Terraform Infrastructure with Service Discovery

Adrian Lund-Lange	*473115*	adrialu@stud.ntnu.no
Vetle T. Moen	*997821*	vetletm@stud.ntnu.no

Abstract

In the cloud computing times we live in today, one might use a multitude of cloud providers. Getting the hang of - and combining - each of their tools is a pain. Terraform lets you use a single language to define and control the orchestration of your infrastructure across a plethora of providers. Once the infrastructure is up and running, which at this point will be dynamic, it introduces new problems; accessing the deployed services without knowing their endpoints beforehand. This is where Consul comes in, adding a layer of service discovery on top of it, allowing the applications deployed in the infrastructure to be registered in the cluster as a service by name. The end result is a fully dynamic and scalable infrastructure where all hosts are reachable with DNS.

November 4, 2018

Contents

1 Introduction **3**

2 Background technology **4**

2.1 Terraform 4

2.2 Ansible 4

2.3 Terraform Inventory 4

2.4 Consul 4

2.5 Fabio 4

3 Survey of similar projects **5**

4 Implementation **6**

4.1 Orchestration 6

4.2 Provisioning 9

4.3 Service Discovery 12

5 Security aspects **15**

5.1 SSH 15

5.2 State files 15

5.3 Secrets 15

5.4 Internal security 15

6 Conclusions **16**

References **16**

A Source Code **18**

1. Introduction

We've become accustomed to OpenStack as a cloud provider during our time at NTNU, but there's *many* others we've yet to try, both public and private. Each of these providers have their own tools and APIs for orchestrating and provisioning, and when you want to mix and match between them you'll quickly run into problems such as compatibility issues or lack of resources such as time just for researching how to use said tools and APIs. When you've finally got your dynamic infrastructure set up and ready to be provisioned with services, you quickly run into another problem; accessing the services. This is typically done with IP addresses, all of which are hard-coded and can often be fluid (DHCP), so that database you were able to reach yesterday has been moved and is no longer reachable today.

These are the problems we wanted to solve as we went exploring the wonderful world of *Infrastructure as Code* and exposing ourselves to tools and services not necessarily covered by the curriculum. The tools we wanted to explore were those that solve these problems, to which the fine people over at *HashiCorp* has possible solutions; *Terraform* for orchestrating an infrastructure, and *Consul* for Service Discovery.

These two tools are not without their shortcomings:

- Terraform creates the infrastructure, but does not handle provisioning (installing applications and configuring them) nicely[1], so how can we better provision our hosts?
- Since we're not provisioning any hosts with Terraform, how can we implement and show off Consul?
- And how can we make this to be *scalable* and *flexible* with regards to service diversity and a dynamic "fleet" of hosts?

These are the questions that lie behind our choice to use the following additional tools:

- Ansible, for provisioning the infrastructure created with Terraform
- Terraform Inventory, to generate a dynamic Ansible inventory from Terraform
- Fabio, to act as a reverse proxy for registered web services in Consul

The symbiosis of these tools work beautifully together, as the remainder of this document will illustrate.

2. Background technology

As our project title suggests, we are aiming to showcase Terraform with service discovery. In order to accomplish this goal we had to research, learn, and implement the following technologies:

2.1. Terraform

Terraform is a infrastructure orchestration tool, which allows for defining one or more infrastructures as code, using one common language. This eliminates the need for in-house tooling such as OpenStack's Heat, AWS' CloudFormation and Google Cloud's Deployment Manager. Since the configuration is written as code, every single bit can be version controlled, shared among team members, and reviewed. One could even add automation into the mix with Continuous Deployment, eliminating the user interaction with the tooling altogether.

2.2. Ansible

Ansible is a provisioning tool that follows a push model. As opposed to Puppet which follows a pull model, Ansible uses SSH to provision hosts, from a single point, making sure they reach a desired state. The hosts themselves do not need to have an agent running and do not need to check their state against a central server at regular intervals. The structure is based on an inventory where a host are grouped together, like with the Puppet + Foreman combination, and it allows groups to have several roles. A role defines one or more tasks to be executed, handling the promise of a desired state.

2.3. Terraform Inventory

Terraform Inventory is a simple utility that is able to extract the hosts from a Terraform deployment state and output a grouped inventory that Ansible is able to use for provisioning. While this tool works fine out of the box, we had to do minor alterations for it to work with our infrastructure, which was easy enough as it's just JSON.

