Real-Time Network Activity Aggregation and Geolocation Integration

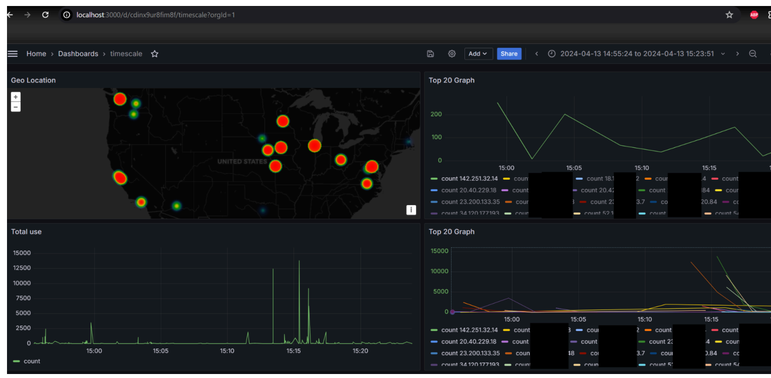
Jason Buchanan

Objective

Data captured using network monitoring tools, such as Wireshark, results in thousands of packets in normal operations. The large output can easily result in data fatigue and require time consuming, manual processing to understand the output and look for meaningful data points. This project intends to create a tool chain that provides real-time, glanceable and easy to understand graphics of the user's network utilization.

This project is written in Java and visualized in Grafana. The underlying technologies include: Spring Boot, TShark as the underlying packet sniffer, JPA and Hibernate for Object Relational Management (ORM), Flux for backpressure management for real-time, parallel processing of packets, MySQL as the datastore and ipinfo.io for their geolocation API

An example of the final output, which will be explained in this paper, can be seen below with geo location heat mapping as well as data splitting based on ip address, with visualizations reactive to timescale changes.



The code for this project can be found on my github account https://github.com/jasonbuchanan145/networking-heatmap

Theory and Formal Specification

Pre-existing Work

My implementation of this project is unique for this task, as far as I can find. However, the final output of a service that monitors network traffic and visualizes it in Grafana has been implemented by others in other ways.

Grafana offers network monitoring tutorials across the entire network using Prometheus and the SNMP protocol, which works well with the goal of overall network monitoring [1]. However, SNMP monitoring, especially in a home environment, can be difficult, if even possible with the user’s networking system.

Examining the manual for Cisco routers [2] shows a complex set up process for SNMP, especially with SNMP v3, which supports authentication and different encryption types such as AES and 3DES [3]

Documentation for home routers is lacking, but posts on various manufacturer community forums, such as Netgear, indicated on some models it was not supported; [4] however, on a couple models, their switches are supported [5]. In the case of SNMP not being supported, the user would have to set up additional hardware to act as a passthrough.

There was also an implementation that used Promtail, Loki and Maxminds [6]. The Promtail in Grafana was intended to monitor ingress data for web servers hosted in Kubernetes, parse log lines and insert parsed data to Prometheus. While this functionality was not the intent of this paper, it provided a good example of how the geo location map plugin for Grafana works for my implementation.

Traffic Size Calculation and Need for Aggregation

The number of bytes in a packet can vary depending on the ethernet specification in use, intermediate hardware and ISPs. While a 64Kb packet is possible, the Maximum Transmission Unit (MTU) reduces this down to as far as 1500 bits or lower [7]. Such a wide variation has led to disjointed information online for what an average packet size is. Because of the lack of consensus, for this calculation I will use my own internet traffic as the basis using the Windows Netstat utility. While working on this report I have used 1.433 Gigabytes which has used 2,007,067 packets. This comes out to an average packet size of approximately 714 bytes.

Given some current home internet speeds can achieve speeds of 1 gigabit per second, using the reference of 714 bytes/frame, this comes to 175,070 frames per second. This project intends to analyze all traffic going to, or incoming from, an external resource. Especially during peak utilization, data captured requires aggregation and queuing of the packet metadata to achieve reasonable space complexity in storage and retrieval as well as stability.

Pseudocode

This section includes pseudocode for the project as part of the formal specification. There are three sections: API side that is used for data collection and aggregation, Grafana side which is used for the display and reporting, and MySQL side which defines the database structure.

