Tero Marttila

Design and Implementation of the `clusterf` Load Balancer for Docker Clusters

School of Electrical Engineering

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo 10.10.2016

Thesis supervisor:

Prof. Raimo Kantola

Thesis advisor:

PhD. Pasi Sarolahti

Aalto University
School of Electrical
Engineering

AALTO-YLIOPISTO
SÄHKOTEKNIIKAN KORKEAKOULU

DIPLOMITYÖN
TIIVISTELMÄ

Tekijä: Tero Marttila

Työn nimi: Docker-ryppäiden `clusterf` -kuormantasaajan suunnittelu ja toteutus

Päivämäärä: 10.10.2016 Kieli: Englanti Sivumäärä:7+97

Professuuri: Tietoverkkotekniikka Koodi: S38

Valvoja: Prof. Raimo Kantola

Ohjaaja: FT Pasi Sarolahti

Docker tarjoaa omavaraisia sovelluslevykuvia joita voidaan siirtää eri tietokoneille ja tavan suorittaa kyseisiä sovelluslevykuvia Docker konttien muodostamissa eristetyssä sovellusympäristöissä. Konttialustat tarjoavat pilvessä ajettavien skaalautuvien sovelluspalveluiden käyttöönottoon vaadittuja peruspalveluita, kuten orkestrointia, verkkoviestintää, palvelinetsintää sekä kuormantasausta.
Tämä työ tutkii konttiverkkojen arkkitehtuuria sekä olemassaolevia pilvikuormantasaajatoteutuksia luodakseen skaalautuvan Linux IPVS-toteutukseen pohjautuvan verkkotason kuormantasaajan toteutusmallin. Tämä `clusterf` kuormantasaaja käyttää kaksikerroksista kuormantasausmallia joka yhdistää eri kuormantasausmenetelmiä saavuttaakkseen sekä skaalautuvuuden että yhteensopivuuden olemassaolevien Docker konttisovelluksien kanssa. Työssä toteutetaan hajautettu ohjauspinta joka tarjoaa automaattisen kuormatasauksen Docker konttisovelluksille.
Tämän `clusterf` kuormantasausmallin toteutusta arvioidaan erillisessä ympäristössä mittaamalla Linux IPVS-toteutuksen tarjoama suorituskyky. Tämän `clusterf` kuormantasaajan skaalautuvuus arvioidaan käyttäen Equal-Cost Multi-Path (ECMP) reititystä sekä IPVS-yhteystilojen synkronointia.
Tutkimustyön tuloksena nähdään että verkkotason IPVS kuormantasaaja suoriutuu huomattavasti paremmin kuin sovellustason HAProxy kuormantasaaja samassa konfiguraatiossa. Tämä `clusterf` toteutus mahdollistaa myös kuormantasaajan skaalautumisen, sallien yhteyksien siirtämisen kuormantasaajalta toiselle.
Nykyinen `clusterf` toteutus perustuu epäsymmetrisen reitityksen käyttöön, joka onnistuu Ethernet-pohjaisessa paikallisverkoissa. Toteutuksen laajentaminen mahdollistaakseen käyttöönoton erilaisia verkkototeutuksia käyttävien pilvialustojen päällä sallisi `clusterf`-kuormantasaajan käytön osana yleistä konttialustaa.

Avainsanat: Containers, Docker, Cloud, Load Balancing, DSR, NAT, Service Discovery

AALTO UNIVERSITY
SCHOOL OF ELECTRICAL ENGINEERING

ABSTRACT OF THE
MASTER'S THESIS

Author: Tero Marttila

Title: Design and Implementation of the `clusterf` Load Balancer for Docker Clusters

Date: 10.10.2016 Language: English Number of pages:7+97

Professorship: Networking Technology Code: S38

Supervisor: Prof. Raimo Kantola

Advisor: PhD. Pasi Sarolahti

Docker uses the Linux container `namespace` and `cgroup` primitives to provide isolated application runtime environments, allowing the distribution of self-contained application images that can be run in the form of Docker containers. Container platforms provide the infrastructure needed to deploy horizontally scalable container services into the Cloud, including orchestration, networking, service discovery and load balancing.

This thesis studies container networking architectures and existing Cloud load balancer implementations to create a design for a scalable load balancer using the Linux IPVS network-level load balancer implementation. The `clusterf` load balancer architecture uses a two-level load balancing scheme combining different packet forwarding methods for scalability and compatibility with existing Docker applications. A distributed load balancer control plane is implemented to provide automatic load balancing for Docker containers.

