linkerd

A Service Mesh for Kubernetes

Using Linkerd to add Reliability, Security and Performance to your Kubernetes Application

Published by

Buoyant

Table of Contents

I. Introduction
Page 3

What is a service mesh?
Is the service mesh a networking model?
What does a service mesh actually do?
Why is the service mesh necessary?

II. A Service Mesh for Kubernetes
Page 8

1. Top-line service metrics 9
2. Encrypting all the things 13
3. Continuous deployment via traffic shifting 17
4. Dogfood, ingress & edge routing 30
5. Staging microservices without the tears 37
6. Distributed tracing made easy 44
7. Linkerd as an ingress controller 51
8. gRPC for fun and profit 59

III. Conclusion
Page 64

The future of the service mesh

Introduction

Over the past year, the *service mesh* has emerged as a critical component of the cloud native stack. High-traffic companies like PayPal, Lyft, Ticketmaster and Credit Karma have all added a service mesh to their production applications, typically alongside components like Kubernetes and Docker. This past January, Linkerd, the open source service mesh for cloud native applications, became an official project of the Cloud Native Computing Foundation.

But what is a service mesh, exactly? And why is it suddenly relevant?

This ebook defines the service mesh and traces its lineage through shifts in application architecture over the past decade. We provide a series of hands-on tutorials on how to use Linkerd as a service mesh with Kubernetes. Finally, we describe where the service mesh is heading, and what to expect as this concept evolves alongside cloud native adoption.

After reading this book, you should not only know what a service mesh is, you should be armed with concrete ways to use the Linkerd service mesh to make your Kubernetes application safer, faster, and more resilient.

WHAT IS A SERVICE MESH?

A service mesh is a dedicated infrastructure layer for handling service-to-service communication. It's responsible for the delivery of requests through the complex topology of services that comprise a modern, cloud native application. In practice, the service mesh is typically implemented as an array of lightweight network proxies that are deployed alongside application code, without the application needing to be aware. (But there are variations to this idea, as we'll see.)

The concept of the service mesh as a separate layer is tied to the rise of the cloud native application. In the cloud native model, a single application might consist of hundreds of services; each service might have thousands of instances; and each of those instances might be in a constantly changing state as they are dynamically scheduled by an orchestrator like Kubernetes. Not only is service communication in this world incredibly complex, it's a pervasive and fundamental part of runtime behavior. Managing it is vital to ensuring end-to-end performance and reliability.

IS THE SERVICE MESH A NETWORKING MODEL?

The service mesh is a networking model that sits at a layer of abstraction above TCP/IP. It assumes that the underlying L3/L4 network is present and capable of delivering bytes from point to point. (It also assumes that this network, as with every other aspect of the environment, is unreliable; the service mesh must therefore also be capable of handling network failures.)

In some ways, the service mesh is analogous to TCP/IP. Just as the TCP stack abstracts the mechanics of reliably delivering bytes between network endpoints, the service mesh abstracts the mechanics of reliably delivering requests between services. Like TCP, the service mesh doesn't care about the actual payload or how it's encoded. The application has a high-level goal ("send something from A to B"), and the job of the service mesh, like that of TCP, is to accomplish this goal while handling any failures along the way.

Unlike TCP, the service mesh has a significant goal beyond "just make it work": it provides a uniform, application-wide point for introducing visibility and control into the application runtime. The explicit goal of the service mesh is to move service communication out of the realm of the invisible, implied infrastructure, and into the role of a *first-class member of the ecosystem*—where it can be monitored, managed and controlled.

WHAT DOES A SERVICE MESH ACTUALLY DO?

Reliably delivering requests in a cloud native application can be incredibly complex. A service mesh like Linkerd manages this complexity with a wide array of powerful techniques: circuit-breaking, latency-aware load balancing, eventually consistent ("advisory") service discovery, retries, and deadlines. These features must all work in conjunction, and the interactions between these features and the complex environment in which they operate can be quite subtle.

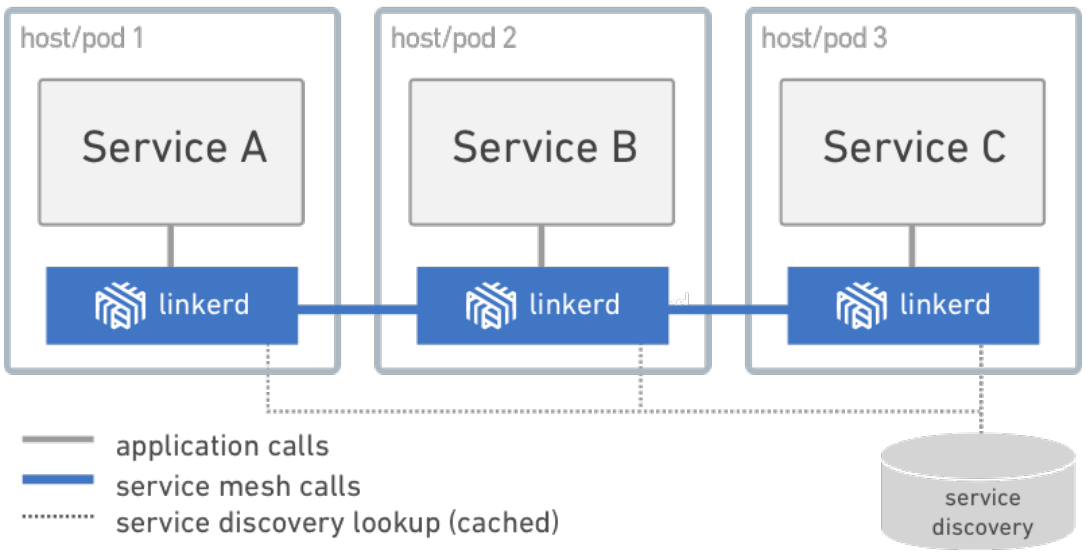
For example, when a request is made to a service through Linkerd, a very simplified timeline of events is as follows:

1. Linkerd applies dynamic routing rules to determine which service the requester intended. Should the request be routed to a service in production or in staging? To a service in a local datacenter or one in the cloud? To the most recent version of a service that's being tested or to an older one that's been vetted in

production? All of these routing rules are dynamically configurable, and can be applied both globally and for arbitrary slices of traffic.
2. Having found the correct destination, Linkerd retrieves the corresponding pool of instances from the relevant service discovery endpoint, of which there may be several. If this information diverges from what Linkerd has observed in practice, Linkerd makes a decision about which source of information to trust.
3. Linkerd chooses the instance most likely to return a fast response based on a variety of factors, including its observed latency for recent requests.
4. Linkerd attempts to send the request to the instance, recording the latency and response type of the result.
5. If the instance is down, unresponsive, or fails to process the request, Linkerd retries the request on another instance (but only if it knows the request is idempotent).
6. If an instance is consistently returning errors, Linkerd evicts it from the load balancing pool, to be periodically retried later (for example, an instance may be undergoing a transient failure).
7. If the deadline for the request has elapsed, Linkerd proactively fails the request rather than adding load with further retries.
8. Linkerd captures every aspect of the above behavior in the form of metrics and distributed tracing, which are emitted to a centralized metrics system.

And that's just the simplified version–Linkerd can also initiate and terminate TLS, perform protocol upgrades, dynamically shift traffic, and fail over between datacenters.

Linkerd instances form a service mesh, allowing application code to communicate reliably.



The Linkerd service mesh manages service-to-service communication and decouples it from application code.

It's important to note that these features are intended to provide both point-wise resilience and application-wide resilience. Large-scale distributed systems, no matter how they're architected, have one defining characteristic: they provide many opportunities for small, localized failures to escalate into system-wide catastrophic failures. The service mesh must be designed to safeguard against these escalations by shedding load and failing fast when the underlying systems approach their limits.

WHY IS THE SERVICE MESH NECESSARY?

The service mesh is ultimately not an introduction of new functionality, but rather a shift in where functionality is located. Web applications have always had to manage the complexity of service communication. The origins of the service mesh model can be traced in the evolution of these applications over the past decade and a half.

Consider the typical architecture of a medium-sized web application in the 2000's: the three-tiered app. In this model, application logic, web-serving logic, and storage logic are each a separate layer. The communication between layers, while complex, is limited in scope—there are only two hops, after all. There is no "mesh", but there is communication logic between hops, handled within the code of each layer.

When this architectural approach was pushed to very high scale, it started to break. Companies like Google, Netflix, and Twitter, faced with massive traffic requirements, implemented what was effectively a predecessor of the cloud native approach: the application layer was split into many services (sometimes called "microservices"), and the tiers became a topology. In these systems, a generalized communication layer became suddenly relevant, but typically took the form of a "fat client" library—Twitter's Finagle, Netflix's Hystrix, and Google's Stubby being cases in point.

In many ways, libraries like Finagle, Stubby, and Hystrix were the first service meshes. While they were specific to the details of their surrounding environment, and required the use of specific languages and frameworks, they were forms of dedicated infrastructure for managing service-to-service communication, and (in the case of the open source Finagle and Hystrix libraries) found use outside of their origin companies.

Fast forward to the modern cloud native application. The cloud native model combines the microservices approach of many small services with two additional factors: containers (e.g., Docker) which provide resource isolation and dependency management), and an orchestration layer (e.g., Kubernetes) which abstracts away the underlying hardware into a homogenous pool.

These three components allow applications to adapt with natural mechanisms for scaling under load and for handling the ever-present partial failures of the cloud environment. But with hundreds of services or thousands of instances, and an orchestration layer that's rescheduling instances from moment to moment, the path that a single request follows through the service topology can be incredibly complex, and since containers make it easy for each service to be written in a different language, the library approach is no longer feasible.

This combination of complexity and criticality motivates the need for a dedicated layer for service-to-service communication decoupled from application code and able to capture the highly dynamic nature of the underlying environment. This layer is the service mesh.

A service mesh like Linkerd provides critical features to multi-service applications running at scale such as:

- **Baseline resilience**: retry budgets, deadlines, and circuit breaking.
- **Top-line service metrics**: success rates, request volumes, and latencies.
- **Latency and failure tolerance**: Failure- and latency-aware load balancing that can route around slow or broken service instances.
- **Distributed tracing** a la Zipkin and OpenTracing
- **Service discovery**: locate destination instances.
- **Protocol upgrades**: wrapping cross-network communication in TLS, or converting HTTP/1.1 to HTTP/2.0.
- **Routing**: route requests between different versions of services, failover between clusters, etc.

Using Linkerd with Kubernetes

Linkerd is not just for Kubernetes, but Kubernetes is one of the most popular use cases. To that end, the following section provides a series of hands-on tutorials for how to use Linkerd as a service mesh on Kubernetes, including:

1. Top-line service metrics
2. Encrypting all the things
3. Continuous deployment via traffic shifting
4. Dogfood environments, ingress and edge routing
5. Staging microservices without the tears
6. Distributed tracing made easy
7. Linkerd as an ingress controller
8. gRPC for fun and profit

Note that these sections do not necessarily build on one another. If there's one particular feature that seems exciting to you, feel free to jump straight to that section!

And, of course, there are many other ways to take advantage of Linkerd's powerful feature set with Kubernetes that are not covered here—for example, using Linkerd to integrate Kubernetes with legacy, non-Kubernetes applications, or to seamlessly shift services in and out of Kubernetes. For these use cases and more, be sure to follow the Buoyant blog.

Let's get started!

CHAPTER 1

Top-line service metrics

In this chapter, we'll show you how to use Linkerd as a service mesh on Kubernetes, and how it can capture and report top-level service metrics such as success rates, request volumes, and latencies without requiring changes to application code. We will just focus on visibility: how a service mesh can automatically capture and report top-line metrics, such as success rate, for services.

USING LINKERD FOR SERVICE MONITORING IN KUBERNETES

One of the advantages of operating at the request layer is that the service mesh has access to protocol-level semantics of success and failure. For example, if you're running an HTTP service, Linkerd can understand the semantics of 200 versus 400 versus 500 responses and can calculate metrics like success rate automatically. (Operating at this layer becomes doubly important when we talk about retries—more on that in later articles.)

Let's walk through a quick example of how to install Linkerd on Kubernetes to automatically capture aggregated, top-line service success rates without requiring application changes.

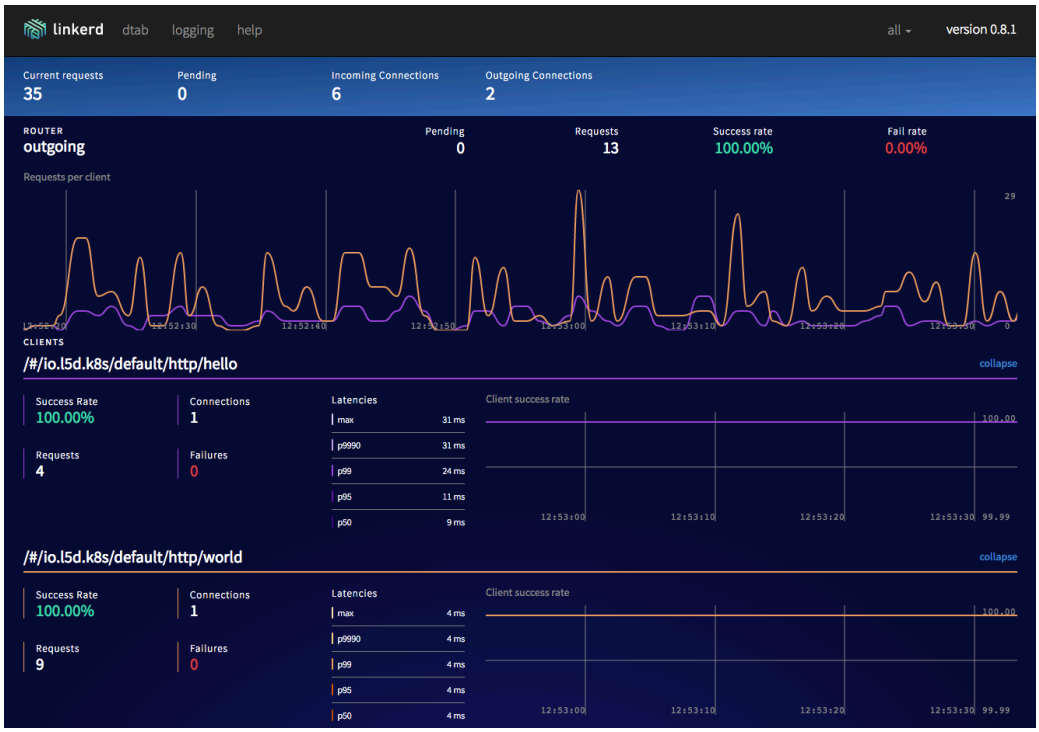
STEP 1: INSTALL LINKERD

Install Linkerd using this Kubernetes config. This will install Linkerd as a DaemonSet (i.e., one instance per host) running in the default Kubernetes namespace:

```
kubectl apply -f https://raw.githubusercontent.com/linkerd/linkerd-examples/master/k8s-daemonset/k8s/linkerd.yml
```

You can confirm that installation was successful by viewing Linkerd's admin page:

```
INGRESS_LB=$(kubectl get svc l5d -o jsonpath="{.status.loadBalancer.ingress[0].*}") open http://$INGRESS_LB:9990 # on OS X
```



STEP 2: INSTALL THE SAMPLE APPS

Install two services, “hello” and “world”, using this hello-world config. This will install the services into the default namespace:

```
kubectl apply -f https://raw.githubusercontent.com/linkerd/linkerd-examples/master/k8s-daemonset/k8s/hello-world.yml
```

These two services function together to make a highly scalable, “hello world” microservice (where the hello service, naturally, calls the world service to complete its request).