2.4. Consul

Consul is a tool used to created a distributed service mesh, allowing hosts and services to be connected, secured, and configured across clouds and platforms. We'll use Consul for its Service Discovery and DNS features, creating a dynamic, self-discovering, healthy cluster of hosts and services, with high availability as a focus.

2.5. Fabio

Fabio is a HTTP reverse proxy that is tightly integrated with Consul, which also features load balancing. While traditional load balancers and reverse proxies require a pre-defined configuration to operate, Fabio retrieves its "configuration" from Consul, by the tags on each registered service. In addition, Fabio updates itself when Consul services change, so there's no need for extra management of it, just fire-and-forget.

3. Survey of similar projects

The intention of using Terraform is to move away from vendor-locked-in tools, such as those provided by OpenStack, AWS, Google Cloud, and others. Terraform handles orchestration across a plethora of cloud providers, and is regarded as the best the industry for what it does. Ansible also has orchestration support, but it's far from the maturity level of what Terraform offers.

Our project was mainly about orchestration and service discovery. Earlier lab assignments in the course has covered parts of these topics with OpenStack Heat, Puppet with Foreman and R10k, which gave us ideas about what to create. Due to the nature of what we intended to create, there is no "one solution fits all", and we had to try things out for ourselves, experimenting with the tools and utilizing their documentations to the fullest.

As far service discovery goes, there are a handful of options. *etcd* is one of them, but it has more of a focus on key/value storage than service discovery (from an initial glance), and Consul was another. The built-in service discovery of Docker Swarm is also a contender, but we wanted something we could have more control over, and being another HashiCorp product, we chose Consul early on.

Throughout the course we've been using Puppet for provisioning, and while it would get the job done, during our progress with Terraform we found out that setting it up right would require a lot of 3rd-party services, such as DNS, role management (like Foreman), and some way of keeping it version controlled (r10k pulling in intervals). So once we finished our first take on Terraform we explored for alternative solutions that would give us more time to spend on our original topic, while still learning something new. We ended up using Ansible, as we found it worked quite well when used in combination with Terraform and an additional tool to bridge the gap between them, provisioning based on known facts instead of defining groups at a later stage.

Lastly, to demonstrate it all working, we created a simple web application in Go that simply lists the hosts registered by Consul. Making sure we could *fully* demonstrate the capabilities for service discovery, we set up a reverse proxy called *Fabio* to load balance between all the hosts serving that web application in a round-robin fashion. Fabio was not our initial pick - that honor goes to Haproxy - but we ended up using Fabio since it integrates so tightly with Consul, further showing off Consul's strengths.

4. Implementation

We started off planning out what we wanted to implement, and to keep the infrastructure rather simple for demonstration purposes we decided on creating a few servers running Consul (to maintain quorum, of which it wants at least 3 for failure tolerances[2]), a few servers running a web application[3] (to demonstrate load balancing), a proxy server to load balance the web servers, and a manager for remote access to it all.

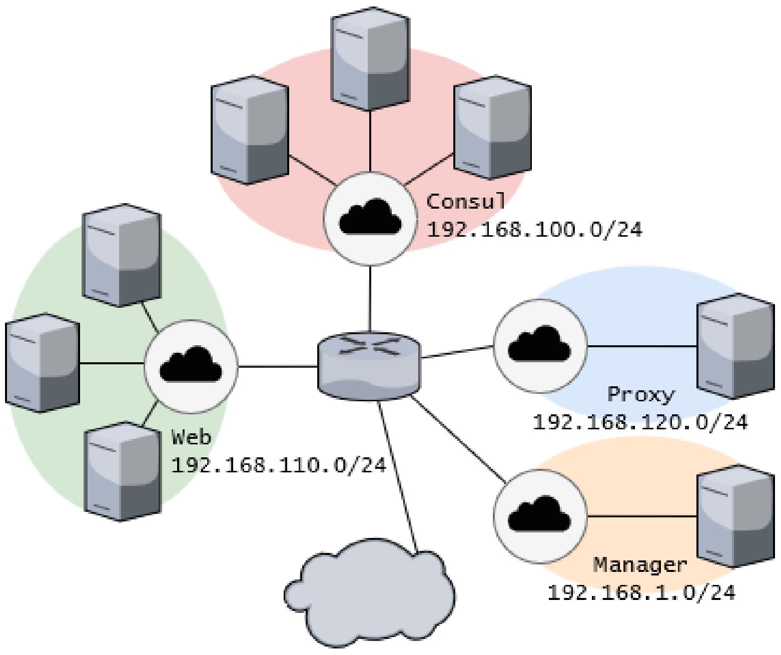

Figure 1: Network diagram

After deciding on the infrastructure we started reading up on the technologies we chose, which started with Terraform, before further moving on to provisioning, then lastly service discovery.

