Service Mesh Overhead Measurement in Kubernetes Container Networks

Andrew Wei Leiyi Zhang Xinghan Zhao
nowei@cs leiyiz@cs zhaox27@cs

Supplementary Material: Proposal, Milestone 1, Milestone 2, Slides
Github Repo: https://github.com/Stosswelle/KuberNerds

1. Introduction

Technology stacks are moving to the cloud. As such, there are more tools and architectures designed for the cloud to help deploy and establish reliable, resilient services. One tool for such deployments is Kubernetes, which is built with the idea of containerizing services into nodes, which contain pods that run on Docker images. Another such tool is the service mesh, which abstracts away the communication complexities within services and allows developers to focus their efforts on developing the core logic of their applications.

There is an overhead with the deployment of service meshes, as we can't abstract away all the complexities of networking between nodes without some implicit cost. In this project, we will explore this cost through the deployment of Istio, a service mesh, on simple applications and dig into what types of overhead exists between deployments that use a service mesh and those that do not.

2. Problem

Kubernetes is an open source container orchestration system that is widely used for building backend services both on bare-metal servers and cloud platforms. Service Meshes coordinate service-to-service communication between microservices deployed in Kubernetes and abstracts away the direct communication through the use of proxy sidecars.

We are going to measure and examine the overhead of deploying services with and without a service mesh. We consider overhead in terms of communication overhead and overhead on CPU consumption. This is an important problem because service meshes are becoming increasingly common as the networking backbone of distributed applications, so characterization and intuition about the overhead associated with service meshes is necessary for future improvements and design decisions.

3. High-level Approach

We learned how to use Kubernetes and deploy applications with a service mesh locally, performing CPU/memory overhead and network packet monitoring before moving on to performing the same measurements when deployed on AWS.

We measure the size and types of the packets being sent between nodes and the CPU/memory usage of nodes in a deployment with and without a service mesh. We use a wire sniffing tool to capture packets between nodes and use a service mesh health metrics aggregator to determine CPU and memory usage. We have more about what tools and application architectures we used in Appendix A.

4. Experiment and Results

We ran the BookInfo application (Figure in 2, Appendix A) on a local machine and AWS EKS (Elastic Kubernetes Service) and recorded some performance metrics and networking packets. Then we wanted to see how the overhead scales with the complexity of the application so we ran a very simple stateless and (essentially) compute-less application (Figure 1, Appendix A) and recorded the performance metrics.

4.1 Network Traffic

By analyzing the packets captured by Wireshark, we are able to look into the details of communication between pods. In this section, we used Bookinfo which is the sample application provided by Istio. We have more about Istio and Bookinfo application in Appendix A.2

4.2.1 Communication Overhead

We first analyze communications in Minikube without Istio deployed. If there is no external access, then every pod is quiet. When an external HTTP request arrives, it will be forwarded to ProductPage by the gateway (172.17.0.1). The gateway uses the ARP protocol to request the MAC address of a destination pod, and pods ask the Kubernetes internal cluster DNS service for the destination cluster IP address. Since most packets, especially TCP packets, have the gateway's IP address as the source or destination IP address, we believe that the gateway handles all of the internal communications, which means the packet from one pod to another will first be forwarded to the gateway, then to the actual destination pod.

Things go differently after we enable Istio and re-deployed the application. First, the number of packets flowing over the virtual network increases dramatically. Even without external access, there will always be a regular health check communication between

the gateway and each pod one time per two seconds. Each health check is one complete HTTP communication, including TCP connection establishment, HTTP request and reply, and TCP connection release. Second, we notice SSL/TLS handshake packets when pods communicate with each other, which means they are actually using HTTPS. Third, we believe that the Istio sidecars, which are running concurrently with application components on each pod, handle all of the internal communications, because all HTTP requests to other services are now sent to localhost (127.0.0.1).

On AWS EKS, we observe a very similar situation. When Istio is enabled, a large number of health check packets appear, SSL/TLS is used over HTTP communication, and communication is handled by istio sidecars running on localhost. The only big difference we notice is that when Istio is disabled, there is no longer a virtual gateway handling packets between pods. Each pod will directly communicate with the cluster IP address returned by the DNS service. There might be some other, much smaller differences that we didn't notice, which we believe don't matter too much.

4.2.2 Packet Size

Comparing HTTP request packets for the same application component (i.e. GET /productpage), we found that Istio made some modifications to HTTP headers. By setting field x-forwarded-client-cert, the overall packet size was increased by around 30% - 50%. The exact number depends on the clients or proxies that current request has flowed through.

