Jaeger-as-a-Service

Team 4 - Aditya Kulkarni, Omkar Vaidya

ABSTRACT

Modern-day applications are microservice-based applications. These applications face a challenge with respect to debugging and observing distributed transactions due to absence of in-memory calls or stack traces. To get a picture of the request in an application that flows through several services, distributed tracing is implemented. It is a method to profile and monitor applications built using the microservices architecture to collect information and observe requests as they propagate through different processes. The Jaeger distributed tracing framework is used in the project for creating distributed traces for requests in the application. The aim of the project is to explore the Jaeger distributed tracing framework and work on auto instrumenting the task of distributed tracing for services using Jaeger. This involves separating the application code and the tracing logic irrespective of the number of services, which would eliminate the need for manual insertion of tracer for each service added into the application. This would mean implementing Jaeger as a 'managed service'. Different concepts such as Pods and Sidecar pattern are used in order to achieve the main aim of the project and is successful in achieving auto instrumentation of distributed tracing component in microservice-based applications.

1. INTRODUCTION

In today's scenario, modern applications can be found everywhere. These applications consist of different services, unlike Monolithic applications. The Microservices architectural style structures an application as a collection of loosely coupled services. It can be logically thought of as a single application even if there are many services involved internally to fulfill requests to the end-user. But due to different elements involved in these applications, microservices introduce complexities concerning debugging, observing distributed transactions over different services, monitoring application health, and tracing the traffic flow through the distributed system. If a request is routed through several services tracing it would be difficult. Even if we keep traces of the functioning of different individual services, it would be difficult to collect trace logs and jot down the request execution path between different services in case of an application failure. There needs to be some kind of mechanism to keep traces between several processes. This is where distributed tracing comes into the picture. In the modern-day scenario, distributed tracing has become a crucial component for the observability of performance monitoring and troubleshooting. It is a new form of tracing which is more suitable for microservice-based applications. It can be defined as a technique to collect information and observe requests as they propagate through the system between processes. This allows us to see traces from end to end, locate failures, and improve the overall performance. In this project, we work on Jaeger - an open-source, end-to-end distributed tracing framework [2]. Jaeger is one of the popular distributed tracing frameworks compatible with the OpenTracing format and outputs traces into a timeline view to understand the flow of the request. To implement distributed tracing and capture the right information, it can be done manually or efforts can be made to automate it. Doing it manually requires a person to write logic to capture the right information before and after every operation. But for automating it, it would require auto instrumentation of code that would incorporate traces for different services and automatically instrument tracing logic into some common libraries. The main aim of the project is to automate the task of distributed tracing for the Jaeger framework and make it as a Managed Service.

2. DISTRIBUTED TRACING

Distributed tracing can be defined as different methods used to profile, monitor, and observe requests as they propagate through different services in microservice-based applications. It is useful for pinpointing failure and finding the causes of poor performance in applications. It can be said as a diagnostic technique that discloses how different sets of services coordinate and handle individual user requests. The tracing starts at the entry-point of a request in an application and will have a unique identifier generated for each request. As it transmits from service to service, each service adds some information to the trace such as request arrival time for a service, time to execute a request in service, etc. Using this information it is possible to create a picture of the entire request execution path (trace) for an end-to-end request. To create the distributed traces the Jaeger framework supports tracing via OpenTracing - an open standard

for distributed tracing. Using open tracing, we can collect traces into spans and also add some span context to it. A trace can comprise of single and multiple spans. A span represents a logical unit of work with associated time intervals and metadata. The metadata is usually tags - which are key: value pairs that enable user-defined annotation of spans to query, filter, and comprehend trace data.[3] The span context carries data across process boundaries i.e help in propagating data between two services. It consists of two major components - an implementation-dependent state to refer distinct spans within a trace and baggage items which may be useful to have data available for access throughout the trace. The logs help in capturing span specific logging messages. They help document a specific event within the span. A single trace typically shows the activity for an individual transaction or request within the application being monitored, from the browser or mobile device down through to the database and back. In aggregate, a collection of traces can show which backend service or database is having the biggest impact on performance affecting the user experience of the application. Distributed tracing provides end-to-end visibility and reveals service dependencies – showing how the services respond to each other. By being able to visualize transactions in their entirety, you can compare anomalous traces against performant ones to see the differences in behavior, structure, and timing. This information allows you to better understand the culprit in the observed symptoms and jump to the performance bottlenecks in your systems.