4.1. Orchestration

Terraform supports a plethora of providers, as of the time of writing, 90 providers are officially supported[4], and another ~80 supported by the community[5]. This ranges from anything like major cloud providers like AWS, Azure and Google Cloud, to Docker/Kubernetes, CloudFlare DNS, managing CVS organizations, monitoring systems, databases and more. Another nice thing with Terraform is that you can use multiple providers at the same time. Need to create a new AWS instance and update CloudFlare DNS entries to access it? Terraform can do this in a single definition, no problem!

With our plan to demonstrate service discovery using a web application, we opted to use our internal OpenStack solution at NTNU and only dip our toes into the world of Terraform, but there is much more to it than what we're exploring. Since OpenStack is not considered a "major" cloud provider it's kind of treated like a second-rate citizen by Terraform, which means that it's not covered by official guides like other providers might be. This increases the difficulty of getting started, but having an existing understanding of how a "stack" is built with OpenStack's own tools gave us the edge in adopting the Terraform way of creating definitions.

4.1.1. Workflow

Deploying a Terraform infrastructure has three steps:

- **Writing** definitions in the domain-specific language *HCL*
- **Previewing** changes to be applied
- **Applying** changes to provision the infrastructure

Aside from writing the definition files, everything is done with the single `terraform` command-line interface:

```
# write definition file
vim main.tf
# initialize terraform, downloading any plugins used
terraform init
# preview changes before applying them
terraform plan
# apply changes
terraform apply
# destroy the infrastructure
terraform destroy
```

Terraform tracks the currently provisioned infrastructure in a file named `terraform.tfstate`, which it will reference to when doing subsequent changes to the infrastructure. The "state file" is by default stored locally, so when working in teams or when Terraform is run in an automated fashion, storing the this remotely is encouraged, which is what the "backend" type is for:

```
terraform {
  backend "s3" {
    bucket = "mybucket"
    key    = "path/to/key"
    region = "eu-west-2"
  }
}
```

This will store, use, and update the state of the infrastructure in an AWS S3 bucket, which is one of the many options[6]. We'll come back to this file in chapter 5.2, but for our simple use-case we opted to forgo this feature and keep our state file stored locally.

4.1.2. Preparation

We had the option right off the bat to either use the OpenStack-provided environment files for authentication, which Terraform will attempt to use if it notices we're using OpenStack resources (we'll come back to this), but another option was to provide credentials right in the configuration like so:

```
provider "openstack" {
  user_name   = "admin"
  tenant_name = "admin"
  password    = "pwd"
  auth_url    = "http://myauthurl:5000/v2.0"
}
```

We opted to use the traditional environment way instead, that way we could keep our infrastructure versioned publicly without exposing our credentials. This also means that we don't have to define the `provider` block at all, Terraform is smart enough to realize what we want to do.

4.1.3. Definitions

Terraform defines most things as *resources*, and to create a new instance we define it like so:

```
resource "openstack_compute_instance_v2" "manager" {
  name        = "manager"
  image_name  = "${var.image_name}"
  flavor_name = "${var.flavor_name}"
  key_pair    = "${openstack_compute_keypair_v2.remote.name}"
  user_data   = "${data.template_cloudinit_config.manager.rendered}"

  security_groups = [
    "${openstack_compute_secgroup_v2.remote.name}",
  ]
```

```
    network = {
      uuid = "${openstack_networking_network_v2.manager.id}"
    }
  }
```

We're using a resource named "openstack_compute_instance_v2", and we name it "manager". A resource has options, of which this example uses a few to define typical parameters for an instance in OpenStack, such as the name, image, flavor and so on. Another important option not covered in this example is the "count" option, which lets us define how many replicas we want of the same resource. We've used this option extensively in our implementation.