In addition, we've also noticed another HTTP field called x-envoy-peer-metadata, which has something to do with Istio. We brought this up in our presentation, and ran another experiment later that night, after which we believe it is only a special case. We described the problem we had and how we fixed it after presentation in Appendix D.

4.2 CPU/Memory

4.2.1 Local Machine

For measuring latency and CPU/Memory utilization on a local machine, we sent 1500 requests sequentially with 50ms wait time in between receiving the previous result and sending out the next. The example microservice app is running on minikube inside a docker container that is allocated 6 cpu cores and 10G of memory.

The summary of result is as following:

Running Stand-alone:

- 36.81 seconds spent waiting for the server's response in total
- The total cpu usage rate of this service at idle is about 0.014 - 0.017cpu
- When loaded, the cpu usage peaked at around 0.50 cpu
- The total cpu seconds used to process the 1500 requests is 59.33 cpu-second
- The memory idle usage is about 423.2 MB
- The memory usage when loaded is 476.44MB, resulting in 53.2 MB difference

Running with Istio:

- 46.15 seconds spent waiting for the server's response in total, resulting in a 25% increase.
- The total cpu usage rate of this service at idle is about 0.043 cpu
- When loaded, the cpu usage peaked at around 0.49, which is no different than running without Istio
- The total cpu seconds used to process the 1500 requests is 67.33 cpu-second, which is 14% higher
- The memory idle usage is about 683.72 MB, which is about 60% higher than before
- The memory usage when loaded is 747.97MB, resulting in a 64.25 MB difference. That difference is about 20% more than before.

4.2.2 AWS EKS Cluster

For measuring latency and CPU/Memory utilization on an AWS EKS Cluster using 2 m5 EC2 deployments (64-bit, 2 vCPU, 17.1 GiB Memory, and 1 x 420 Instance Storage), we still sent the same 1500 requests sequentially with 50ms wait time in between. The difference is that we repeated this on both the bookinfo app and then the simple app as well.

The summary of result for running **bookinfo** is as following:

Running Stand-alone:

- 66.67 seconds spent waiting for the server's response in total
- The total cpu usage rate of this service at idle is about 0.008 cpu
- When loaded, the cpu usage peaked at around 0.326 cpu
- The total cpu seconds used to process the 1500 requests is 47 cpu-second
- The memory idle usage is about 354.9 MB
- The memory usage when loaded is 400.6 MB, resulting in 45.7 MB difference

Running with Istio:

- 74.88 seconds spent waiting for the server's response in total, resulting in a 12.3% increase.

- The total cpu usage rate of this service at idle is about 0.015 cpu
- When loaded, the cpu usage peaked at around 0.333, which is no different than running without Istio
- The total cpu seconds used to process the 1500 requests is 51.42 cpu-second, which is 10% higher
- The memory idle usage is about 611 MB, which is about 72% higher than before
- The memory usage when loaded is 658 MB, resulting in a 47 MB difference. That difference is about the same as running istio-free.

The summary of result for running **simple service** is as follows:

Running Stand-alone:
- 41.60 seconds spent waiting for the server's response in total
- The total cpu usage rate of this service at idle is about 0.008 cpu
- When loaded, the cpu usage peaked at around 0.0254 cpu
- The total cpu seconds used to process the 1500 requests is 3.91 cpu-second
- The memory idle usage is about 100.8 MB
- The memory usage when loaded is 102.2 MB, resulting in 1.4 MB difference

Running with Istio:
- 58.26 seconds spent waiting for the server's response in total, resulting in a 40% increase.
- The total cpu usage rate of this service at idle is about 0.01 cpu
- When loaded, the cpu usage peaked at around 0.037, which is not that different than running without Istio
- The total cpu seconds used to process the 1500 requests is 6.8 cpu-second, which is 70% higher
- The memory idle usage is about 233.8 MB, which is about 133% higher than before
- The memory usage when loaded is 235.3 MB, resulting in a 1.5 MB difference. That difference is about the same as running istio-free.

5. Future Work

I think something we would like to do in the future is to write a much more complex microservice architecture with both higher component counts and a more complex, realistic topology. Perhaps a good way to do that is to find a blogpost by some company that is running a microservice architecture such as Netflix and then simulate/recreate their topology.

We can also try different Service Mesh implementations and see if the cost differs between different implementations.

We can also try heavier workloads, as our workloads are not representative of the traffic that a high-traffic service would experience, which may not fully expose the fragility of service mesh architectures.

