Project Proposal

Observability Query Language for Linkerd using GraphQL

By
Tarun Pothulapati

1. Table Of Contents

2. Project Description

There's a lot of important observability data in Linkerd. Between Prometheus and tap, you can get all kinds of data that explain how your system is working. Unfortunately, these two pieces of data are separate and difficult to introspect and understand. Prometheus maintains the aggregates of request data i.e total_requests, etc while Tap helps you see requests in real time happening in between your microservices. Now, Linkerd's dashboard and CLI interact with two different endpoints to show metrics based on the user's request.

This Project aims to create a common query Interface that integrates with existing metric backends i.e Prometheus and Tap Server and aggregates data. There is a lot of interrelated data coming from different back-ends. Having a **GraphQL** Endpoint in the front that would make queries to the corresponding backends and get the data would improve the developer experience. Users can just send an HTTP request in a JSON format of what they want (i.e Query), GraphQL makes corresponding requests to the backends and returns the data to the client. If more components are added to Linkerd that have metrics, these backends can be easily integrated by just updating the GraphQL endpoint.

The GraphQL service is essentially divided into two parts i.e The Type System, where the service can define all the types (referencing each other) it exposes and the Resolver Functions that actually get the data. Having the Type System separated from the Backend essentially allows clients who want to query generate the Client Code from these types. There are also Client Code generators that make this easy without breaking existing functionality. On the Server side, More metrics can be added by updating the type system and the resolver functions.

3. Impact to the Community

By having a single endpoint, where users can query for metrics, adding more front-ends will be easy. The Linkerd dashboard which now uses Tap for request level data and Prometheus for aggregated data can rely on this single endpoint for queries. The service mesh dashboards that are already built by the service mesh community like Kiali, etc can also use this endpoint to query data without having to query multiple places.

By making the interface easy, It's essentially empowering the community to build things like alerts, better interfaces, etc on top of it. The possibilities are huge.

4. Deliverables

The Deliverables are marked into must-have and good-to-have. As the names imply must-have will be done by the end of GSoC. If there is more time left, good-to-have things will also be worked on.

Must-Have

1. A Stateless Graphql Endpoint in Go with both unit and e2e tests.
2. Support all Prometheus Queries supported by the public API of Linkerd right now.
3. Support Subscribing to a specific query from Prometheus Queries for streaming data.
4. Support streaming of Tap Queries through the GraphQL endpoint through subscribe requests.
5. Use Dataloader to batch and cache query requests on the GraphQL endpoint.

Good-To-Have

6. Make the CLI use the GraphQL endpoint for Tap, Stat, and Top Queries.
7. Make the Dashboard also use the GraphQL endpoint for metrics.

5. Implementation

5.1. Project Structure and Libraries Used

The GraphQL service will be built as a part of the Controller Docker file and will be deployed as a part of the Linkerd Control plane by including it as a separate service in the helm installation template of Linkerd. It will then interact with the Tap Server and Prometheus Instances which are already a part of the Linkerd Control Plane.

The Type System will be defined in the types/ dir while the query, subscription fields go into their specific directories. As the resolver functions interact with either the Prometheus or the Tap Server, the code to talk with the backends will be separated into a different folder and will be called by the resolver functions.

The GraphQL implementation in Golang project can be used in the service along with the http-handler which acts as middleware for HTTP Requests and sends it to the GraphQL library. The Downside is it all uses dynamic types for resolver functions as addressed here.

The other good library option is gqlgen, which takes a different approach of generating graphql code and resolver functions from the type system thereby creating static types. Also Supports Subscriptions and Data Loader functionalities.

The choice of the library would be made by discussing with the mentor and maintainers during the community bonding period.

5.2. GraphQL Type System involving all the metrics in Linkerd

As the Type System plays a crucial role in empowering the users to just get what data they want, It is very important to get the type system right and in an object format that would abstract away the underlying differences of metrics coming from different endpoints and in different formats.

The Type System Planned for now is

```
/* Represents a single resource like Deployment, DaemonSet,
etc. Can also contain sub-resources like pods, etc. and
also a parent resource by which it was created. */
type Resource {
   name: String!
   type: String
   isLive: Bool
   isReady: Bool
   isMeshed: Bool
   labels: [String]!
   parent: Resource
   children: [Resource]
   incoming: [Resource]
   outgoing: [Resource]
   metrics: [Metrics]
}

/*
Represents the defined statistics for a corresponding edge
(btw two resources) and the time-window. If the edge is
Null, it would mean for all requests.
 */
type Metrics {
   edge: Edge
   time_window: String
   success_count: Int
```

```
   failure_count: Int
   latency_ms_p50: Int
   latency_ms_p95: Int
   latency_ms_p99: Int
   tls_request_count: Int
   actual_success_count: Int
   actual_failure_count: Int
}

/*
Represents requests between two Resources and is used for
Requests/Response Metrics.
*/
type Edge {
   from: Resource
   to: Resource
}

/*
Represents the requests and response happening for a
corresponding edge. Used to serve tap queries.
*/
type Exchange {
   edge: Edge

   request: Request
   response: Response
}

/*
Traffic is used to get the overall request level statistics
in an edge.
*/
type Traffic {
 edge: Edge
```

```
 request: Request
 count: Int
 best: Int
 worst: Int
 last: Int
 success_rate: Int
}

/* Represents a single request */
type Request {
   method: String
   path: String
   authority: String
}

/* Represents the metrics for a single request */
type Response {
   latency: Int
   http_status: Int
   grpc_status: Int
}
```

Queries on this Type System are very flexible. For Example, for a Traffic Query, if the from field of the edge is null, then the service would give back the stats from all resources to the specified resource. By having this flexibility, Users should be able to get only what they requested.