Notice the use of variables here, `${var.image_name}` uses a variable defined elsewhere, `${openstack_networking_network_v2.manager.id}` uses data from another resource, defined like this:

```
resource "openstack_networking_network_v2" "manager" {
  name           = "manager-net"
  admin_state_up = "true"
}
```

In which there's a "name" field, which when referenced to is using that name. We could just write `uuid = "manager-net"`, but referencing instead of hard-coding is always the better option.

There's also *data* types, describing types of data that could be later used in a resource, such as Cloud-Init configurations, which we used to create and use SSH keys that each instance used to accept remote connections from our manager instance.

```
data "tls_public_key" "manager" {
  private_key_pem = "${file("id_rsa")}"
}

data "template_file" "manager_keys" {
  template = "${file("templates/add_keys.sh.tpl")}"

  vars {
    private_key = "${data.tls_public_key.manager.private_key_pem}"
    public_key  = "${data.tls_public_key.manager.public_key_openssh}"
  }
}

data "template_cloudinit_config" "manager" {
  part {
    filename     = "add_keys.sh"
    content_type = "text/x-shellscript"
    content      = "${data.template_file.manager_keys.rendered}"
  }
}
```

This is the data type referenced in the instance resource from above, the `user_data` option. In here we're defining a cloud-init shell script that will add the private key on the manager, used to connect to all the other hosts.

These are the patterns in which we describe our entire infrastructure, its instances, security groups, floating IPs, networks and subnets, even keypairs can be created on-demand for our internal network, all using the same domain-specific language of which most HashiCorp tools use; "HCL".

The definitions used to describe our infrastructure are mostly alike, a combination of resources and other types intertwined, with subtle differences to define what they do differently. We encourage the reader to browse the definitions in our repository (see Appendix A).

4.1.4. Extending Terraform

This is a part we didn't venture into, as we didn't have a particular need for it in our infrastructure, but Terraform has support for modules and plugins.

Modules are basically subsets of definitions, where its possible to group up resources that are closely connected. This resembles stacks in OpenStack, but are not as restrictive. We played around with this initially before we ended up forgoing it completely for a (slightly) less structured setup, as we found ourselves limited to the interactions between modules (you can't reference resources within a module as a variable outside of it).

Plugins are used to create new types of resources, such as providers or data types. The existing set of official and community-provided providers and provisioners were enough to create our infrastructure. We didn't delve into this part of Terraform, but we acknowledge its existence and the work put into the extensibility of Terraform.

4.2. Provisioning

Ansible is based on a pre-defined list of hosts called an "inventory", hosts gets assigned roles, and each role gets tasks to perform. Tasks can be anything from installing software, configuring it or making sure it's running, idempotently. Best practice suggests defining one role for one type of service (e.g. installing and configuring one piece of software and its dependencies), and have one base/common role which will install packages that would be needed on all machines. Roles are used in a "playbook", where hosts are assigned to roles based on pre-defined groups, facts gathered about them, or a combination of both. This lets us provision only database software to database hosts, or installing software based on the operating system they are running.

The first order of business was getting familiar with Ansible's directory structure, its definition language, and understanding how to navigate the documentation.

4.2.1. Directory structure

The playbook is located in the root directory, and roles in a sub-directory named `roles`. All files Ansible reads must be written in YAML, with the exception of the inventory. Roles are each in their own sub-directory of the `roles` directory, and within each role there's the following sub-directories:

- `tasks`: Defines a set of tasks to be executed, such as "Install package X", which will use a module to perform the task. A module could be something like "apt" which installs packages on Debian-based systems, or "service" which manages system services. You can also define conditions by which the task should run, e.g. only restart a service if the configuration was changed. Tasks must be defined in `tasks/main.yml`, but should you need further segregation, tasks in other relative files can be *imported*.

- `handlers`: If a task needs to trigger some other action based on a condition, then this can be defined as a "handler" in `handlers/main.yml`. Handlers are just tasks that gets triggered by *actual* tasks. As an example, we use it to restart our Consul service after copying over a configuration template, but only if the configuration was altered.

- `templates`: Holds templates files defined in the *Jinja2* templating language. Ansible can populate these files with variables before copying them to the destined host.

- `files`: This can contain shell scripts, binaries, or whatever else you want copied to the host.

- `vars`: If you need to specify role-wide variables, such as a version number for a package, you can define this here. Variables can also be defined per-task instead, and you can also override variables from the playbook.

- `defaults`: If a variable is not defined in the playbook, Ansible will use the defaults declared here to populate the variables defined in *vars/main.yml*.

