TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Karl-Andreas Turvas 194145IACB

DYNAMIC INFRASTRUCTURE EXPLORATION: ORCHESTRATION LAB WITH HASHICORP

Project Report

Tallinn 2023

List of Abbreviations and Terms

SSH	Secure Shell
PoE	Power Over Ethernet
HAT	Hardware Attached on Top
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
TLS	Transport Layer Security
SSL	Secure Sockets Layer
BIOS	Basic Input/Output System
CPU	Central Processing Unit
GPU	Graphics Processing Unit
RAM	Random Access Memory
RAID	Redundant Array of Independent Disks
NAT	Network Address Translation
IP	Internet Protocol
DNS	Domain Name System
DDNS	Dynamic Domain Name System
IaC	Infrastructure as Code
VM	Virtual Machine
VE	Virtual Environment
ISP	Internet Service Provider

Table of Contents

1. Introduction

This report aims to provide an overview of the author's efforts in the Computers and Systems Project (IAS1420) course. The project's objective was to construct a virtualized lab environment for a single-zone container orchestration cluster using HashiCorp solutions. The author opted to using infrastructure as code approach at the project's core: infrastructure and configurations should be defined and managed using code and be reproducible. For the orchestration cluster using HashiCorp Nomad, the author's approach diverged from the all-batteries-included industry standard approach of using Kubernetes emphasizing the exploration of containerized services to acquire expertise in designing, building, and managing contemporary dynamic infrastructure.

Following chapters will give an account of the motivations for building such a (seemingly overly) complex environment, the process of building it, key learning outcomes and proposed improvements.

2. Motivations for Orchestration Environments

In the olden days services were primarily installed and administered on dedicated hardware hosts. The demand for resource efficiency and the isolation of services spurred the evolution of virtualization solutions, resulting in escalating complexity and abstractions layers. From virtual machines to containers to orchestrating said containers, each step has promised new opportunities to make infrastructure more cost-effective, resource economic and resilient.

Virtual machines enable better utilization of a single physical machine by dividing one physical host into many virtual ones. Virtualizing hosts provides complete isolation from the physical host operating system and kernel, thus allowing secure isolation between services running on hosts. However bare services running on virtual hosts still rely on the host operating system, so differing versions of dependencies or changes in one part of the system might affect others. Good isolation comes at a great resource cost of virtualizing an entire host from hardware to kernel to operating system.

Containers operate at the operating system level, allowing the segregation of services with minimal overhead and providing isolation from the host operating system without a strong security boundary. Containers solve the inefficiency and the problem of relying on host operating system by running a bare minimum of an entirely new operating system on top of the host operating system. It isolates the service and its dependencies from the host, but still shares the same kernel and resources. Lack of security from containers means administrators need to put more effort into securing the permissions and capabilities of containers as well as configuring ways of isolating communications.

And finally, orchestrators automate roll-outs and roll-backs of containerized services to a number of hosts in the orchestrated cluster. Orchestration promises to separate the containerized services from hosts, removing the need to rely on any particular host, be it virtual or physical, thus providing the peace of mind from physical host failures: if one host dies, the orchestrator will put it on another and the service will continue to work.

Thus an orchestrated environment seemed enticing for its benefits:

- safety from host failures.
- automatic service roll-outs.
- automatic load balancing.

- automatic routing and certificates for each service.
- ease of deployment of containers as opposed to the more involved process of writing and managing configurations for each service manually.

Before this project the author had administration experience with virtual machines and containers, but only some user experience with orchestration using Kubernetes. Motivation of this project was to evaluate the viability of orchestration in the context of a home laboratory. The author hypothesized that the effort would not pay off as the complexity and resource overhead of running orchestration for a handful of services in home setting would be too great. Either way the gained experience would be useful in understanding modern infrastructure and for future endeavours.

3. Building the Lab

The hardware for this project was entirely composed of personal computing equipment that the author had already acquired for their home laboratory purposes. The author's home laboratory consisted of a single 36-unit rack with 1GB/s networking through all ports of switches and routers, in addition to 250mbit/s duplex speed modem connection to the internet. The laboratory network was physically isolated from the home network using a second router. Hardware used for the project consisted of two routers, two switches, a HP DL380 G9 server and two Raspberry Pi 3B+ [1] devices that were connected to the second, laboratory router using a switch. Map of the physical network connections can be seen on Figure 1.