The `clusterf` load balancer is evaluated using a testbed environment, measuring the performance the network-level Linux IPVS implementation. The scalability of the `clusterf` load balancer is tested using Equal-Cost Multi-Path (ECMP) routing and IPVS connection synchronization

The result is that the network-level Linux IPVS load balancer performs significantly better than the application-level HAProxy load balancer in the same configuration. The `clusterf` design allows for horizontal scaling with connection failover between IPVS load balancers.

The current `clusterf` implementation requires the use of asymmetric routing within a network, such as provided by local Ethernet networks. Extending the `clusterf` design to support deployment onto existing Cloud infrastructure platforms with different networking implementations would qualify the `clusterf` load balancer for use in container platforms.

Keywords: Containers, Docker, Cloud, Load Balancing, DSR, NAT, Service Discovery

Symbols and Abbreviations

BGP	Border Gateway Protocol
DHCP	Dynamic Host Configuration Protocol
DNS	Domain Name System
DNAT	Destination Network Address Translation (NAT)
DSR	Direct Server Return (load balancing)
ECMP	Equal-Cost Multi-Path (routing)
HTTP	HyperText Transfer Protocol
IaaS	Infrastructure as a Service
ICMP	Internet Control Message Protocol
IP	Internet Protocol
IPv4	Internet Protocol Version 4
IPv6	Internet Protocol Version 6
LVS, IPVS	Linux Virtual Server, IP Virtual Server (Linux)
NAT	Network Address Translation
OSPF	Open Shortest Path First
PaaS	Platform as a Service
RFC	Request For Comments (IETF)
SaaS	Software as a Service
SNAT	Source Network Address Translation (NAT)
TCP	Transport Control Protocol
TLS	Transport Layer Security
UDP	User Datagram Protocol
VIP	Virtual IP
VRRP	Virtual Router Redundancy Protocol
VXLAN	Virtual eXtensible LAN

Aknowledgements

I would like to thank Prof. Jörg Ott for providing me with the time and resources necessary to study Docker container platforms and develop the `clusterf` load balancer at the Aalto University Department of Communications and Networking.

I would like to thank my parents and family for their support in getting me to sit down and actually write this thesis, from the first few pages to the very last week of editing.

I would like to thank my supervisor Prof. Raimo Kantola and advisor PhD. Pasi Sarolahti for their comments and review of my thesis text.

I would like to thank my colleague at Comnet MSc. Arseny Kurnikov for their support in the implementation and testing of the `clusterf` load balancer.

This work was supported by the Academy of Finland project "Cloud Security Services" (CloSe) [1] at Aalto University Department of Communications and Networking.

Otaniemi, 09.10.2016

Tero Marttila

[1] `https://wiki.aalto.fi/display/CloSeProject/`

1 Introduction

Cloud computing can be defined as the combination of three different service models: Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS) [1]. An IaaS Cloud uses Virtual Machines (VMs) to allocate and share physical machine resources between multiple operating systems by virtualizing low-level resources such as disk controllers, network interface cards and bootloaders. The provider of an IaaS Cloud platform controls both the physical network and server infrastructure. A PaaS Cloud provides a high-level application environment including the tooling and infrastructure required to provide software applications and services in the Cloud (SaaS). Application containers can be used as the foundation of such a platform for horizontally scalable cloud services. Container platforms designed to be portable across different infrastructures can be used for the local development of cloud applications and deployment onto cloud server infrastructures.

Containers allocate and share operating system resources between multiple processes by virtualizing high-level resources such as process management interfaces, the TCP/IP network stack, or the filesystem namespace. Mature container-based platforms such as Google's Borg [2] demonstrate the strengths of containers when operating cloud services at large scales. Google Borg is the primary container platform used to deploy thousands of different applications across clusters of tens of thousand of servers within the Google infrastructure [2]. Google Borg is used to run a wide variety of applications in the form of containers, including end-user facing services such as Gmail and Google Web Search, infrastructure services such as BigTable, and batch-processing infrastructure such as MapReduce [2].