- `meta`: Holds information about a role with which platforms it supports, versions, etc.

4.2.2. YAML

YAML stands for *YAML Ain't Markup Language* and is data definition language similar to JSON or XML, made especially to make JSON or XML more human-friendly. An example from our playbook:

```
- hosts: all
  roles:
    - common
    - dotfiles
    - netplan
    - consul
```

At the top we declare that "all" hosts will have these roles, and Ansible will make it so. YAML is easy to understand and work with, however it's strict with regards to indentation. One misplaced space would result in an invalid configuration, much like Python.

4.2.3. Ansible documentation

The documentation for Ansible is quite newcomer friendly in the sense that every page follows the exact same structure with synopsis, parameters, examples, etc. You can search for a specific version, the latest version, or if you're feeling brave; use the development version, which we ended up doing for certain tasks.

4.2.4. Workflow

A typical flow for creating a role would be to first make a plan for what you want it to do. For this we'll use our Consul roles as an example. In order to get the Consul "raft" (more on this in 4.3.1) up and running we need two things; the Consul application itself, and a service configuration.

First task is to download the application, move it to a location on the hosts' PATH, and set its permissions:

```
- name: Install from archive
  unarchive:
    src: https://releases.hashicorp.com/consul/{{ consul_version }}/consul_{{ consul_version }}_lin
    dest: /usr/local/bin
    mode: 0755
    owner: root
    group: root
    remote_src: yes
  vars:
    consul_version: 1.3.0
  become: yes
```

The "name" field is treated like a like a comment which will be printed out verbatim while the task is executing. The "unarchive" field describes a module which downloads the archived binary from HashiCorp's website, extracts and moves it to its destination on the host, and also makes it executable. We're using variables local to the task itself for the version, and we're using "become" to make sure we have administrative rights while executing the task.

Next we want consul to run as a service on our host system. For that we need a systemd service configuration, which we've defined in a template for the role:

```
[Unit]
Description=Consul Startup process
After=network.target

[Service]
Type=simple
ExecStart=/bin/bash -c '/usr/local/bin/consul agent -config-dir /etc/consul.d/'
TimeoutStartSec=0
Restart=always
RestartSec=3
```

```
[Install]
WantedBy=default.target
```

Now that we have that ready to go, we use a task to copy it to the host machine:

```
- name: Add service configuration
  copy:
    src: consul.service
    dest: /etc/systemd/system/consul.service
    owner: root
    group: root
    mode: 0644
  become: yes
  notify: reload_consul_service
```

Here we're using the "copy" module to copy the file to the host, again making sure it has the correct attributes and using "become" to execute the task with administrative rights. It also uses the "notify" option, which will trigger a *handler* if this task resulted in a changed state. The handler is defined in `roles/consul/handlers/main.yml` like so:

```
- name: reload_consul_service
  systemd:
    daemon_reload: yes
  become: yes
```

The only thing that distinguishes this from a task is that the "name" field is now used more like a function name instead of a comment, and must match the "notify" field's value. All this handler does it letting systemd know that a new service was either created or updated.

Consul now has a service and is ready to run, but has not been configured yet. Since we have both "server" and "client" configurations for Consul we ended up separating these into different roles, which we'll come back to in chapter 4.3. Assuming the configuration file exists in the template directory, we'll create the directory where it should reside, then copy it to the host:

```
- name: Create configuration directories
  file:
    path: /etc/consul.d
    state: directory
    mode: 0755
  become: yes

- name: Add server configuration
  template:
    src: consul.json.j2
    dest: /etc/consul.d/config.json
    owner: root
    group: root
    mode: 0644
  become: yes
  notify: restart_consul_service
```

The "template" module will do the same job as the "copy" module, except it also attempt to inject variables into the file before copying it, using the *Jinja2* templating language, hence the ".j2" suffix to the file name. Again, we're using "notify" to trigger a handler which will restart the service upon a change:

```
- name: restart_consul_service
  service:
    name: consul
    state: restarted
  become: yes
```

Lastly, we need to make sure the service is both started and enabled, so it can start on its own should an outage occur:

```
- name: Start service
  service:
    name: consul
    enabled: yes
    state: started
  become: yes
```

This should be everything needed to define a role that will create, configure and start a Consul service on any given host, being able to idempotently do so, and also be able to trigger service restarts when configurations gets changed, either from updates or drifts.