You can see this in action by sending traffic through Linkerd’s external IP:

```
http_proxy=$INGRESS_LB:4140 curl -s http://hello
```

You should see the string “Hello world”.

STEP 3: INSTALL LINKERD-VIZ

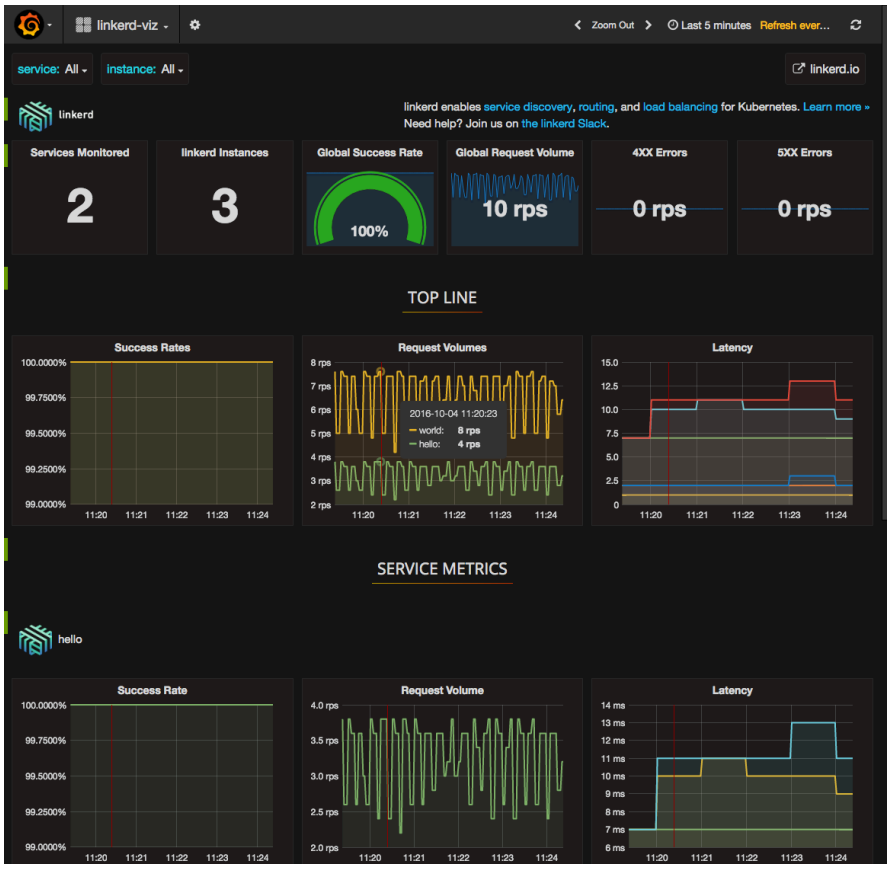
Finally, let's take a look at what our services are doing by installing linkerd-viz. Linkerd-viz is a supplemental package that includes a simple Prometheus and Grafana setup and is configured to automatically find Linkerd instances. Install linkerd-viz using this linkerd-viz config. This will install linkerd-viz into the default namespace:

```
kubectl apply -f https://raw.githubusercontent.com/linkerd/linkerd-viz/master/k8s/linkerd-viz.yml
```

Open linkerd-viz's external IP to view the dashboard:

```
VIZ_INGRESS_LB=$(kubectl get svc linkerd-viz -o
jsonpath="{.status.loadBalancer.ingress[0].*}") open http://
$VIZ_INGRESS_LB # on OS X
```

You should see a dashboard, including selectors by service and instance. All charts respond to these service and instance selectors:



The linkerd-viz dashboard includes three sections:

- **TOP LINE**: Cluster-wide success rate and request volume.
- **SERVICE METRICS**: One section for each application deployed. Includes success rate, request volume, and latency.
- **PER-INSTANCE METRICS**: Success rate, request volume, and latency for each node in your cluster.

With just three simple commands we were able to install Linkerd on our Kubernetes cluster, install an app, and use Linkerd to gain visibility into the health of the app's services. Of course, Linkerd is providing much more than visibility: under the hood, we've enabled latency-aware load balancing, automatic retries and circuit breaking, distributed tracing, and more. In the following chapters, we'll walk through how to take advantage of all these features.

CHAPTER 2

Encrypting all the things

In this chapter, we'll show you how to use Linkerd as a service mesh to add TLS to all service-to-service HTTP calls, without modifying any application code.

In the first chapter, we showed you how you can easily monitor top-line service metrics (success rates, latencies, and request rates) when Linkerd is installed as a service mesh. In this chapter, we'll show you another benefit of the service mesh approach: it allows you to decouple the application's protocol from the protocol used on the wire. In other words, the application can speak one protocol, but the bytes that actually go out on the wire are in another.

In the case where no data transformation is required, Linkerd can use this decoupling to automatically do protocol upgrades. Examples of the sorts of protocol upgrades that Linkerd can do include HTTP/1.x to HTTP/2, thrift to thrift-mux, and, the topic of this article, HTTP to HTTPS.

When Linkerd is deployed as a service mesh on Kubernetes, we place a Linkerd instance on every host using DaemonSets. For HTTP services, pods can send HTTP traffic to their host-local Linkerd by using the `http_proxy` environment variable. (For non-HTTP traffic the integration is slightly more complex.)

In a Buoyant blog post last year, we showed you the basic pattern of using Linkerd to "wrap" HTTP calls in TLS by proxying at both ends of the connection, both originating and terminating TLS. However, now that we have the service mesh deployment in place, things are significantly simpler. Encrypting all cross-host communication is largely a matter of providing a TLS certificate to the service mesh.

Let's walk through an example. The first two steps will be identical to what we did in chapter 1—we'll install Linkerd as a service mesh and install a simple microservice "hello world" application. If you have already done this, you can skip straight to Step 3.

STEP 1: INSTALL LINKERD

We can install Linkerd as a service mesh on our Kubernetes cluster by using this Kubernetes config. This will install Linkerd as a DaemonSet (i.e., one instance per host) in the default Kubernetes namespace:

```
kubectl apply -f https://raw.githubusercontent.com/linkerd/linkerd-examples/master/k8s-daemonset/k8s/linkerd.yml
```

You can confirm that installation was successful by viewing Linkerd's admin page (note that it may take a few minutes for the ingress IP to become available):

```
INGRESS_LB=$(kubectl get svc l5d –o
jsonpath="{.status.loadBalancer.ingress[0].*}")
open http://$INGRESS_LB:9990 # on OS X
```

STEP 2: INSTALL THE SAMPLE APPS

Install two services, "hello" and "world", using this hello-world config. This will install the services into the default namespace:

```
kubectl apply -f https://raw.githubusercontent.com/linkerd/linkerd-examples/master/k8s-daemonset/k8s/hello-world.yml
```

These two services function together to make a highly scalable, "hello world" microservice (where the hello service, naturally, must call the world service to complete its request).

At this point, we actually have a functioning service mesh and an application that makes use of it. You can see the entire setup in action by sending traffic through Linkerd's external IP:

```
http_proxy=$INGRESS_LB:4140 curl -s http://hello
```

If everything's working, you should see the string, "hello world".

STEP 3: CONFIGURE LINKERD TO USE TLS

Now that Linkerd is installed, let's use it to encrypt traffic. We'll place TLS certificates on each of the hosts, and configure Linkerd to use those certificates for TLS.

We'll use a global certificate (the mesh certificate) that we generate ourselves. Since this certificate is not tied to a public DNS name, we don't need to use a service like Let's Encrypt. We can instead generate our own CA certificate and use that to sign our mesh certificate ("self-signing"). We'll distribute three things to each Kubernetes host: the CA certificate, the mesh key, and the mesh certificate.

The following scripts use sample certificates that we've generated. *Please don't use these certificates in production*. For instructions on how to generate your own self-signed certificates, see our previous post, where we have instructions on how to generate your own certificates).

STEP 4: DEPLOY CERTIFICATES AND CONFIG CHANGES TO KUBERNETES

We're ready to update Linkerd to encrypt traffic. We will distribute the sample certificates as Kubernetes secrets.

```
kubectl apply -f https://raw.githubusercontent.com/linkerd/linkerd-examples/master/k8s-daemonset/k8s/certificates.yml
```

Now we will configure Linkerd to use these certificates by giving it this configuration and restarting it:

```
kubectl delete ds/l5d configmap/l5d-config kubectl apply -f https://raw.githubusercontent.com/linkerd/linkerd-examples/master/k8s-daemonset/k8s/linkerd-tls.yml
```

STEP 5: SUCCESS!

At this point, Linkerd should be transparently wrapping all communication between these services in TLS. Let's verify this by running the same command as before:

```
http_proxy=$INGRESS_LB:4140 curl -s http://hello
```

If all is well, you should still see the string, "hello world"—but under the hood, communication between the hello and world services is being encrypted. We can verify this by making an HTTPS request directly to port 4141, where Linkerd is listening for requests from other Linkerd instances:

```
curl -skH 'l5d-dtab: /svc=>/#/io.l5d.k8s/default/admin/l5d;' https://
$INGRESS_LB:4141/admin/ping
```

Here we're asking curl to make an HTTPS call, and telling it to skip TLS validation (since curl is expecting a website, not Linkerd). We're also adding a dtab override to route the request to the Linkerd instance's own admin interface. If all is well, you should again see a successful "pong" response. Congratulations! You've encrypted your cross-service traffic.

CONCLUSION

In this chapter, we've shown how a service mesh like Linkerd can be used to transparently encrypt all cross-node communication in a Kubernetes cluster. We're also using TLS to ensure that Linkerd instances can verify that they're talking to other Linkerd instances, preventing man-in-the-middle attacks (and misconfiguration!). Of course, the application remains blissfully unaware of any of these changes.

TLS is a complex topic and we've glossed over some important security considerations for the purposes of making the demo easy and quick. Please make sure you spend time to fully understand the steps involved before you try this on your production cluster.

Finally, adding TLS to the communications substrate is just one of many things that can be accomplished with a service mesh.

CHAPTER 3

Continuous deployment via traffic shifting

Beyond service discovery, top-line metrics, and TLS, Linkerd also has a powerful routing language called *dtabs* that can be used to alter the ways that requests—even individual requests—flow through the application topology. In this chapter, we'll show you how to use Linkerd as a service mesh to do blue-green deployments of new code as the final step of a CI/CD pipeline.

In previous chapters, we've shown you how you can use a service mesh like Linkerd to capture top-line metrics and transparently add TLS to your application, without changing application code.

In this chapter we'll show you an example of how to use Linkerd's routing rules, called dtabs, to automatically alter traffic flow through your application at the end of a CI/CD pipeline to perform a blue-green deployment between old and new versions of a service.

Continuous deployment (CD) is an extension of continuous integration (CI), in which code is pushed to production on a continuous basis, tightly coupled to the development process. While it requires powerful automation, minimizing the time between development and deployment allows companies to iterate very rapidly on their product.

For multi-service or microservice architectures, the final step of the CD process, the deployment itself can be risky because so much runtime behavior is determined by the runtime environment, including the other services that are handling production traffic. In these situations, gradual rollouts such as blue-green deployments become increasingly important.

Coordinating traffic shifting across multiple Linkerd instances requires a centralized traffic control tool. For this we recommend Namerd, a service with an API that serves routing rules backed by a consistent store. You can read more about how Namerd integrates with production systems in our Bouyant blog post covering routing in Linkerd.

We'll demonstrate a blue-green deployment using an example app from the linkerd-examples Github repo. The example app is a contrived "hello world" microservice application, consisting a of "hello" service that handles incoming requests and calls a "world" service before returning a response. With Jenkins as our automation server, we'll deploy a new version of the world service using the Jenkins Pipeline Plugin.

Before we start continuously deploying, we'll need to initially deploy the hello world app to Kubernetes, routing requests using Linkerd and Namerd. We can do this easily by using the Kubernetes configs in the linkerd-examples repo.

STEP 1: INSTALL NAMERD

We'll start by installing Namerd, which will manage the dtabs that we use to orchestrate our blue-green deployments. Please note that our Namerd configuration uses the ThirdPartyResource APIs, which requires a cluster running Kubernetes 1.2+ with the ThirdPartyResource feature enabled.