The history of mature container-based platforms such as Google Borg can be traced back to the implementation of shared resource isolation primitives for optimizing server resource utilization [2]. Critical production services require the reservation of sufficient resources for handling peak loads, leading to poor resource utilization during idle periods. Robust resource isolation primitives allow these idle resources to be used by lower-priority batch jobs that are insensitive to latency and can be quickly preempted on demand [3]. However, the development and deployment of services in the form of distributed applications requires systems to support their management and operation. The rapid evolution of support services within Google's Borg ecosystem demonstrated the applicability of the application container abstraction for more than just managing shared resources. Over time, the nature and significance of these early container-based platforms expanded, forming an Application-oriented Infrastructure platform [4].

Docker is an open-source Linux container management system that is rapidly gaining adoption in the development of Cloud applications [5]. Docker emphasizes the application-oriented aspect of containers, with containers being used to operate individual applications rather than opaque virtual machines. Docker Images can be

used to package applications into a single easily distributable image, including all runtime resources and dependencies. Docker combines the ease of building Docker images when developing applications with the flexibility of running Docker Containers when deploying applications. Docker Containers provide an isolated runtime environment for each application, such that the application images do not need to include any low-level components such as disk management or network configuration. The Application Container model simplifies the deployment of applications by providing an abstraction layer where Docker manages the infrastructure resources such as the storage and network. The Docker Engine can be used to deploy any application developed as a Docker Image onto any developer machine or cloud server configured to run Docker Containers.

While Docker presents a solution for many of the problems related to application development, deploying container-based applications into the Cloud requires the support of a container-oriented cloud infrastructure platform [6]. An Application-oriented Infrastructure platform [4] supports the development and deployment of distributed cloud-native applications, allowing them to scale horizontally in the Cloud. Such an application infrastructure platform requires numerous components, including a container engine for running applications, an orchestration system to manage containers, a service discovery mechanism to allow containers to communicate across the network, and a load-balancing layer to distribute traffic across multiple service instances.

1.1 Research Problems

The objective of this work is the implementation of a scalable network-layer load-balancer within the scope of a cluster of machines running Docker applications. Docker container networking concepts and existing implementations of cloud load balancers such as Google Maglev [7] and Microsoft Ananta [8] are studied to construct a design for a scalable network-level load balancer using Linux IPVS. We implement the `clusterf` control plane to integrate this network-level load balancer design with Docker, providing automatic load balancing for container services within a Docker cluster. The research questions for this work are:

- Do network-level load balancing methods have scalability advantages over application-level load balancing methods?
- How is the design of a network-level load balancer determined by the network architecture?

1.2 Thesis Structure

This work is structured as follows. Chapter 2 presents a theoretical model for a container platform with a focus on container networking, discussing orchestration, service discovery and load balancing methods. Chapter 3 studies various implementations of Docker container platform components, focusing on components relevant to networking and load balancing. Chapter 4 presents the implementation of the `clusterf` load balancer for Docker clusters, using a design for a scalable network-level load balancing based on the Linux IPVS implementation discussed in Section 3.4.5. Chapter 5 discusses the methods used to evaluate the design and implementation of the `clusterf` load balancer, presenting and analyzing the results in Chapter 6. Finally, we conclude with Chapter 7, answering our research questions based on the design work in Chapter 4 and the results in Chapter 6.

2 Background

The dynamic nature of Cloud computing leads to changes in application design [9] when scaling an application from a developer's machine to a service with millions of users. For a traditional application running on a single machine, vertical scaling involves upgrading the machine for improved performance and reliability. The vertical scaling approach keeps the application software simple, but it is ultimately limited by the cost of sufficiently powerful and reliable hardware. Cloud infrastructure uses a different approach, running large numbers of commodity server machines optimized for cost and efficiency over reliability. For a distributed application capable of running across multiple machines, horizontal scaling involves increasing the number of machines for improved performance and reliability.

Horizontal scaling of a service involves additional complexity due to the need for a distributed systems design. Increasing the capacity of the service requires a method for dynamically distributing load across multiple instances, allowing the addition of more instances to increase capacity. However, increasing the number of independent instances also means that it is more likely for some instance to fail. Horizontal scaling also requires a fault tolerance mechanism to maintain the reliability of the service. Any cloud service running only a single instance is limited in terms of both performance and reliability.

This Chapter provides an overview of container platform infrastructure components that ease the development of horizontally scalable applications using containers. Containers provide isolated runtime environments for application, allowing the use of self-contained application images and immutable infrastructure for deterministic deployments. Container network allows container applications running within a cluster of machines to connect to services exposed by other containers within the same cluster, while also allowing those containers to be run on any networked machine. Service discovery allows container applications to resolve the current network location of a service, even as new containers providing the service are started, or old ones are removed. Load balancing allows the deploying of horizontally scalable applications, distributing processing workload across multiple service instances.