One thing to note here is when the handlers are executed. You'd think that the handler runs *immediately* after the task succeeded, but it actually only triggers when the entire *role* has finished. This lets us tell the service to restart upon configuration changes, even if it hasn't been enabled yet, since it will be by the end of the role.

4.2.5. Terraform Inventory

With a dynamic infrastructure, a static host inventory would be a pain to manage. This is why we went searching and quickly found an utility named *Terraform Inventory*, which is able to extract hosts' IPs and groupings from a Terraform state file and create inventories for Ansible dynamically. This (quite popular) tool had some minor flaws, which we handled by wrapping it in a Python script, but otherwise it's a great compliment to our infrastructure to make it as seamless and non-interactive as possible.

4.3. Service Discovery

One of the features Consul provides is *service discovery*; being able to query registered services (and hosts) by name instead of depending on hard-coded IPs, which is never going to work well in a dynamic infrastructure. Once configured, we'll be able to reach the "demo" service by querying `demo.service.consul`, or the web2 host (by hostname) with `web2.node.consul`. If there are more than one service with the same name, Consul will load balance between them, which is an nice additional feature.

As with the other parts of this project, we had to learn how to use and configure it, and how to register and manage services.

4.3.1. Service cluster

With Consul we have a few definitions to get a grip on:

- **Agent**: Consul runs as an agent on each host, and depending on its configuration, it will run as either a "server" or as a "client". We defined the agent as a system service in 4.2.4.
- **Server**: Agents in server-mode exist in a peer set and will form a "raft" with one or more servers. Less than 3 is highly discouraged due to failure tolerances. More than 5 results in increased cost with little to no benefit.
- **Raft**: When a set of Consul servers need to establish a leader they use the a protocol called "raft"[7]. The reason why we need a raft is to maintain consistency (based on the CAP theorem[8]) throughout our cluster, with a recommended minimum of 3 servers for high-availability[2].
- **Quorum**: This is a fancy word for majority, and a raft needs to maintain quorum in order to remain functional. If there are 5 servers in a Raft, we need at least 3 alive and healthy to keep our cluster consistent.
- **Client**: Agents in client-mode will run on every host machine that provides and registers a service within the cluster. We also use it for DNS purposes in our infrastructure.
- **Cluster**: This is not a term used much by Consul, but we'll use it to refer to all hosts running Consul, either as server or as client.

4.3.2. Configuration

As mentioned already, we used Ansible to install Consul and making sure it's configured correctly. The only piece we didn't mention was the Consul configuration, which we had to separate by the type of agent Consul would run as; "server" or "client".

For a Consul raft to initialize, it needs to be "bootstrapped"; to elect a leader among the servers. This can be done manually by "bootstrapping" one of the servers and making the others join the raft, then restart the first one without bootstrapping enabled. Later versions of Consul has seen some improvements to this; we bootstrap *all* of the servers and tell them to expect a certain amount of servers to complete the raft, then they all stop bootstrapping once the quorum is met. This is the "bootstrap_expect" option at play.

Here's the JSON configuration we used for our servers:

```
{
  "server": true,
  "datacenter": "skyhigh",
  "data_dir": "/var/consul",
  "bootstrap_expect": {{ groups['consul'] | length }},
  "retry_join": [{% for host in groups['consul'] %}
      "{{ hostvars[host].ansible_default_ipv4.address }}"{{ ", " if not loop.last else "" }}
  {% endfor %}],
  "client_addr": "0.0.0.0",
  "ports": {
    "dns": 53
  },
  "recursors": ["1.1.1.1", "1.0.0.1"],
  "enable_syslog": true,
}
```

We've set the field "server" to `true`, specified ports that differ from the default, set out outbound DNS addresses (recursors), set the "bootstrap_expect" option and "retry_join" which is a list of all servers' IP addresses which becomes the raft.