To install Namerd in the default Kubernetes namespace, run:

```
kubectl apply -f https://raw.githubusercontent.com/linkerd/linkerd-examples/master/k8s-daemonset/k8s/namerd.yml
```

You can confirm that installation was successful by viewing Namerd's admin page (note that it may take a few minutes for the ingress IP to become available):

```
NAMERD_INGRESS_LB=$(kubectl get svc namerd -o jsonpath="{.status.loadBalancer.ingress[0].*}") open http://$NAMERD_INGRESS_LB:9990 # on OS X
```

The admin page displays all configured Namerd namespaces, and we've configured two namespaces—"external" and "internal". For the sake of continuous deployment, we're mostly concerned with the "internal" namespace.

In addition to the admin UI, we can also use the namerctl utility to talk directly to Namerd. This utility will be used by the deploy script to start sending traffic to newly deployed services. To install it locally, run:

```
go get -u github.com/linkerd/namerctl
go install github.com/linkerd/namerctl
```

The utility uses the `NAMERCTL_BASE_URL` environment variable to connect to Namerd. In order to connect to the version of Namerd that we just deployed to Kubernetes, set the variable as follows:

```
export NAMERCTL_BASE_URL=http://$NAMERD_INGRESS_LB:4180
```

And now try using `namerctl` to display the internal dtab:

```
$ namerctl dtab get internal
# version MjgzNjk5NzI=
/srv          => /#/io.l5d.k8s/default/http ;
/host         => /srv ;
/tmp          => /srv ;
/svc          => /host ;
/host/world   => /srv/world-v1 ;
```

The last line of the dtab maps the logical name of the `world` service to the currently deployed version of the world service, `world-v1`. In a production system, versions could be shas, dates, or anything else that guarantees name uniqueness. We'll use this dtab entry to safely introduce new versions of the world service into production.

STEP 2: INSTALL LINKERD

Next we'll install Linkerd and configure it to resolve routes using Namerd. To install Linkerd as a DaemonSet (i.e., one instance per host) in the default Kubernetes namespace, run:

```
kubectl apply -f https://raw.githubusercontent.com/linkerd/linkerd-examples/master/k8s-daemonset/k8s/linkerd-namerd.yml
```

You can confirm that installation was successful by viewing Linkerd's admin UI (note that it may take a few minutes for the ingress IP to become available):

```
L5D_INGRESS_LB=$(kubectl get svc l5d -o
jsonpath="{.status.loadBalancer.ingress[0].*}") open http://
$L5D_INGRESS_LB:9990 # on OS X
```

We'll use the admin UI to verify steps of the blue-green deploy.

STEP 3: INSTALL THE SAMPLE APPS

Now we'll install the hello and world apps in the default namespace, by running:

```
kubectl apply -f https://raw.githubusercontent.com/linkerd/linkerd-
examples/master/k8s-daemonset/k8s/hello-world.yml
```

At this point, we actually have a functioning service mesh and an application that makes use of it. You can see the entire setup in action by sending traffic through Linkerd's external IP:

```
$ curl $L5D_INGRESS_LB
Hello (10.196.2.5) world (10.196.2.6)!!
```

If everything is working, you'll see a "hello world" message similar to that above, with the IPs of the pods that served the request.

Continuous deployment

We'll now use Jenkins to perform blue-green deploys of the "world" service that we deployed in the previous step.

SET UP JENKINS

Let's start by deploying the buoyantio/jenkins-plus Docker image to our Kubernetes cluster. This image provides the base `jenkins` image, along with

the `kubectl` and `namerctl` binaries that we need, as well as additional plugins and a pre-configured pipeline job that we can use to run deployments. The pipeline job makes use of the Jenkins Pipeline Plugin and a custom Groovy script that handles each of the steps in the blue-green deploy for us.

To deploy the Jenkins image to the default Kubernetes namespace, run:

```
kubectl apply -f https://raw.githubusercontent.com/linkerd/linkerd-examples/master/k8s-daemonset/k8s/jenkins.yml
```

You can confirm that installation was successful by opening up the Jenkins web UI (note that it may take a few minutes for the ingress IP to become available):

```
JENKINS_LB=$(kubectl get svc jenkins -o jsonpath="{.status.loadBalancer.ingress[0].*}") open http://$JENKINS_LB
# on OS X
```

You should see a “hello_world” job in the UI.

COMMITTING CODE

Now it’s time to make some code changes to the world service, and have the Jenkins job deploy them to production for us. To do this, start by forking the linkerd-examples repo in the Github UI. Once you’ve created a fork, clone your fork locally:

```
git clone https://github.com/esbie/linkerd-examples.git
cd linkerd-examples
```

For the sake of this example, we’re going to change a text file that controls the output of the world service. By default, the world service outputs the string “world”:

```
$ cat k8s-daemonset/helloworld/world.txt
world
```

Let’s spice that up a bit:

```
echo "hal, open the pod bay doors" > k8s-daemonset/helloworld/world.txt
```

And commit it:

```
git commit -am "Improve the output of the world service"
git push origin master
```

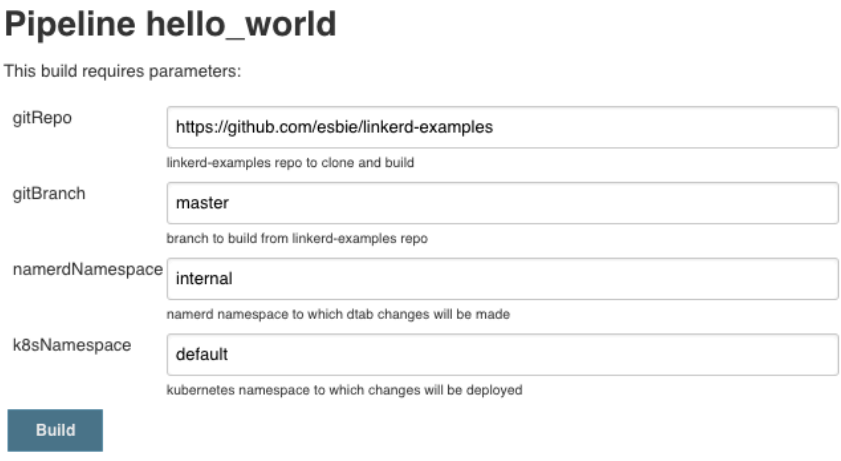
Now it's time to get this critical change into production.

RUNNING THE JOB

With our change committed and pushed to our fork of the `linkerd-examples` repo, we can kick off the Jenkins "hello_world" pipeline job to safely deploy the change into production. Each of the 6 steps in the pipeline job is controlled by a custom Groovy script and described below in more detail. The deploy is fully automated, with the exception of three places in the pipeline where it pauses for human-in-the-loop verification of critical metrics before proceeding.

BUILD WITH PARAMETERS

To start the deploy, click into the "hello_world" job in the Jenkins UI, and then click "Build with the parameters" in the sidebar. You'll be taken to a page that lets you customize the deploy, and it will look something like this:



Change the value of the `gitRepo` form field to point to your fork of the `linkerd-examples` repo, and then click the "Build" button. Note that if you pushed your changes to a separate branch in your fork, you should also change the value of the `gitBranch` form field to match your branch name.

CLONE

The first step in the pipeline is to clone the git repo using the build parameters specified above. Pretty straightforward.

DEPLOY

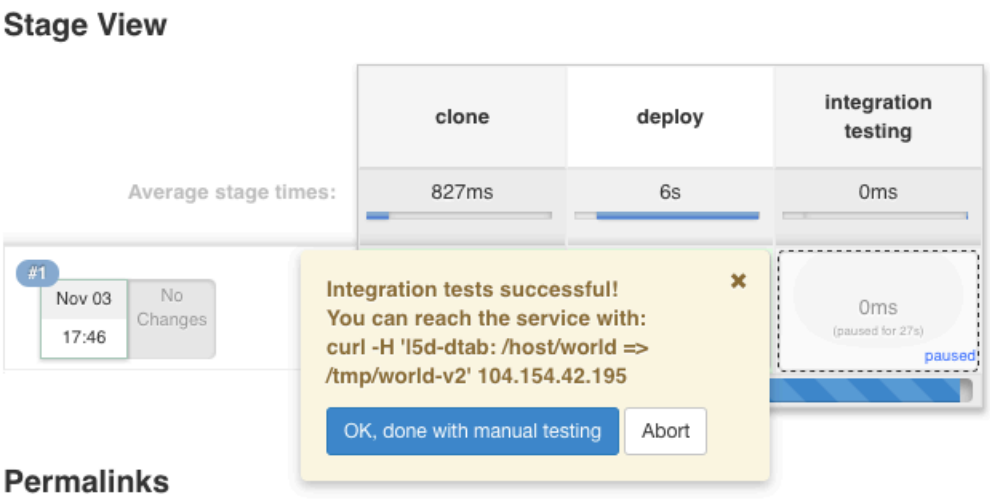
The second step in the deploy pipeline is to actually deploy the new version of the world service to our cluster, without sending it any traffic. The script determines that the currently deployed version of the world service is `world-v1`, so it creates a new service called `world-v2` and deploys that to our Kubernetes cluster. At this point you will see two different versions of the world service running simultaneously:

```
$ kubectl get po | grep world
world-v1-9eaxk                  1/1       Running   0          3h
world-v1-kj6gi                  1/1       Running   0          3h
world-v1-vchal                  1/1       Running   0          3h
world-v2-65y9g                  1/1       Running   0          30m
world-v2-d260q                  1/1       Running   0          30m
world-v2-z7ngo                  1/1       Running   0          30m
```

Even with the `world-v2` version fully deployed, we still have not made any changes to production traffic! Linkerd and Namerd are still configured to route all world service traffic to the existing `world-v1` version. Fully deploying a new version of the service before sending it any traffic is key to performing a blue-green deploy.

INTEGRATION TESTING

Once the new version of our service is deployed, the script performs a test request to make sure the new version can be reached. If the test request succeeds, it pauses the deploy and waits for us to acknowledge that the newly deployed version looks correct before proceeding.



At this point, we want to make sure that the new pods are running as expected—not just by themselves, but in conjunction with the rest of the production environment. Normally this would involve a deployment to a separate staging cluster, combined with some mechanism for sending or replaying production traffic to that cluster.

Since we're using Linkerd, we can significantly simplify this operation by taking advantage of Linkerd's per-request routing to accomplish the same thing without a dedicated staging environment. At ingress, we can tag our request with a special header, `l5d-dtab`, that will instruct Linkerd to route this request through the production cluster, but replace all service calls to `world-v1` with calls to `world-v2` instead *for this request only*.

The Jenkins UI provides the dtab override that we need to route requests to the new version of our service, and using that information we can make our own test request:

```
$ curl -H 'l5d-dtab: /host/world => /tmp/world-v2' $L5D_INGRESS_LB
Hello (10.196.2.5) hal, open the pod bay doors (10.196.1.17)!!
```

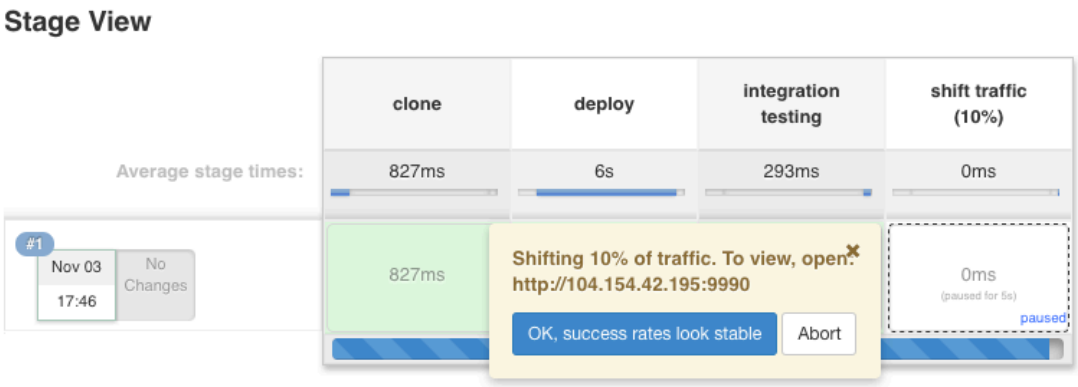
Success! Our request is being routed to the `world-v2` service, which is returning the new world text that we added on our branch. Even though we can reach the new service, it's worth noting that we *still* have not changed the behavior of any production traffic, aside from the request that we just made. We can verify that by omitting the `l5d-dtab` header and ensuring that we still get the `world-v1` response:

```
$ curl $L5D_INGRESS_LB
Hello (10.196.2.5) world (10.196.2.6)!!
```

If everything looks good, we can proceed to the next step in the pipeline by clicking the "Ok, I'm done with manual testing" button in the Jenkins UI.

SHIFT TRAFFIC (10%)

After some manual testing, we're ready to start the blue-green deployment by sending 10% of production traffic to the newly deployed version of the service. The script makes the change in routing policy and again pauses, asking us to confirm that everything looks OK with 10% traffic before proceeding.