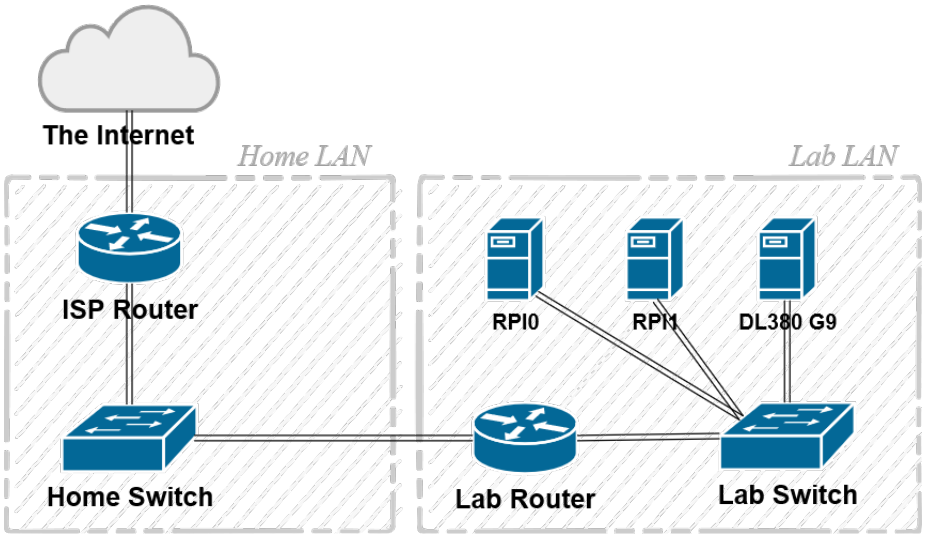

Figure 1. Physical network connections map

The most notable limitation arising from this cost effective approach of using existing hardware, that was not designed for orchestration originally, was the necessity to simulate a multi-host environment through virtualization as there was only one physical machine with enough resources to run orchestrated services at larger scale. This virtualization-centric approach influenced the realism of the experimental setup: certain nuances and intricacies that might have been observed in a dedicated hardware environment were potentially overlooked or simplified in the virtualized setup. Moreover the proposed safety from host failures in this projects simulated environment only applies to virtual machines which are unlikely to fail one by one: if something were to happen to the physical host, then the whole cluster would fail. So while simulating a multi host cluster with virtualization can still be considered a viable way of testing things out, it can not be considered a production

ready system for providing services.

Following chapters detail the set up of hardware, operating system and configuration management of the virtualization machine used of orchestration.

3.1 Hardware

Hardware used specifically for the project was a single HP DL380 G9. Configuring CPUs and storage proved relatively straightforward: the task entailed maximizing the board's capacity. In this instance, the accommodation permitted the insertion of 2 CPUs and 8 disks, as only one of the three potential drive bays was integrated into the board.

The allocation of RAM led to additional deliberation, despite how enticing it is to fill all 24 slots for a cumulative memory capacity of 384GB. For best performance, all four memory channels of each CPU should be populated with identical DIMM setup to allow parallel access across the entire memory bus. This means installing 8, 16, or 24 DIMMs total. However populating all sockets per memory channel, meaning all 24 slots, memory controller significantly lowers the frequency as detailed in a blog post detailing an investigation into the NUMA architecture [2]. The mitigation of this frequency reduction, was imperative due to the consequential overhead in running multiple virtual machines.

Consequently, reading the DL380 G9 user manual about the RAM configurations [3] it was written that opting for a configuration of 24 modules of 16GB RAM would lead to a reduction in memory speed from the specified 2133MHz of the sticks to 1600MHz. Recognizing the significance of maintaining optimal performance a decision was made to embrace a faster but lower capacity alternative. Accordingly, the decision was made to leave the third slots of the memory channels unoccupied and a configuration of 16 modules of 16GB RAM was selected. This choice was predicated on the premise that 256GB was deemed very likely already sufficient for the intended purposes without knowing exactly how many virtual machines would be used in the end.

For the two Raspberry Pis, additional heatsinks were installed for better cooling performance and a power over Ethernet hardware attached on top (HAT henceforth) module was installed to provide both power and networking over the same Ethernet cable, simplifying cabling. No other configuration was done for the Raspberry Pis.

3.1.1 BIOS

Having configured the hardware, initiating the system setup and addressing BIOS configurations was next. On booting the system one faulty RAM stick was detected and promptly replaced. From settings, of virtualization was enabled, power-saving and throttling settings disabled for maximal performance. Additionally, the Integrated Lights-Out (ILO) [4] the hardware and firmware-based remote management platform was confirmed to be working.

Subsequently, the HP Smart Storage Administrator [3] interface was used to set up the eight 600GB physical disks into a singular logical volume configured with RAID10 using the machines installed storage controller. With this configuration it would be possible to replace disks seamlessly without rebooting, on failure or otherwise. Recognizing the constrained size of the disks, a decision was made to forgo the standby status for additional unconfigured disks, typically employed for disaster recovery scenarios.

3.1.2 Operating System: Proxmox Virtual Environment

For virtualization Proxmox Virtual Environment [5] server management platform was used for its free and open-source nature. Additionally it used widespread industry solutions like Debian [6] at its core, standard Linux and OVS [7] networking, popular storage options, and the widespread KVM hypervisor [8], coinciding with author's previous experience and also ensuring that support and solutions would be easily available. Its nonrestrictive of the free version of the operating system and single-install approach rendered it a fitting and advantageous solution for the project's requirements compared to other virtualization platforms which would have required investing into licences or would have been more complicated to set up.

3.1.3 Operating System: Raspberry Pi OS

For the Raspberry Pis, once again a Debian-based operating system was chosen and the Raspberry Pi OS [9] was installed on both devices. Raspberry Pis were called 'rpi0' and 'rpi1'.

For simpler management, Dnsmasq [10] was installed and configured on 'rpi0' to provide internal domain name resolution to all physical and virtual hosts.

3.2 Virtualization

Adhering to the infrastructure as code approach set by the author opted for the HashiCorp Packer [11] for creating the operating system images for virtual machine clone templates and HashiCorp Terraform [12] for provisioning virtual machines from the clone templates on the Proxmox platform.

3.2.1 Operating System Image Builds Using Packer

The code used for building the operating system image clone templates can be found at GitHub [13].

Ubuntu 22 [14], a Debian based Linux operating system, was chosen as the operating system for all virtual machine to keep consistency of managing hosts in the lab since Proxmox and Raspberry Pis all ran Debian based operating systems.

Using Packer the following steps were automated and made reproducible as code:

- Latest Ubuntu 22 image was downloaded.
- Clound-init was configured [15]. The heart of our automation lies in the configuration of cloud-init [15] in the template. Cloud-init allows for the customization of the machine on provisioning and each boot, this is required for the provisioning process to be automated.
- Additionally upgrades were applied and unnecessary files cleaned to preserve disk space, free disk space was zeroed to keep the image smaller.
- The root account was locked, and SSH root login prohibited.
- Finally, the configured machine was converted into a Proxmox clone template which was later used to create the machines comprising of the project's infrastructure.

