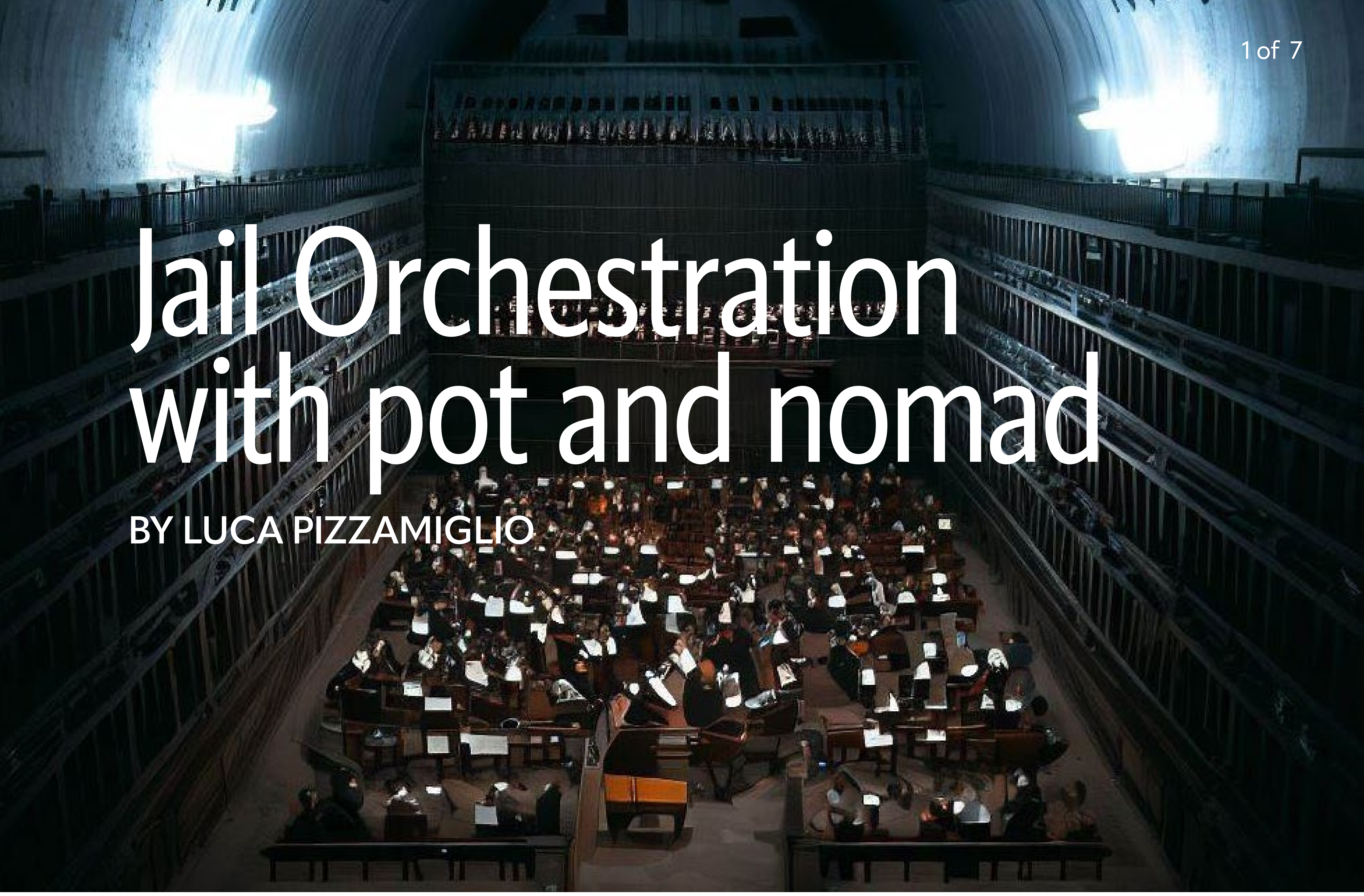


Jail Orchestration with pot and nomad

BY LUCA PIZZAMIGLIO

Containers are a great tool to distribute horizontally scalable applications on many servers. When the number of applications and their cardinality grows, the number of containers can easily become hard to manage manually.

Container orchestrators are applications that aim to simplify the management of a large fleet of containers, hiding the complexity and improving reliability, especially in an environment made very dynamic by autoscaling and continuous deployment. In this article, we will talk about a setup based on FreeBSD, using pot, a jail framework that supports jail images, and nomad, a container agnostic orchestrator developed by HashiCorp.