A design for a generic container platform allows application architectures developed in the form of application containers on a local machine to be deployed across heterogneous cloud platform infrastructures. The design of a generic container platform must allow for implementation differences across different local development environments and cloud infrastructure platforms. These differences are particularly apparent in the design of container network architectures, requiring the use of techniques such as Network Address Translation (NAT) and overlay networking.

Using a container platform as an abstraction layer for the deployment of cloud applications allows the use of cloud infrastructure services as commodity resources, easing the portability of cloud service deployments across competing cloud infras-

tructure providers. Applications dependent on specific cloud infrastructure services within a cloud provider's platform lead to cloud vendor lock-in. [10]

2.1 Docker Containers

Containers can be considered as a lightweight alternative to virtual machines, shifting the virtualization layer from low-level machine resources to the virtualization of operating system resources. Containers extend the typical multi-processing model of operating systems to further isolate groups of processes by virtualizing their view of system resources such as the process management interfaces, the TCP/IP network stack, or the filesystem namespace. Processes executing within different containers all share the same operating system kernel, but cannot see resources outside of their container.

Docker is a container management tool which provides an image format for distributing self-contained application bundles, and a container runtime for executing the application images within isolated runtime environments using Linux containers [11]. Docker's philosophy of Application Containers differs from traditional container management systems in that Docker containers are designed to execute individual applications, rather than a traditional operating system. The Docker Engine initializes each container with pre-configured runtime environment, allowing the use of minimal application images as small as a single statically built binary. Docker application Images are designed to be portable across different machines and infrastructures, providing a consistent runtime environment whether run on a developer machine or on cloud servers.

Docker's images are designed to be self-contained application bundles that can be distributed and run on any architecture-compatible Linux machine without any dependencies on the host environment. Images are built from a source-language `Dockerfile` file format which describes the base image to extend, and the execution steps used to set up the desired environment within the container. Each image also includes metadata about the runtime environment for the container, including the application command line to run within the container, additional filesystem volumes to mount for storing data, network TCP/UDP ports exposed by the service, and additional metadata labels. The application runtime interface provided by Docker is that of a generic UNIX process, meaning environment variables and an executable binary with arguments (or shell command). This avoids the need to impose any requirements for the use of a specific application-level framework, and maintains compatibility with existing applications intended to be run as UNIX services.

On Linux, Docker uses the kernel's `cgroup` [12] and `namespace` [13] facilities to create an isolated environment for the container. Linux cgroups are mainly used to manage resource limits for contained processes, whereas the different namespaces virtualize various aspects of the runtime environment. The Mount namespace is used

to present a copy of the Docker image as the root filesystem for the container, as well as internal system mounts and optional persistent data volumes. The Network namespace is used to create and configure a separate in-kernel networking context for the container, including an isolated set of network interfaces, IP routing tables, netfilter firewall rules, and TCP/UDP ports.

Docker's Application Containers philosophy is evident in the way that these namespaces are configured and managed. Both the mount and network namespaces are fully initialized and configured by Docker when creating the container, and thus the application images can be kept free of any infrastructure-specific artifacts such as bootloader or network configuration. The Docker engine also provides facilities for the administration, centralized logging and monitoring of application containers, avoiding the need for running auxilliary components such as traditional SSH or Syslog daemons within each container.

The behavior of Docker's filesystem namespaces is also related to various aspects of application design that arise when developing applications for deployment on cloud infrastructure [9]. Docker's container runtime environment is designed to give each container a private and ephemeral copy of the image: any changes to files in the Docker image are invisible to other containers, and are discarded when the container is restarted. Immutable Infrastructure means that rather than modifying the files included in the image, the running container is replaced with a new container running an updated version of the image. Infrastructure as Code means that the updated images are built from the modified Dockerfile, which can be managed as code using a Version Control System. Applications must therefore enforce a separation between code and data, by configuring explicit data volumes for any files written by the application. Docker can then provide various infrastructure-specific volume drivers to provide persistence for these data volumes across container restarts.

2.2 Orchestration

When moving from application development to operating a production service, performance and reliability requirements can quickly grow beyond the scale of a single machine. Where an operating system manages the tasks running on a single machine, orchestration manages the tasks running across an entire cluster of machines: a distributed operating system.

