

# Todo list

□ : TLDR what have I done . . . . .	i
□ Introduction: container glossary entry . . . . .	1
□ <b>Introduction - what flow stuff does or doesn't</b> . . . . .	2
□ <b>Previous Work - What was coders</b> . . . . .	3
□ <b>Previous Work - What was missing</b> . . . . .	3
□ <b>Previous Work - some filler</b> . . . . .	3
□ <b>Project Aims - 2nd draft</b> . . . . .	4
□ <b>Project Aims - 1st draft</b> . . . . .	4
Figure: <i>networkcard</i> → <i>libpcap/af_packet/pf_ring</i> → <i>gopacket</i> → <i>output</i> .	5
□ Overhead: Existing solution architectures . . . . .	6
□ Protocol: calculate difference between raw packet and packet message . . .	11
□ multi packet: gRPC: HTTP/2 transport (any thoughts on SCTP) . . . . .	11
□ multi packet: figure of message? . . . . .	11
□ Message Design: 5 Message Design . . . . .	12
□ Agent: 8 Agent design . . . . .	14
□ Collector: 9 Collector design . . . . .	15
□ Query API: 10 Query API design . . . . .	15

## Final Year Project

---

■ UI: 11 UI Design . . . . .	15
■ Syntetic workload: 12 Syntetic workload . . . . .	16
■ deployment: 13 Deployed . . . . .	16

[Figure 1 about here.]

# AN EXTENSIBLE FRAMEWORK FOR PORTABLE AND DISTRIBUTED PACKET CAPTURE

By

CHARLES DE FREITAS

Student ID: 1810538

Course: MSci Computer Science

Supervisor: Dr Ian Batten

Word Count: 1421

## ABSTRACT

Use of network monitoring solutions as for a long time been standard practice. With many powerful protocols and technologies being adopted; often thanks to their excellent performance, or alternatively their deep integration in a complete network infrastructure solution. New and novel approaches to achieving the same results in virtual and dynamic environments, where a unique host may be much more short lived and more focused in it's activity.

long sentence

The use of network monitoring for both per-host insights and traffic flow statistics opens the way for endless realworld applications of this data. This work implements extensible capture and decoding of network events, which can then be aggregated into the broader framework. The

This sounds a bit odd

: TLDR what have I done

# Contents

	Page
<b>Abstract</b> . . . . .	i
<b>Contents</b> . . . . .	ii
<hr/>	
<b>1 Introduction</b> . . . . .	1
1.1 Previous Work . . . . .	3
1.2 Project Aims . . . . .	4
1.3 Related Work . . . . .	5
1.3.1 Packet capture . . . . .	5
1.3.1.1 Current options . . . . .	5
1.3.1.2 reading off the wire . . . . .	6
1.3.1.3 decoding . . . . .	6
1.3.2 Flow monitoring . . . . .	6
1.3.2.1 Current options . . . . .	6
1.3.2.2 correlating packets . . . . .	6
1.3.2.3 Flow Statistics . . . . .	6
1.3.2.4 Overhead . . . . .	6
1.3.2.5 IPFIX . . . . .	6
1.3.3 Application vs Network monitoring? . . . . .	7
<b>2 Design</b> . . . . .	8
2.1 Planning . . . . .	8
2.1.1 Portability . . . . .	9
2.1.2 Extensibility . . . . .	11
2.1.3 Scalability . . . . .	11
2.2 Protocol . . . . .	11
2.2.1 event types . . . . .	11
2.2.1.1 single packet . . . . .	11
2.2.1.2 multi packet . . . . .	11
2.3 Message Design . . . . .	12
2.3.1 Flows . . . . .	12
2.3.2 Rich Details . . . . .	12