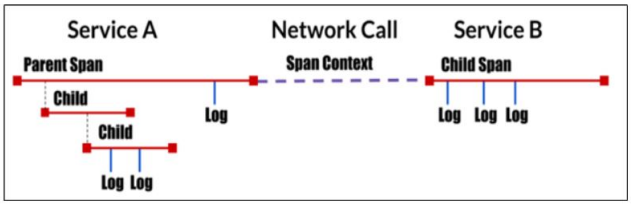

Figure 1: Trace

3. JAEGER DISTRIBUTED TRACING

Jaeger is an open-source, end-to-end distributed tracing framework. Every time an action takes place in an application, a request is executed by the Jaeger architecture which may require dozens of different services to participate to produce a response. [2] Jaeger performs distributed tracing by following a request path through various microservices that make up an application. Jaeger is compatible with the OpenTracing format. Some of the benefits of Jaeger are that it allows monitoring of distributed transactions, optimize performance and latency, and perform root cause analysis. Since we have a trace of all services we get a full picture to identify the root cause of a failure in the application. It is also used distributed context propagation which lets us connect data from different components together to create a complete end to end trace. The distributed tracing framework is made up of different components that work together to collect, store, and display tracing data. The different components of the architecture are client, agent, collector, storage, query service, and console. [9] The Jaeger client or clients are language-specific implementations of OpenTracing API and are used to inject the tracer into the services in the application. The Jaeger agent is a server queue that listens for spans sent over UDP and batches it and forwards it ahead to the collector. The Jaeger agent needs to be placed on the same host as the instrumented application. The Jaeger collector receives traces from the agent and places them into an internal queue for processing. The storage is a data store that keeps all the trace logs in the application which are retrieved by the query service. The Jaeger Console provides a user interface that allows visualizing the distributed tracing data. Jaeger outputs the traces into a timeline view to help understand the flow of the request and detecting bottlenecks. [1]

4. PODS

In Kubernetes, it does not run containers directly as it is. Instead, it wraps one or more containers into a higher-level structure called a Pod. Therefore a Pod is the smallest deployable computing unit that you can create and manage in Kubernetes. [6] It represents a single instance of a running process in the cluster. Pods can contain one or more containers (eg: Docker containers) and when a pod runs these multiple containers the containers are managed as a single entity and share the Pod's resources. The communication and data sharing is simplified when the pod consists of multiple containers. Since all the containers in the pod share the same network namespace - they can locate each other and communicate via localhost.[5] Containers easily communicate with other containers in the same pod as if they were on the same machine but maintain a degree of isolation between them. Pods can communicate with other pods by using the IP addresses of the pods or by referencing resources that reside in other pods. Pods can consist of an 'init container' that runs when the pod is started and is required before other application containers are run in the pod. Additionally, pods can also be replicated for the healthy functioning of the application under a heavy workload by enabling replica pods to be created and can be shut down automatically based on the changes in demand.

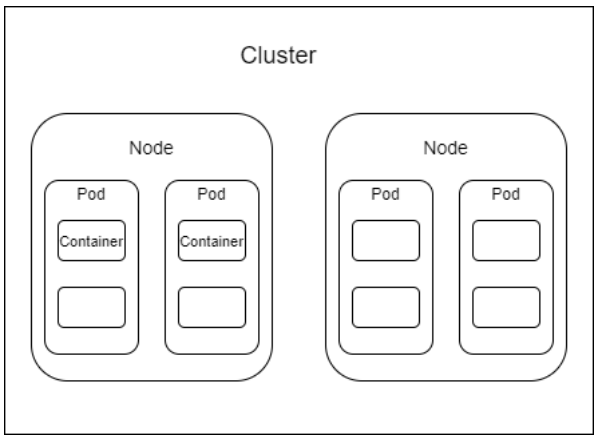

Figure 2: Pods

5. SIDECAR PATTERN

In the context of cloud-native applications, a Sidecar pattern is one in which a separate process is deployed alongside