The Mesos [14] platform is an example of an orchestration system that serves as a foundation for the Mesosphere DC/OS distributed operating system. Both Apache Mesos and Google Borg [4] share a common motivation of efficiently sharing machine resources between different kinds of distributed applications. Both systems replaced specific distributed batch processing frameworks (Google's Global Work Queue and Apache's Hadoop) with more general mechanisms suitable for running other workloads such as latency-sensitive production services. Whereas Google Borg

is a monolithic system with a single master and agent process, Apache Mesos decomposes the system into a minimal master reponsible for resource allocation, external schedulers and modular executors. A scheduler is responsible for assigning tasks to machines, and an executor is responsible for executing tasks in an isolated environment. Different Mesos frameworks can then provide optimal scheduling strategies and execution environments for different workloads sharing a cluster of machines.

Clusters consist of large numbers of machines interconnected by a network. Both Borg [4] and Mesos [14] discuss cluster sizes on the order of tens of thousands of machines. The orchestration system consists of a central master for the cluster, and an agent process on each machine. The agents register with the master, which maintains a registry of available machines and their resources, in terms of quantities such as CPUs, memory and disk. To execute tasks, the master provides a management interface for users to submit job specifications, which define the characteristics and resource requirements of the tasks to execute. The master schedules the job onto a set of machines matching the job's constraints, and hands the job over to the machine agents for task execution. The machine agent then executes each task, using an isolation mechanism such as Linux containers to enforce the resource allocation for different tasks sharing the same machine. The machine agent must also track the lifetime of the task, in order to release its resources for further use once it exits. The master must deal with unexpected changes to jobs, such as task failure or machine failure.

The specific scheduling requirements for an orchestration system vary depending on the different kinds of workloads being run. Batch processing workloads such as Hadoop can be optimized using data locality in order to minimize inter-machine network traffic. When the input dataset for a job is distributed across disk storage on the cluster machines, tasks should ideally be run on the same machine that their slice of the input dataset is stored on. For batch-processing pipelines, the scheduler must also be aware of task completion in order to start further jobs which depend on the output of the completed task. For long-running production services, support for runtime support configuration changes to existing jobs is highly desireable, such as rolling upgrades to individual tasks to minimize service downtime.

The task execution environment is fundmental in achieving the dual benefits of performance scalability combined with efficient resource utilization, which relies on isolation between mixed-priority tasks sharing machine resources. Critical production services that handle network requests are highly latency-sensitive, while batch processing workloads are generally insensitive to latency and considered best-effort. High-priority production services must be scheduled with sufficient guaranteed resources to handle peak load, which leaves significant amounts of resources underutilized during idle periods. With an execution environment capable of providing sufficiently strict task isolation and scheduling, this otherwise idle capacity can be utilized by overcomitted resources for lower-priority tasks, which can be preempted under higher-priority task load spikes [4].

Both Apache Mesos and Google Borg use a task execution environment based on Linux containers with their cgroup [12] features to provide the necessary performance isolation. Using Linux containers with their namespace [13] features also solves the operational aspects of executing tasks on different machines, as the semantics provided by application containers align closely with the concept of a workload unit to be scheduled onto a cluster of machines. Container images provide the application portability needed to transparently execute arbitrary applications on arbitrary machines.

2.3 Container Networking

Container platforms using Linux network namespaces allow each container to behave as a virtual network host with independent network interfaces, IP addresses, routes and TCP/UDP port numbers. Providing each application container with its own network identity can be considered a design requirement for a container platform [4]. This requirement is discussed in more detail in Section 2.4 on Service Discovery, considering the additional complexity introduced by sharing a common TCP/UDP port space between different containers.

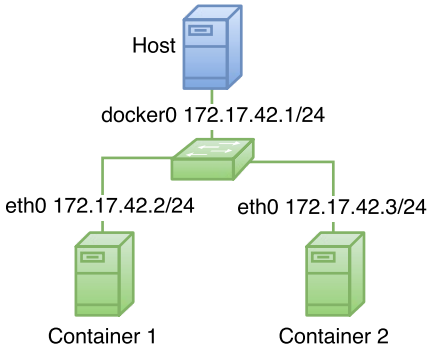

Figure 1: Minimalist container networking architecture, with the host machine in blue, and the virtual container network in green.

