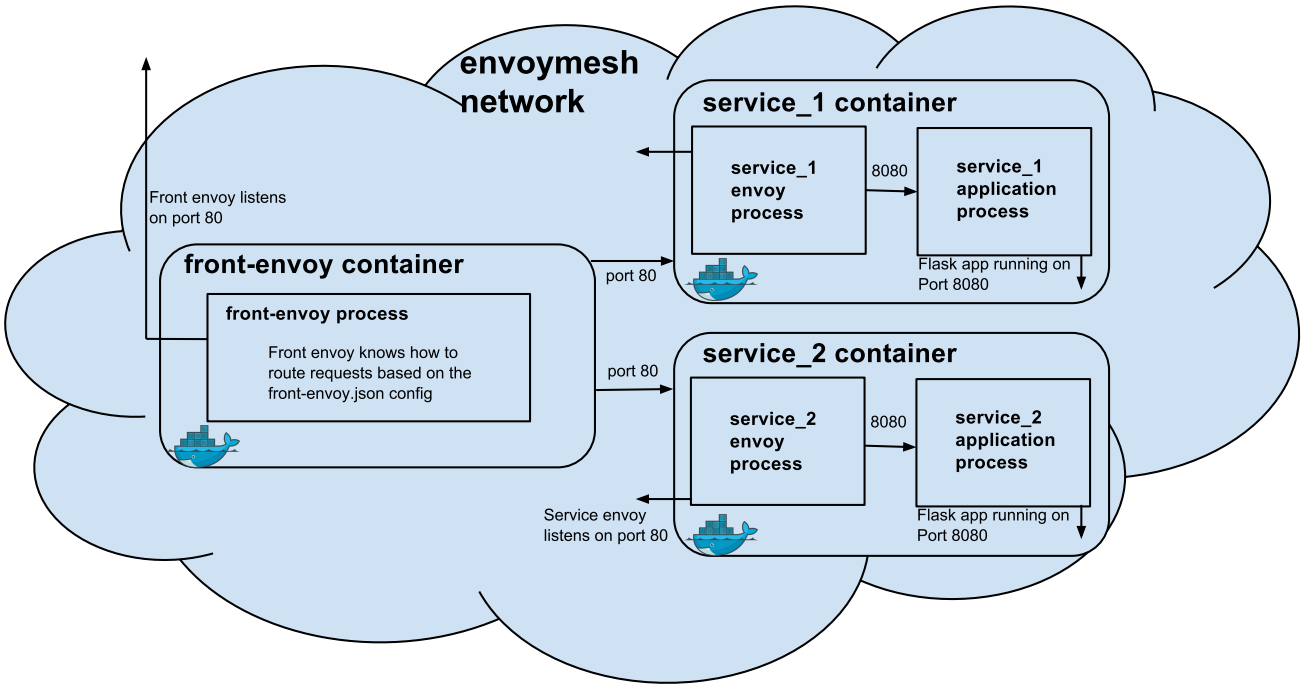
Front Proxy

To get a flavor of what Envoy has to offer as a front proxy, we are releasing a docker compose sandbox that deploys a front envoy and a couple of services (simple flask apps) colocated with a running service envoy. The three containers will be deployed inside a virtual network called `envoymesh`.

Below you can see a graphic showing the docker compose deployment:



All incoming requests are routed via the front envoy, which is acting as a reverse proxy sitting on the edge of the `envoymesh` network. Port `80` is mapped to port `8000` by docker compose (see /examples/front-proxy/docker-compose.yml). Moreover, notice that all traffic routed by the front envoy to the service containers is actually routed to the service envoys (routes setup in /examples/front-proxy/front-envoy.yaml). In turn the service envoys route the request to the flask app via the loopback address (routes setup in /examples/front-proxy/service-envoy.yaml). This setup illustrates the advantage of running service envoys collocated with your services: all requests are handled by the service envoy, and efficiently routed to your services.

Running the Sandbox

The following documentation runs through the setup of an envoy cluster organized as is described in the image above.