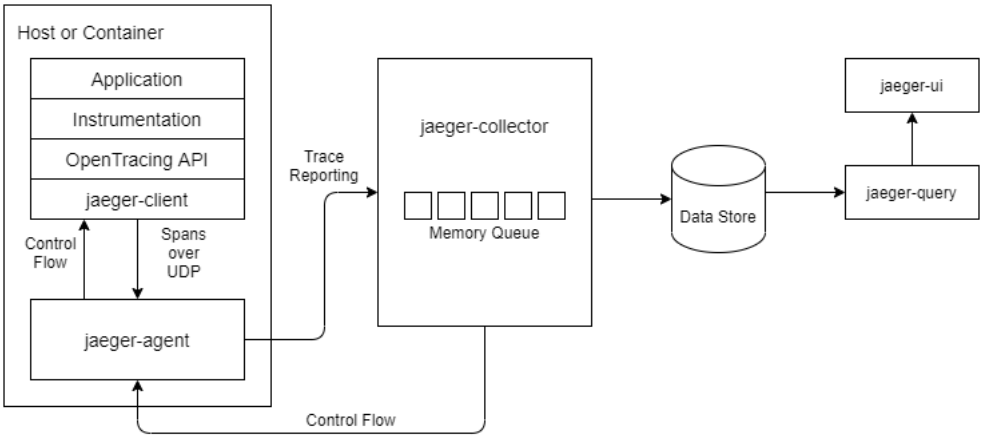

Figure 3: Jaeger Architecture

the primary application which provides additional functionality.[8] In a typical service mesh architecture, the sidecar constitutes a data plane while a control-plane GUI or CLI manages configuration for the data plane [11].

Some of the common functionalities are:

- Discovering Services in the cluster - In a service mesh, ephemeral primary application processes (Pods or containers) typically live behind a Load Balancer service with a static IP address. To communicate with the desired application processes, the service IP address needs to be obtained. The control-plane has this information available. Having a sidecar process obviates the need for the application processes to talk to the control plane components while also providing a clean abstraction between business logic and platform-infrastructure specific logic.
- Periodically checking the health of services - The sidecar can perform periodic checks to health check endpoints and recognize failure in case of multiple failed attempts.
- Routing outbound requests to correct service clusters - Individual API Services register routes they handle. The sidecar can identify which service cluster can accept this request.
- Periodically checking the health of services - The sidecar can perform periodic checks to health check endpoints and recognize failure in case of multiple failed attempts.
- Routing outbound requests to correct service clusters - Individual API Services register routes they handle. The sidecar can identify which service cluster can accept this request.
- Authentication and Authorization - Incoming requests can be checked to see if they have the right authentication and authorization.
- Observability - Sidecars are the perfect candidates for logging, monitoring, and distributed tracing of both inbound and outbound requests.

Benefits:

- Application and sidecars may be written in different languages - Applications are increasingly being written in varied languages. In terms of observability, the logging, monitoring, and tracing client libraries may not be available for the language the applications are written in. Here, a sidecar allows the applications to take advantage of specific languages while still providing the core functionalities mentioned above.
- In a common deployment environment of Kubernetes, the sidecar and the primary application can be deployed alongside each other in a single abstraction of a Pod, as mentioned above. Pods are analogous to a host machine where the containers share resources like volume and networking namespaces. This allows sidecars to communicate with the application containers through localhost, thus incurring minimal extra latency.
- All the Sidecar containers in a cluster can be updated without affecting the application containers.

