation of requirements for network data collection 24
3.4 Justification of whether a variation meets the requirement 25

4.1 Lab environment hardware specifications 38
4.2 VMs & hosts 39
4.3 Subnets for the 3-tier application 40

5.1 Revisiting the use cases 50

6.1 Pull requests and their direct links 56

Bibliography

[1] Traces from opentelemetry. [Online]. Available: https://opentelemetry.io/docs/concepts/signals/traces/

[2] Metrics from opentelemetry. [Online]. Available: https://opentelemetry.io/docs/concepts/signals/metrics/

[3] Metrics from opentelemetry. [Online]. Available: https://opentelemetry.io/docs/concepts/signals/logs/

[4] Baggage from opentelemetry. [Online]. Available: https://opentelemetry.io/docs/concepts/signals/baggage/

[5] Http trace-context field. [Online]. Available: https://www.w3.org/TR/trace-context/

[6] Ipfix vs. netflow. [Online]. Available: https://blog.gigamon.com/2019/09/17/ipfix-vs-netflow/

[7] Srv6. [Online]. Available: https://info.support.huawei.com/info-finder/encyclopedia/en/SRv6.html

[8] Path tracing. [Online]. Available: https://www.segment-routing.net/path-tracing/pt-tutorial/

[9] Tcpdump official website. [Online]. Available: https://www.tcpdump.org/

[10] Rfc 9197 data fields for in situ operations, administration, and maintenance (ioam). [Online]. Available: https://www.rfc-editor.org/rfc/rfc9197.html

[11] Huawei - what is ioam. [Online]. Available: https://info.support.huawei.com/info-finder/encyclopedia/en/IOAM.html

[12] Ip flow information export (ipfix) entities - iana. [Online]. Available: https://www.iana.org/assignments/ipfix/ipfix.xhtml

[13] Cisco nexus 9000 series nx-os system management configuration guide, release 7.x. [Online]. Available:

https://www.cisco.com/c/en/us/td/docs/switches/datacenter/nexus9000/sw/7-x/system_management/configuration/guide/b_Cisco_Nexus_9000_Series_NX-OS_System_Management_Configuration_Guide_7x/b_Cisco_Nexus_9000_Series_NX-OS_System_Management_Configuration_Guide_7x_chapter_011100.html

[14] Management of large scale netflow data by distributed systems. [Online]. Available: https://ntnuopen.ntnu.no/ntnu-xmlui/handle/11250/2410246

[15] Tracking network flows with ovn cni. [Online]. Available: https://docs.okd.io/4.10/networking/ovn_kubernetes_network_provider/tracking-network-flows.html

[16] Wikipedia - netflow. [Online]. Available: https://en.wikipedia.org/wiki/Netflow

[17] Netstream packets. [Online]. Available: https://support.huawei.com/enterprise/en/doc/EDOC1100127059/5dcef217/netstream-packetss

[18] Netstream packet samping. [Online]. Available: https://support.huawei.com/enterprise/en/doc/EDOC1100127059/60f64bd/netstream-packet-sampling

[19] Ipv6 in-situ operations, administration, and maintenance. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2665963820300270

[20] Otel collector. [Online]. Available: https://opentelemetry.io/docs/collector/

[21] Ios xr 7.8.1 on cisco asr 9000 series. [Online]. Available: ttps://www.cisco.com/c/en/us/td/docs/routers/asr9000/software/asr9k-r7-8/general/release/notes/b-release-notes-asr9000-r781.html

[22] Ios xr 7.8.1 on cisco 8000 series. [Online]. Available: https://www.cisco.com/c/en/us/td/docs/iosxr/cisco8000/general/78x/release/notes/b-release-notes-cisco8k-r781.html

[23] Rfc3954 - cisco systems netflow services export version 9. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc3954

[24] Elasticsearch netflow/ipfix integration. [Online]. Available: https://docs.elastic.co/en/integrations/netflow

[25] Fortigate firewall. [Online]. Available: https://www.fortinet.com/de/products/next-generation-firewall

[26] Netflow configuration on a fortigate. [Online]. Available: https://community.fortinet.com/t5/FortiGate/Technical-Tip-How-to-Configure-Netflow/ta-p/196080

[27] Advanced vector extensions - intel. [Online]. Available: https://de.wikipedia.org/wiki/Advanced_Vector_Extensions

[28] Opentelemetry javascript instrumentation. [Online]. Available: https://opentelemetry.io/docs/instrumentation/js/automatic/

[29] Opentelemetry react instrumentation. [Online]. Available: https://developers.redhat.com/articles/2023/03/22/how-enable-opentelemetry-traces-react-applications#10_step_opentelemetry_demonstration

[30] Instrument nginx. [Online]. Available: https://opentelemetry.io/blog/2022/instrument-nginx/

[31] Opentelemetry python instrumentation. [Online]. Available: https://opentelemetry.io/docs/instrumentation/python/automatic/

[32] softflowd-timestamps-wrong. [Online]. Available: https://forum.netgate.com/topic/62036/softflowd-nfdump-timestamps-wrong/3

[33] Peeling away timing error in netflow data. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-642-19260-9_20

[34] Kubernetes documentation. [Online]. Available: https://kubernetes.io/docs/home/

[35] Opentelemetry. [Online]. Available: https://opentelemetry.io/docs/what-is-opentelemetry/