2.3.3	Sampling . . . . .	13
<b>3</b>	<b>Implementation . . . . .</b>	<b>14</b>
3.1	Architecture . . . . .	14
3.2	Agent . . . . .	14
3.2.1	Cross Platform Support . . . . .	15
3.2.2	Packet Capture . . . . .	15
3.2.3	Decoding . . . . .	15
3.2.3.1	Performance . . . . .	15
3.3	Collector . . . . .	15
3.3.1	RPC Service . . . . .	15
3.3.2	Database Connections . . . . .	15
3.4	Query API . . . . .	15
3.4.1	Request Mapping . . . . .	15
3.4.2	Performance . . . . .	15
3.5	UI . . . . .	15
<b>4</b>	<b>Evaluation . . . . .</b>	<b>16</b>
4.1	Syntetic workload . . . . .	16
4.2	deployment . . . . .	16
4.3	Use Cases . . . . .	17
4.3.1	Types of Use cases evaluated . . . . .	17
4.3.1.1	Hypervisor . . . . .	17
<b>5</b>	<b>Conclusion . . . . .</b>	<b>18</b>
<b>A</b>	<b>Benchmarks . . . . .</b>	<b>19</b>
<hr/>		
	<b>Glossary . . . . .</b>	<b>20</b>
	<b>Acronyms . . . . .</b>	<b>21</b>
	<b>Nomenclature . . . . .</b>	<b>22</b>
	<b>Figures . . . . .</b>	<b>22</b>
	<b>Tables . . . . .</b>	<b>22</b>
	<b>Code Snippets . . . . .</b>	<b>22</b>
	<b>References . . . . .</b>	<b>24</b>
	<b>WIP . . . . .</b>	<b>36</b>

# Chapter One

## Introduction

Network monitoring to some degree has long been a widespread practice. Ranging from capturing a single stream of packets, to watching 1000s of devices communicate across a vast corporate network. Having access to structured network traffic can help an individual track down issues in their own connectivity or that of an application, right through to being just as applicable to performing complex and automated analysis of traffic, in an attempt to detect DDOS attacks or measure QoS. This data could simply be a dump of packets, or sampled data that was observed from multiple points over a network.

The process for capturing traffic on a single host are readily available and accessible to the average user. Anyone can spin up Wireshark and tell it to give them all of the DNS requests made, which can then be explored through the structured breakdown of each individual packet; alternatively saving this data to a file for later examination. The same logic hold for monitoring the performance of a website, one could simply collect packets on the computer running the website and use this to make an educated guess as to where issues could be arising in the network.

However, this type of approach will immediately run into issues. However, problems can quickly begin to crop up. Wireshark will seriously struggle to capture a high throughput of data, while loading large numbers of packets will cause issues with the GUI and exploring the breakdown of each packet or flow. Let alone sifting through ever growing records of traffic. This approach is also fundamentally limited by the scope of the traffic seen, you can only get a somewhat limited view from the perspective of a single computer on a network. This is where purpose build solutions come into play. Enterprise Network switches and routers that come with built in support for capturing traffic are readily available (albeit less accessible to

Introduction:  
container glos-  
sary entry

List of “IPFIX”  
projects [list](#)

the individual user). By shifting this responsibility to the infrastructure, we have also shifted the point of observation to one that sees traffic destined for multiple hosts. Often also taking advantage of specialized hardware which can deal with much larger volumes of traffic. When collecting from multiple points, this data can be cross-referenced to create much more contextually aware outputs.

The most immediate issue that crops up with dedicated solutions is one of cost. To get the full advantages of a wide observation domain, compatible devices will need to be deployed throughout the network; otherwise, what is effectively the same issue as mentioned before when capturing from a single host. The scope of the data is limited to that one point.

As previously mentioned, most all enterprise network devices will support some form of network monitoring. With protocols such as: sFlow, Network (plus jFlow if you're feeling fancy) and more recently IPFIX being the de facto choices in industry. While these protocols have been designed for enterprise scale and there is an abundance of solutions for processing this data, supported hardware can be vastly expensive; and standalone applications for actually producing this data are scarce or expensive in their own right.

### **Introduction - what flow stuff does or doesn't**

Flow monitoring protocols offer the facility for distributed traffic monitoring, enabling a plethora of use cases for the dataset produced.

Specialised tools do exist, they may take the form of x or y However there may not be one for all use cases. You may end up needing multiple solutions.

---

What if we could create a generic framework for handling the collection and processing of packets, for the purposes of distributed monitoring of a network domain? Could we build upon the strong properties of industry tried and tested protocols. Taking advantage of the excellent tooling of modern stuff, along with making it generic enough to be applied to an extremely broad range of target platforms and environments.

This project explores the current solutions to network observability, where these show their strengths and weaknesses. It addresses the requirements and expectations such a framework would need. And then implements a full end-to-end solution for distributed network flow and event monitoring which demonstrate the performance and portability needed, while additionally making it trivial to extend the system to



support structured processing of new (or the same) protocols.

## 1.1 Previous Work

---

### Previous Work - What was coders

- I made coders
- It was cool

I wanted to explore this area after working on a project that focused on provisioning container based Integrated Development Environments (IDEs). While this was a somewhat contrived use case, I had hoped to be able to view the same type of ‘per-host’ statistics that you would get from a normal monitoring solution.