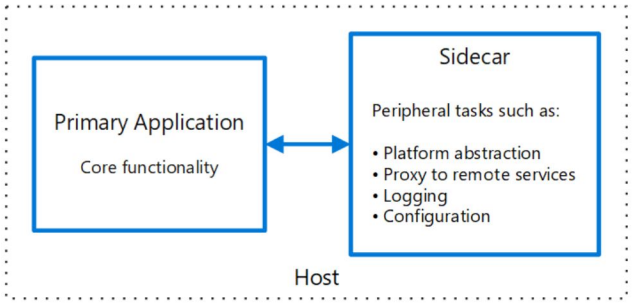

Figure 4: Sidecar

6. JAEGER - MANAGED SERVICE

Now, onto the crux of this implementation. To provide Jaeger as a "managed service", we have abstracted away

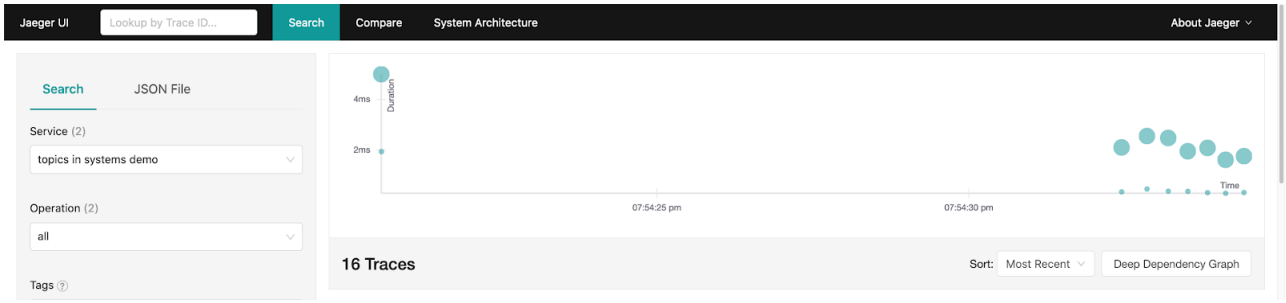

Figure 5: Jaeger UI

from the primary application, the client-library instrumentation of Jaeger into a separate sidecar proxy. The proxy determines if the packet is originating from the service or is received from other services. Accordingly, Jaeger tracing spans are started and ended by the proxy. Thus, all the Jaeger logic lives in the sidecar, oblivious to the application container.

Here, iptables are key to route incoming and outgoing through the proxy. More specifically, we map the service port (80) on the eth0 interface to proxy port (8000). We use the lo interface to route Pod-internal traffic directly to the application service container without the hop of the proxy.[10]

This networking needs to be set up before any container in the Pod is started. Fortunately, Kubernetes has a specific type of container, called init container, for this use case. Init containers are specified in the Pod specification and run before the application and sidecar containers.

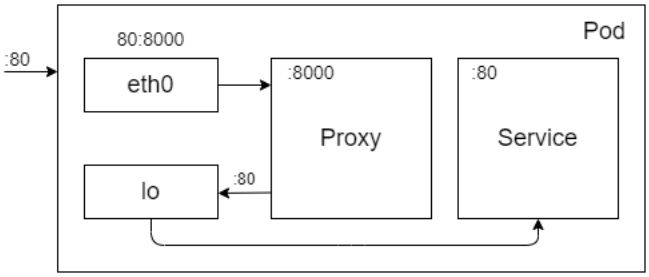

Figure 6: Jaeger Managed Implementation

7. IMPLEMENTATION WITH KUBERNETES

Here, we describe the steps needed to test our implementation in the Kubernetes environment.

Packaging

1. Write separate Dockerfiles for the individual service, proxy, and init containers.
2. Build the Dockerfiles into Docker images.
3. Push the Docker images to a registry like Docker Hub (or AWS Elastic Container Registry).
4. Write the YAML Pod spec, which will contain the image names and configuration of the service container, sidecar proxy container, and init container.