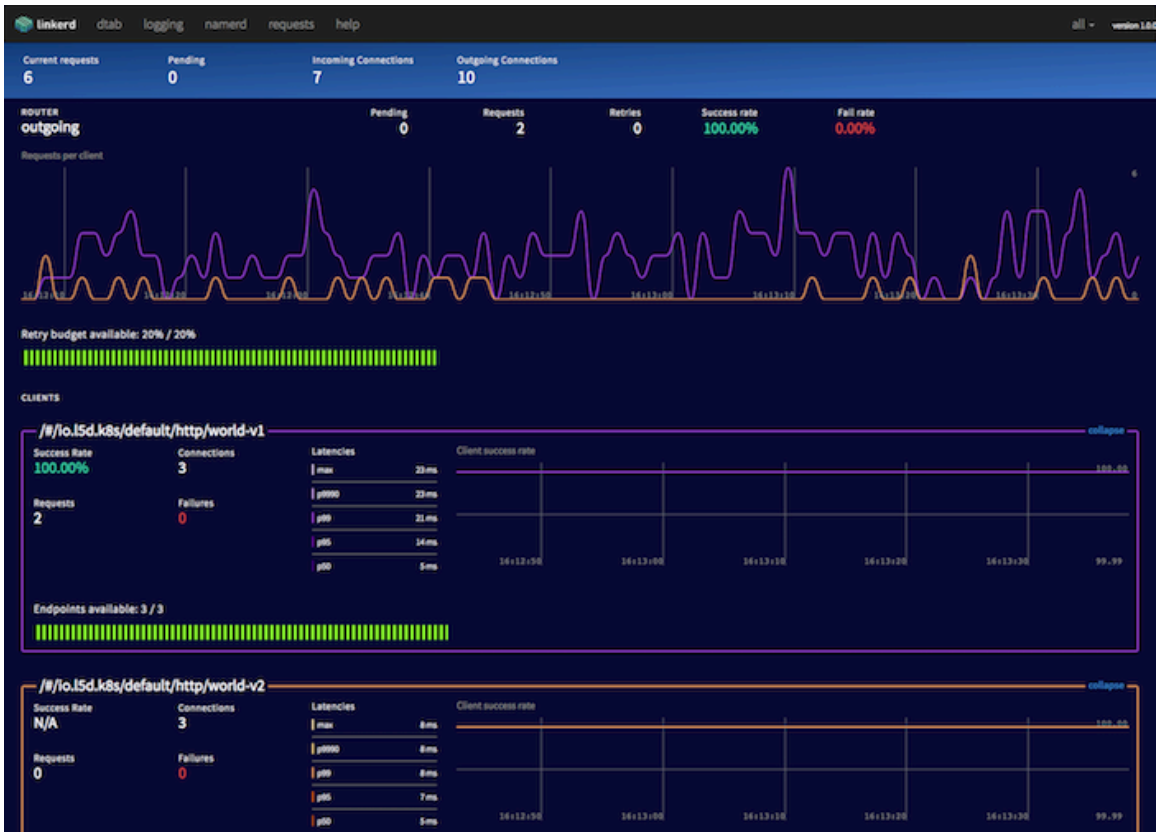


Note that if the user aborts on any pipeline step, the script assumes there was something wrong with the new service, and automatically reverts the routing change, sending all traffic back to the original service. Since we're not tearing down instances of the old version of the service while shifting traffic, reverting traffic back can happen quickly, minimizing the impact of a bad deploy.

We can verify that our service is taking 10% of requests by sending it 10 requests and hoping that the odds are in our favor:

```
$ for i in {1..10}; do curl $L5D_INGRESS_LB; echo ""; done
Hello (10.196.2.5) world (10.196.1.16)!!
Hello (10.196.2.5) world (10.196.1.16)!!
Hello (10.196.2.5) hal, open the pod bay doors (10.196.2.13)!!
Hello (10.196.2.5) world (10.196.2.6)!!
Hello (10.196.1.13) world (10.196.2.6)!!
Hello (10.196.1.13) world (10.196.2.6)!!
Hello (10.196.2.5) world (10.196.1.16)!!
Hello (10.196.2.5) world (10.196.2.6)!!
Hello (10.196.1.14) world (10.196.2.6)!!
Hello (10.196.1.14) world (10.196.1.16)!!
```

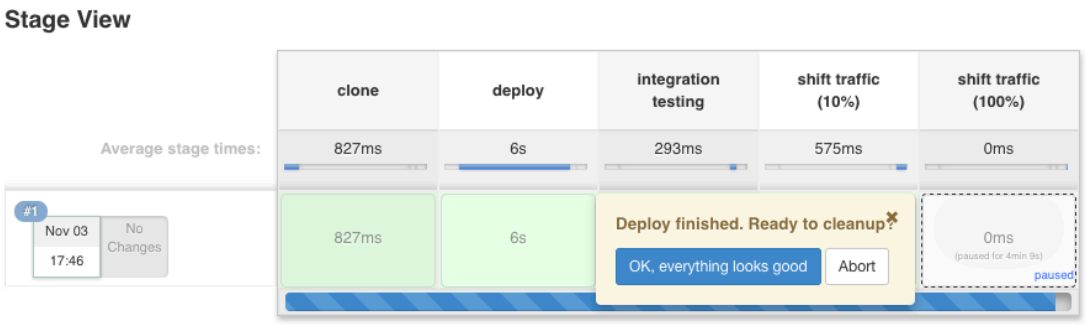
Looking good! Now is also a good time to check Linkerd's admin dashboard, to verify that the new service is healthy. If your application were receiving a small amount of steady traffic, then the dashboard would look like this:



We can see right away that the `world-v2` service is taking roughly 10% of traffic, with 100% success rate. If everything looks good, we can proceed to the next step by clicking the “Ok, success rates look stable” button in the Jenkins UI.

SHIFT TRAFFIC (100%)

In this step the script shifts additional traffic to the new version of our service. For a concise example, we’re moving immediately to 100% of traffic, but in a typical deployment you could include additional intermediary percentages as separate steps in the pipeline.



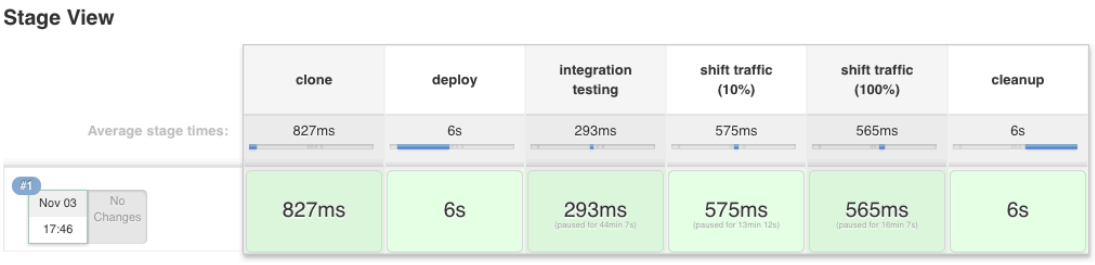
We can verify that the new service is serving traffic by sending it a request without a dtab override header:

```
$ curl $L5D_INGRESS_LB
Hello (10.196.2.5) hal, open the pod bay doors (10.196.2.13)!!
```

Once we’re confidant that `world-v2` is successfully handling 100% of production traffic, we can proceed to the final step by clicking the “Ok, everything looks good” button in the Jenkins UI.

CLEANUP

In the final step, the script finalizes the deploy by making the routing rules to route traffic to the new version of the service permanent. It also tears down the previous version of the service that was still running in our cluster but not receiving any traffic.



The final version of Namerd's dtab is now:

```
$ namerctl dtab get internal
# version MTIzMzU0OTE=
/srv          => /#/io.l5d.k8s/default/http ;
/host         => /srv ;
/tmp          => /srv ;
/http/*/*     => /host ;
/host/world   => /srv/world-v2 ;
```

We can verify that the old service has been torn down by looking at the world service pods that are currently deployed to our cluster.

```
$ kubectl get po | grep
world world-v2-65y9g          1/1       Running   0          1h
world-v2-d260q                1/1       Running   0          1h
world-v2-z7ngo                1/1       Running   0          1h
```

Everything looks good. Kicking off a subsequent pipeline job will deploy a `world-v3`version of the service, gradually shift traffic over, and then promote it to the current version when the deploy successfully completes.

CONCLUSION

In this chapter, we've shown a basic workflow incorporating Linkerd, Namerd, and Jenkins to progressively shift traffic from an old version to a new version of a service as the final step of a continuous deployment pipeline. We've shown how Linkerd's ability to do per-request routing actually lets us stage the new version of the service without needing a

separate staging cluster, by using the `l5d-dtab` header to stitch the new service into the production topology *just for that request*. Finally, we've shown how percentage-based traffic shifting can be combined with a Jenkins `input` step to allow for human-in-the-loop verification of metrics as traffic moves from 0% to 100%.

This was a fairly simple example, but we hope it demonstrates the basic pattern of using service mesh routing for continuous deployment and provides a template for customizing this workflow for your own organization.

CHAPTER 4

Dogfood environments, ingress and edge routing

In this chapter, we'll show you how to use a service mesh of Linkerd instances to handle ingress traffic on Kubernetes, distributing traffic across every instance in the mesh. We'll also walk through an example that showcases Linkerd's advanced routing capabilities by creating a *dogfood* environment that routes certain requests to a newer version of the underlying application, e.g. for internal, pre-release testing.

This chapter is about using Linkerd as an ingress point for traffic to a Kubernetes network. As of 0.9.1, Linkerd supports the Kubernetes Ingress resource directly, which is an alternate, and potentially simpler starting point for some of the use cases in this article. For information on how to use Linkerd as a Kubernetes ingress controller, please see the later chapter entitled ***Linkerd as an ingress controller***.

In previous chapters, we've shown you how you can use Linkerd to capture top-line service metrics, transparently add TLS across service calls, and perform blue-green deploys. These chapters showed how using Linkerd as a service mesh in environments like Kubernetes adds a layer of resilience and performance to internal, service-to-service calls. In this chapter, we'll extend this model to ingress routing.

In this chapter, we won't use the built-in Ingress Resource that Kubernetes provides (again, see the later chapter entitled *Linkerd as an ingress controller* for that use case). Kubernetes Ingress Resources are a convenient way of doing basic path and host-based routing, but they are fairly limited. In the examples below, we'll be reaching far beyond what they provide.

STEP 1: DEPLOY THE LINKERD SERVICE MESH

Starting with our basic Linkerd service mesh Kubernetes config from the previous articles, we'll make two changes to support ingress: we'll modify the Linkerd config to add an additional logical router, and we'll tweak the VIP in the

Kubernetes Service object around Linkerd. (The full config is here: Linkerd-ingress.yml.)

Here's the new `ingress` logical router on Linkerd instances that will handle ingress traffic and route it to the corresponding services:

```
routers:
- protocol: http
  label: ingress
  dtab: |
    /srv                    => /#/io.l5d.k8s/default/http ;
    /domain/world/hello/www => /srv/hello ;
    /domain/world/hello/api => /srv/api ;
    /host                   => /$/io.buoyant.http.domainToPathPfx/
domain ;
    /svc                    => /host ;
  interpreter:
    kind: default
    transformers:
    - kind: io.l5d.k8s.daemonset
      namespace: default
      port: incoming
      service: l5d
  servers:
  - port: 4142
    ip: 0.0.0.0
```

In this config, we're using Linkerd's routing syntax, dtabs, to route requests from domain to service—in this case from "api.hello.world" to the `api` service, and from "www.hello.world" to the `world` service. For simplicity's sake, we've added one rule per domain, but this mapping can easily be generified for more complex setups. (If you're a Linkerd config aficionado, we're accomplishing this behavior by combining Linkerd's default header token identifier to route on the Host header, the `domainToPathPfx` namer to turn dotted hostnames into hierarchical paths, and the transformer to send requests to the corresponding host-local Linkerd.)

We've added this ingress router to every Linkerd instance—in true service mesh fashion, we'll fully distribute ingress traffic across these instances so that no instance is a single point of failure.

We also need modify our Kubernetes `Service` object to replace the `outgoing` VIP with an `ingress` VIP on port 80. This will allow us to send ingress traffic directly to the Linkerd service mesh—mainly for debugging purposes, since the this traffic will not be sanitized before hitting Linkerd. (In the next step, we'll fix this.)

The Kubernetes change looks like this:

```
---
apiVersion: v1
kind: Service
metadata:
  name: l5d
spec:
  selector:
    app: l5d
  type: LoadBalancer
  ports:
  - name: ingress
    port: 80
    targetPort: 4142
  - name: incoming
    port: 4141
  - name: admin
    port: 9990
```

All of the above can be accomplished in one fell swoop by running this command to apply the full Linkerd service mesh plus ingress Kubernetes config:

```
$ kubectl apply -f https://raw.githubusercontent.com/linkerd/linkerd-examples/master/k8s-daemonset/k8s/linkerd-ingress.yml
```

STEP 2: DEPLOY THE SERVICES

For services in this example, we'll use the same hello and world configs from the previous blog posts, and we'll add two new services: an api service, which calls both `hello` and `world`, and a new version of the world service, `world-v2`, which will return the word "earth" rather than "world"—our growth hacker

team has assured us their A/B tests show this change will increase engagement tenfold.

The following commands will deploy the three hello world services to the default namespace:

```
$ kubectl apply -f https://raw.githubusercontent.com/linkerd/linkerd-examples/master/k8s-daemonset/k8s/hello-world.yml
$ kubectl apply -f https://raw.githubusercontent.com/linkerd/linkerd-examples/master/k8s-daemonset/k8s/api.yml
$ kubectl apply -f https://raw.githubusercontent.com/linkerd/linkerd-examples/master/k8s-daemonset/k8s/world-v2.yml
```

At this point we should be able to test the setup by sending traffic through the `ingress` Kubernetes VIP. In the absence of futzing with DNS, we'll set a Host header manually on the request:

```
$ INGRESS_LB=$(kubectl get svc l5d -o jsonpath="{.status.loadBalancer.ingress[0].*}")
$ curl -s -H "Host: www.hello.world" $INGRESS_LB
Hello (10.0.5.7) world (10.0.4.7)!!
$ curl -s -H "Host: api.hello.world" $INGRESS_LB
{"api_result":"api (10.0.3.6) Hello (10.0.5.4) world (10.0.1.5)!!"}
```

Success! We've set up Linkerd as our ingress controller, and we've used it to route requests received on different domains to different services. And as you can see, production traffic is hitting the `world-v1` service—we aren't ready to bring `world-v2` out just yet.

STEP 3: A LAYER OF NGINX

At this point we have functioning ingress. However, we're not ready for production just yet. For one thing, our ingress router doesn't strip headers from requests, which means that external requests may include headers that we do not want to accept. For instance, Linkerd allows setting the `l5d-dtab` header to apply routing rules per-request. This is a useful feature for ad-hoc staging of new services, but it's probably not appropriate calls from the outside world!