~~Finally, we'll shut down AWS cluster and split the cost among three of us :)~~

6 Conclusion

In this project, we learned about Kubernetes and Istio, how to deploy services using these tools on Minikube and AWS, and also how to monitor these deployments in terms of network traffic and CPU usage by using Ksniff and Prometheus.

On the network traffic, we found out that Istio enforces regular health checks between gateways and each pod one time per two seconds. If the application we deploy is relatively idle, this would dominate the overhead of kubernetes internal communications. We also found out that Istio increases HTTP request packet size by 30% - 50% by setting up some specific fields in HTTP headers.

On the CPU/Memory side, we found out that the CPU overhead is not that significant and decreases when the cpu load of the services increases. However, the memory penalty for Istio is pretty heavy, sitting around 70% higher memory usage, or 250 MB more for the bookinfo app. This may be alleviated by having services that are more memory-heavy (spinning up multiple threads and/or heavier computation etc) and because people have so much memory nowadays, the extra memory hit should not be that important anyway.

The Latency penalty is also not significant in real life situations when clients are connecting to a server or cloud that is far away from clients. Especially when some pages are taking nearly 10 seconds to load, istio induced latency in those cases should be insignificant for clients to notice.

We conclude by noting that this project has given us insight on the overheads of service mesh on simple microservice architectures, giving us better intuition on the cost of such a network abstraction.

7. References

What is a Service Mesh?:
https://www.redhat.com/en/topics/microservices/what-is-a-service-mesh
Kubernetes Service Mesh: A Comparison of Istio, Linkerd and Consul:
https://platform9.com/blog/kubernetes-service-mesh-a-comparison-of-istio-linkerd-and-consul/
Learning Kubernetes:
https://kubernetes.io/docs/tutorials/kubernetes-basics/
Kubernetes Cluster:
https://kubernetes.io/docs/tutorials/kubernetes-basics/create-cluster/cluster-intro/
Run your first application on Kubernetes:
https://medium.com/@m.sedrowski/run-your-first-application-on-kubernetes-e54d5194e84b
Docker walkthrough:
https://www.docker.com/blog/containerized-python-development-part-1/
Minikube start:
https://minikube.sigs.k8s.io/docs/start/
Minikube tutorial:
https://kubernetes.io/docs/tutorials/hello-minikube/
ksniff:
https://github.com/eldadru/ksniff
Istio docs:
https://istio.io/latest/docs/setup/getting-started/
Monitoring with Prometheus:
https://sysdig.com/blog/kubernetes-monitoring-prometheus/
https://blog.marcnuri.com/prometheus-grafana-setup-minikube/
Setting up kubernetes on AWS:
https://towardsdatascience.com/kubernetes-application-deployment-with-aws-eks-and-ecr-4600e11b2d3c
Istio on EKS:
https://aws.amazon.com/blogs/opensource/getting-started-istio-eks/
x-forwarded-client-cert:
https://www.envoyproxy.io/docs/envoy/latest/configuration/http/http_conn_man/headers#x-forwarded-client-cert

8. Appendix

A. Tools

We learned more about and gained more experience working with the following tools:

A.1 Minikube

Minikube is the recommended way of running Kubernetes on a local desktop computer for educational purposes. It runs inside a docker container and will simulate most behavior of kubernetes running on linux instances or cloud.

A.2 Istio and BookInfo application

Istio is a service mesh developed by Google, IBM and Lyft, it utilizes the open source envoy proxy developed by Lyft as the sidecar proxy in each pod. Istio provides an example application called BookInfo and it is a web server that has 6 different components talking to each other.