Deployment

1. Create a Kubernetes cluster with at-least 1 node (local: kind/minikube setup).
2. Deploy two separate Pods for the two services, namely service-A and service-B.
3. Deploy the built-in all-in-one Jaeger Operator which includes all Jaeger components required for registering, collecting, and displaying a trace.[4]

Exact steps for execution (Tested on a macOS system)

Setting up a Kubernetes cluster [7]

brew install kind
kind create cluster

Building and Push Docker images

cd init/
docker build -t [docker_hub_user]/init-container:latest -f initDockerfile .
docker push [docker_hub_user]/init-container:latest
cd proxy/
docker build -t [docker_hub_user]/sidecar-proxy:latest -f proxyDockerfile .
docker push [docker_hub_user]/sidecar-proxy:latest
cd serviceA/
docker build -t [docker_hub_user]/client-service:latest -f serviceADockerfile .
docker push [docker_hub_user]/client-service:latest
cd serviceB/
docker build -t [docker_hub_user]/server-service:latest -f serviceBDockerfile .
docker push [docker_hub_user]/server-service:latest

Deploying Pods on the cluster

kubectl apply -f yamls/jaeger.yaml [all-in-one image]
kubectl apply -f yamls/serviceA.yaml
kubectl apply -f yamls/serviceB.yaml

8. RESULTS AND DISCUSSIONS

The end result, i.e. the trace being generated, is the same as that of our initial prototype. It uses the same built-in Jaeger UI to display the traces. This further demonstrates the inconspicuous and unobstrusive nature of our implementation.

In Figure.5, Smaller dot denotes a short span (Service 2 to 3). Longer dot denotes a longer span (Service 1 to Service 2)

9. EVALUATION

We discuss the three parameters of the system which are minimally affected by our implementation.

- Execution Time - The additional time incurred by adding the proxy is in a few milliseconds. The reason being the hops from proxy to the service and back take place through localhost, the (lo) interface.

- Header Complexity - The proxy adds the same Jaeger tracing headers that a non-auto-instrumented version adds. As a result, the header complexity with the addition of the proxy remains the same.

- Init Container Delay - As said earlier, we use init containers to set up the initial networking required to forward requests from the application container to the proxy and back. This init containers is essentially a shell script which contains the required iptables rules. The result is a small delay before starting up a Pod. Note that this delay only occurs once per Pod, when it starts.

10. CONCLUSION

We were able to achieve separation of tracing logic into sidecar proxy containers. This fits well into the current cloud-native service mesh pattern. We were able to successfully and efficiently auto-instrument application code with tracing logic with no additional effort for developers. It's remarkably easy to maintain and roll out updates to tracing logic or even swap out tracing libraries within the sidecar proxy.

11. REFERENCES

[1] Jaeger tracing client library github. https://github.com/jaegertracing/jaeger-client-go.

[2] Official jaeger documentation. https://www.jaegertracing.io/. [Accessed November 29, 2020].

[3] Opentracing overview. https://opentracing.io/docs/overview/.

[4] Operator for kubernetes. https://www.jaegertracing.io/docs/1.21/operator/.

[5] Pod kubernetes engine documentation google cloud. https://cloud.google.com/kubernetes-engine/docs/concepts/pod.

[6] Pods. https://kubernetes.io/docs/concepts/workloads/pods/.

[7] Quick start. https://kind.sigs.k8s.io/docs/user/quick-start/.

[8] Sidecar pattern - cloud design patterns. https://docs.microsoft.com/en-us/azure/architecture/patterns/sidecar.

[9] Understanding jaeger. https://docs.openshift.com/container-platform/4.1/service_mesh/service_mesh_arch/ossm-jaeger.html.

[10] V. N. O. S. Enthusiast, V. Noronha, and R. More. Hand-crafting a sidecar proxy and demystifying istio. https://venilnoronha.io/hand-crafting-a-sidecar-proxy-and-demystifying-istio, Mar 2019.

[11] M. Klein. Service mesh data plane vs. control plane. https://blog.envoyproxy.io/service-mesh-data-plane-vs-control-plane-2774e720f7fc, Oct 2017.