More Types can be added as needed. During the Community Bonding Period, The design will be formulated by writing a design doc(as a part of a Github Issue) and reviewing it with other contributors to make sure everything is covered in a usable format.

5.3. Data Loader for Batching and Caching

The GraphQL API directly is prone to the n+1 problem, for example, if the user wants to see pod level metrics for all pods of a particular resource, the query would be like

```
Query
{
        resource(name: “web”, type: “service”)
        {
                Children
                {
                        metrics
                }
        }
}
```

Then, this query would first have to get all the children for the resource, and then for each child, a metrics request has to be sent indicating a total of n+1 requests. Using the Data Loader library in Go which executes the batch function provided for similar requests coalesced into one, this would decrease the load on Prometheus.

The Data Loader also has simple caching functionality which can be used to not hit Prometheus multiple times for the same query in a single request.

5.4. Handling Subscriptions for both Prom and Tap Queries

Subscription is a way to stream responses to users until they terminate the connections. This is very important for front-end interfaces where the data has to be updated automatically. By having streaming built into our endpoint, user’s don’t have to poll the GraphQL endpoint repeatedly. Instead, they can just subscribe to what they need, and GraphQL endpoint would maintain a WebSocket connection with the user and then stream data as it is updated in the backend. The GraphQL library that we are using already supports Subscriptions as a Type by using WebSockets under HTTP.

As TAP Server is implemented to stream real-time data about requests and responses that are happening in between the micro-services, The GraphQL endpoint would have a TAP Client as one backend and would stream data to clients, as the TAP client gets data from the TAP Server.

For the Prometheus Queries, as there isn’t a direct way of streaming data from the Prometheus Instance, (Even the Grafana dashboard polls the Prometheus instance at a specified period for new data.). Our GraphQL endpoint would be polling the Prometheus instance at every specified period and will just push that data to the subscribed clients.

6. Project Timeline

Timespan	Activity
7 May - 26 May	*Community Bonding Period:* Sketch out all the details as design docs and present to the community.
Actual Start of the Work Period	
27 May - 02 June	**4.1** Basic Project Structure Creation along with DockerFile and generator files.
03 June - 09 June	**4.2** Define the Type System for Prom Queries and work on Prometheus Client which gets the data.
10 June - 16 June	**4.2** Integrate the Prom Client into the Resolver Functions.
17 June - 23 June	**4.3** Add Streaming of Prom Queries using GraphQL Subscriptions.
Phase 1 Evaluation due	
24 June - 30 June	**4.3** Tests for Prom Queries, Subscriptions, and Final Touches.
01 July - 07 July	**4.4** Define the Type System for TAP Subscriptions.
08 July - 14 July	**4.4** Write the TAP Client Code to get data from the TAP Server.
15 July - 21 July	**4.4** Implement the Resolver functions for fields that involve TAP Data.
Phase 2 Evaluation due	
22 July - 28 July	**4.5** Data Loader Improvements for prom queries.
29 July - 04 August	**4.1** Tests and Final Touches for Tap Queries along with a sample project on using the GraphQL endpoint.
05 August - 11 August	**4.6** Make the linkerd CLI use the GraphQL endpoint for the Tap, Stat, and Top Queries.
12 August - 18 August	Final improvements and evaluation preparation.
Final evaluation due	

7. About Me

Personal Information

Name: Tarun Pothulapati
Email: tarunpothulapati@outlook.com
Phone No: +919398529136
College: Vardhaman College Of Engineering, Hyderabad, India.
Github Username: pothulapati
Timezone: GMT +05:30 (India)
Website: www.tarunpothulapati.com
Twitter: www.twitter.com/pothulapati

Why Me

I have always been passionate about Go and Kubernetes. I have been a part of the CNCF and Kubernetes community and contributed to multiple projects like Cluster-Api-Provider-Azure, Virtual-kubelet, etc. Linkerd has been a fascinating project to me, as its impact is huge and it's solving a very important problem i.e to provide observability, reliability, and security for microservices.

Availability and Communication

- Exams end on May 2nd. Classes start in July but I won't be having any exams until September. So, It won't be a problem working on the project.
- I will be working full time during the GSoC Period i.e (min of 40 hours per week).
- I will be available throughout the program on the Linkerd slack channel and will be in constant communication with mentors.

Experience with Go and Kubernetes

- I have been contributing to the Linkerd project with the following PR's
  #2416 Wire up stats and dashboards for Kubernetes Job Resources.
  #2439 Add job resources to dashboard and Grafana.
  #2292 Ignoring Resources with the empty kind field.
  #2293 Add ep alias for endpoints command.
- I have also been working on a Kubernetes controller project which helps you manage Azure Service Fabric Mesh resources right from the Kubernetes API server using CRD's and kubebuilder which can be found here

- I have been coding in Golang for the past one year and have been making some small utility projects with it.

My resume can be found here.