The idea for this project primarily stemmed from a previous system I developed. The purpose of which was to offer fast access to pre-configured Integrated Development Environment (IDE) on demand. By using kubernetes (K8S) as a container orchestrator, the system was capable of dynamically placing each new IDE in the same isolated network (or separately). This was with the intention of allowing further expansion to include networked resources that could be shared among a team. See Figure A.1 on Page 26 Here I want to talk about the previous project I worked on and the issues it raised.

---

### Previous Work - What was missing

something

---

### Previous Work - some filler

Since the power of these monitoring solutions come from having multiple points of reference, I wanted a platform independent solution for aggregating network events and statistics. Making it easy to monitor traffic with reference to any existing or new hosts you might have. Along with being able to apply this to anything from a physical host to a container.

[Figure 2 about here.]

## 1.2 Project Aims

### Project Aims - 2nd draft

- Provide a mechanism for collecting traffic from as broad a range of platforms as possible
- Build a light weight application for capturing and transforming network traffic into structured events
- portability!!!

The aim of this project is to build a framework for easily implementing aggregated traffic capture and the subsequent structuring and collating of this data.

The focus of this project will be specifically on the portability of the final implementation for traffic capture.

repetitive sentence start

The framework will also be Secure by design (SBD).

While existing protocols such as IPFIX set out that “sensitive data **should** be transmitted to the Collecting Process using a means that secures its contents against eavesdropping” [4, p. 55].

Since this project encompasses both and implementation, it will make no assumptions about the integrity of the network [1]

---

### Project Aims - 1st draft

The aim of this project was to build a system for easy collection and aggregation of network related events and statistics. With a specific focus on the ability to target a wide range of platforms / architectures; while also making it straightforward to

deploy in both virtual and physical environments.

The next core focus was that the data captured should have the potential to carry protocol specific information. e.g. including the raw query provisioning container based IDEs. While this was a somewhat contrived use case, I had hoped to be able to view the same type of ‘per-host’ statistics that you would get from a normal monitoring solution. in DNS events.


expand on reason  
for rich events

---

## 1.3 Related Work

### 1.3.1 Packet capture

#### 1.3.1.1 Current options



Missing  
figure

```
networkcard- > libpcap/af_packet/pf_ring- >  
gopacket- > output
```

### 1.3.1.2 reading off the wire

### 1.3.1.3 decoding

## 1.3.2 Flow monitoring

### 1.3.2.1 Current options

### 1.3.2.2 correlating packets

### 1.3.2.3 Flow Statistics

### 1.3.2.4 Overhead

Due to the volumes of data a network may be subjected to, there is the risk of each component of the collection process becoming overloaded. Sampling of the packets can significantly reduce the volume of traffic produced [2] (*TableX*). goes in depth on the bottlenecks and achieved performance with flow monitoring.

Overload

Overhead: Existing solution architectures

### 1.3.2.5 IPFIX

Internet Protocol Flow Information Export (IPFIX) is a protocol used in the transmission of traffic flow information across a network. By purely describing how information should be structured, regardless of transport [3] protocol. IPFIX implementations MUST support Stream Control Transmission Protocol (SCTP) but may also support TCP and UDP independently [4].

this isn't the same as my current def of flows

What was previously described as a flow differs from what is referred to here. A Flow is defined as a set of packets or frames passing an Observation Point in the network during a certain time interval. All packets belonging to a particular Flow have a set of common properties [4, p. 8] .

The core principle of capturing traffic flows is observing traffic from multiple points, as to collect it and process as one data-set.

[Figure 3 about here.]

[Figure 4 about here.]

### **1.3.3 Application vs Network monitoring?**

# Chapter Two

## Design

While working on the project it went through several iterations to further refine the design.

### 2.1 Planning

As set out in the project aims, there are 3 key points I want to address with this solution:

- Portability, to maximise the variety of platforms we will be able to monitor.
- Extensibility, it should be trivial to extend currently supported protocols; along with adding support for completely new ones.
- Scalability, any real world setting where traffic monitoring is needed will have large volumes of **traffic**, making operation at scale a fundamental requirement. numbers!?

### 2.1.1 Portability

When I say portability, I mean being able to get the code to compile for multiple platforms, as apposed to being able to run the same binary in more places.