Notice the weird-looking `{{ ... }}` and `{\% ... \%}` parts? That's the Jinja2 templating at work, which will get the number of servers we have for "bootstrap_expect", and also dynamically create a valid JSON list of all IPs of the Consul servers in our infrastructure, gathered from Ansible variables and facts.
The `"retry_join"` field would end up looking like the following once it's copied to each host:

```
"retry_join": ["192.168.100.10", "192.168.100.12", "192.168.100.14"],
```

The configuration for all the other hosts which will join the raft as "clients" has the following configuration:

```
{
  "datacenter": "skyhigh",
  "data_dir": "/var/consul",
  "retry_join": [{% for host in groups['consul'] %}
      "{{ hostvars[host].ansible_default_ipv4.address }}"{{ ", " if not loop.last else "" }}
  {% endfor %}],
  "client_addr": "0.0.0.0",
  "ports": {
    "dns": 53
  },
  "recursors": ["1.1.1.1", "1.0.0.1"],
  "enable_syslog": true
}
```

Pretty much the same thing, except we've removed the "server" and "bootstrap_expect" fields.
Configuring Consul is a breeze once you get to understand the concepts and read up on what all of the options do, for which the documentation is quite good.

4.3.3. Registering services

Once the raft is set up and every client has connected, we deploy our demo web application to our "web" hosts, again by using Ansible. The only thing special about this is how Consul gets to know about it; by registering the running web application as a service.

The way we register a service is as easy as configuring Consul itself – it's actually exactly the same, because service definitions *are* Consul configurations, as Consul can take multiple (hence the `-config-dir` parameter in the Systemd service defined in chapter 4.2.4):

```
{
    "service": {
        "name": "demo",
        "port": 8080
    }
}
```

You give the service a name, tell it what port it will be available on, then restart Consul on the host that provides it and it'll be available throughout the cluster. There's more options to a service configuration, such as tags and health checks, but the above example is all that is needed for the service to be recognized. If more than one host registers a service with the same name and port, they will be load balanced between automatically when queried, based on randomness and the "health" of the node providing the service. The health of a node or service is pre-defined (is the service/node up?), or it can be explicitly defined as we will see in 4.3.4.

4.3.4. Load Balancing

While the built-in load balancing of Consul services is a nice feature right out of the box, we still don't have access to the service outside of the internal network. For that we'll need a reverse proxy that will be able to serve the services, like a web server.

Fabio is a reverse-proxy with load balancing features, it's configuration-less in the sense that it's self-configuring, getting all the necessary information it needs from Consul service's *tags*. With our existing web application, all we had to do was modify the service definition to this:

```
{
    "service": {
        "name": "demo",
        "port": 8080,
        "tags": ["urlprefix-:80 proto=tcp"],
        "check": {
            "name": "/health",
            "http": "http://{{ ansible_default_ipv4.address }}:8080/health",
            "interval": "10s"
        }
    }
}
```

There's two additions here; one is the "tags" section in which we specify which URL prefix should be used to access the service (in which we've omitted so the application is available on the root path "/"), the port it should be served on (port 80 is the default HTTP port), and the protocol it should be served over. The second addition is a health check from the application itself. Fabio will only work with services that are properly health-checked (making sure it's alive and well), which is an optional feature of Consul.

As previously mentioned in 4.2.4, since we're using Ansible to provision everything, including the service configuration and Fabio, we can use template variables to specify the IP address of each host that serves this application instead of hard-coding it.

With Fabio in place, we've reverse-proxied and load balanced the service, no matter how many exists, which makes it dynamically scalable!

5. Security aspects

As with anything that touches the public web, security is a concern. We inherently designed our infrastructure to have as few weak points as possible by only having two hosts accessible from the outside. The manager acts as a "bastion" host for SSH access (and thus Ansible) for every host on the internal network by proxying connections through itself. The other host accessible from the outside is the proxy host, which is where Fabio runs to serve the demo web application. Aside from only giving those two hosts public IP addresses, we also strengthened our internal network with OpenStack security groups.

5.1. SSH

To secure remote connections we've hardened the hosts by only allowing remote access by pubkey, and disallowed remote access to the root user. Further improvements would consist of letting the manager only accessible through VPN, not exposing it to the public web at all, and/or using multi-factor authentication like 2FA. For our demonstration purposes we didn't use either of these two additional layers of security, but we acknowledge their importance in a production environment.

5.2. State files

The state files Terraform operates with can (and most likely will) contain information about the infrastructure that we don't want to end up in the wrong hands. One solution to this is keeping it in an external backend (see 4.1.1), out of the way of both the public and even the team members working with it. Further improving this would be to automate Terraform entirely by utilizing a Continuous Deployment environment, only allowing machine users to access the state file and deploy the infrastructure.