For example, we can use the `l5d-dtab` header to override the routing logic to use `world-v2` rather than the production `world-v1` service the outside world:

```
$ curl -H "Host: www.hello.world" -H "l5d-dtab: /host/world => /srv/
world-v2;" $INGRESS_LB Hello (10.100.4.3) earth (10.100.5.5)!!
```

Note the **earth** in the response, denoting the result of the `world-v2` service. That's cool, but definitely not the kind of power we want to give just anyone!

We can address this (and other issues, such as serving static files) by adding nginx to the mix. If we configure nginx to strip incoming headers before proxying requests to the Linkerd ingress route, we'll get the best of both worlds: an ingress layer that is capable of safely handling external traffic, and Linkerd doing dynamic, service-based routing.

Let's add nginx to the cluster. We'll configure it using this nginx.conf. We'll use the `proxy_pass` directive under our virtual servers `www.hello.world` and `api.hello.world` to send requests to the Linkerd instances, and, for maximum fanciness, we'll strip Linkerd's context headers using the `more_clear_input_headers` directive (with wildcard matching) provided by the Headers More module.

(Alternatively, we could avoid third-party nginx modules by using nginx's `proxy_set_header` directive to clear headers. We'd need separate entries for each `l5d-ctx-` header as well as the `l5d-dtab` and `l5d-sample` headers.)

Note that as of Linkerd 0.9.0, we can clear incoming `l5d-*` headers by setting `clearContext: true` on the ingress router server. However, nginx has many features we can make use of (as you'll see presently), so it is still valuable to use nginx in conjunction with Linkerd.

For those of you following along at home, we've published an nginx Docker image with the *Headers More* module installed (Dockerfile here) as buoyantio/nginx:1.11.5. We can deploy this image with our config above using this Kubernetes config:

```
$ kubectl apply -f https://raw.githubusercontent.com/linkerd/linkerd-
examples/master/k8s-daemonset/k8s/nginx.yml
```

After waiting a bit for the external IP to appear, we can test that nginx is up by hitting the simple test endpoint in the nginx.conf:

```
$ INGRESS_LB=$(kubectl get svc nginx —o
jsonpath="{.status.loadBalancer.ingress[0].*}")
$ curl $INGRESS_LB
200 OK
```

We should be able to now send traffic to our services through nginx:

```
$ curl -s -H "Host: www.hello.world" $INGRESS_LB
Hello (10.0.5.7) world (10.0.4.7)!!
$ curl -s -H "Host: api.hello.world" $INGRESS_LB
{"api_result":"api (10.0.3.6) Hello (10.0.5.4) world (10.0.1.5)!!"}
```

Finally, let's try our header trick and attempt to communicate directly with the `world-v2` service:

```
$ curl -H "Host: www.hello.world" -H "l5d-dtab: /host/world => /srv/
world-v2;" $INGRESS_LB
Hello (10.196.1.8) world (10.196.2.13)!!
```

Great! No more **earth**. Nginx is sanitizing external traffic.

STEP 4: TIME FOR SOME DELICIOUS DOGFOOD!

Ok, we're ready for the good part: let's set up a dogfood environment that uses the `world-v2` service, but only for some traffic!

For simplicity, we'll target traffic that sets a particular cookie, `special_employee_cookie`. In practice, you probably want something more sophisticated than this—authenticate it, require that it come from the corp network IP range, etc.

With nginx and Linkerd installed, accomplishing this is quite simple. We'll use nginx to check for the presence of that cookie, and set a dtab

override header for Linkerd to adjust its routing. The relevant nginx config looks like this:

```
if ($cookie_special_employee_cookie ~* "dogfood") {
  set $xheader "/host/world => /srv/world-v2;";
}
proxy_set_header 'l5d-dtab' $xheader;
```

If you've been following the steps above, the deployed nginx already contains this configuration. We can test it like so:

```
$ curl -H "Host: www.hello.world" --cookie
"special_employee_cookie=dogfood" $INGRESS_LB
Hello (10.196.1.8) earth (10.196.2.13)!!
```

The system works! When this cookie is set, you'll be in dogfood mode. Without it, you'll be in regular, production traffic mode. Most importantly, dogfood mode can involve new versions of services that appear *anywhere* in the service stack, even many layers deep—as long as service code forwards Linkerd context headers, the Linkerd service mesh will take care of the rest.

CONCLUSION

In this chapter, we saw how to use Linkerd to provide powerful and flexible ingress to a Kubernetes cluster. We've demonstrated how to deploy a nominally production-ready setup that uses Linkerd for service routing. And we've demonstrated how to use some of the advanced routing features of Linkerd to decouple the *traffic-serving* topology from the *deployment topology*, allowing for the creation of dogfood environments without separate clusters or deploy-time complications.

CHAPTER 5

Staging microservices without the tears

Staging new code before exposing it to production traffic is a critical part of building reliable, low-downtime software. Unfortunately, with microservices, the addition of each new service increases the complexity of the staging process, as the dependency graph between services grows quadratically with the number of services. In this chapter, we'll show you how one of Linkerd's most powerful features, *per-request routing*, allows you to neatly sidestep this problem.

For a video presentation of the concepts discussed in this chapter see Alex Leong's meetup talk, Microservice Staging without the Tears.

Linkerd acts as a transparent request proxy that adds a layer of resilience to applications by wrapping cross-service calls with features like latency-aware load balancing, retry budgets, deadlines, and circuit breaking.

In addition to improving application resilience, Linkerd also provides a powerful routing language that can alter how request traffic flows between services at runtime. In this chapter, we'll demonstrate Linkerd's ability to do this routing, not just globally, but on a per-request basis. We'll show how this *per-request routing* can be used to create ad-hoc staging environments that allow us to test new code in the context of the production application, without actually exposing the new code to production traffic. Finally, we'll show how (in contrast to staging with a dedicated staging environment) ad-hoc staging requires neither coordination with other teams, nor the costly process of keeping multiple deployment environments in sync.

WHY STAGE?

Why is staging so important? In modern software development, code goes through a rigorous set of practices designed to assess *correctness*: code review, unit tests, integration tests, etc. Having passed these hurdles, we move to assessing *behavior*: how fast is the new code? How does it behave under load? How does it interact with runtime dependencies, including other services?

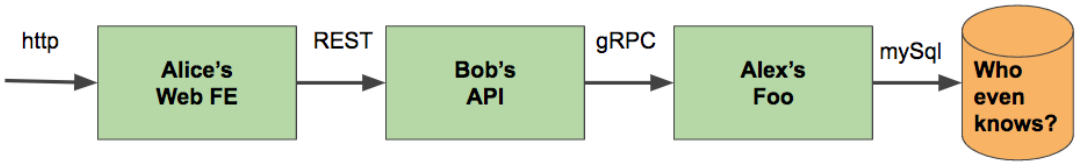
These are the questions that a staging environment can answer. The fundamental principle of staging is that the closer to the production environment, the more realistic staging will be. Thus, while mocks and stub implementations make sense for tests, for staging, we ideally want actual running services. The best staging environment is one in which the surrounding environment is exactly the same as it will be in production.

WHY IS STAGING HARD FOR MICROSERVICES?

When your application consists of many services, the interaction between these services becomes a critical component of end-to-end application behavior. In fact, the more that the application is disaggregated into services, the more that the runtime behavior of the application is determined not just by the services themselves, but by the interactions between them.

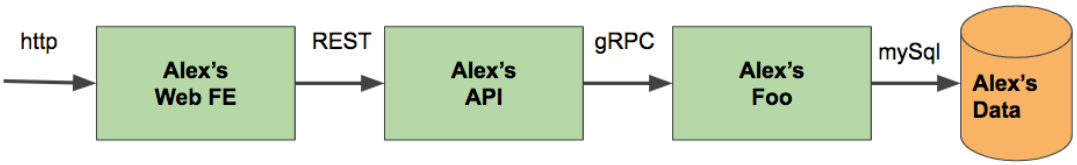
Unfortunately, increasing the number of services doesn't just increase the importance of proper staging, it also increases the difficulty of doing this properly. Let's take a look at a couple common ways of staging, and why they suffer in multi-service environments.

A frequent choice for staging is the shared staging cluster, wherein your staged service is deployed into a dedicated staging environment alongside other staged services. The problem with this approach is that there is no isolation. If, as in the diagram below, Alex deploys his Foo service and sees weird behavior, it's difficult to determine the source—it could be due to the staging deploys of Alex, Alice, or Bob, or simply the mock data in the database. Keeping the staging environment in sync with production can be very difficult, especially as the number of services, teams, and releases all start to increase.



An alternative to the shared environment that addresses the lack of isolation is the "personal" or per-developer, staging cluster. In this model, every developer

can spin up a staging cluster on demand. To keep our staging effective, staging a service requires staging its upstream and downstream dependencies as well. (For example, in the diagram below, Alex would need to deploy Web FE and API in order to ensure the changes he made to his Foo service are correctly reflected there.) Unfortunately, maintaining the ability to deploy arbitrary subsets of the application topology on demand also becomes very complex, especially as the application topology becomes larger, and as services have independent deployment models.

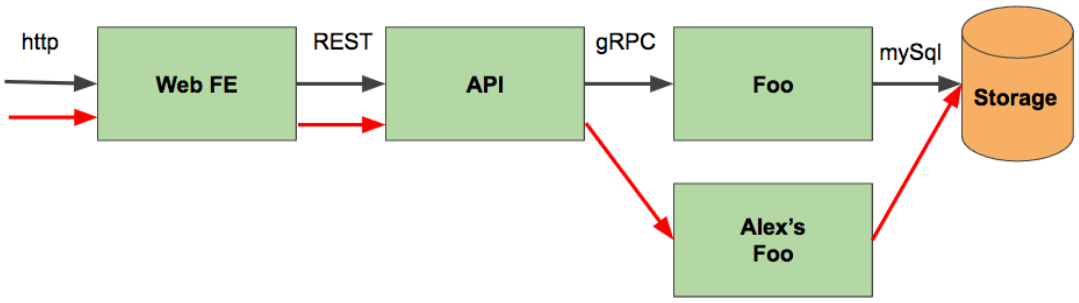


Finally, there is the (sadly prevalent) option of simply deploying fresh code into production and rolling it back when flaws are discovered. Of course, this is rather risky, and may not be an option for applications that handle, e.g., financial transactions. There are many other ways you could obtain a staging environment, but in this article, we'll describe a straightforward, tear-free approach.

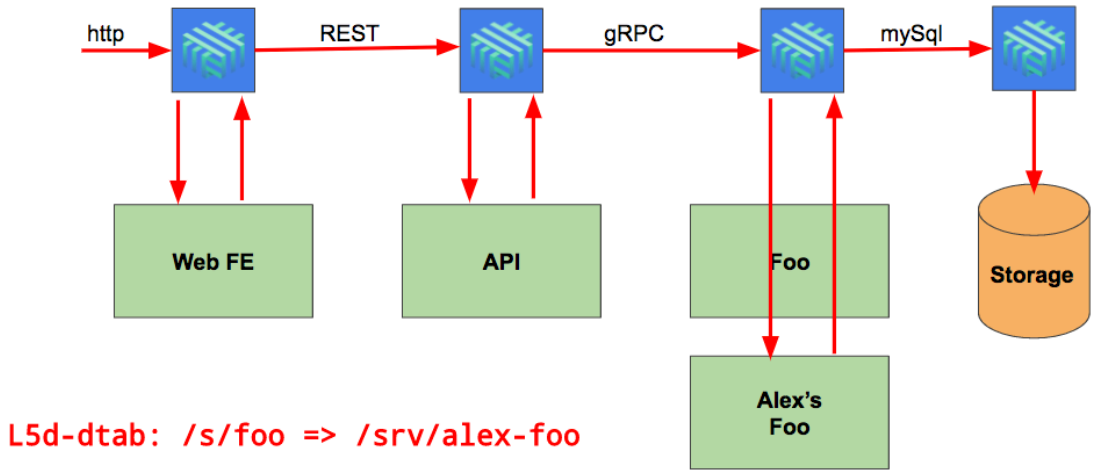
A BETTER PATH

Fortunately, with Linkerd, we can do staging without incurring the costs detailed above, by creating *ad-hoc staging environments*. In fact, one of the prime motivations for the routing layer in Finagle, the library underlying Linkerd, was solving this very problem at Twitter.

Let's consider again the goal of staging Alex's Foo service. What if, rather than deploying to a separate environment, we could simply substitute Foo-staging in place of Foo-production, for a specific request? That would give us the ability to stage Foo safely, against the production environment, without requiring any deployment other than that of Foo-staging itself. This is the essence of ad-hoc staging environments. The burden on the developer is now greatly eased: Alex must simply stage his new code, set a header on ingress requests, and voila!



Happily, Linkerd's per-request routing allow us to do just this. With Linkerd proxying traffic, we can set a routing "override" for a particular request using the `l5d-dtab` header. This header allows you to set routing rules (called, in Finagle parlance, "Dtabs") for that request. For example, the dtab rule `/s/foo => /srv/alex-foo` might override the production routing rule for Foo. Attaching this change to a *single request* would allow us to reach Alex's Foo service, but only for that request. Linkerd propagates this rule, so any usage of Alex's Foo service anywhere in the application topology, for the lifetime of that request, will be properly handled.



TRYING THIS AT HOME

We've already seen an example of this in Chapter 5 on dogfood environments. We deployed a `world-v2` service, and we were able to send individual dogfood requests through this service via a simple header containing a routing override. Now, we can use this same mechanism for something else: setting up an ad hoc staging environment.

Let's deploy two versions of a service and use Linkerd's routing capabilities to test our new service before using it in production. We'll deploy our `hello` and `world-v1` services as our running prod services, and then we'll create an ad-hoc staging environment to stage and test a new version of world, `world-v2`.

STEP 1: DEPLOY LINKERD AND OUR HELLO-WORLD SERVICES

We'll use the hello world service from the previous blog posts. This consists of a hello service that calls a world service.