As previously mentioned, relying on hardware that supports something like NetFlow removes the need for additional devices but it does limit you to these ones.

both the range of target architectures and deployment scenarios

The goal is to develop a stand alone binary that, given a sequence of packets will pick out the appropriate packets and extract the desired information. This can be broken down to three main points: collecting, decoding and marshaling; see Figure A.4 on Page 29 .

[Figure 5 about here.]

To minimise the footprint of said program, a compiled language over one which is interpreted seems like the logical choice.

While relying on a complete infrastructure and monitoring solution, you are bound to encounter some amount of lock-in.

Fundamentally, by being able to target (build for) a greater range of Operating Systems (OSs), it will allow for more flexible deployments.

this argument is coming from my personal use of unifi. . .

cite? this sounds a bit rubbish



## 2.1.2 Extensibility

## 2.1.3 Scalability

does this clash with performance?

## 2.2 Protocol

rich traffic

To replicate the rich details that are available through Wireshark [5, Section 6.1] without generating prohibitively large network overhead, the data must be extracted and encoded at the point of collection. Take the case of summarising a TCP stream. Including full packets would amplify traffic.

expand on this

Protocol: calculate difference between raw packet and packet message

comparing encoding size to PCAP?

### 2.2.1 event types

#### 2.2.1.1 single packet

DNS / TLS

#### 2.2.1.2 multi packet

Flow statistics

multi packet: gRPC: HTTP/2 transport (any thoughts on SCTP)

gRPC uses HTTP/2

HTTP/2 vs SCTP [here](#)

Protocol buffers are a language-neutral, platform-neutral, extensible mechanism for serializing structured data .

At the core of the framework is the message definitions.

multi packet: figure of message?

```
message Flow {  
    uint32 src_port = 1;  
    uint32 dst_port = 2;  
    string src_address = 3;  
    string dst_address = 4;  
    google.protobuf.Timestamp timestamp = 5;  
    uint64 id = 6;  
    repeated Packet packets = 7;  
    uint64 packet_count = 8;  
}
```

Listing 1: Flow Definition

The base purpose being to define the details that can be associated with a given event. The structure of a messages is intentionally very loose, allowing for it to be tailored to any level of detail. By using protocol buffer as a base throughout, I'm able to update existing definitions without affecting any existing functionality.

define event

Given the goals of **extensibility** and **performance** I ended up with some key similarities with IPFIX.

## 2.3 Message Design

Message Design: 5 Message Design

Since I wanted to be able to export detailed protocol information without the overhead of sending all the captured traffic, the message structure needed to be able to carry these additional details.

### 2.3.1 Flows

### 2.3.2 Rich Details

The risk of data amplification. Packet capture on low end / embedded devices may be CPU or memory bound

### **2.3.3 Sampling**

# Chapter Three

## Implementation

### 3.1 Architecture

[Figure 6 about here.]

### 3.2 Agent

Agent: 8 Agent design

Based on the existing solutions that have been discussed, I chose to start with a push based agent.

[Figure 7 about here.]

14 implications  
of performance  
overhead when  
monitoring on  
host

### 3.2.1 Cross Platform Support

### 3.2.2 Packet Capture

### 3.2.3 Decoding

#### 3.2.3.1 Performance

## 3.3 Collector

Collector: 9 Collector design

### 3.3.1 RPC Service

### 3.3.2 Database Connections

## 3.4 Query API

Query API: 10 Query API design

### 3.4.1 Request Mapping

### 3.4.2 Performance

## 3.5 UI

UI: 11 UI Design

# Chapter Four

## Evaluation

When it came to evaluation, I wanted to bring it back to each point set out in the project planning.

[link](#)

### 4.1 Syntetic workload

Syntetic workload: 12 Syntetic workload

### 4.2 deployment

deployment: 13 Deployed

In relation to how to deploy the system (and external components) for a greater load.

## 4.3 Use Cases

### 4.3.1 Types of Use cases evaluated

#### 4.3.1.1 Hypervisor

Deployed to a **Physical** host (Hypervisor) which itself is hosting a variety of virtual machines and containers. What kinds of agents have been deployed?

What scale is this intended to work at?

how does it match aims?

What are the bits that went well / bad

what would change.

## Chapter Five

## Conclusion



# Appendix One

## Benchmarks

[Table 1 about here.]

[Table 2 about here.]

[Table 3 about here.]

[Table 4 about here.]