5.3. Secrets

As with most infrastructures, especially those concerning web applications, "secret" information such as usernames, passwords, tokens and other forms of credentials are at play. Authentication details for a database needs to be available to both the application that utilizes it, and the system provisioning and creating the database. One solution to this aspect would be to store these credentials in Consul's Key/Value storage[9] or a separate system such as HashiCorp's *Vault*.
We're only mentioning it here for posterity, as we didn't plan to integrate this into our project, it's just something we came by as we explored Consul.

5.4. Internal security

One of the features of Consul is *Connect*, which can tell which nodes are allowed to reach a service. They call it *Intentions*, which is basically ACLs for services using TLS-secured communication and pre-defined rules, and is a nice feature for today's GDPR world. This system also ties in with Consul's Key/Value storage, limiting which nodes/services has access to certain secrets. While this is something we looked at implementing, and it's supposedly rather simple to do, we just didn't have time to look much into it.

6. Conclusions

The infrastructure and patterns we have implemented for this project is suitable for modern microservices where we want services to be completely autonomous without any human interaction on the hosts themselves. If we were to set up an infrastructure with support for end-users and expand our scope to include domain controllers, dedicated DNS, DHCP, email, file-servers, print-servers, etc, we would have implemented a more powerful provisioning system. Such an infrastructure would probably be provisioned with Puppet, using Foreman and R10k for role assignment and deployment of environments.

There are still a lot of tools, services and other integrations we would have liked to have included in our project. CI/CD is one of them, allowing us to create an autonomous self-deployed setup based on Git branches. The main issue was getting a system like the CircleCI integration on GitHub to connect with our infrastructure on OpenStack, as it's not accessible outside the campus network. One possible solution would be to set up GitLab on another host in our network, but we figured it would be a whole lot of work just to implement it, and time was of the essence.

Services we implemented, but later removed:

- **Puppet**: We initially wanted to use Puppet, since we'd already learned how to use and deploy it, but ended up using Ansible due to how well it integrated with Terraform. We'd still prefer to use Puppet over Ansible on a larger scale, so Ansible was used mostly because we wanted to explore alternatives.
- **HAProxy**: We had a complete configuration for one dedicated load balancer, but discovered Fabio which required less effort to configure and was more tightly integrated with Consul, further showing off our initial focus in a better light.

Other services we considered, but did not implement:

- *Logging and Monitoring*: event-based infrastructure scaling and general metrics
- *Database cluster*: for demonstration purposes, but we opted for a simpler demonstration
- *VPN & 2FA*: to demonstrate additional layers of security

If you want to create a flexible infrastructure, then we would highly recommend a look into Terraform, Ansible, and Consul. These tools made it a joy to get our systems and services to communicate. They also have comprehensive and understandable documentation, which made our lives that much easier during our research.

References

[1] HashiCorp, “Terraform provisioners.” [Online]. Available: https://www.terraform.io/docs/provisioners/index.html

[2] ——, “Consul consensus table.” [Online]. Available: https://www.consul.io/docs/internals/consensus.html#deployment-table

[3] A. L. Lange, “Consul demo app.” [Online]. Available: https://github.com/adrialu/consul-web-demo

[4] HashiCorp, “Terraform providers.” [Online]. Available: https://www.terraform.io/docs/providers/index.html

[5] ——, “Terraform community providers.” [Online]. Available: https://www.terraform.io/docs/providers/type/community-index.html

[6] ——, “Terraform backend types.” [Online]. Available: https://www.terraform.io/docs/backends/types/index.html

[7] ——, “Raft protocol overview.” [Online]. Available: https://www.consul.io/docs/internals/consensus.html

[8] Wikipedia, “Cap theorem.” [Online]. Available: https://en.wikipedia.org/wiki/CAP_theorem

[9] HashiCorp, “Kv data.” [Online]. Available: https://learn.hashicorp.com/consul/getting-started/kv.html

A. Source Code

Instead of embedding all our work in this document, we're rather going to refer to the repository we worked with during this project, allowing the reader to experiment with- and try out our infrastructure for themselves.

https://github.com/adrialu/terraform-consul

As a supplement to the infrastructure, we also developed the demo web application ourselves. It's a simple application written in the Go language which will list out all the nodes Consul is aware of, and also highlights the node the visitor has been load balanced to.

https://github.com/adrialu/consul-web-demo

Lastly, Adrian created a plugin for the text editor *Sublime Text* which auto-formats Terraform definition files when saving them to disk, which came in handy to maintain consistency throughout the project development.

https://github.com/p3lim/sublime-terrafmt