Let's deploy our prod environment (Linkerd, and the hello and world services):

```
$ kubectl apply -f https://raw.githubusercontent.com/linkerd/linkerd-examples/master/k8s-daemonset/k8s/linkerd-ingress.yml
$ kubectl apply -f https://raw.githubusercontent.com/linkerd/linkerd-examples/master/k8s-daemonset/k8s/hello-world.yml
```

Let's also deploy the service we want to stage, world-v2, which will return the word "earth" rather than "world".

```
$ kubectl apply -f https://raw.githubusercontent.com/linkerd/linkerd-examples/master/k8s-daemonset/k8s/world-v2.yml
```

STEP 2: USE PER REQUEST OVERRIDES IN OUR AD-HOC STAGING ENVIRONMENT

So now that we have a running world-v2, let's test it by running a request through our production topology, except that instead of hitting `world-v1`, we'll hit `world-v2`. First, let's run an unmodified request through our default topology (you may have to wait for l5d's external IP to appear):

```
$ INGRESS_LB=$(kubectl get svc l5d -o jsonpath="{.status.loadBalancer.ingress[0].*}")
```

```
$ curl -H "Host: www.hello.world" $INGRESS_LB
Hello (10.196.2.232) world (10.196.2.233)!!
```

As we expect, this returns `Hello (.....) World (.....)` from our production topology.

Now, how do we get to the staging environment? All we have to do is pass the following dtab override and requests through the prod topology will go to `world-v2`! A dtab override is another dtab entry that we pass using headers in the request. Since later dtab rules are applied first, this rule will replace (override) our current "/host/world => /srv/world-v1" rule with a rule to send requests with `/host/world` to `/srv/world-v2` instead.

```
$ curl -H "Host: www.hello.world" -H "l5d-dtab: /host/world => /srv/
world-v2;" $INGRESS_LB
Hello (10.196.2.232) earth (10.196.2.234)!!
```

We now see "earth" instead of "world"! The request is successfully served from the world-v2 service wired up to our existing production topology, with no code changes or additional deploys. Success! Staging is now fun and easy.

Dtabs and routing in Linkerd are well documented. During development, you can also make use of Linkerd's "dtab playground" at `http://$INGRESS_LB:9990/delegator`. By going to the "outgoing" router and testing a request name like `/http/1.1/GET/world`, you can see Linkerd's routing policy in action.

IN PRACTICE

In practice, there are some caveats to using this approach. First, the issue of writes to production databases must be addressed. The same dtab override mechanism can be used to send any writes to a staging database, or, with some application-level intelligence, to /dev/null. It is recommended that these rules are not created by hand so as to avoid expensive mistakes with production data!

Secondly, you application needs to forward Linkerd's context headers for this to work.

Lastly, it's important to ensure that the `l5d-dtab` header is not settable from the outside world! In our post about setting up a dogfood environment in Kubernetes, we gave an example nginx configuration for ingress that would strip unknown headers from the outside world—good practice for a variety of reasons.

CONCLUSION

We've demonstrated how to create ad-hoc staging environments with Linkerd by setting per-request routing rules. With this approach, we can stage services in the context of production environment, without modifying existing code, provisioning extra resources for our staging environment (other than for the staging instance itself), or maintaining parallel environments for production and staging. For microservices with complex application topologies, this approach can provide an easy, low-cost way to staging services before pushing to production.

CHAPTER 6

Distributed tracing made easy

Linkerd's role as a service mesh makes it a great source of data around system performance and runtime behavior. This is especially true in polyglot or heterogeneous environments, where instrumenting each language or framework can be quite difficult. Rather than instrumenting each of your apps directly, the service mesh can provide a uniform, standard layer of application tracing and metrics data, which can be collected by systems like Zipkin and Prometheus.

In this chapter we'll walk through a simple example how Linkerd and Zipkin can work together in Kubernetes to automatically get distributed traces, with only minor changes to the application.

In chapter 1, we showed you how you can use Linkerd to capture top-line service metrics. Service metrics are vital for determining the health of individual services, but they don't capture the way that multiple services work (or don't work!) together to serve requests. To see a bigger picture of system-level performance, we need to turn to distributed tracing.

In a recent Buoyant blog post, we covered some of the benefits of distributed tracing, and how to configure Linkerd to export tracing data to Zipkin. In this chapter, we'll show you how to run this setup entirely in Kubernetes, including Zipkin itself, and how to derive meaningful data from traces that are exported by Linkerd.

Before we start looking at traces, we'll need to deploy Linkerd and Zipkin to Kubernetes, along with some sample apps. The linkerd-examples repo provides all of the configuration files that we'll need to get tracing working end-to-end in Kubernetes. We'll walk you through the steps below.

STEP 1: INSTALL ZIPKIN

We'll start by installing Zipkin, which will be used to collect and display tracing data. In this example, for convenience, we'll use Zipkin's in-memory store. (If you plan to run Zipkin in production, you'll want to switch to using one of its persistent backends.)

To install Zipkin in the default Kubernetes namespace, run:

```
kubectl apply -f https://raw.githubusercontent.com/linkerd/linkerd-examples/master/k8s-daemonset/k8s/zipkin.yml
```

You can confirm that installation was successful by viewing Zipkin's web UI:

```
ZIPKIN_LB=$(kubectl get svc zipkin -o jsonpath="{.status.loadBalancer.ingress[0].*}")
open http://$ZIPKIN_LB # on OS X
```

Note that it may take a few minutes for the ingress IP to become available. (Also note that if you're running on Minikube, you need to run a different set of commands to load the web UI.)

But the web UI won't show any traces until we install Linkerd.

STEP 2: INSTALL THE SERVICE MESH

Next we'll install the Linkerd service mesh, configured to write tracing data to Zipkin. To install Linkerd as a DaemonSet (i.e., one instance per host) in the default Kubernetes namespace, run:

```
kubectl apply -f https://raw.githubusercontent.com/linkerd/linkerd-examples/master/k8s-daemonset/k8s/linkerd-zipkin.yml
```

This installed Linkerd as a service mesh, exporting tracing data with Linkerd's Zipkin telemeter. The relevant config snippet is:

```
telemetry:
- kind: io.l5d.zipkin
  host: zipkin-collector.default.svc.cluster.local

 port: 9410
  sampleRate: 1.0
```

Here we're telling Linkerd to send tracing data to the Zipkin service that we deployed in the previous step, on port 9410. The configuration also specifies a sample rate, which determines the number of requests that are traced. In this example we're tracing all requests, but in a production setting you may want to set the rate to be much lower (the default is 0.001, or 0.1% of all requests).

You can confirm the installation was successful by viewing Linkerd's admin UI (note, again, that it may take a few minutes for the ingress IP to become available, depending on the vagaries of your cloud provider):

```
L5D_INGRESS_LB=$(kubectl get svc l5d -o
jsonpath="{.status.loadBalancer.ingress[0].*}")
open http://$L5D_INGRESS_LB:9990 # on OS X
```

STEP 3: INSTALL THE SAMPLE APPS

Now we'll install the "hello" and "world" apps in the default namespace, by running:

```
kubectl apply -f https://raw.githubusercontent.com/linkerd/linkerd-
examples/master/k8s-daemonset/k8s/hello-world.yml
```

Congrats! At this point, we have a functioning service mesh with distributed tracing enabled, and an application that makes use of it.

Let's see the entire setup in action by sending traffic through Linkerd's outgoing router running on port 4140:

```
http_proxy=http://$L5D_INGRESS_LB:4140 curl -s http://hello
Hello () world ()!
```

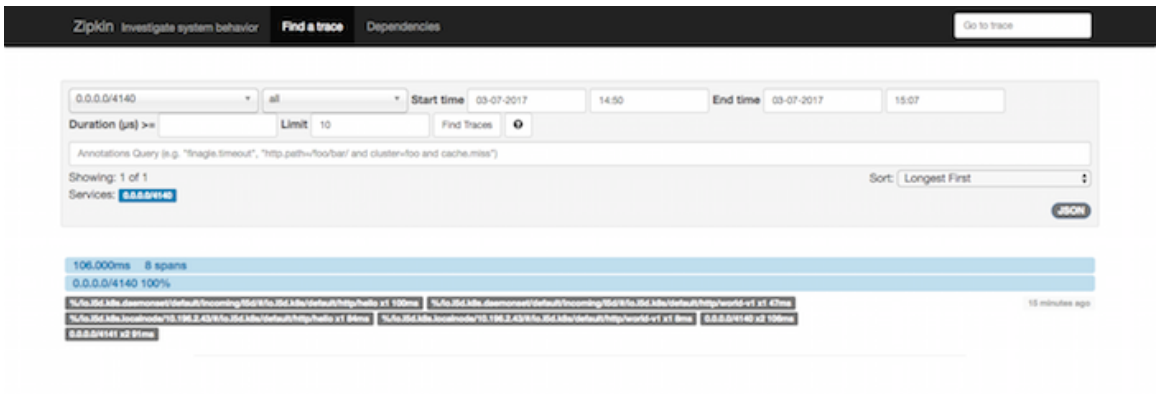
If everything is working, you'll see a "Hello world" message similar to that above, with the IPs of the pods that served the request.

STEP 4: ENJOY THE VIEW

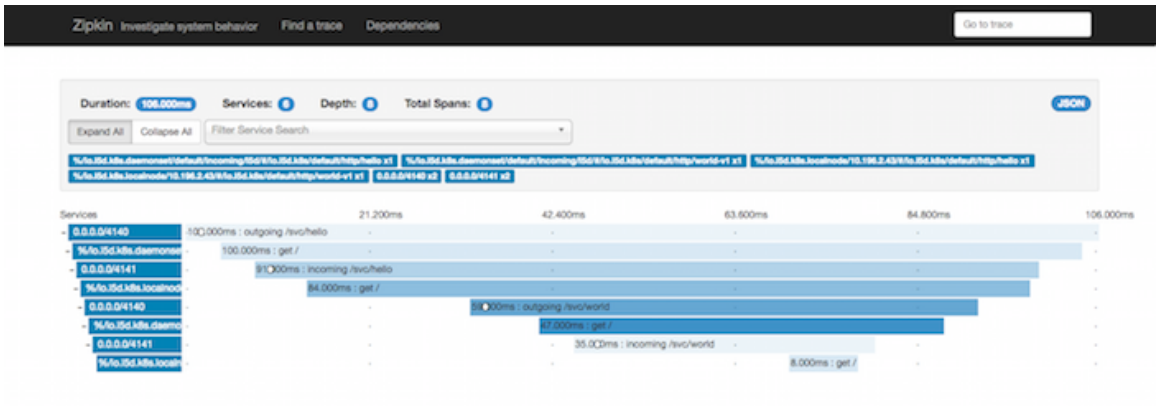
Now it's time to see some traces. Let's start by looking at the trace that was emitted by the test request that we sent in the previous section. Zipkin's UI allows you to search by "span" name, and in our case, we're interested in spans that originated with the Linkerd router running on 0.0.0.0:4140, which is where we sent our initial request. We can search for that span as follows:

```
open http://$ZIPKIN_LB/?serviceName=0.0.0.0%2F4140 # on OS X
```

That should surface 1 trace with 8 spans, and the search results should look like this:



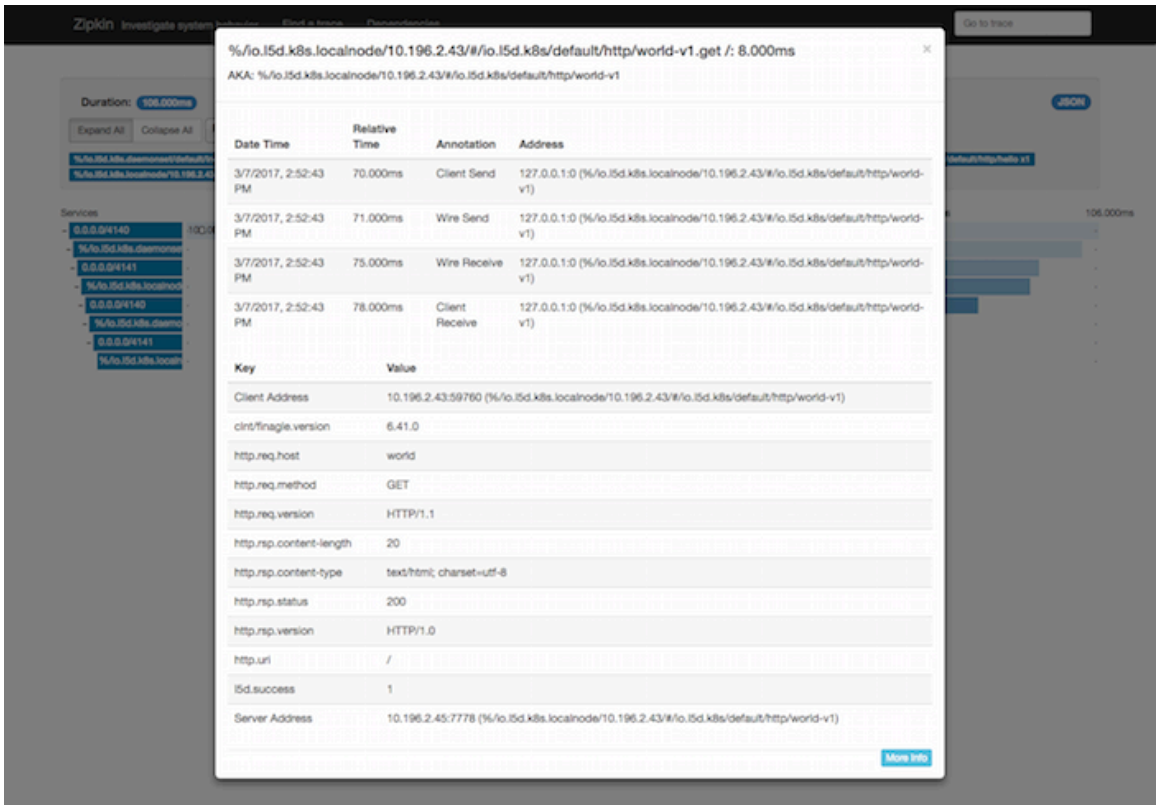
Clicking on the trace from this view will bring up the trace detail view:



From this view, you can see the timing information for all 8 spans that Linkerd emitted for this trace. The fact that there are 8 spans for a request between 2 services stems from the service mesh configuration, in which each request passes through two Linkerd instances (so that the protocol can be upgraded or

downgraded, or TLS can be added and removed across node boundaries). Each Linkerd router emits both a server span and a client span, for a total of 8 spans.