# Glossary

**container** A way of packaging an application and its dependencies so that it can be run in a repeatable manner. . 3, 5

**flow** A sequence of related packets from one source computer to a destination. Also described as . 6

**message** Structure used for representing any features the system will report . 12

**NetFlow** Cisco stuff. 9

**protocol** e.g. DNS. i, 4–6, 8

**protocol buffer** Protocol buffers are a language-neutral, platform-neutral, extensible mechanism for serializing structured data . 11, 12

**traffic flow** A Flow is defined as a set of packets or frames passing an Observation Point in the network during a certain time interval. All packets belonging to a particular Flow have a set of common properties [4, p. 8] . i, 6

# Acronyms

**DNS** Domain Name System. 5, 11

**IDE** Integrated Development Environment. 3, 5

**IPFIX** Internet Protocol Flow Information Export. 4, 6, 12, 24

**K8S** kubernetes. 3

**OS** Operation System. 10

**SBD** Secure by design. 4

**SCTP** Stream Control Transmission Protocol. 6, 11

**TCP** Transmission Control Protocol. 6, 11

**TLS** Transport Layer Security. 11

**UDP** User Datagram Protocol. 6

# Nomenclature

MB/s Mega Bytes per Second

# Figures

A.1	Coders . . . . .	26
A.2	IPFIX Architecture (Source: <a href="#">Wikipedia IP Flow Information Export</a> )	27
A.3	Jaeger Architecture . . . . .	28
A.4	Packet pipeline . . . . .	29
A.5	Component Architecture . . . . .	30
A.6	Agent . . . . .	31

# Tables

A.1	Benchmark Environment . . . . .	32
A.2	Unmarshal Benchmarks . . . . .	33
A.3	Marshal Benchmarks . . . . .	34
A.4	Size Benchmarks . . . . .	35

# Code Snippets

1	Flow Definition . . . . .	12
---	---------------------------	----

# References

- [1] *Fallacies of Distributed Computing*. In: *Wikipedia*. Mar. 17, 2021. URL: [https://en.wikipedia.org/w/index.php?title=Fallacies\\_of\\_distributed\\_computing&oldid=1012709130](https://en.wikipedia.org/w/index.php?title=Fallacies_of_distributed_computing&oldid=1012709130) (visited on 05/14/2021) (cit. on p. 4).
- [2] Rick Hofstede et al. “Flow Monitoring Explained: From Packet Capture to Data Analysis With NetFlow and IPFIX”. In: *IEEE Commun. Surv. Tutor.* 16.4 (2014), pp. 2037–2064. ISSN: 1553-877X. DOI: [10.1109/COMST.2014.2321898](https://doi.org/10.1109/COMST.2014.2321898) (cit. on p. 6).
- [3] R. Stewart (Ed.) *Stream Control Transmission Protocol*. RFC 4960. Updated by RFCs 6096, 6335, 7053, 8899. Fremont, CA, USA: RFC Editor, Sept. 2007. DOI: [10.17487/RFC4960](https://doi.org/10.17487/RFC4960). URL: <https://www.rfc-editor.org/rfc/rfc4960.txt> (cit. on p. 6).
- [4] B. Claise (Ed.), B. Trammell (Ed.), and P. Aitken. *Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of Flow Information*. RFC 7011. Fremont, CA, USA: RFC Editor, Sept. 2013. DOI: [10.17487/RFC7011](https://doi.org/10.17487/RFC7011). URL: <https://www.rfc-editor.org/rfc/rfc7011.txt> (cit. on pp. 4, 6, 20).
- [5] *Wireshark*. URL: <https://www.wireshark.org/> (visited on 04/28/2021) (cit. on p. 11).