API side

startSpringBoot()

cache = select all from ip location database table and map to a java.util.concurrent.ConcurrentHashMap to ensure thread safety

subprocess = start a TShark subprocess filtering for IP source (src) destination (dst), frame length and timestamp, use "-T ek" to have it output to json with each line being one json object for easy mapping to a Java object

flux = create a new flux subscription based on a sink of a stream of the subprocess output

```
# This section happens as part of a parallel flux object  that runs until the program is terminated because the subprocess stream is only closed when interrupted by Java but in this pseudocode it is written as if it's a sequential loop for clarity

flux for each line {
   # It's an index for elastic search inherited from using -T ek we aren't using elastic search so we can ignore this line

   if line starts with "{\"index\""
      continue
   mappedFrame = mapToJavaObject(line)
   makeMetrics(mappedFrame)
}

function makeMetrics(mappedFrame){
   handleIp(mappedFrame.srcIp)
   handleIp(mappedFrame.dstIp)
}

function handleIp(ip){
   if ip matches a pattern equal to a best guess of an IPv4 local traffic or broadcast traffic ip
      return
   # We can assume that if we already fetched the ip location info for an ip odds are it's not going to change

  if cache does not contain(ip)
     cache.add(fetchAndSaveIpLocation(ip))
   incrementOrInsertByIp(ip)
}

function fetchAndSaveIpLocation(ip){
   ipLocation = webclient.getRequest("ipinfo.io/{ip}",ip)
   saveToIpLocationTableWithATransaction(ipLocation)
   return ipLocation
}

Use @Transactional to have spring manage the database transaction
function incrementOrInsertByIp(ip){
   id = query the database if an id exists for {ip} that was made in the last 2 minutes
   if id is not null{
      update db and set count column to count+1 for id
   }else{
      metric = mapIpToMetricObjectInitalizedWithACountOfOne(ip)
      saveMetricToDb(metric)
   }
}
```

Grafana side

Heatmap

```
  Select split(loc) field based on the comma(“,”) to return two columns in the result set
           sum(ip_metrics.`count`)
  From ip location table ip_location join timescale
  Where timescale.timestamp between the value defined by the user in the view of Grafana
  Group by ip.loc
```

Top 20 Ip Graph

```
Select ip, count, time
From timescale ts
```

Inner Join the timescale table on itself (t) with the time filter in the subquery equal to the filter applied in Grafana, grouped by the ip, order by the sum of the count in descending order, limit 20.
Join on ts.ip = t.ip
Group by ts.ip, ts.` time`, ts.`count`
Order by ts.`time`

Total Count

Select sum(`count`)
From timescale
Where time between the timefilter in Grafana
Group by timescale.time

MySQL table definition

There are two tables in this project that are defined by JPA and Hibernate based on annotation in the Java API, with a modification for a foreign key constraint I put in after it was created. Using the MySQL command "Show create table <tableName>" shows the definitions as follows:

Ip_info

```
CREATE TABLE `ip_info` (
  `ip` varchar(255) NOT NULL,
  `city` varchar(255) DEFAULT NULL,
  `country` varchar(255) DEFAULT NULL,
  `loc` varchar(255) DEFAULT NULL,
  `postal` varchar(255) DEFAULT NULL,
  PRIMARY KEY (`ip`)
) ENGINE=InnoDB DEFAULT
CHARSET=utf8mb4
COLLATE=utf8mb4_0900_ai_ci;
```

Timescale

```
CREATE TABLE `timescale` (
  `id` bigint NOT NULL,
  `count` bigint DEFAULT NULL,
  `ip` varchar(255) NOT NULL,
  `time` bigint DEFAULT NULL,
  PRIMARY KEY (`id`),
  KEY `ip` (`ip`),
  CONSTRAINT `timescale_ibfk_1`
FOREIGN KEY (`ip`) REFERENCES
`ip_info` (`ip`)
) ENGINE=InnoDB DEFAULT
CHARSET=utf8mb4
COLLATE=utf8mb4_0900_ai_ci;
```

Design

Technologies used

Summary

This project utilizes several technologies including the following:

- TShark: A command line utility that works as the underlying packet sniffer providing real-time packets information to the parent service
- Java/Spring Boot: The parent process that runs a TShark subprocess and aggregates the data
- Flux: A reactive publisher/subscriber framework in Java that can manage the number of threads relative to changes in back pressure of the live stream of data [8]
- JPA/Jakarta: Object Relational Management framework (ORM) to map Java objects to MySQL entities as well as manage database transactions
- IPInfo.io: A rest API to fetch geo location for a given IP
- MySQL: The datastore that aggregated data is stored and queried
- Docker: Containerization for Grafana runtime and network management

- Grafana: The web UI that queries MySQL and renders dashboards

Dependency Output Examples

Use of these technologies are shown in the formal specification and workflow diagram sections of this paper and won't be explored here. There are two services that a full example is beneficial.