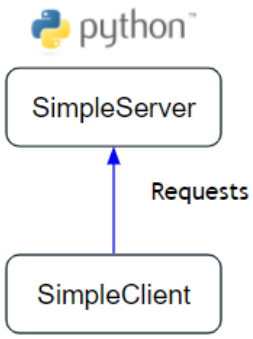

Figure 1. SimpleApplication, SimpleServer is load-balanced with 3 instances

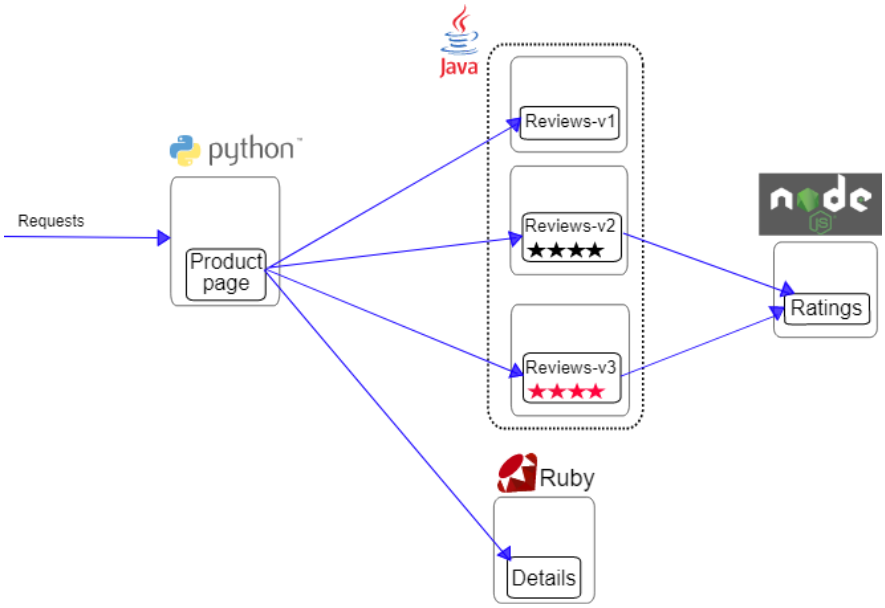

Figure 2. BookInfo application

- one ProductPage, the frontend website
- one Details, providing book information
- three different versions of Reviews, providing book reviews
- one Ratings, providing book ranking information.

When an external HTTP request arrives ProductPage will send HTTP request to Reviews and Details, and Reviews will then send an HTTP request to Ratings (or not, if Reviews-v1 gets called). Note that here ProductPage only calls Reviews service and lets the load balancer of Reviews decide which version, each running on a single pod with a different pod IP address, actually gets called.

A.3 Wireshark and Ksniff

Wireshark is a popular open-source packet sniffer and analysis tool. Users use it to capture network traffic on local or remote machines and store the packet data for offline analysis. We had some experience with it in the undergrad level network course.

Ksniff is a kubectl plugin that utilizes tcpdump and Wireshark to start a remote capture on any pod in a given kubernetes cluster. Using Ksniff, we are able to monitor and see the difference of network traffic with and without Istio deployed.

A.4 Prometheus

Prometheus is used to monitor and collect the cpu and memory usage data. We considered using the metrics-server as it is easy to set up and use. However,

metrics-server does not recommend using it to get accurate usage for cpu and memory as it is designed to be a fast way to monitor scaling requirements.

So we end up learning how to set up Prometheus. Although usually people use Prometheus together with Grafana to visualize the collected data, we found native web-interface of Prometheus good enough for our use. By default, Prometheus pulls metrics from all pods in the default namespace every few seconds, then through its web portal, we can use PromQL to query and filter through the metric to display appropriate metrics we want.

Instead of changing the configuration of Prometheus to target a specific namespace, we opted to deploy Prometheus in its own namespace and because Istio components are already deployed in their own namespace, we leave the default namespace with only the application-related pods.