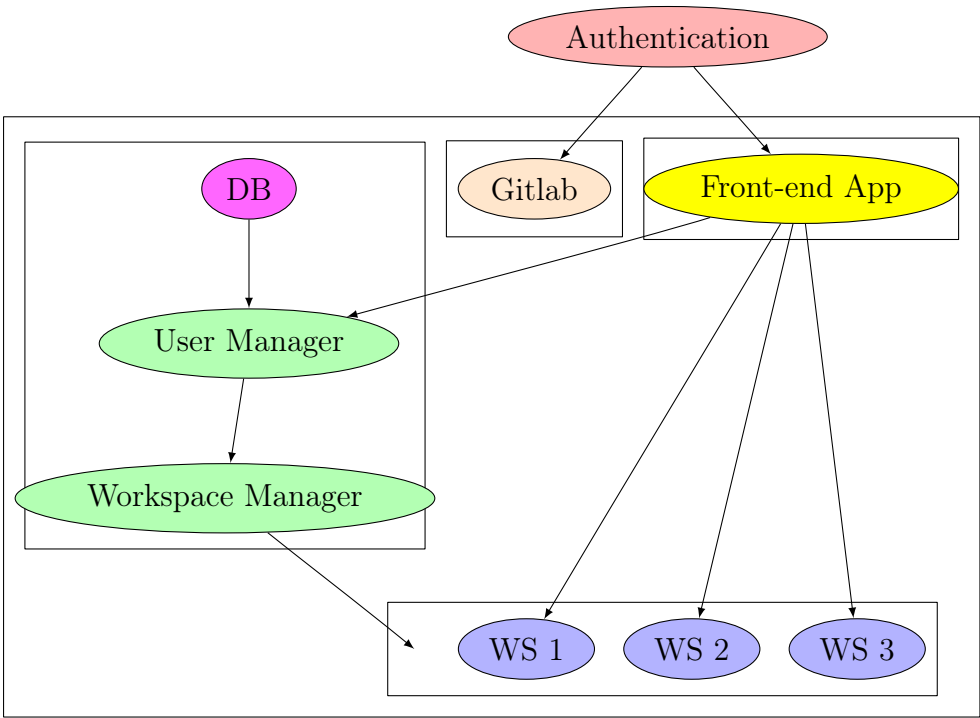
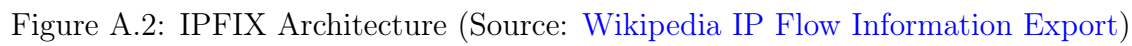