The minimalist container networking architecture shown in Figure 1 contains the minimum of networking components used to allow containers to communicate. The host machine is used to run multiple Docker containers, as discussed previously in Section 2.1 on Docker Containers. A virtual container network is created, using a virtual Ethernet bridge (`docker0`) as described in Section 2.3.1 on Ethernet Bridging. The container network uses a subnet of Internet Protocol (IP) network addresses to allow each container having an individual network address to commnicate, as described in Section 2.3.2 on IP Routing. Expanding the container network to communicate with other machines outside the container platform typically uses Network Address Translation (NAT), as described in Section 2.3.3 on NAT. Expanding the container network to communicate with container networks on other

machines within the container platform typically uses overlay network tunnels, as described in Section 2.3.4.

Container platforms designed to scale from a single developer machine to a horizontally scalable cloud deployment must consider many different scenarios, including a developer's machine roaming between different networks, a variety of cloud infrastructures, or a dedicated network designed for container workloads. The network abstractions exposed to the application containers must remain independent of the infrastructure used to run the containers. A network implementation must not arbitrarily limit the number of containers that can be run on a machine, such as requiring a DHCP lease on the machine's local network for each container. It should also be possible to run the same service as a single container on a development machine, or as large number of horizontally scaled instances scheduled to run on multiple cloud servers.

2.3.1 Ethernet Bridging

A link-layer Ethernet network consists of Ethernet interfaces connected using either a point-to-point link or an Ethernet bridge to forward packets between multiple links. Ethernet networks use a flat address space with 48-bit Ethernet addresses, where any Ethernet address may be used anywhere within the network. Each Ethernet interface connected to the network must have a unique Ethernet address, which are typically pre-assigned when the interface is created. Ethernet networks are modeled as a shared broadcast domain where each transmitted packet is potentially received by all connected interfaces, with spurious packets being discarded. Ethernet packets sent to the broadcast destination address are processed by all interfaces, and provide a means for a host to discover other connected interfaces. The advantage of Ethernet networks is in their simplicity and flexibility, as interfaces can be attached, detached or moved within the network without the need for any reconfiguration. [15, Section 4.3]

Ethernet bridges can minimize flooding by only forwarding unicast traffic to the port closest to the destination, but this requires every bridge on the network to learn all connected interface Ethernet addresses. Ethernet packets sent to unknown address or the broadcast address are flooded to all connected links. The volume of broadcast traffic and the requirement for each bridge to have global knowledge of all connected interfaces limits the ultimate scalability of Ethernet networks. The standard broadcast flooding algorithm always requires a loop-free topology, requiring all traffic across the network to share the same spanning tree of links. [15, Section 4.7]

Traditional machine-level virtualization uses a physical machine to run a virtualization hypervisor, virtualizing the physical machine's resources for use by multiple virtual machines. Virtual machines generally use Ethernet as the networking interface between the hypervisor and virtual machine. The hypervisor exposes a virtual

Ethernet interface to the virtual machine. The virtual machine's operating system uses the virtual Ethernet interface to send Ethernet packets, which are processed by the hypervisor. A common configuration is to use a virtual Ethernet bridge within the hypervisor to forward traffic between the machine's physical Ethernet interface and the internal virtual machine Ethernet links, forming a single Ethernet network spanning both virtual and external machines.

Linux container networking is generally also implemented using virtual `veth` Ethernet interface pairs forming a virtual point-to-point link. A newly created network namespace only contains the default loopback device, and processes running within such an empty network namespace will not have any network connectivity. When creating a new container, a pair of `veth` interfaces is created, and one of the paired network interfaces is moved into the container's network namespace. The remaining `veth` interface is typically attached to a virtual Ethernet bridge on the host machine. This configuration forms a internal virtual Ethernet network within the host, allowing containers to communicate with other containers and the host machine. [16]

It is also possible to connect containers to an external Ethernet network. As in virtual machine networking, the machine's external network interface can be attached to the virtual Ethernet bridge used for container networking. Alternatively, Linux allows the creation of virtual `vlan`, `macvlan` and `ipvlan` devices for a network interface, which can be moved into the container's network namespace. The use of `macvlan` devices can provide significant performance advantages over the use of `veth` interface pairs with a virtual bridge [17]. However, this requires that the external Ethernet network allows the use of multiple virtual Ethernet addresses for packets forwarded across the external network interface. Using Ethernet networking techniques such as `macvlan` for container networking on cloud virtual machines is not always possible: [16]

> Kubernetes is unable to use MACVLAN since most software defined networks (including Google Compute Engine's) don't support this option.