3.2.2 Virtual Machine Provisioning Using Terraform

The code used for provisioning virtual machines from image clone templates can be found at GitHub [16].

Using the template built with Packer, Terraform was used to provision virtual machines on Proxmox and configure them with cloud-init. Using the Terraform definition, a file with the configurations detailing virtualized hardware, hostname, IP, gateway, authorized SSH keys of root user and first boot commands were specified.

Using the boot commands, a script was run such that each machine's disk was expanded to use the full capacity to account for the difference in the templates small disk size and specified larger actual size of the disk for running virtual machines.

On apply, Terraform created three base nodes, which would make up the control plane of the orchestration, three service nodes, which would make up the service layer of the orchestration, and one support node, which would be used for any required additional supporting services like certificate authority for internal certificates and Ansible to configure all nodes. Support node was provisioned with 1GB of RAM, 2 CPUs and a 50GB boot disk, all base and service nodes were provisioned with 12GB of RAM, 4 CPUs and a 150GB boot disk each. This number of base and service nodes and their resource amounts were chosen on the basis of Nomad reference architecture for small clusters [17] which stated that for one machine failover without intervention at least three machines are needed for each layer (so three for control layer and three for service layer).

After provisioning there were three physical machines, two Raspberry Pis (named rpi0 and rpi1), one HP DL380 G9 (named pve0) where on its Proxmox platform were seven provisioned virtual machines: three base nodes (named vb0, vb1, vb2 where vb stands for Virtual Base node), three service nodes (named vs0, vs1, vs2 where vs stands for Virtual Service node), and an additional head node. A simplified diagram of the physical and virtual nodes can be seen on Figure 2.

3.3 Configuration Management

Ansible [18] was used for configuration of all physical and virtual hosts. All chapters forward imply that the steps described were documented in a reproducible infrastructure as code approach with Ansible and not manually. The Ansible code used for managing all machines can be found at GitHub [19].

The design of the Ansible code was made to be as simple as possible: each host would be part of only one host group and one host group only, making it easy to find what was being run on each host at a glance.

Each host group's playbook and variables file can be found in the playbooks directory. There the playbook defines the connection parameters, additional variable files, and the list of roles to be applied to the hosts in a host group. Variable files then configure how each role applied should be run on the host. Each role is separated to configure one service or aspect of a host. Idempotency of the roles was key in ensuring reproducibility and consistency of the configurations, guaranteeing that the resulting configuration on hosts

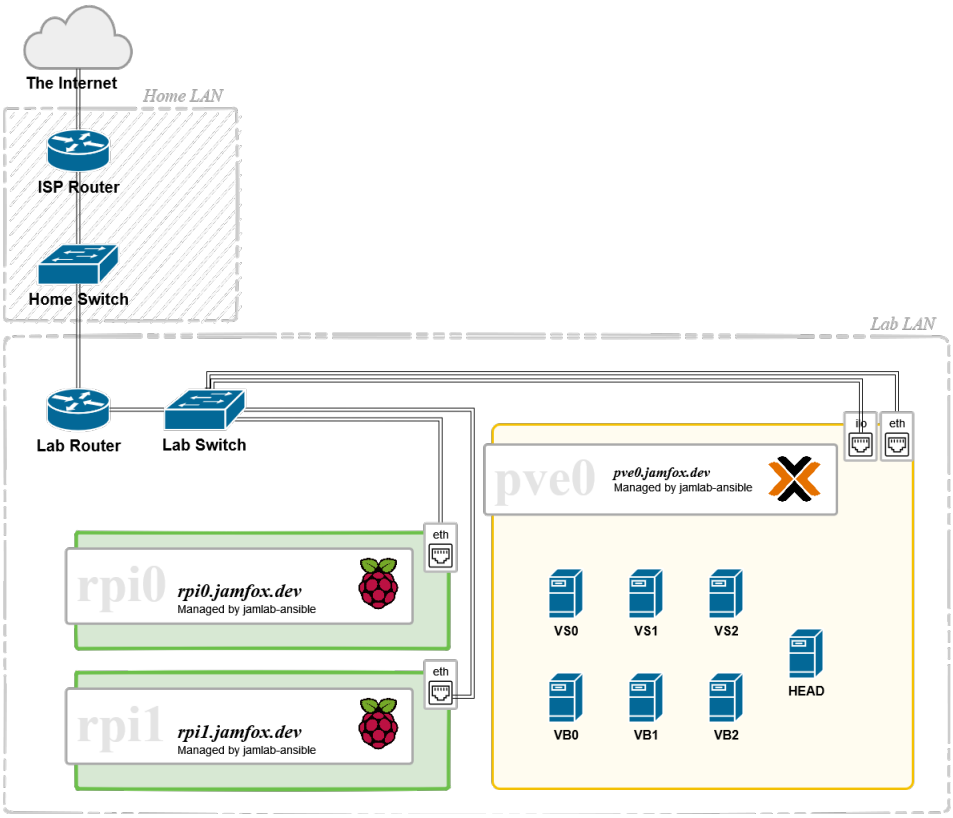

Figure 2. Simple overview of the physical and virtual machines

would be the same no matter how many times the management system was run.