B. Wireshark Diagrams

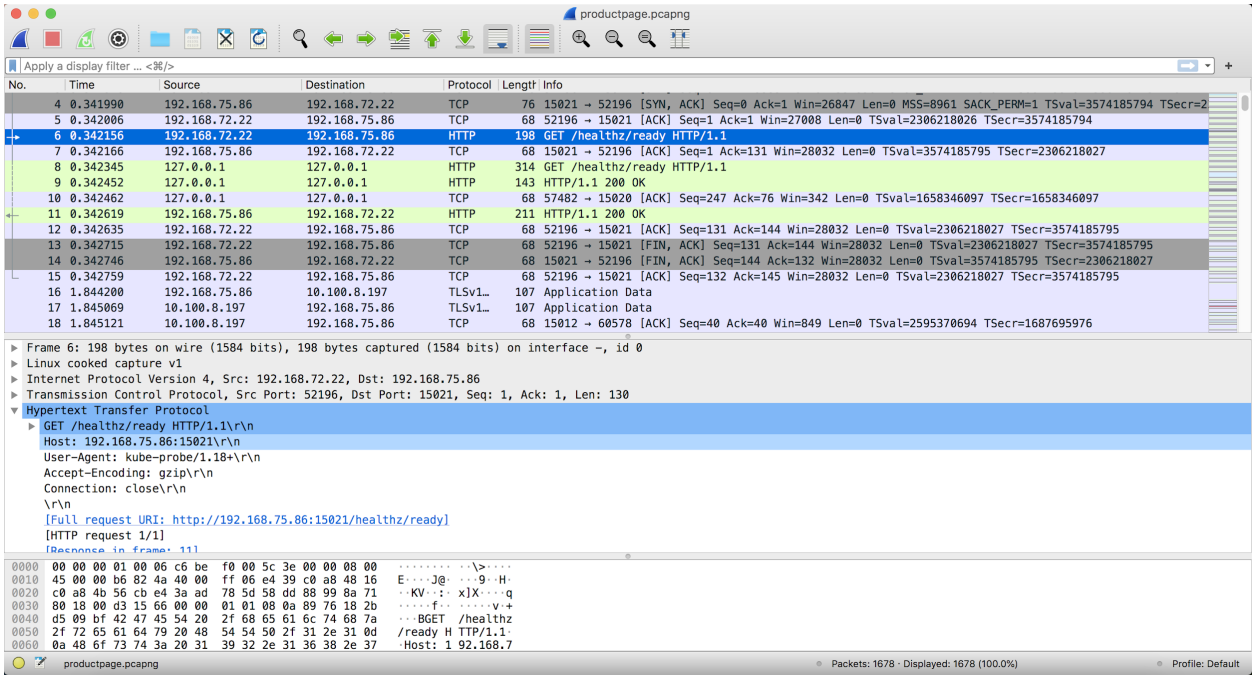

Figure 3. Wireshark records for running bookinfo on AWS EKS with Istio enabled

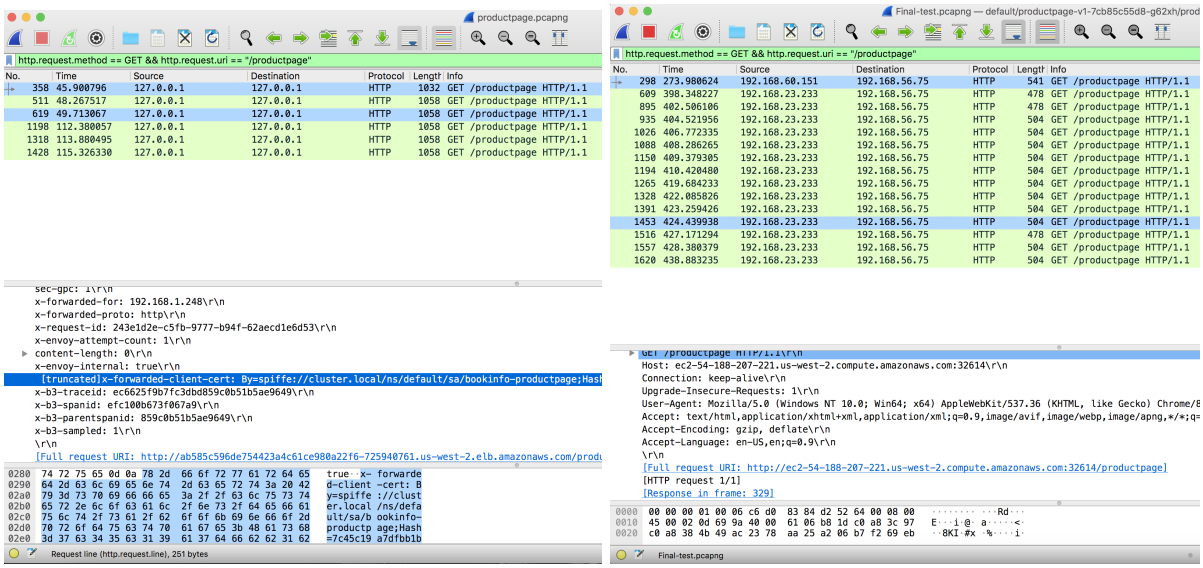

Figure 4. External “GET /productpage” HTTP requests when running bookinfo on EKS. The first one has Istio enabled and the second one doesn’t. Packet length difference is noticeable.