Clicking on a span will bring up additional details for that span. For instance, the last span in the trace above represents how long it took the world service to respond to a request—8 milliseconds. If you click on that span, you'll see the span detail view:



This view has a lot more information about the span. At the top of the page, you'll see timing information that indicates when Linkerd sent the request to the service, and when it received a response. You'll also see a number of key-value pairs with additional information about the request, such as the request URI, the response status code, and the address of the server that served the request. All of this information is populated by Linkerd automatically, and can be very useful in tracking down performance bottlenecks and failures.

A NOTE ABOUT REQUEST CONTEXT

In order for distributed traces to be properly disentangled, we need a little help from the application. Specifically, we need services to forward Linkerd's "context headers" (anything that starts with `l5d-ctx-`) from incoming requests to outgoing requests. Without these headers, it's impossible to align outgoing requests with incoming requests through a service. (The hello and world services provided above do this by default.)

There are some additional benefits to forwarding context headers, beyond tracing. From our previous blog post on the topic:

Forwarding request context for Linkerd comes with far more benefits than just tracing, too. For instance, adding the `l5d-dtab` header to an inbound request will add a dtab override to the request context. Provided you propagate request context, dtab overrides can be used to apply per-request routing overrides at any point in your stack, which is especially useful for staging ad-hoc services within the context of a production application. In the future, request context will be used to propagate overall *latency budgets*, which will make handling requests within distributed systems much more performant.

Finally, the `L5d-sample` header can be used to adjust the tracing sample rate on a per-request basis. To guarantee that a request will be traced, set `L5d-sample: 1.0`. If you're sending a barrage of requests in a loadtest that you don't want flooding your tracing system, consider setting it to something much lower than the steady-state sample rate defined in your Linkerd config.

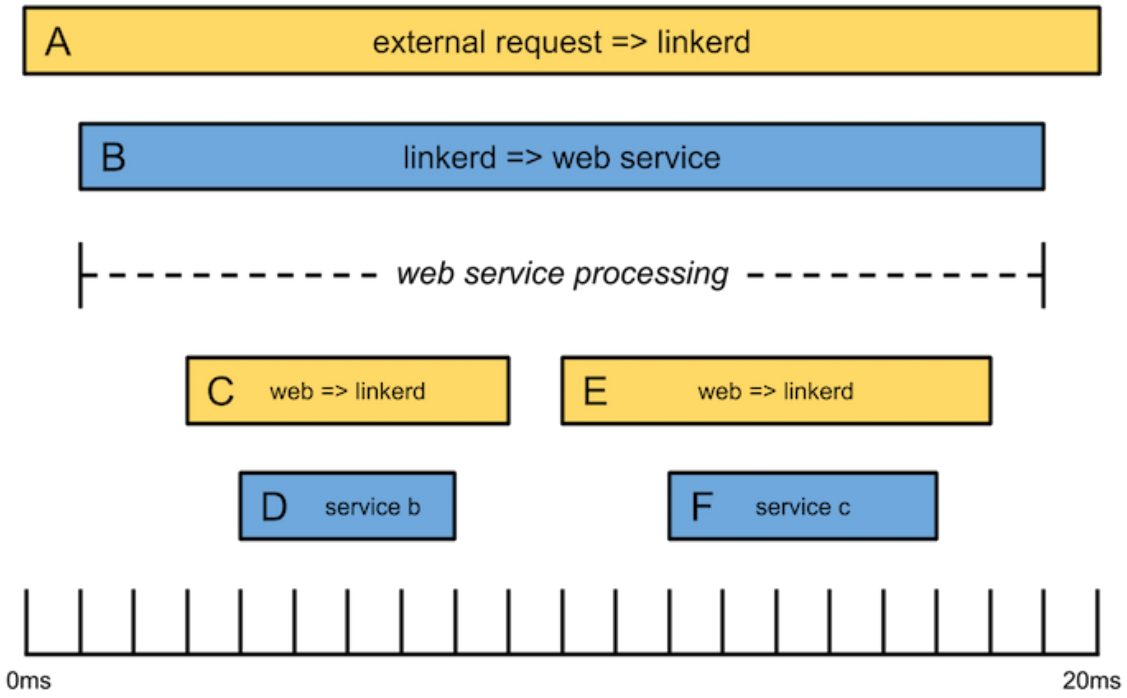
CONCLUSION

We've demonstrated how to run Zipkin in Kubernetes, and how to configure your Linkerd service mesh to automatically export tracing data to Zipkin. Distributed tracing is a powerful tool that is readily available to you if you're already using Linkerd. Check out Linkerd's Zipkin telemeter configuration reference, and find us in the Linkerd Slack if you run into any issues setting it up.

APPENDIX: UNDERSTANDING TRACES

In distributed tracing, a trace is a collection of spans that form a tree structure. Each span has a start timestamp and an end timestamp, as well as additional

metadata about what occurred in that interval. The first span in a trace is called the root span. All other spans have a parent ID reference that refers to the root span or one of its descendants. There are two types of spans: server and client. In Linkerd's context, server spans are created when a Linkerd router receives a request from an upstream client. Client spans are created when Linkerd sends that request to a downstream server. Thus the parent of a client span is always a server span. In the process of routing a multi-service request, Linkerd will emit multiple client and server spans, which are displayed as a single trace in the Zipkin UI. For instance, consider the following trace:



In this example, an external request is routed by Linkerd to the "Web" service, which then calls "Service B" and "Service C" sequentially (via Linkerd) before returning a response. The trace has 6 spans, and a total duration of 20 milliseconds. The 3 yellow spans are *server spans*, and the 3 blue spans are *client spans*. The *root span* is Span A, which represents the time from when Linkerd initially received the external request until it returned the response. Span A has one child, Span B, which represents the amount of time that it took for the Web service to respond to Linkerd's forwarded request. Likewise Span D represents the amount of time that it took for Service B to respond to the request from the Web service. For more information about tracing, read our previous blog post, Distributed Tracing for Polyglot Microservices.

Chapter 7:

Linkerd as an ingress controller

Linkerd is designed to make service-to-service communication internal to an application safe, fast and reliable. However, those same goals are also applicable at the edge. In this chapter, we'll demonstrate a new feature of Linkerd which allows it to act as a Kubernetes ingress controller, and show how it can handle ingress traffic both with and without TLS.

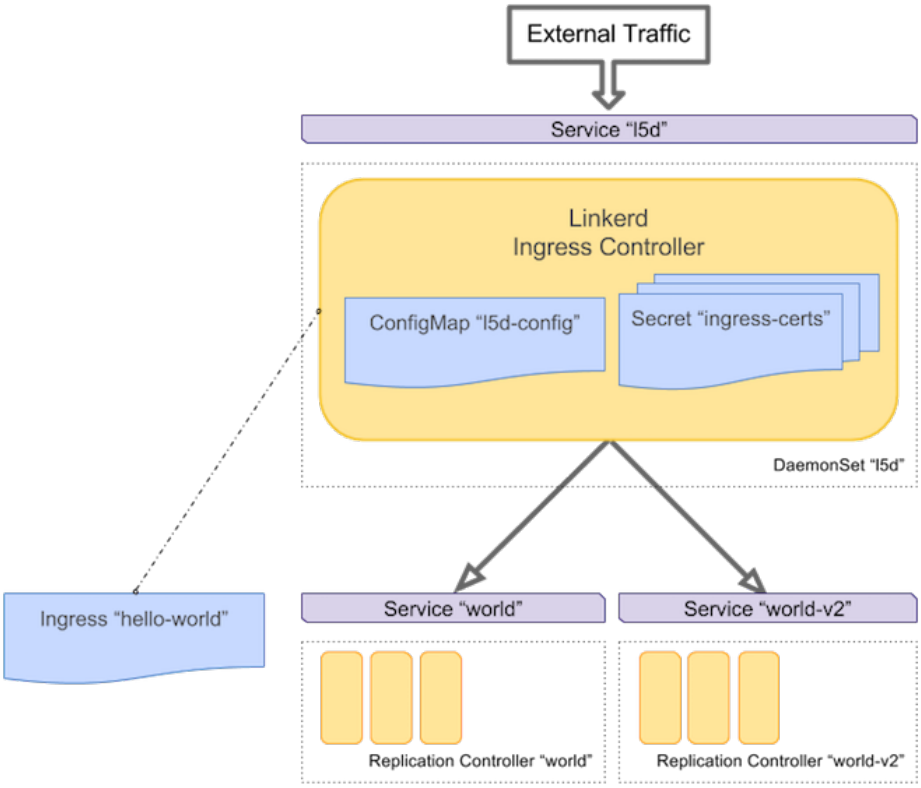
In previous chapters, we explored how to receive external requests by deploying Linkerd as a Kubernetes DaemonSet and routing traffic through the corresponding Service VIP. In this post, we'll simplify this setup by using Linkerd as a Kubernetes ingress controller, taking advantage of features introduced in Linkerd 0.9.1.

This approach has the benefits of simplicity and a tight integration with the Kubernetes API. However, for more complex requirements like on-demand TLS cert generation, SNI, or routing based on cookie values (e.g. the employee dogfooding approach discussed in previous chapters), combining Linkerd with a dedicated edge layer such as Nginx may still be necessary.

What is a Kubernetes ingress controller? An ingress controller is an edge router that accepts traffic from the outside world and forwards it to services in your Kubernetes cluster. The ingress controller uses HTTP host and path routing rules defined in Kubernetes' ingress resources.

INGRESS HELLO WORLD

Using a Kubernetes config from the linkerd-examples repo, we can launch Linkerd as a dedicated ingress controller. The config follows the same pattern as our previous posts on k8s daemonsets: it deploys an `l5d-config` ConfigMap, an `l5d` DaemonSet, and an `l5d` Service.



STEP 1: DEPLOY LINKERD

First let's deploy Linkerd. You can of course deploy into the default namespace, but here we've put Linkerd in its own namespace for better separation of concerns:

```
$ kubectl create ns l5d-system $ kubectl apply -f https://
raw.githubusercontent.com/linkerd/linkerd-examples/master/k8s-
daemonset/k8s/linkerd-ingress-controller.yml -n l5d-system
```

You can verify that the Linkerd pods are up by running:

```
$ kubectl get po -n l5d-system
NAME         READY     STATUS    RESTARTS   AGE
l5d-0w0f4    2/2       Running   0          5s
l5d-3cmfp    2/2       Running   0          5s
l5d-dj1sm    2/2       Running   0          5s
```

And take a look at the admin dashboard (This command assumes your cluster supports LoadBalancer services, and remember that it may take a few minutes for the ingress LB to become available.):

```
$ L5D_SVC_IP=$(kubectl get svc l5d -n l5d-system -o
jsonpath="{.status.loadBalancer.ingress[0].*}") $ open http://
$L5D_SVC_IP:9990 # on OS X
```

Let's take a closer look at the ConfigMap we just deployed. It stores the `config.yaml` file that Linkerd mounts on startup.

```
$ kubectl get cm l5d-config -n l5d-system -o yaml
apiVersion: v1
data:
  config.yaml: |-
    namers:
    - kind: io.l5d.k8s

 routers:
  - protocol: http
    identifier:
      kind: io.l5d.ingress
    servers:
      - port: 80
        ip: 0.0.0.0
        clearContext: true
    dtab: /svc => /#/io.l5d.k8s

 usage:
   orgId: linkerd-examples-ingress
```

You can see that this config defines an HTTP router on port 80 that identifies incoming requests using ingress resources (via the `io.l5d.ingress` identifier).

The resulting namespace, port, and service name are then passed to the Kubernetes namer for resolution. We've also set `clearContext` to `true` in order to remove any incoming Linkerd context headers from untrusted sources.

STEP 2: DEPLOY THE HELLO WORLD APPLICATION

Now it's time to deploy our application, so that our ingress controller can route traffic to us. We'll deploy a simple app consisting of a hello and a world service.

```
$ kubectl apply -f https://raw.githubusercontent.com/linkerd/linkerd-
examples/master/k8s-daemonset/k8s/hello-world.yml
$ kubectl apply -f https://raw.githubusercontent.com/linkerd/linkerd-
examples/master/k8s-daemonset/k8s/world-v2.yml
```

You can again verify that the pods are up and running:

```
$ kubectl get po
NAME              READY     STATUS    RESTARTS   AGE
hello-0v0vx       1/1       Running   0          5s
hello-84wfp       1/1       Running   0          5s
hello-mrcfr       1/1       Running   0          5s
world-v1-105tl    1/1       Running   0          5s
world-v1-1t6jc    1/1       Running   0          5s
world-v1-htwsw    1/1       Running   0          5s
world-v2-5tl10    1/1       Running   0          5s
world-v2-6jc1t    1/1       Running   0          5s
world-v2-wswht    1/1       Running   0          5s
```

At this point, if you try to send an ingress request, you'll see something like:

```
$ curl $L5D_SVC_IP
Unknown destination: Request("GET /", from /184.23.234.210:58081) / no
ingress rule matches
```

STEP 3: CREATE THE INGRESS RESOURCE

In order for our Linkerd ingress controller to function properly, we need to create an ingress resource that uses it.

```
$ kubectl apply -f https://raw.githubusercontent.com/linkerd/linkerd-
examples/master/k8s-daemonset/k8s/hello-world-ingress.yml
```

Verify the resource:

```
$ kubectl get ingress
NAME          HOSTS       ADDRESS   PORTS     AGE
hello-world   world.v2              80        7s
```

This `hello-world` ingress resource references our backends (we're only using `world-v1` and `world-v2` for this demo):

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: hello-world
  annotations:
    kubernetes.io/ingress.class: "linkerd"
spec:
  backend:
    serviceName: world-v1
    servicePort: http
rules:
- host: world.v2
  http:
    paths:
    - backend:
        serviceName: world-v2
        servicePort: http
```

The resource:

- Specifies `world-v1` as the default backend to route to if a request does not match any of the rules defined.
- Specifies a rule where all requests with the host header `world.v2` will be routed to the `world-v2` service.
- Sets the `kubernetes.io/ingress.class` annotation to "linkerd". Note, this annotation is only required if there are multiple ingress controllers running in the cluster. GCE runs one by default; you may choose to disable it by following these instructions.

That's it. You can exercise these rules by curling the IP assigned to the l5d service loadbalancer.