To make running multiple playbooks easier, a script was written in bash. The code for the script can be found at GitHub [20]. The script provided the following advantages:

- Runs playbooks in parallel
- Shows filtered logs with only errors and changes for actionable readability
- Easily targets specific playbooks using playbooks names or string matches
- Pull the latest of specified git branch for runs
- Use specific git branches for runs
- Skips unreachable hosts
- Runs a post exit script

3.3.1 Security Configurations

On all machines unattended upgrades were installed to provide security updates on all nodes as soon as they became available. On top of that Ansible system itself would upgrade all packages on all hosts to the latest version on each run in an effort to keep the machines

as up to date as possible.

Iptables [21] was used to limit access using packet filtering rulesets using deny-by-default principles. Internal LAN was allowed on all hosts, external access was configured on case by case and need by need basis. External access was configured using country specific IP sets enabling the potential to use port forwarding to expose services to the internet by region. While this firewall enabled a more secure way of exposing services, port forwaring was not used in the end to expose services to the internet.

Most crucially password authentication was disabled over SSH, thus only allowing public key authentication rendering brute force attacks on remote access virtually impossible.

4. The Orchestration Environment

For orchestration, out of all the choices currently used in the industry [22], HashiCorp Nomad [23] was chosen as opposed to all-batteries-included industry standard approach of Kubernetes [24] emphasizing the exploration of containerized services to acquire expertise in designing, building orchestration environments from scratch and managing them.

4.1 Nomad as Orchestrator

Nomad concentrates solely on cluster management and scheduling, adhering to the Unix philosophy of maintaining a concise scope. It integrates with tools such as HashiCorp Consul [25] for service discovery and service mesh.

In short HashiCorp themselves describe Nomad as providing scheduling, self-healing, rollouts and rollbacks, storage and auto scaling. Consul adds service discovery and service meshing on top of Nomad and what Nomad was missing and needed was load balancing and configuration management, both of which can be run on Nomad but do not come with it out of the box, so to say. The previously mentioned Ansible configuration management in Chapter 3 was run outside of the Nomad cluster, from the virtual machine called 'head'.

4.1.1 Configuring Nomad

As they were designed from the start to be compatible, Consul was used for service discovery and service meshing. Consul was run alongside Nomad and together they form the heart and foundation of the orchestration.

The Ansible Community Nomad [26] and Consul [27] roles were used to install, configure and manage the orchestrator and its nodes.

The primary challenge in configuring Nomad and Consul using roles stemmed from the intricate task of establishing accurate configurations to facilitate encrypted communication both between nodes and between Consul and Nomad.

Step CA [28] was set up on the support node called 'head' as the certificate authority. Certificate authority was used for requesting internal certificates to all orchestration cluster nodes, ensuring that all traffic to, from and between Nomad nodes was able to be encrypted

and verified using the certificates.

Furthermore, Podman [29] was used as the modern container engine of choice instead of the default Docker engine [30] to ensure that only a rootless user with fine-grain tuned permissions could run the daemon and containers. This ensured greater security and isolation of containerized services from the host operating system (criticality of this was briefly mentioned in Chapter 2).

4.2 Load balancing

Load balancing is the act of routing and balancing traffic through a balancer to appropriate destinations. The fundamental idea behind load balancing is to distribute incoming network traffic and workloads across multiple servers. This approach enhances the overall performance, availability, and scalability of a system.

When employing orchestration tools such as Nomad, the destinations may correspond to any number of nodes that may be hosting multiple services, each on different, ephemeral ports, thus increasing routing and load balancing complexity. In some part of the load balancing system, it must be integrated to work with service discovery which would be able to tell the load balancer(s) on which host and port services were running on.

Load balancing was solved using according to HashiCorp's recommended architecture [31] which meant using an external load balancer that would route to live orchestration nodes which would have their own internal load balancers that would then direct traffic to appropriate host and port in the cluster.

For the external load balancer, HAProxy [32] was installed and configured on the Raspberry Pi called 'rpi1'. All ingress Nomad traffic was routed through 'rpi1' running HAProxy which checked which Nomad nodes are alive. On receiving a request, HAProxy initiated an encrypted HTTPS connection, terminated it at 'rpi1' and then forwarded plain HTTP requests to live Nomad node internal load balancers.

On the external load balancer, wildcard certificates for the lab domain names were configured on HAProxy with certbot [33] using Let's Encrypt [34] as the provider. Let's Encrypt is a non-profit certificate authority that provides free SSL/TLS certificates for websites. SSL/TLS certificates are used to encrypt the data transmitted between a user's web browser and a service, ensuring that sensitive information such as login credentials, and other personal data is transmitted securely. Compared to the previously mentioned internal certificate authority using Step CA, Let's Encrypt is a publicly trusted certificate

authority. This means that its root certificate is included in major web browsers and operating systems, and its certificates are automatically trusted by default, making it the better choice for exposing services to the internet.

On the topic of exposing services to the internet, since the authors home environment did not have a static public IP, port forwaring solutions were not considered. Instead Cloudflare Tunnel [35] was configured on the external balancer machine. Cloudflare Tunnel provides a secure way to connect the balancer to Cloudflare without a publicly routable IP address. With Tunnel, a lightweight daemon creates outbound-only connections to Cloudflare's global network, making it possible to use Cloudflares public DNS to point domains to Tunnel which can be configured to utilize Cloudflare's firewall and intrusion protection including an interface to configure fine-grained access rights (such as limiting some services to IPs of a specific country or requiring email confirmation before getting access to a service).

For the internal load balancers, Fabio [36] was used for its native integration with Consul service discovery [37]. Nomad configurations needed to exhibit dynamic and self-healing attributes in case of application or node failures. Fabio was deployed as a system job type to ensure it was running on all possible nodes to ensure failover and even traffic distribution.

Sequence diagram of the communications needed to create a successful communication from a client to a containerized service can be seen at Figure 3.