Figure A.1: Coders





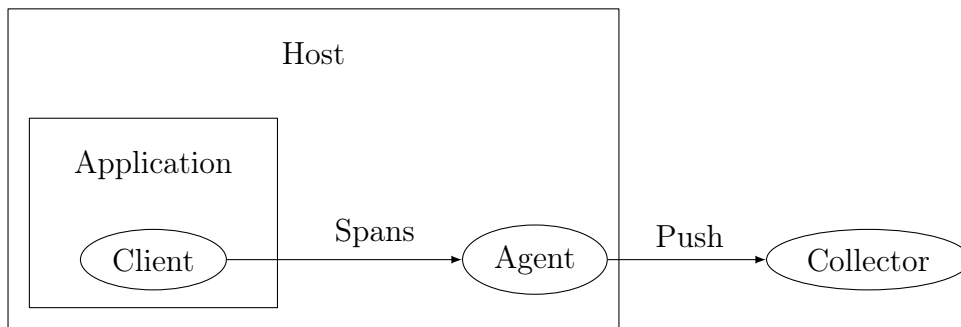


Figure A.3: Jaeger Architecture

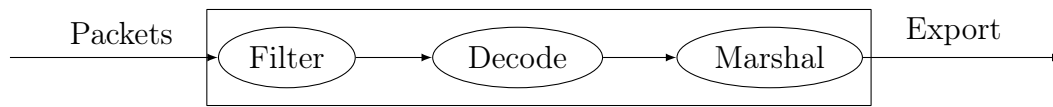


Figure A.4: Packet pipeline

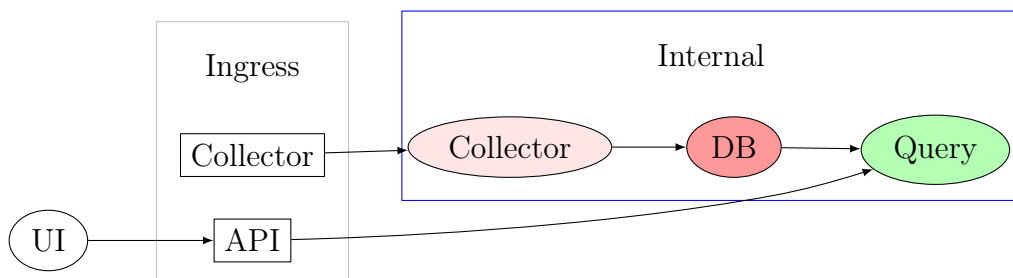


Figure A.5: Component Architecture

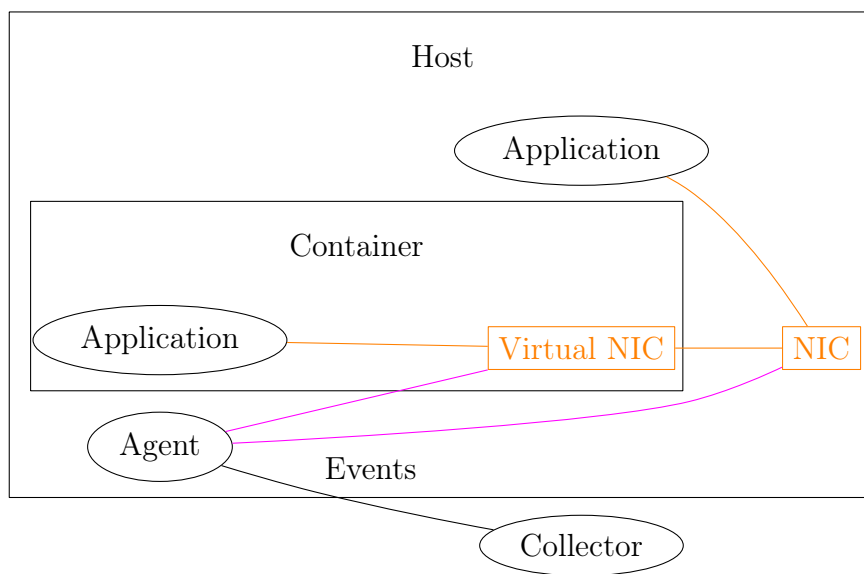


Figure A.6: Agent

GOOS	GOARCH	CPU
linux	amd64	AMD Ryzen 5 5600X 6-Core Processor

Table A.1: Benchmark Environment

---

Message	loops	(ns) per loop	I/O (MB/s)
Agent	2003037	590.8	885.28
Endpoint	10596949	108.1	583.01
Flow	810357	1543	516.41
Packet	1721414	671.9	308.07
Http	1533000	804.6	548.07
Dns	1548782	766.8	507.31
Tls	1733568	714.0	473.41
Arp	1178860	998.3	690.18

---

Table A.2: Unmarshal Benchmarks

---

Message	loops	(ns) per loop	I/O (MB/s)
Agent	3045099	392.6	1332.12
Endpoint	11553470	116.7	539.83
Flow	1000000	1237	644.46
Packet	1556341	689.0	300.42
Http	1702892	614.0	718.25
Dns	1843177	680.5	571.67
Tls	1961786	661.3	511.14
Arp	1523559	790.6	871.49

---

Table A.3: Marshal Benchmarks



---

Message	loops	(ns) per loop	I/O (MB/s)
Agent	63025371	18.83	28043.78
Endpoint	198427028	6.135	10106.47
Flow	5098138	219.5	3607.41
Packet	7151492	175.0	1143.15
Http	8916307	129.6	3440.40
Dns	11554758	119.2	3322.87
Tls	9748363	129.0	2651.38
Arp	10249390	119.8	5742.23

---

Table A.4: Size Benchmarks

# WIP

## Eval

To evaluate the effectiveness of my solution, I explored a series of realistic use-cases, along with generated benchmarks.

I explored the limitations that arise due to external components that the system must interact with.

Also testing how performance varies on the different platforms that I have been able to target. For the most part, I used libpcap for the platform dependant packet capture. This adds its own set of limits and advantages.

## Benchmarks

A critical requirement at both the point of observation and when ingesting is the speed at which we can transform the data.

Firstly there is capturing packets, since this is something handled by the underlying implementation it isn't worth an in-depth review.

Next is the decoding of packets after they were captured. This handled by the gopacket library, which has flexible support for only attempting to decode packets to the desired layers. Additionally, it is able to do TCP stream reassembly without copying/allocations (assuming packets arrive in order)

I had to get the best performance out of gopacket (use it properly)

Having decoded the packets, the appropriate information needs to be carried over to the new format.

Marshall The first benchmark is marshalling performance.

Here you can see the results for the currently supported protocols and features...

These are generated from the protobufs, and work by populating the structure with seeded random data. With the results consisting of...

Unmarshall

Discussion

The main advantage of this approach is how the amount of code that needs to be maintained is very small.

Adding protocols

Add support for decoding the packets Define the message to represent it Add or description to the message Define any additional API mappings

API performance

I wrote a post-processor that reads the database definitions and generates a set of translations for web requests to SQL queries.

Either the request path or the query parameters can be used to bind a request to filter the resulting query.

Additional parameters like  $id_{eq}$  or  $id_{ne}$  can also be mapped to SQL clauses.