C. Prometheus Diagrams

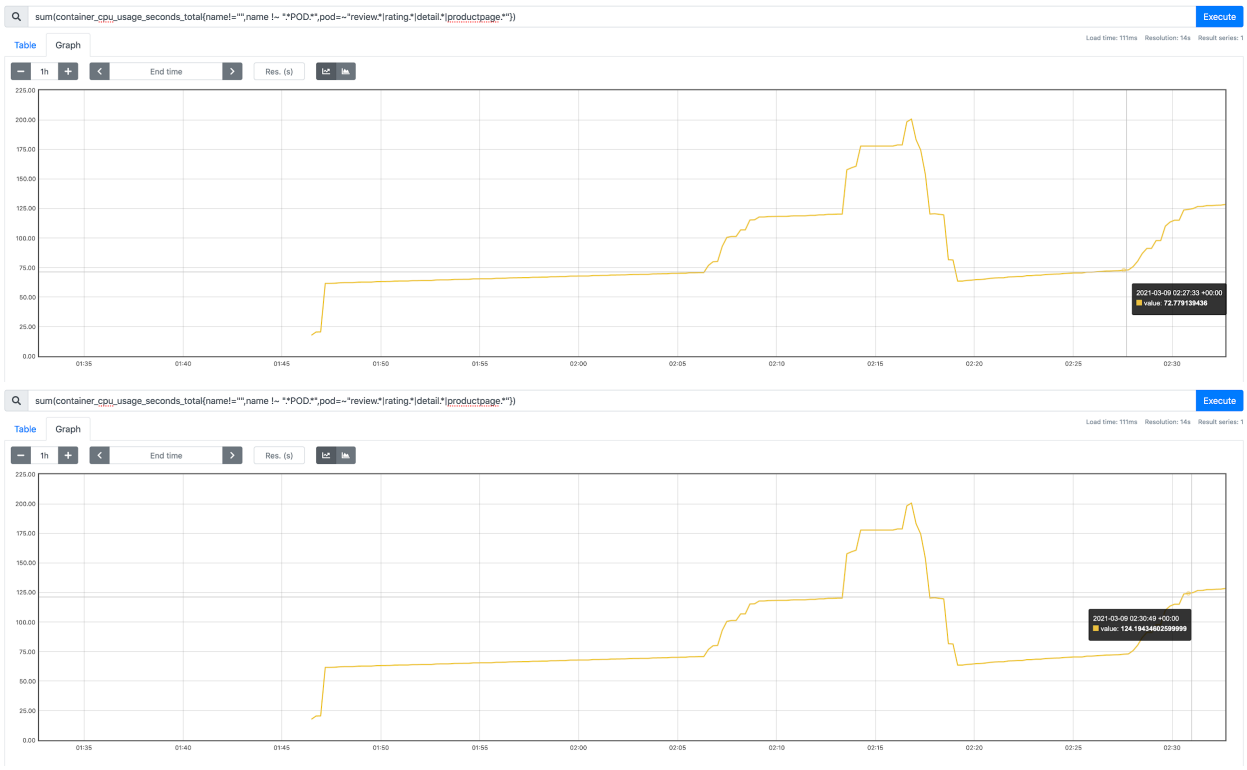

Figure 5. The total cpu usage running with istio before and after processing 1500 requests

D. Our Experiment After The Presentation

At first, we ran two experiments on AWS EKS and compared the packet length of HTTP “GET /productpage” requests, the results are as follows:

1. With Istio enabled, externally accessed through Istio Ingress Gateway: ~1000 B
2. Without Istio, externally accessed through Istio Ingress Gateway: ~2000 B

We checked the details of HTTP headers, and found out that:

1. With Istio enabled, the header includes a 251B field “x-forwarded-client-cert”
2. Without Istio, the header includes a 1041B field “x-envoy-peer-metadata”

Therefore we made a false assumption in the presentation that Istio sometimes adds overheads (client-cert) but sometimes helps get rid of some unnecessary data “peer-metadata”. However, that actually doesn’t make sense. So we had a meeting with Bowen after the presentation, and we decided to run our experiment again:

- This time, we disabled Istio, and didn’t use Istio Ingress Gateway to access the productpage. Instead, we exposed the service and made it publicly accessible by a domain name and port number combination. This time, the received HTTP “GET / productpage” request is ~500B, and we didn’t find any client-cert or peer-metadata fields in the HTTP header.

This should be the correct experimental result. We’ve compared it with our previous experiment which has Istio enabled and gave our conclusion in section 4.2.2.

Since we are confident that we didn’t do anything wrong in our experiment with Istio enabled, we believe that normally there shouldn’t be a x-envoy-peer-metadata. We don’t actually know the exact reason why our previous experiment without Istio was unsuccessful (packet size ~2000B with a huge x-envoy-peer-metadata). It might be caused by an inappropriate use of Istio, since we used Istio Ingress Gateway but didn’t deploy Istio sidecars on each pod. The only thing we are sure of is that x-envoy-peer-metadata was set by the Istio Gateway. We also failed to find any useful information about what on earth this field means, which remains a mystery both to us and to Bowen.