The System Architecture

To explain how an orchestrator works, we need to introduce a few services and explain their role.

The nomad client

A nomad client is a server that receives orders from the orchestrator to execute containers. Nomad clients can also be referred to as nodes, like in kubernetes.

In large installations, the majority of servers are nomad clients, as they are responsible to execute users' applications. In the cloud native jargon, nomad clients form the data plane of the cluster.

Nomad clients can support multiple container drivers: some drivers are operating system agnostic, while other drivers, like docker or pot, are available only on specific operating systems.

To orchestrate FreeBSD jails using pot, we need nomad clients based on FreeBSD.

The nomad server

The nomad servers are the machines that implement the orchestrator. The nomad server is responsible for keeping the state of the cluster and for scheduling containers to nomad clients. Multiple instances (3 to 5) are required to provide redundancy and to share the load. In the cloud native jargon, nomad servers form the control plane of the cluster.

Users are going to interact with nomad servers to deploy applications to the cluster. The nomad servers are responsible to keep the state of the cluster healthy, and to reschedule containers in case of client failures.

Nomad servers can run on any supported operating system. To orchestrate jails, at least one nomad client has to be based on FreeBSD.

The container registry

The orchestrator (nomad servers) is the "brain" of the cluster and it assigns containers to clients that support the container types. This means that:

- any nomad clients can be selected to run supported containers,
- clients don't know in advance which containers are going to host,
- a client can execute multiple instances of the same containers.

Every client needs a service to download the image of the container they are assigned to execute. This requirement is fulfilled by the container registry, a service that provides the images of containers.

When the orchestrator selects a client to execute a container, that client is going to download the container image from a registry and then start the container.

In our examples, we are going to use potluck, a public container registry maintained by the pot community, that builds images from an open source catalog of image recipes. However, as container images contain binaries, we strongly encourage everyone to have their own local registry, for security reasons.

In pot, images are files that are downloaded via `fetch(1)`, so a registry can be just a simple web server.

Users are going to interact with nomad servers to deploy applications to the cluster.

The service catalog

A service catalog is a list of services enriched by additional information like all the container addresses that implement those services.

When an orchestrator schedules a container implementing a service, it's also going to register the container address to the list of containers implementing that service.

The service catalog can be also configured to periodically check the health status of the service for all containers, so the list of container addresses only contains healthy addresses.

The service catalog that we are going to use is based on consul, a service mesh application also developed by HashiCorp.

The ingress (optional)

Because of the dynamic nature of the orchestrator, it can be very hard to know where our services are running. Every time a new deployment occurs, containers are scheduled to potentially different nodes on different ports. With ingress, we define a proxy/load balancer that is configured to provide a fixed entry point to our services.

For instance, we can configure the proxy in a way that the URL path (i.e., https://example.com/foo) provides information about the targeted service (i.e. redirecting to the containers implementing the service foo) Another common way is to use the host header.

Ingress proxies dynamically maintain the list of valid container's addresses, by continuously interacting with the service catalog.

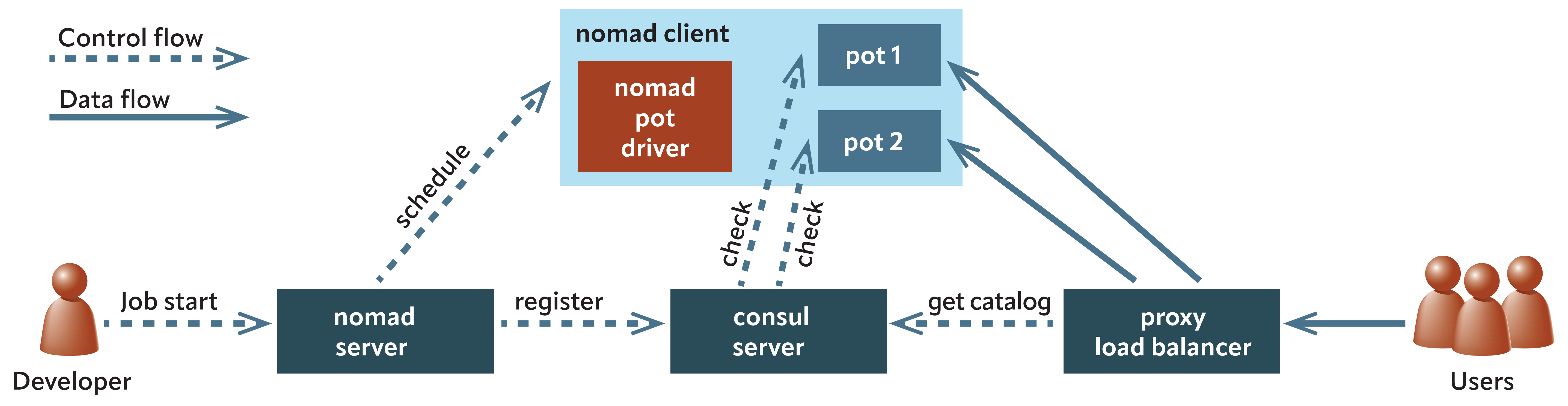
For our examples, we are using traefik, an ingress proxy implemented by traefix lab.