Using Ethernet bridging for virtual machine networking has the benefit of allowing the use of live migration. Live migration allows a running virtual machine to be moved between different physical machines within the same virtualization cluster without restarting the virtual machine or changing the virtual machine's network address. A virtualization cluster supporting live migration requires a shared network infrastructure, allowing the virtual machine's Ethernet network interface to be reattached to a different location within the cluster network. Containers generally do not support live migration, and any method of migrating a container from one machine to another involves restarting the container. Container platforms used to deploy horizontally scaled applications can use dynamic service discovery and load-balancing, dynamically routing traffic for container services to service instances with dynamic network addresses within the cluster.

2.3.2 IP Routing

Network-layer IP routing differs from Ethernet networks in the use of a hierarchial addressing scheme, where hosts are assigned addresses based on their location within the network. Global IP addresses are allocated in large chunks for use by an Autonomous System (AS), which may be further subdivided into smaller subnets routable by routers within the AS. Each AS presents a coherent routing policy to neighboring AS routers, whereby routers in a different AS do not need to be aware of the interior network structure. The use of hierarchially assigned network address prefixes allows IP networks to scale to the size of the Internet by aggregating the routing information for large quantities of individual host addresses into a single fixed-length routing entry. The use of global IP addresses requires each attached host to be configured with a unique IP address allocated and routed by the network it is connected to. [15, Section 5.6.2]

IP network addresses are allocated and routed based on a bitwise prefix, which can be represented using the Classless Inter-Domain Routing (CIDR) address notation. The CIDR network prefix `192.0.2.0/24` matches all 32-bit IPv4 addresses with a 24-bit prefix matching `192.0.2.0`, covering 256 addresses from `192.0.2.0` to `192.0.2.255`. [15, Section 5.6.2]

IP network packets between hosts are forwarded over a link-layer protocol such as Ethernet. Each Ethernet interface on a network host is configured using a local address and subnet mask, which can be expressed using the CIDR notation `192.0.2.100/24`. IP hosts use the Address Resolution Protocol (ARP) to resolve the Ethernet address of IP hosts connected to the same network [15, Section 5.6.3]. The host will send an ARP query on that interface for any destination address within `192.0.2.0/24`, using the Ethernet broadcast destination address and the local IP source address `192.0.2.1`. Hosts connected to an Ethernet network will respond to queries for their configured interface address, returning an ARP reply using unicast Ethernet addresses. Communication between IP hosts connected to the same Ethernet network is straightforward, assuming that all hosts have a matching interface route and use source addresses within the network prefix.

An IP router with multiple interfaces connecting different networks must be capable of forwarding incoming packets between interfaces. IP routers use a set of configured routing rules to forward packets with a matching IP destination address via the correct interface and link-level destination. Each routing rule contains the CIDR prefix for matching the destination address, the outbound interface and the next-hop address. The routing table can contain multiple overlapping routes for an address, and the route with the longest matching prefix is used. A default route of `0.0.0.0/0` matches all destination addresses that do not have a more specific route.

An IP host with a single network interface and IP network address only needs to process packets associated with its own local address, requiring a minimal routing

table. A machine being connected to a network uses a mechanism such as Dynamic Host Configuration Protocol (DHCP) to configure the local interface address and routes [15, Section 5.5.3]. A DHCP client on the machine uses Ethernet broadcast messages to request a DHCP lease from a DHCP server attached to the Ethernet network, which allocates an appropriate IP address within the network for use by the client. The DHCP lease will contain the IP address used to configure the local interface, a subnet mask used to configure the interface route for the local network, and a default gateway used to configure a default route for reaching all other IP destinations outside of the local network. Different configuration methods can be used for virtual machines in cloud environments, but the resulting IP host routing is generally similar.

The scalability of IP networks relies on the centralized allocation of network addresses, which comes at the cost of requiring explicit mechanisms for allocating addresses and configuring routing rules. Each IP host must be allocated a unique IP address within the local network, each local network must be allocated a unique subnet within the AS, and each AS must be allocated a unique network within the Internet. For two hosts connected to different IP networks to communicate, they must both use appropriate local address routable by every IP router on the path between the endpoints. If a host attempts to use an inappropriate local source address, any reply packets will not be routed back to the host.