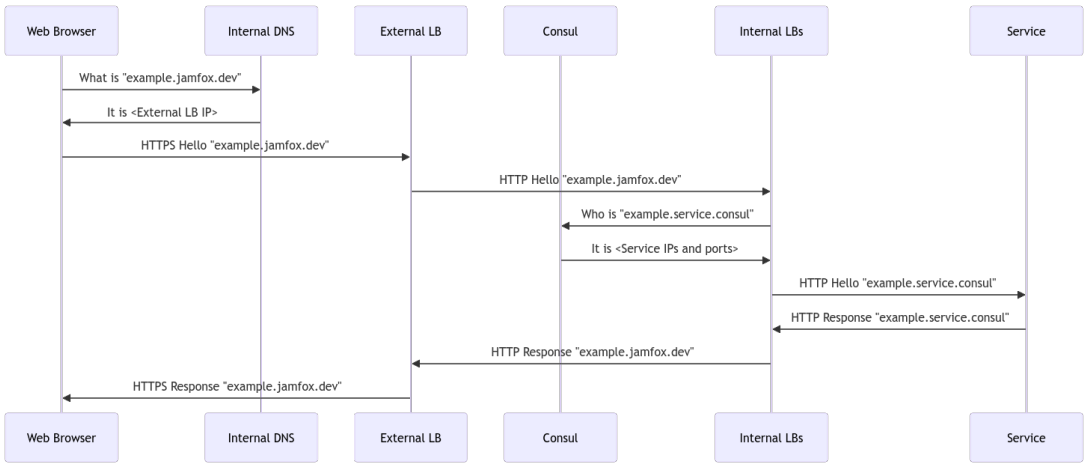

Figure 3. Sequence Diagram of Service Communication

4.3 Consul Service Meshing

HashiCorp describes service mesh as "...a dedicated network layer that provides secure service-to-service communication within and across infrastructure." In dynamic containerized orchestrated environments it becomes an especially hard task to isolate services from each other, even more so when a service has multiple instances and/or supporting containers alongside it. Service meshing is the solution in where each service with multiple containers has their own set of support containers that bind them together with encryption that only they can read, thus optimizing and isolating traffic.

Of course this means even more overhead for each service running to keep the meshes running also increasing load on the service discovery for each additional mesh created as each mesh is managed on the basis of where which host and port each of the services containers is currently at.

The service mesh was enabled to be used with Nomad services using Consul Connect [38]. To use service meshing, it had to be defined in a services definition.

5. Key Outcomes

5.1 Overview of the Final Architecture

Finally there were three physical machines:

- 'rpi0' where the internal DNS provided name resolution for the laboratory network.
- 'rpi1' where the HAProxy external load balancer accepted client requests for the containerized services.
- 'pve0' the Proxmox Virtual Environment virtualization machine where the virtualized orchestration services were provisioned and ran from.

On the 'pve0' there were seven virtual machines:

- three base nodes, making up the control plane of Nomad orchestration that provided scheduling, self-healing, roll-outs and rollbacks, storage and auto scaling for containerized services.
- three service nodes, making up the service layer of the orchestration, in other words the nodes where the services were running on.
- one support node, which was used to run the Step CA internal certificate authority and Ansible configuration system.

Diagram of the whole project with both physical and virtual machines along with key services can be seen on Figure 4.

5.2 Testing failures

When one of the virtual machines from the control plane was shut down to test a host failure the cluster continued to function and on reboot, the initially shut down machine was rejoined to the cluster automatically.

When two of the virtual machines from the control plane were shut down to test catastrophic failure, the cluster operation was halted as having only one node available causes a loss of quorum and data loss, since at least two servers must be up so they can vote for the leader and compare the data. Functions were restored after rebooting shut down nodes and forcefully telling the cluster to elect last remaining node as the leader and to use its data