Step 1: Install Docker

Ensure that you have a recent versions of `docker, docker-compose` and `docker-machine` installed.

A simple way to achieve this is via the Docker Toolbox.

Step 2: Docker Machine setup

First let's create a new machine which will hold the containers:

```
$ docker-machine create --driver virtualbox default
$ eval $(docker-machine env default)
```

Step 4: Clone the Envoy repo, and start all of our containers

If you have not cloned the envoy repo, clone it with `git clone git@github.com:envoyproxy/envoy` or `git clone https://github.com/envoyproxy/envoy.git`:

```
$ pwd
envoy/examples/front-proxy
$ docker-compose up --build -d
$ docker-compose ps
        Name                       Command               State      Ports
-------------------------------------------------------------------------------------------------
--------------
example_service1_1      /bin/sh -c /usr/local/bin/ ... Up       80/tcp
example_service2_1      /bin/sh -c /usr/local/bin/ ... Up       80/tcp
example_front-envoy_1   /bin/sh -c /usr/local/bin/ ... Up       0.0.0.0:8000->80/tcp,
0.0.0.0:8001->8001/tcp
```

Step 5: Test Envoy's routing capabilities

You can now send a request to both services via the front-envoy.

For service1:

```
$ curl -v $(docker-machine ip default):8000/service/1
*   Trying 192.168.99.100...
* Connected to 192.168.99.100 (192.168.99.100) port 8000 (#0)
> GET /service/1 HTTP/1.1
> Host: 192.168.99.100:8000
> User-Agent: curl/7.43.0
> Accept: */*
>
< HTTP/1.1 200 OK
< content-type: text/html; charset=utf-8
< content-length: 89
< x-envoy-upstream-service-time: 1
< server: envoy
< date: Fri, 26 Aug 2016 19:39:19 GMT
< x-envoy-protocol-version: HTTP/1.1
<
Hello from behind Envoy (service 1)! hostname: f26027f1ce28 resolvedhostname: 172.19.0.6
* Connection #0 to host 192.168.99.100 left intact
```

For service2:

```
$ curl -v $(docker-machine ip default):8000/service/2
*   Trying 192.168.99.100...
* Connected to 192.168.99.100 (192.168.99.100) port 8000 (#0)
> GET /service/2 HTTP/1.1
> Host: 192.168.99.100:8000
> User-Agent: curl/7.43.0
> Accept: */*
>
< HTTP/1.1 200 OK
< content-type: text/html; charset=utf-8
< content-length: 89
< x-envoy-upstream-service-time: 2
< server: envoy
< date: Fri, 26 Aug 2016 19:39:23 GMT
< x-envoy-protocol-version: HTTP/1.1
<
Hello from behind Envoy (service 2)! hostname: 92f4a3737bbc resolvedhostname: 172.19.0.2
* Connection #0 to host 192.168.99.100 left intact
```

Notice that each request, while sent to the front envoy, was correctly routed to the respective application.

Step 6: Test Envoy's load balancing capabilities

Now let's scale up our service1 nodes to demonstrate the clustering abilities of envoy.:

```
$ docker-compose scale service1=3
Creating and starting example_service1_2 ... done
Creating and starting example_service1_3 ... done
```