TShark

TShark is the underlying packet sniffer for the API and runs as a subprocess and outputs a live stream. TShark has most of the capabilities of wireshark, however it is controllable based on command line arguments. An example of the data can be seen below

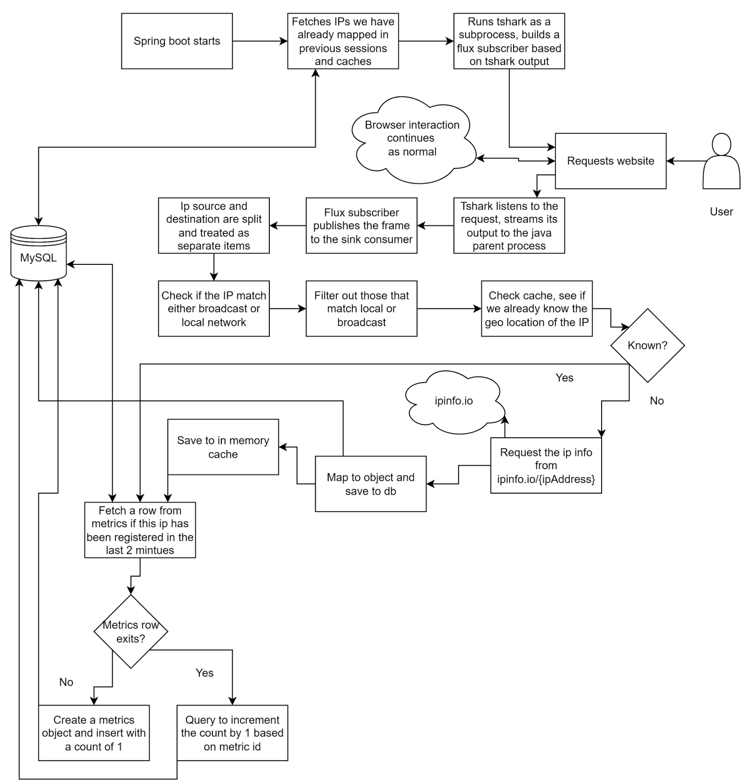
tshark -i Wi-Fi -T ek -e ip.src -e ip.dst -e frame.len
Capturing on 'Wi-Fi'
{"index":{"_index":"packets-2024-04-16","_type":"doc"}}
{"timestamp":"1713287091964","layers":{"ip_src":["192.168.0.214"],"ip_dst":["23.200.133.235"],"frame_len":["111"]}}
{"index":{"_index":"packets-2024-04-16","_type":"doc"}}
{"timestamp":"1713287091992","layers":{"ip_src":["23.200.133.235"],"ip_dst":["192.168.0.214"],"frame_len":["74"]}}

IpInfo.io

IpInfo offers both browser and rest based information on a best guess of the geolocation of an ip address as well as other data points. The Java API uses ipinfo's API over REST to resolve a geo location. An example, using University of South Dakota's ip of 206.209.15.51 can be seen below
curl ipinfo.io/206.209.15.51

{
  "ip": "206.209.15.51",
  "city": "Vermillion",
  "region": "South Dakota",
  "country": "US",
  "loc": "42.7794,-96.9292",
  "org": "AS11736 University of South Dakota",
  "postal": "57069",
  "timezone": "America/Chicago",
  "readme": "https://ipinfo.io/missingauth"
}

Workflow Diagram



Testing and Alternative Implementations Attempted

Testing and Debugging

Because this project is based on chaining multiple services together, there are no formal unit tests and testing relies on manual validation. For a future version there could be automated tests if one puts in the time into mocking services with Mockito or other mock implementations, H2 for database mocking, etc.

Manual testing was done by monitoring my own traffic, viewing logs and debugging visualizations using MySQL and the Grafana data transform utilities. There were also a couple stress tests with running speed tests from Google.

I also did some calculations on the count of frames it was reporting and compared it to the Netstat utility to see if it was reasonable (similar to the Traffic Size Calculation section of this paper).

Dropwizard and Prometheus

The most common similar implementations of this project use Prometheus, detailed in the previous works section. Prometheus is a NoSQL datastore that is usually used when scraping metrics from another service [9]. In the initial implementation I wanted to use Dropwizard with Prometheus which allows, to the second, information on rolling decaying averages as an exponential decay function [10]. However, after many hours of attempting to get it to work with Grafana I could not get it working because Grafana kept looking at each entry in the store as a document as opposed to one document with multiple ips. For example it would look at the entry of ip x as one "datasource" and the entry for ip as another datasource when it's all one datasource. This issue has to be an issue with how I was utilizing the Drop Wizard API but I eventually gave up on this in favor of pivoting to MySQL and manual aggregation on insert to the database.

Future Enhancements

Most future enhancements focus on increasing performance, however there are a couple others that increase functionality and ease of use.

TShark Subprocess Utilization

There is additional load to the Java API in filtering out broadcast traffic such as ARP, DHCP, etc. Any performance impact was not noticeable in local testing. During a demonstration in a large university network, the system was impacted. The noisy network contributed to a reduction in performance as the API processed the frames, only to throw a significant percentage of them after it mapped them to Java objects. This could be mitigated by setting additional capture filtering when initializing the TShark subprocess so that this traffic does not make it to the Java API.