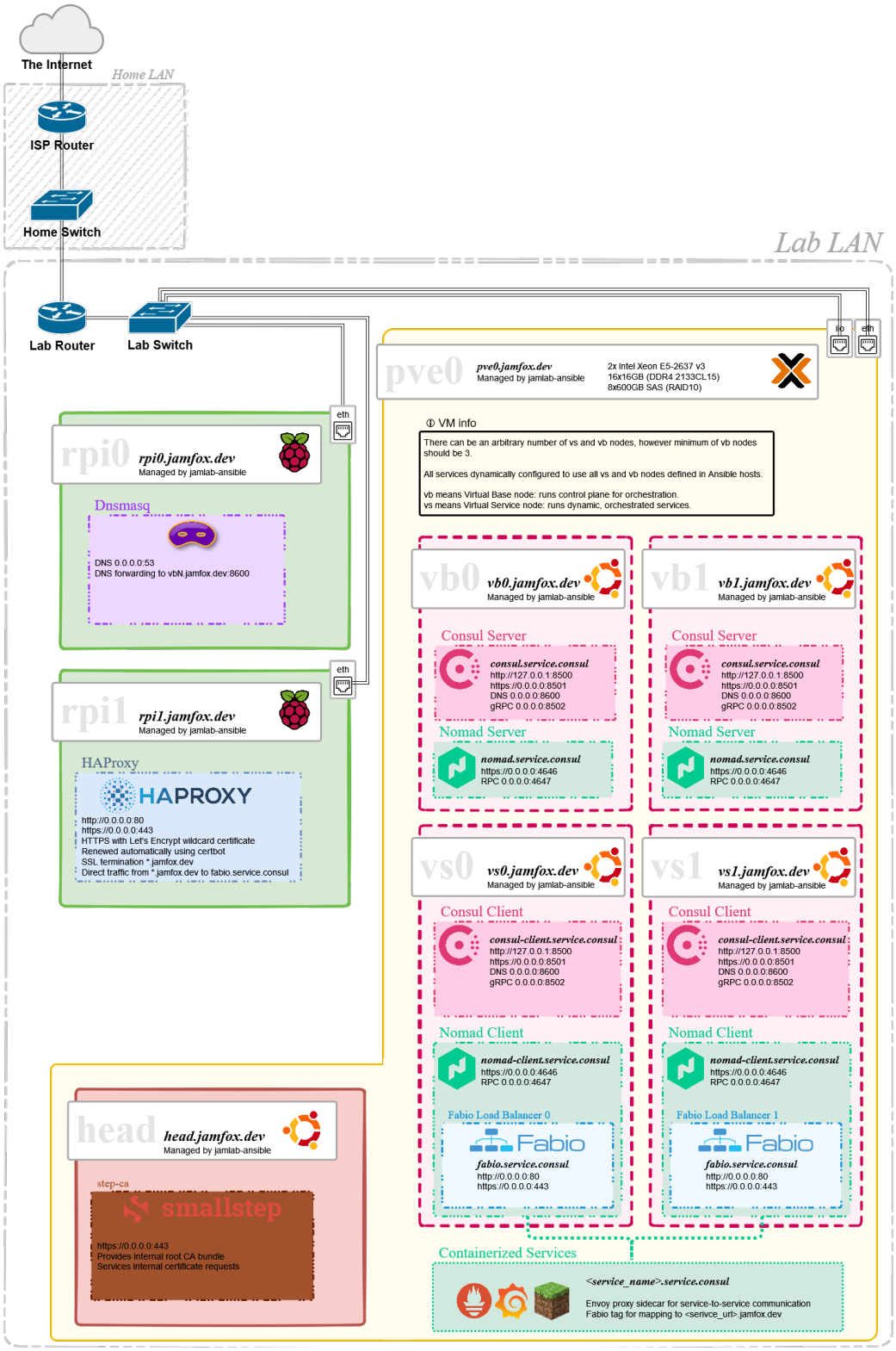

Figure 4. Overview of the physical, virtual machines and key services

(even if it may not have been the latest).

Shutting down service nodes was also tested. The containerized services continued to

function until at least one node was still up. Thus the project could have been done with only two service nodes instead of three as was done. With base nodes, it is required to have three servers to have a fault tolerance of one machine since base nodes need to elect a cluster leader. Service nodes however do not need to have a leader and as such having two servers is enough to have fault tolerance of one machine. Having fault tolerance of one for services also means that services that would need to be on multiple servers would not be able to function if only one is remaining.

Downtime when a node service was shut down and a service without multiple instances was transferred from one node to a remaining one was less than a minute. This, while significant is still quite fast and came with the benefit that it was automatic whereas without an orchestrator the administrator would have to get notified of failure and fix it manually which would in likelihood lead to longer downtime.

5.3 Load Balancing Orchestrated services

Load balancing orchestrated services proved to be the most fascinating for the author. Having previous experience with fairly basic reverse proxy definitions where in a configuration a static machine with a static port was defined for any particular service, the dynamic nature of orchestrated services added some exciting complexity.

No longer having a static host or a static port to rely on highlights the need for a good service discovery in dynamic environments like this very well. Not only that, but integrating service discovery into load balancers needs great care as most proxies do not support service discovery integration out of the box and require custom configuration. Custom configuration then means taking extra care to make sure that the configuration is secure and efficient.

5.4 Service Meshing

HashiCorp describes service mesh as "...a dedicated network layer that provides secure service-to-service communication within and across infrastructure." In dynamic containerized orchestrated environments it becomes an especially hard task to isolate services from eachother, even more so when a service has multiple instances and/or supporting containers alongside it. Service meshing is the solution in where each service with multiple containers has their own set of support containers that bind them together with encryption that only they can read, thus optimizing and isolating traffic.

Of course this means even more overhead for each service running to keep the meshes running also increasing load on the service discovery for each additional mesh created as each mesh is managed on the basis of where which host and port each of the services containers is currently at.

5.5 Container Service Definitions

Writing orchestration service definitions to run on Nomad at this point in time is still a complex task that requires fairly good knowledge of the underlying infrastructure. It would be hard for someone unfamiliar with Nomad to write a service definition without some assistance from the administrator as some of the configuration is specific to the way infrastructure was configured (which variable to use for external domain name, which for internal, how to configure the service mesh etc) making them very leaky abstractions [39] by default. This could be overcome from the administration side by automating a lot of it behind the scenes, providing customization only for vital components. However, for domain names for example this is not possible and would require good guides along with templates to make it understandable for the uninitiated to jump right into running their services on the orchestrated cluster.

5.6 Viability of Orchestration Environments for Home Laboratory Purposes

Orchestrated environment seemed enticing for its benefits of not having to worry about host failures, automation of roll-outs, load balancing, routing, certificates for each service and the ease of managing containers as opposed to manual configuration management. However during the project, the author concluded that the resource and management overhead is just too great for it to pay off. In the end it costs more time and money to run orchestration when compared to running plain containers or services directly on host manually.

In larger environments where up-time is vital, orchestration can provide many advantages, but in small environments the cost of running the orchestration cluster most likely exceeds the requirements for all services running on the cluster many times over.

6. Improvements

While the project resulted in great experience with dynamic orchestrated experience for the author and resulted in a working self-healing environment with a fault tolerance of one machine, there were three main criticisms that could be improved on.

First, monitoring and logging was not solved in a readable and actionable way, only out of box solutions of the installed services were used, no collection or analysis tools were used.