```
$ curl $L5D_SVC_IP
```

```
world (10.0.4.7)!
$ curl -H "Host: world.v2" $L5D_SVC_IP
earth (10.0.1.5)!
```

While this example starts with totally new instances, it's just as easy to add an ingress identifier router to a pre-existing linked setup. Also, although we employ a DaemonSet here (to be consistent with the rest of the Service Mesh for Kubernetes series), utilizing a Kubernetes Deployment for a Linkerd ingress controller works just as well. Using Deployments is left as an exercise for the reader. :)

INGRESS WITH TLS

Linkerd already supports TLS for clients and servers within the cluster. Setting up TLS is described in much more detail in chapter 3. In this ingress controller configuration, Linkerd expects certs to be defined in a Kubernetes secret named `ingress-certs` and to follow the format described as part of the ingress user guide. Note that there's no need to specify a TLS section as part of the ingress resource: Linkerd doesn't implement that section of the resource. All TLS configuration happens as part of the `l5d-config` ConfigMap.

The Linkerd config remains largely unchanged, save updating the server port to `443` and adding TLS file paths:

```
...
servers:
- port: 443
  ip: 0.0.0.0
  clearContext: true
  tls:
    certPath: /io.buoyant/linkerd/certs/tls.crt
    keyPath: /io.buoyant/linkerd/certs/tls.key
...
```

The l5d DaemonSet now mounts a secret volume with the expected name: `ingress-certs`

```
spec:
  volumes:
  - name: certificates
    secret:
      secretName: ingress-certs
  ...
  containers:
  - name: l5d
```

```
    ...
    ports:
    - name: tls
      containerPort: 443
      hostPort: 443
    ...
    volumeMounts:
    - name: "certificates"
      mountPath: "/io.buoyant/linkerd/certs"
      readOnly: true
    ...
```

And the updated Service config exposes port `443`.

A reminder that the certificates we're using here are for testing purposes only. Create the Secret, delete the DaemonSet and ConfigMap, and re-apply the ingress controller config:

```
$ kubectl apply -f https://raw.githubusercontent.com/linkerd/linkerd-
examples/master/k8s-daemonset/k8s/ingress-certificates.yml -n l5d-
system $ kubectl delete ds/l5d configmap/l5d-config -n l5d-system
$ kubectl apply -f https://raw.githubusercontent.com/linkerd/linkerd-
examples/master/k8s-daemonset/k8s/linkerd-tls-ingress-controller.yml -
n l5d-system
```

You should now be able to make an encrypted request:

```
# Example requires this development cert: https://
raw.githubusercontent.com/linkerd/linkerd-examples/master/k8s-
daemonset/certificates/cert.pem
# The cert expects "hello.world" host, so we add an /etc/hosts entry,
eg:
# 104.198.196.230 hello.world
# where "104.198.196.230" is the ip stored in $L5D_SVC_IP
$ curl --cacert cert.pem -H "Host: world.v2" https://hello.world
$ earth (10.0.1.5)!
```

CONCLUSION

Linkerd provides a ton of benefits as an edge router. In addition to the dynamic routing and TLS termination described in this chapter, it also pools connections, load balances dynamically, enables circuit breaking, and supports distributed tracing. Using the Linkerd ingress controller and the Kubernetes configuration referenced in this chapter, you gain access to all these features in an easy to use, Kubernetes-native approach. Best of all, this method works seamlessly with the rest of the service mesh, allowing for operation, visibility, and high availability in virtually any cloud architecture.

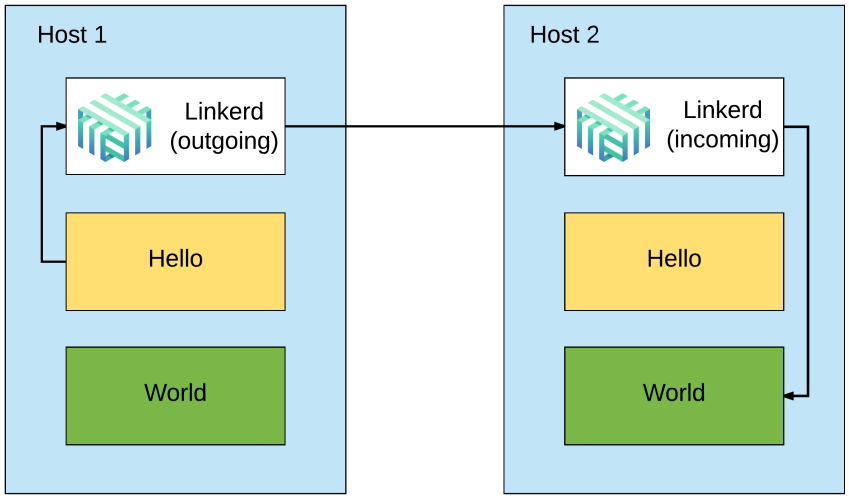
CHAPTER 8

gRPC for fun and profit

As of Linkerd 0.8.5, released in early 2017, Linkerd supports gRPC and HTTP/2! These powerful protocols can provide significant benefits to applications that make use of them. In this post, we'll demonstrate how to use Linkerd with gRPC, allowing applications that speak gRPC to take full advantage of Linkerd's load balancing, service discovery, circuit breaking, and distributed tracing logic.

In this chapter, we'll use our familiar `hello world` microservice app and configs, which can be found in the `linkerd-examples` repo (k8s configs here and `hello world` code here).

The `hello world` application consists of two components—a `hello` service which calls a `world` service to complete a request. `hello` and `world` use gRPC to talk to each other. We'll deploy Linkerd as a DaemonSet (so one Linkerd instance per host), and a request from `hello` to `world` will look like this:



DaemonSet deployment model: one Linkerd per host.

As shown above, when the `hello` service wants to call `world`, the request goes through the *outgoing* router of its host-local Linkerd, which does not send the request directly to the destination `world` service, but to a Linkerd instance running on the same host as `world` (on its *incoming* router). That Linkerd instance then sends the request to the `world` service on its host. This three-hop model allows Linkerd to

decouple the application's protocol from the transport protocol—for example, by wrapping cross-node connections in TLS. (For more on this deployment topology, see Chapter 2, Pods are great, until they are not.)

TRYING THIS AT HOME

Let's see this setup in action! Deploy `hello`, `world`, and Linkerd to the default k8s namespace:

```
kubectl apply -f https://raw.githubusercontent.com/BuoyantIO/linkerd-
examples/master/k8s-daemonset/k8s/hello-world-grpc.yml
kubectl apply -f https://raw.githubusercontent.com/BuoyantIO/linkerd-
examples/master/k8s-daemonset/k8s/linkerd-grpc.yml
```

Once Kubernetes provisions an external LoadBalancer IP for Linkerd, we can do some test requests! Note that the examples in these blog posts assume k8s is running on GKE (e.g. external loadbalancer IPs are available, no CNI plugins are being used). Slight modifications may be needed for other environments—see our Flavors of Kubernetes wiki page for environments like Minikube or CNI configurations with Calico/Weave.

We'll use the helloworld-client provided by the `hello world` docker image in order to send test gRPC requests to our `hello world` service:

```
$ L5D_INGRESS_LB=$(kubectl get svc l5d -o
jsonpath="{.status.loadBalancer.ingress[0].*}")
$ docker run --rm --entrypoint=helloworld-client buoyantio/helloworld:
0.1.3 $L5D_INGRESS_LB:4140 Hello (10.196.1.242) world (10.196.1.243)!!
```

It works!

We can check out the Linkerd admin dashboard by doing:

```
$ open http://$L5D_INGRESS_LB:9990 # on OSX
```

And that's it! We now have gRPC services talking to each other, with their HTTP/2 requests being routed through Linkerd. Now we can use all of Linkerd's awesome

features, including per-request routing, load balancing, circuit-breaking, retries, TLS, distributed tracing, service discovery integration and more, in our gRPC microservice applications!

HOW DID WE CONFIGURE LINKERD FOR GRPC OVER HTTP/2?

Let's take a step back and examine our config. What's different about using gRPC rather than HTTP/1.1? Actually, not very much! If you compare our Linkerd config for routing gRPC with the config for plain old HTTP/1.1, they're quite similar (full documentation on configuring an HTTP/2 router can be found here).

The changes you'll notice are:

PROTOCOL

We've changed the router `protocol` from `http` to `h2` (naturally!) and set the `experimental` flag to `true` to opt in to experimental HTTP/2 support.

```
routers:
- protocol: h2
experimental: true
```

IDENTIFIER

We use the header path identifier to assign a logical name based on the gRPC request. gRPC clients set HTTP/2's `:path` pseudo-header to `/package.Service/Method`. The header path identifier uses this pseudo-header to assign a logical name to the request (such as `/svc/helloworld.Hello/Greeting`). Setting `segments` to 1 means we only take the first segment of the path, in other words, dropping the gRPC `Method`. The resulting name can then be transformed via a dtab where we extract the gRPC service name, and route the request to a Kubernetes service of the same name. For more on how Linkerd routes requests, see our routing docs.

```
identifier:
  kind: io.l5d.header.path
```

```
  segments: 1
```

DTAB

We've adjusted the dtab slightly, now that we're routing on the `/serviceName` prefix from the header path identifier. The dtab below transforms the logical name assigned by the path identifier (`/svc/helloworld.Hello`) to a name that tells the io.l5d.k8s namer to query the API for the `grpc` port of the `hello` Service in the default namespace (`/#/io.l5d.k8s/default/grpc/Hello`).

The domainToPathPfx namer is used to extract the service name from the package-qualified gRPC service name, as seen in the dentry `/svc => /$/io.buoyant.http.domainToPathPfx/grpc`.

Delegation to `world` is similar, however we've decided to version the `world` service, so we've added the additional rule `/grpc/World => /srv/world-v1` to send requests to world-v1.

Our full dtab is now:

```
/srv         => /#/io.l5d.k8s/default/grpc;
/grpc        => /srv;
/grpc/World => /srv/world-v1;
/svc         => /$/io.buoyant.http.domainToPathPfx/grpc;
```

CONCLUSION

In this chapter, we've seen how to use Linkerd as a service mesh for gRPC requests, adding latency-aware load balancing, circuit breaking, and request-level routing to gRPC apps. Linkerd and gRPC are a great combination, especially as gRPC's HTTP/2 underpinnings provide it with powerful mechanisms like multiplexed streaming, back pressure, and cancelation, which Linkerd can take full advantage of. Because gRPC includes routing information in the request, it's a natural fit for Linkerd, and makes it very easy to set up Linkerd to route gRPC requests. For more on Linkerd's roadmap around gRPC, see our blog post on the topic.

Finally, for a more advanced example of configuring gRPC services, take a look at our Gob microservice app. In that example, we additionally deploy Namerd, which we use to manage our routing rules centrally, and update routing rules without redeploying Linkerd. This lets us to do things like canarying and blue green deploys between different versions of a service.

For more information on Linkerd, gRPC, and HTTP/2 head to the Linkerd gRPC documentation as well as the config documentation for HTTP/2.

Conclusion

The service mesh is a critical component of the cloud native stack. It is a dedicated infrastructure layer for making service-to-service communication safe, fast, and reliable. Linkerd provides a robust platform for building a service mesh in Kubernetes. This ebook provides several hands-on tutorials describing the steps to get up and running with Linkerd and Kubernetes.

Linkerd is part of the Cloud Native Computing Foundation and has a thriving community of contributors and users. Adopters range from startups like Monzo, which is disrupting the UK banking industry, to high scale Internet companies like PayPal, Ticketmaster, and Credit Karma, to established companies like Houghton Mifflin Harcourt that have been in business for hundreds of years. By following the tutorials in this ebook, you can join this active and rapidly-growing community!

THE FUTURE OF THE SERVICE MESH

While service mesh adoption in the cloud native ecosystem is growing rapidly, there is an extensive and exciting roadmap ahead still to be explored. The requirements for serverless computing (e.g. Amazon's Lambda) fit directly into the service mesh's model of naming and linking, and form a natural extension of its role in the cloud native ecosystem. The roles of service identity and access policy are still very nascent in cloud native environments, and the service mesh is well poised to play a fundamental part of the story here. Finally, the service mesh, like TCP/IP before it, will continue to be pushed further into the underlying infrastructure. Just as Linkerd evolved from systems like Finagle, the current incarnation of the service mesh as a separate, user-space proxy that must be explicitly added to a cloud native stack will also continue to evolve.

We hope you've found this ebook useful. We're extremely excited about the future of Linkerd, and of Kubernetes, and we hope you join us in our quest to make cloud native applications safe, fast, ansd resilient!

About Buoyant

Buoyant, founded by ex-Twitter infrastructure engineers William Morgan and Oliver Gould, gives enterprises the tools to operate their cloud native applications safely, reliably, and effectively. Buoyant builds practical tools to identify, mitigate, and prevent production incidents in microservices and cloud native apps—all without affecting the pace of product development.

Buoyant is creator of Linkerd, the open source *service mesh* for cloud-native applications. Created in 2016, Linkerd brings scalable, production-tested reliability to cloud native applications in the form of a service mesh, a dedicated infrastructure layer for service communication that adds resilience, visibility and safety without requiring complex application integration. In January 2017, Linkerd was voted into the the Cloud Native Computing Foundation as its fifth project, alongside Kubernetes, Prometheus, OpenTracing and Fluentd.

Buoyant