Nomad-pot-driver

Nomad has been built to support several container technologies and different operating systems. In fact, a nomad package is available and HashiCorp also provides binary blobs for FreeBSD.

Nomad has a plugin architecture that allows it to extend it to support new container technologies. Esteban Barrios wrote and opensourced the nomad-pot-driver plugin. This plugin works as an interface between a nomad client and pot, to provide the features needed to orchestrate jails.

The orchestrator schedules workloads to the client that uses the plugin to interface with pot.



Overview of the architecture and the foles of the different services.

Minipot

Minipot is a package that installs and configures all the aforementioned services on a single FreeBSD machine and is the reference installation that we are going to use to show our examples.

Minipot is a configuration useful for testing, but not for professional installation, as it's going to concentrate all service on one machine only, reducing a cluster to a single node that does it all.

In particular, it is going to install and configure consul, traefik, and nomad. Nomad is going to run as client and as server, playing the dual role of orchestrator and executor.

There is a detailed guide on how to install minipot on this blog post on the Klara website.

Schedule a Job

Once minipot is initialized and all services are running, we can use the following job description file to start a job in nomad

```
job "nginx-minipot" {
  datacenters = ["minipot"]
  type        = "service"
4    group "group1" {
5      count = 1
6      network {
7        port "http" {}
8      }

9      task "www1" {
10       driver = "pot"

11       service {
12         tags = ["nginx", "www"]
13         name = "hello-web"
14         port = "http"

15         check {
16           type      = "tcp"
17           name      = "tcp"
18           interval  = "5s"
19           timeout   = "2s"
20         }
21       }

22       config {
23         image = "https://potluck.honeyguide.net/nginx-nomad"
24         pot = "nginx-nomad-amd64-13_1"
25         tag = "1.1.13"
26         command = "nginx"
27         args = ["-g","’daemon off;’"]

28         port_map = {
29           http = "80"
30         }
31       }

32       resources {
33         cpu = 200
34         memory = 64
35       }
36     }
37   }
38 }
```

The job stanza describes all the details nomad needs to schedule the job.

The job "nginx-minipot" (1) has one group named "group1" (4), that has one task called "www1" (9).

The task "www1" is a pot container (10), the image registry is potluck (23), the pot image is nginx (24) and the version is 1.1.13 (25). By specifying that the task is based on the pot driver, the orchestrator is going to schedule this job on a client with pot support. In our example, server and client are the same machine.

Compared to the common use of jails, where the bootstrap happens using rc script, we are going to run nginx directly (26), without any other additional services. The args parameter (27) is important to allow nomad to properly follow the container lifecycle and to capture its logs. Pot is going to take care of initializing the network and everything that is needed.

The port_map stanza (28) and the network stanza (6) are saying that nginx is going to listen to port 80 in the jail, but the nomad client will use a different port ("http"), dynamically assigned by the nomad server.

The service stanza (11) provides the information that nomad is going to use to register the service to consul. In our example, the task "www1" implements the service "hello-web" (11) and it will be registered to consul, using the port "http" (14), assigned by the nomad server. For the IP address, the nomad server is going to use the nomad client IP address, defined during the scheduling operation.

In our example, the job is also configuring a tcp health check that consul is going to run every 5 seconds to determine the health status of the instance.

This job description needs to be stored in a file (i.e., nginx.job) and any user can launch the service via

```
$ nomad job nginx.job
```

NOTE: The first deployment can take some time as the client needs to download the image. On slow connections, the first deployment can even fail because of deployment timeouts. Once the fetch is completed, it is safe to re-run the deployment that will be executed in few seconds.

Check on nomad

Once the job is scheduled, it is possible to check the status of the deployment via command line:

```
$ nomad jobs allocs nginx-minipot
ID        Node ID   Task Group  Version  Desired  Status   Created     Modified
636d3241  c375b833  group1      3        run      running  28m43s ago  28m27s ago
$ nomad alloc status 636d3241
[...]
Allocation Addresses:
Label  Dynamic  Address
*http  yes      2003:f1:c709:de00:faac:65ff:fe86:9458:22854
[...]
$ curl "[2003:f1:c709:de00:faac:65ff:fe86:9458]:22854"
```

"Allocation" is the name used by nomad to identify a container, an instance of the task.