Second, end-to-end encryption should be used from the external load balancer to services. In this case the HTTPS connection was terminated at the external load balancer, putting a lot of trust on the security of the internal network.

Third, load balancing could be solved more efficiently by integrating Consul service discovery directly with the external load balancer so there would be no need for internal ones on each machine as the external load balancer is a single point of failure anyway.

6.1 Monitoring and Logging

In orchestration environments, the importance of monitoring and logging cannot be overstated. Traditionally, logging existed as a fundamental component, capturing system events and activities. However, the increased volume of logs and the complexity of orchestration environments makes it challenging to find and analyze information effectively.

Consul and Nomad provided logs for their operations and were readable. However while Nomad does log orchestrated services running on it, finding, accessing, and reading them is a time consuming task and made troubleshooting an inefficient process.

In most environments like this, in the authors experience, usually slow networking and storage become bottlenecks. The authors networking was built on slow maximum speed of duplex 1GB/s and while considered fast for spinning mechanical disks, the 15K SAS disks still pale in comparison to solid state storage options. Both faster networking and faster storage is usually recommended or even required for larger environments [40]. While these bottlenecks were not apparent during this project (especially since only lightweight services were tested), either way it was impossible to provide empirical data to back up claims of performance without proper monitoring.

Efficient monitoring tools offer real-time visibility into the various components of an orchestration environment. They enable the tracking of resource utilization, system bottlenecks, and the overall performance of orchestrated processes. This real-time feedback is invaluable for identifying issues promptly and optimizing system efficiency. Logging, when coupled with advanced analysis tools, transforms into a source for understanding system behavior and troubleshooting, rather than drowning in a sea of raw log data.

Monitoring and logging tools also contribute significantly to security efforts. By tracking and analyzing system events, it is possible to detect and respond to security incidents in a more easily visible and timely manner. This proactive approach is essential for safeguarding potential sensitive data of services.

Investing time in monitoring and log analysis tools is a logical step towards improving this project's security as well as management and troubleshooting efficiency.

6.2 HTTPS Termination at the External Load Balancer

In the quest for enhanced security in orchestration environments, the implementation of HTTPS encryption is a pivotal step. However, when terminating HTTPS at the external load balancer as was done in this project (described in the orchestration chapter 4), it's crucial to recognize that while external communications are encrypted, the internal network becomes a focal point for trust and potential vulnerabilities.

HTTPS termination at the external load balancer means that external traffic is decrypted and thus in a readable state within the internal network. While this ensures secure external interactions, internal communications might be susceptible to unauthorized access or monitoring within the network if a node is compromised or a bug is planted.

Encrypting external communications was vital and non-negotiable and while network segmentation of the laboratory environment and trust in it is greater than in the communications of the wide web, implementing end-to-end encryption within the internal network to mitigate the risk of unauthorized access is highly encouraged.

6.3 Eliminate Internal Load Balancers

Load balancing was solved using an external load balancer that routed to internal load balancers running on each service node of the cluster. This was done because the external load balancer of choice had no direct integration with Consul service discovery. It would

be more efficient to choose and spend more time configuring service discovery to work directly with the external load balancer, thus eliminating the need for internal ones, reducing resource overhead for running services.

7. Summary

The goal of the report was to outline the author's efforts in the Computers and Systems Project (IAS1420) course in creating a virtualized lab environment for a single-zone container orchestration cluster. This project diverged from the mainstream Kubernetes approach, opting for HashiCorp solutions to gain hands-on experience in designing, building, and managing dynamic infrastructure.

In conclusion the exploration revealed that the orchestration demands a significant upfront investment in time and effort. It calls for a more nuanced comprehension of underlying components compared to the straightforward manual deployment approach of utilizing plain containers without an orchestration layer. Furthermore, in the author's view the considerable physical infrastructure and management overhead associated with establishing and overseeing the orchestration layer imply that the advantages of scalability and flexibility become worthwhile only in larger environments and do not provide a lot in smaller environments (like the author's home lab) where a person may only be running a few of services at most. The resource overhead for running orchestration might exceed that of the service running on it, making it a very inefficient endeavour even if it does provide safety and ease of recovery from host failures.

Despite the author's decision not to employ the container orchestration approach for managing home services, the experience lead to numerous learning outcomes in building, administrating, and understanding modern dynamic infrastructure methodologies as well as ideas for further learning and improvement of the project.

References

[1] Raspberry Pi Ltd. *Raspberry Pi 3 Model B+*. 2023. URL: https://www.raspberrypi.com/products/raspberry-pi-3-model-b-plus/. Used 2023.28.11.

[2] Frank Denneman. *NUMA Deep Dive Part 4: Local Memory Optimization*. 2016. URL: https://frankdenneman.nl/2016/07/13/numa-deep-dive-4-local-memory-optimization/. Used 2023.28.11.

[3] L.P Hewlett-Packard Development Company. *HP ProLiant DL380 Gen9 Server User Guide*. 2014. URL: https://raw.githubusercontent.com/JamFox/docs.jamfox.dev/master/docs/content/homelab/attachments/DL380Gen9-UserGuide.pdf. Used 2023.28.11.

[4] Hewlett Packard Enterprise Development LP. *HPE Integrated Lights-Out (iLO)*. 2023. URL: https://www.hpe.com/us/en/hpe-integrated-lights-out-ilo.html. Used 2023.28.11.

[5] Proxmox Server Solutions GmbH. *Proxmox Virtual Environment - Open-Source Server Virtualization Platform*. 2023. URL: https://www.proxmox.com/en/proxmox-virtual-environment/overview. Used 2023.28.11.