All these possible clauses generated at compile time as what is effectively an  $O(1)$  Time complexity lookup due to it being a static hashmap. Show some traces to demonstrate performance.

UI

The ui is a responsive progressive web app built using react admin.

This provides an easy starting point for displaying a basic table view of resources exposed by the api

It consists of:

Resources

A resource describes an individual type that the api exposes, e.g. DNS.

For a resource I define a list view. Which describes how to represent one record in a table.

Data provider This handles mapping ui elements and views to specific api endpoints.

Compared to ipfix or netflow...

Fundamentally there are a lot of similarities between my solution and the protocol design of ipfix. Both are routed in the idea that the underlying structure should be easily extensible allowing for arbitrary levels of detail or to focus down the information collected.

In the case of ipfix, the rfc sets out that an implementation MUST support sctp but MAY support tcp or udp. With this flexibility it allows for an implementation to be purpose built for use cases where one transport might make more sense. While SCTP is a great idea, (whhhhhyyy)

While rfc 7011 (section 11) comprehensively addresses the security considerations for the communication between exporters and collectors, highlighting both why the traffic should be confidential but also should not be interrupted (for accounting & forensic purposes)

gRPC (literally has testimonials from cisco and juniper ) While ipfix does set out requirments for how to secure traffic, it explicitly states "Information Element containing end-user payload information is exported, it SHOULD be transmitted to the Collecting Process using a means that secures its contents against eavesdropping."

I chose to use gRPC as my primary transport implementation.

It offers Streaming Blocking / non blocking Cancellation / timeout Lots of language support Pluggable authentication

---

texcount
----------

---

File: main.tex  
Encoding: utf8  
Sum count: 1421  
Words in text: 1312  
Words in headers: 82  
Words outside text captions, etc.: 25  
Number of headers: 50  
Number of floats/tables/figures: 6  
Number of math inlines: 2

Number of math displayed: 0

Subcounts:

```

text+headers+captions #headers/#floats/#inlines/#displayed
0+0+0 1/0/0/0 _top_
655+1+0 1/0/0/0 Chapter: Introduction
0+2+1 1/1/0/0 Section: Previous Work
0+2+0 1/0/0/0 Section: Project Aims
0+2+0 1/0/0/0 Section: Related Work
0+9+0 4/0/1/0 Subsection: Packet capture
126+10+17 6/2/1/0 Subsection: Flow monitoring
0+4+0 1/0/0/0 Subsection: Application vs Network monitoring?
15+1+0 1/0/0/0 Chapter: Design
74+1+0 1/0/0/0 Section: Planning
155+1+2 1/1/0/0 Subsection: Portability
0+1+0 1/0/0/0 Subsection: Extensibility
0+1+0 1/0/0/0 Subsection: Scalability} \thought{does this clash with performance?
44+1+0 1/0/0/0 Section: Protocol} \thought{rich traffic
92+6+2 3/0/0/0 Subsection: event types} \subsubsection{single packet} \acrshort{dns
32+2+0 1/0/0/0 Section: Message Design
0+1+0 1/0/0/0 Subsection: Flows
18+2+0 1/0/0/0 Subsection: Rich Details
0+1+0 1/0/0/0 Subsection: Sampling
0+1+0 1/0/0/0 Chapter: Implementation
0+1+2 1/1/0/0 Section: Architecture
18+1+1 1/1/0/0 Section: Agent} \task{8 Agent design
0+3+0 1/0/0/0 Subsection: Cross Platform Support
0+2+0 1/0/0/0 Subsection: Packet Capture
0+2+0 2/0/0/0 Subsection: Decoding
0+1+0 1/0/0/0 Section: Collector} \task{9 Collector design
0+2+0 1/0/0/0 Subsection: RPC Service
0+2+0 1/0/0/0 Subsection: Database Connections
0+2+0 1/0/0/0 Section: Query API} \task{10 Query API design
0+2+0 1/0/0/0 Subsection: Request Mapping
0+1+0 1/0/0/0 Subsection: Performance
0+1+0 1/0/0/0 Section: UI} \task{11 UI Design
20+1+0 1/0/0/0 Chapter: Evaluation
0+2+0 1/0/0/0 Section: Syntetic workload} \task{12 Syntetic workload
15+1+0 1/0/0/0 Section: deployment} \task{13 Deployed
0+2+0 1/0/0/0 Section: Use Cases
48+6+0 2/0/0/0 Subsection: Types of Use cases evaluated
0+1+0 1/0/0/0 Chapter: Conclusion

```