The IPv6 address is the nomad client one.

The port 22854 is the port nomad chose to direct the nomad client to the port 80 of the container.

To see the port redirection setup, we can use the command:

```
$ sudo pot show
```

An alternative of the CLI, the nomad server is configured to also provide a powerful web UI reachable at `localhost:4646`

From there, we can see the "nginx-minipot" job and navigate to see all information about the cluster, allocations, clients, and so on.

From nomad, we can see directly the status of all containers.

By reaching the allocation page, we can click on the "exec" button to run a /bin/sh shell into the running containers.

Check on consul

Via CLI, we can see the list of services in the consul catalog (consul catalog services), but not the details. However, we can check the status of the "hello-web" service, by reaching the web UI interface at

```
localhost:8500
```

From here, we can navigate to check the status of the service "hello-web" and its check "tcp".

Check on traefik

The proxy traefix is configured to route traffic on port 8080, but provides a web-ui to monitor the status at port 9200 (`localhost:9200`). Traefik is also configured to sync the service catalog from consul.

By selecting the http services, we can see our "hello-web" service (marked as `hello-web@consulcatalog`).

By clicking on the service, we can see the service details and the routing options.

The configuration is based on the host header, in our case "hello-web.minipot".

Now we can reach the service hello-web via the ingress:

```
$ curl -H Host:hello-web.minipot http://127.0.0.1:8080
```

Alternatively, we can add the entry

```
127.0.0.1 hello-web.minipot
```

to `/etc/hosts` and then use the hostname directly:

```
$ curl http://hello-web.minipot:8080
```

We will obtain the same output as we had by point curl directly to the jail.

Horizontal scaling

To see the orchestrator in action, we can now simply change the count from 1 to 2 in the job file (row 5) and re-submit the job via

```
$ nomad run nginx.job
```

Once scheduled, the running nomad allocations are now 2, the service "hello-web" in consul has two instances, as the servers in traefik.

- To verify the round-robin distribution of the ingress, we can
- Tail the logs of one container (`$ nomad alloc logs -f allocation1`)
- Tail the logs of the other container (`$ nomad alloc logs -f allocation2`)
- Execute curl on the ingress (`$ curl -H Host:hello-web.minipot http://127.0.0.1:8080`)

At every execution of curl, the proxy distributes the requests between the containers as is possible to see from their logs.

Tear down everything

To stop our examples, we suggest this tear down operation:

- Stop the nomad jobs (`$ nomad stop nginx-minipot`)
- Stop traefik (`$ sudo service traefik stop`)
- Stop nomad (`$ sudo service nomad stop`)
- Stop consul (`$ sudo service consul stop`)

From playground to production

Minipot is a single node installation useful as a playground, to learn or to test things locally.

A production environment should be shaped in a different way:

- 3 or 5 distinct consul servers
- 3 or 5 distinct nomad servers
- 2 ingress proxy servers (HA configuration)

Several nomad client servers (depends on the expected workload and reliability requirements like the overprovision factor)

It is worth mentioning that the aforementioned setup can mix different operating systems: the only servers that have to run FreeBSD are the nomad clients targeting jail/pot workloads.

Nomad or consul servers can run on Linux or Solaris, allowing you to reuse pieces of infrastructure that may already be available to you.

As ingress proxy, we used traefik, which natively syncs with consul. However, it's possible to use other services, like nginx or ha-proxy, together with consul-template, to achieve the same result. In such a configuration, consul-template is responsible for detecting changes in consul,to render the proxy configuration template, and to notify the proxy of the new configuration.

Additionally, the configuration of all services would need improvements, like adding authentication to nomad.

Minipot is a single node installation useful as a playground, to learn or to test things locally.

Credits

I want to highlight the community effort needed to reach this point, from Esteban Barrios, the first developer of the nomad-pot-driver, to Michael Gmelin (grembo@), who really helped a lot to increase the reliability and the stability of this solutions. I also want to mention Stephan Lichtenauer and Bretton Vine, who worked on Potluck, the public image registry, and on numerous other projects like the Ansible playbook and blog posts dedicated to pot and nomad use cases.

LUCA PIZZAMIGLIO is a port committer in the FreeBSD project and a member of the port manager team (portmgr). In 2017, he started the pot project, with the goal of understanding how containers on FreeBSD could look.