[6] Inc. Software in the Public Interest. *Debian – The Universal Operating System*. 2023. URL: https://www.debian.org/. Used 2023.28.11.

[7] The Linux Foundation. *Open vSwitch: Production Quality, Multilayer Open Virtual Switch*. 2016. URL: https://www.openvswitch.org/. Used 2023.28.11.

[8] The Linux Kernel community. *Kernel Virtual Machine*. 2016. URL: https://www.linux-kvm.org/page/Main_Page. Used 2023.28.11.

[9] Raspberry Pi Ltd. *Raspberry Pi OS*. 2023. URL: https://www.raspberrypi.com/software/. Used 2023.28.11.

[10] Simon Kelley. *Dnsmasq*. 2023. URL: https://dnsmasq.org/. Used 2023.28.11.

[11] HashiCorp. *Packer by HashiCorp*. 2023. URL: https://www.packer.io/. Used 2023.28.11.

[12] HashiCorp. *Terraform by HashiCorp*. 2023. URL: https://www.terraform.io/. Used 2023.28.11.

[13] Karl-Andreas Turvas. *jamlab-packer*. 2023. URL: https://github.com/JamFox/jamlab-packer. Used 2023.28.11.

[14] Canonical Ltd. *Ubuntu 22.04.3 LTS (Jammy Jellyfish)*. 2023. URL: https://www.releases.ubuntu.com/22.04/. Used 2023.28.11.

[15] Canonical Ltd. *cloud-init: The standard for customising cloud instances*. 2023. URL: https://cloud-init.io/. Used 2023.28.11.

[16] Karl-Andreas Turvas. *jamlab-terraform*. 2023. URL: https://github.com/JamFox/jamlab-terraform. Used 2023.28.11.

[17] Hashicorp. *Nomad Reference Architecture*. 2023. URL: https://developer.hashicorp.com/nomad/tutorials/enterprise/production-reference-architecture-vm-with-consul. Used 2023.28.11.

[18] Inc. Red Hat. *Ansible is Simple IT Automation*. 2023. URL: https://www.ansible.com/. Used 2023.28.11.

[19] Karl-Andreas Turvas. *jamlab-ansible*. 2023. URL: https://github.com/JamFox/jamlab-ansible. Used 2023.28.11.

[20] Karl-Andreas Turvas. *ansible-parallel*. 2023. URL: https://github.com/JamFox/ansible-parallel. Used 2023.28.11.

[21] The Netfilter. *netfilter: iptables*. 2023. URL: https://www.netfilter.org/projects/iptables/index.html. Used 2023.28.11.

[22] Jordan Webb. *The container orchestrator landscape*. 2022. URL: https://lwn.net/Articles/905164/. Used 2023.28.11.

[23] Hashicorp. *Nomad by Hashicorp*. 2023. URL: https://www.nomadproject.io/. Used 2023.28.11.

[24] The Linux Foundation and The Kubernetes Authors. *Kubernetes: Production-Grade Container Orchestration*. 2023. URL: https://kubernetes.io/. Used 2023.28.11.

[25] Hashicorp. *Consul by Hashicorp*. 2023. URL: https://www.hashicorp.com/products/consul. Used 2023.28.11.

[26] Brian Shumate. *Ansible-Nomad*. 2023. URL: https://github.com/ansible-community/ansible-nomad. Used 2023.28.11.

[27] Brian Shumate. *Ansible-Consul*. 2023. URL: https://github.com/ansible-community/ansible-consul. Used 2023.28.11.

[28] Inc. Smallstep Labs. *Step CA: Open-Source Certificate Authority & PKI Toolkit*. 2023. URL: https://smallstep.com/certificates/. Used 2023.28.11.

[29] Podman. *Podman*. 2023. URL: https://podman.io/. Used 2023.28.11.

[30] Docker Inc. *Docker Engine overview*. 2023. URL: https://docs.docker.com/engine/. Used 2023.28.11.

[31] Hashicorp. *Managing External Traffic with Application Load Balancing*. 2023. URL: https://developer.hashicorp.com/nomad/tutorials/load-balancing/external-application-load-balancing. Used 2023.28.11.

[32] HAProxy Technologies. *HAProxy: The Reliable, High Performance TCP/HTTP Load Balancer*. 2023. URL: http://www.haproxy.org/. Used 2023.28.11.

[33] Electric Frontier Foundation. *Certbot*. 2023. URL: https://certbot.eff.org/. Used 2023.28.11.

[34] Internet Security Research Group (ISRG). *Let's Encrypt*. 2023. URL: https://letsencrypt.org/. Used 2023.28.11.

[35] Inc. Cloudflare. *Cloudflare Tunnel*. 2023. URL: https://developers.cloudflare.com/cloudflare-one/connections/connect-networks/. Used 2023.28.11.

[36] Frank Schröder and Education Networks of America. *Fabio LB*. 2022. URL: https://fabiolb.net/. Used 2023.28.11.

[37] Hashicorp. *Load Balancing with Fabio*. 2023. URL: https://developer.hashicorp.com/nomad/tutorials/load-balancing/load-balancing-fabio. Used 2023.28.11.

[38] Hashicorp. *Consul service mesh*. 2023. URL: https://developer.hashicorp.com/consul/docs/connect. Used 2023.28.11.

[39] Joel Spolsky. *The Law of Leaky Abstractions*. 2002. URL: https://www.joelonsoftware.com/2002/11/11/the-law-of-leaky-abstractions/. Used 2023.28.11.

[40] SUSE Rancher. *SUSE Harvester: Hardware and Network Requirements*. 2023. URL: https://docs.harvesterhci.io/v1.2/install/requirements#hardware-requirements. Used 2023.28.11.