Back Pressure and Thread Performance Tuning

This implementation allows Java defaults to determine how many threads to create and destroy to process the backpressure as quickly as possible. This may not be desirable behavior based on how people utilize the internet. For example, visiting web pages or downloading files results in a spike in data utilization and then a lower idling while interacting with the downloaded data. Some lag in the queue should be acceptable to help even out performance so that the API can process additional back pressure. Allowing for additional backpressure by limiting the queue consumers also protects the system against an overall performance degradation.

Database Batching

For every frame, there was one database query to update the metrics, whether that be to increase the count by 1 or to insert a new row if a metric did not exist for the timeframe. Round tripping the database at every frame results in a significant amount of database transactions. Being able to build a batch update and insert could help increase performance by reducing database calls.

Improved IPv6 Support

While the program can handle IPv6 addresses, it does not apply proper filtering. In the heatmap we would want to ignore Unique Local Addresses (ULA) and other traffic that would not go external to the network or where the destination is the user we would only want to extract the source ip and discard the destination ip.

Containerizing the Java API with TShark

The Java API is not containerized because Docker has a virtualization layer between Docker and the host. The Docker documentation indicates this is not a blocker for containerizing and I should have been able to listen directly to the host. I was not able to get this working and abandoned it. Getting this working would further the goal of having this project be easy to use for a basic home user.

Conclusion

In this project, I delivered a unique implementation of live aggregation of TShark data with geo location information and used Grafana to create a UI. I utilized several technologies to achieve this goal and made it fairly easy to use. There are some shortcomings, particularly around performance, that would have to be addressed. This project still achieved its overall goal and provided a unique and easy to use tool chain for a user to understand their network activity

Work Cited

Use of AI and Writing Aids Disclaimer

During development of this project I had a couple issues that I used ChatGPT to help resolve - primarily for integrating the Grafana built in functions for timescale filtering to my database, but also a couple of issues in the API. Code derived from ChatGPT constitutes less than 10% of my overall code in this project.

In this document the sources section uses the citation machine from citationmachine.net strictly for generating APA citations for URLs I provided to it, not for other writing or research tasks.

Sources

[1] Miroy, K. (2022, September 30). *Grafana & Prometheus SNMP: Beginner’s Network Monitoring Guide*.

Grafana Labs. https://grafana.com/blog/2022/01/19/a-beginners-guide-to-network-monitoring-with-grafana-and-prometheus/

[2] Cisco. (n.d.-b). SNMP Version 3. https://www.cisco.com/c/en/us/td/docs/ios-xml/ios/snmp/configuration/xe-3se/5700/snmp-xe-3se-5700-book/nm-snmp-snmpv3.pdf

[3] Cisco. (n.d.-a). AES and 3-des encryption support for SNMP version 3. https://www.cisco.com/c/en/us/td/docs/ios-xml/ios/snmp/configuration/xe-3se/3650/snmp-xe-3se-3650-book/nm-snmp-encrypt-snmp-support.pdf

[4] NetGear Contributors. (2016, July 21). *Is it possible to setup snmp or remote logging on the R8000?*. NETGEAR ® COMMUNITY. https://community.netgear.com/t5/Nighthawk-Wi-Fi-5-AC-Routers/Is-it-possible-to-setup-SNMP-or-remote-logging-on-the-R8000/m-p/1114567#M35940

[5] NetGear. (n.d.). *How do I configure snmpv3 users on My Netgear GS728TPv2, GS728TPPV2, GS752TPV2, or GS752TPP Switch?*. NETGEAR KB. https://kb.netgear.com/000058234/How-do-I-configure-SNMPv3-users-on-my-NETGEAR-GS728TPv2-GS728TPPv2-GS752TPv2-or-GS752TPP-switch

[6] Promtail agent. Grafana Labs. (n.d.). https://grafana.com/docs/loki/latest/send-data/promtail/

[7] Roy, S. (2023, June 10). *Maximum packet size for a TCP connection*. Baeldung on Computer Science. https://www.baeldung.com/cs/tcp-max-packet-size

[8] *Class Flux*. Flux (reactor-core 3.6.5). (n.d.). https://projectreactor.io/docs/core/release/api/reactor/core/publisher/Flux.html

[9] Prometheus. (n.d.). *Getting started: Prometheus*. Prometheus Blog. https://prometheus.io/docs/prometheus/latest/getting_started/

[10] Drop Wizard. (n.d.). *Metrics mind the gap.* Metrics Core | Metrics. https://www.dropwizard.io/projects/metrics/en/release-3.0.x/manual/core.html