Now if we send a request to service1 multiple times, the front envoy will load balance the requests by doing a round robin of the three service1 machines:

```
$ curl -v $(docker-machine ip default):8000/service/1
*   Trying 192.168.99.100...
* Connected to 192.168.99.100 (192.168.99.100) port 8000 (#0)
> GET /service/1 HTTP/1.1
> Host: 192.168.99.100:8000
> User-Agent: curl/7.43.0
> Accept: */*
>
< HTTP/1.1 200 OK
< content-type: text/html; charset=utf-8
< content-length: 89
< x-envoy-upstream-service-time: 1
< server: envoy
< date: Fri, 26 Aug 2016 19:40:21 GMT
< x-envoy-protocol-version: HTTP/1.1
<
Hello from behind Envoy (service 1)! hostname: 85ac151715c6 resolvedhostname: 172.19.0.3
* Connection #0 to host 192.168.99.100 left intact
$ curl -v $(docker-machine ip default):8000/service/1
*   Trying 192.168.99.100...
* Connected to 192.168.99.100 (192.168.99.100) port 8000 (#0)
> GET /service/1 HTTP/1.1
> Host: 192.168.99.100:8000
> User-Agent: curl/7.43.0
> Accept: */*
>
< HTTP/1.1 200 OK
< content-type: text/html; charset=utf-8
< content-length: 89
< x-envoy-upstream-service-time: 1
< server: envoy
< date: Fri, 26 Aug 2016 19:40:22 GMT
< x-envoy-protocol-version: HTTP/1.1
<
Hello from behind Envoy (service 1)! hostname: 20da22cfc955 resolvedhostname: 172.19.0.5
* Connection #0 to host 192.168.99.100 left intact
$ curl -v $(docker-machine ip default):8000/service/1
*   Trying 192.168.99.100...
* Connected to 192.168.99.100 (192.168.99.100) port 8000 (#0)
> GET /service/1 HTTP/1.1
> Host: 192.168.99.100:8000
> User-Agent: curl/7.43.0
> Accept: */*
>
< HTTP/1.1 200 OK
< content-type: text/html; charset=utf-8
< content-length: 89
< x-envoy-upstream-service-time: 1
< server: envoy
< date: Fri, 26 Aug 2016 19:40:24 GMT
< x-envoy-protocol-version: HTTP/1.1
<
Hello from behind Envoy (service 1)! hostname: f26027f1ce28 resolvedhostname: 172.19.0.6
* Connection #0 to host 192.168.99.100 left intact
```

Step 7: enter containers and curl services

In addition of using `curl` from your host machine, you can also enter the containers themselves and `curl` from inside them. To enter a container you can use `docker-compose exec <container_name> /bin/bash`. For example we can enter the `front-envoy` container, and `curl` for services locally:

```
$ docker-compose exec front-envoy /bin/bash
root@81288499f9d7:/# curl localhost:80/service/1
Hello from behind Envoy (service 1)! hostname: 85ac151715c6 resolvedhostname: 172.19.0.3
root@81288499f9d7:/# curl localhost:80/service/1
Hello from behind Envoy (service 1)! hostname: 20da22cfc955 resolvedhostname: 172.19.0.5
root@81288499f9d7:/# curl localhost:80/service/1
Hello from behind Envoy (service 1)! hostname: f26027f1ce28 resolvedhostname: 172.19.0.6
root@81288499f9d7:/# curl localhost:80/service/2
Hello from behind Envoy (service 2)! hostname: 92f4a3737bbc resolvedhostname: 172.19.0.2
```

Step 8: enter containers and curl admin

When envoy runs it also attaches an `admin` to your desired port. In the example configs the admin is bound to port `8001`. We can `curl` it to gain useful information. For example you can `curl` `/server_info` to get information about the envoy version you are running. Additionally you can `curl` `/stats` to get statistics. For example inside `frontenvoy` we can get:

```
$ docker-compose exec front-envoy /bin/bash
root@e654c2c83277:/# curl localhost:8001/server_info
envoy 10e00b/RELEASE live 142 142 0
root@e654c2c83277:/# curl localhost:8001/stats
cluster.service1.external.upstream_rq_200: 7
...
cluster.service1.membership_change: 2
cluster.service1.membership_total: 3
...
cluster.service1.upstream_cx_http2_total: 3
...
cluster.service1.upstream_rq_total: 7
...
cluster.service2.external.upstream_rq_200: 2
...
cluster.service2.membership_change: 1
cluster.service2.membership_total: 1
...
cluster.service2.upstream_cx_http2_total: 1
...
cluster.service2.upstream_rq_total: 2
...
```

Notice that we can get the number of members of upstream clusters, number of requests fulfilled by them, information about http ingress, and a plethora of other useful stats.