Routers use routing control protocols to exchange routing information, originating local routes and propagating routes advertised by other routers. Routers within an AS will generally use an interior routing protocol to exchange specific routes for each network within the AS, ensuring optimal routing for packets within a reasonably sized network. Routers in different ASs will use the Border Gateway Protocol (BGP) to originate aggregated routes for their interior networks, and propagate router advertisements from other ASs. [15, Section 5.5.6]

In order for a router to provide IP connectivity to hosts on a local network, a network subnet must be allocated and advertised for routing. While DHCP can be used to automatically allocate individual addresses for hosts within an Ethernet network, using ARP to providing zero-configuration routing, no similar IPv4 protocol exists for automatically allocating entire network subnets for use by a router. The limited availability of IPv4 addresses makes it difficult to provide any dynamic mechanisms for address allocation, and the additional complexity of dynamic routing is rarely needed outside of carrier and dedicated server networks. The IPv4 access networks at the edge of the Internet are thus typically configured to only provide host-level connectivity, lacking any option for a host to upgrade itself into a network router, providing IP network connectivity for multiple local virtual machines or containers.

2.3.3 Network Address Translation

Network Address Translation (NAT) allows an IP router to use normal host configuration mechanisms such as DHCP for external network configuration, while also providing routing for an internal network of hosts. NAT avoids the requirement for allocating globally routable addresses by introducing a separate internal network address space. NAT avoids the requirement for neighboring routers to have any routes for the internal network addresses by modifying the packets being forwarded, using the external network host source address when forwarding packets to the external network. A NAT router can operate using a single external network address, while allowing the use of arbitrary private RFC1918 [18] address space on the internal network. [19]

Compared to the stateless nature of IP routing, where every packet contains sufficient header information to be forwarded independently of other packets, NAT routing is stateful. When an outgoing packet from an internal network host is forwarded by replacing the internal source address with the external network address, the incoming reply packet on the external network interface will not contain the internal network address of the originating host. Sharing the same external network address for all outgoing packets requires the NAT router to use additional transport-layer header information from incoming packets to determine the internal network destination address for forwarding. Whereas the NAT router must be able to determine the internal network host address from the incoming packet's 5-tuple of network-layer addresses and transport-layer port numbers, the same transport port number space is also used independently by each internal network host. NAT is thus closely tied to transport-layer client-server semantics, using SNAT techniques for clients, and DNAT techniques for servers.

Source NAT (SNAT) is used for internal network clients establishing connections to external network servers, sharing the NAT router's external network source address. Each internal network host will independently choose an ephemeral source port for the outgoing connection, and different internal network hosts may use overlapping source ports for connections to the same external server and port. The NAT router will create a connection state recording the internal network host address associated with each outgoing 5-tuple, also rewriting the source port as necessary to guarantee unique 5-tuples. The 5-tuple of each incoming packet will be compared with the connection state table, and an incoming packet matching a recorded state entry will be rewritten to the corresponding internal network host address and port. SNAT routing is transparent to internal network hosts behaving strictly as clients, where the NAT router is able to use and record a unique 5-tuple for each outgoing connection. SNAT does not generally allow internal network hosts to act as servers accepting connections from unkonwn external clients, as an SNAT router will generally discard incoming packets for unknown 5-tuples.

Destination NAT (DNAT) can be used to expose internal network servers by al-

locating specific ports to forward incoming connections to an explicitly configured internal network host. DNAT implementations generally also allow rewriting the internal destination port, allowing the use of multiple distinct external ports to forward connections to multiple internal hosts using the same internal destination port. The use of DNAT requires a configuration mechanism for allocating and configuring the port-forwarding rules, which is problematic in cases where the NAT router and the internal host machines are in different administrative domains.

The stateful nature of NAT routing complicates the design of a scalable network architecture where the capacity or reliability requirements preclude the use of a single router for a given set of internal hosts. When distributing packets between multiple routers, the same router must be used for both the initial connection packet and the reverse path. Only the router having forwarded the initial packet for a connection will have the state required to translate any further packets associated with the same connection, including any reply packets. Any fault-tolerance mechanism requires the NAT connection states to be synchronized between machines, as a restarted NAT router will lose all connection states. Any connections being redirected to a different NAT router lacking the connection state will be cut, forcing clients to reconnect. NAT is best used as close to the edge of the network as possible, limiting the resulting scalability bottlenecks.

Container networking engines designed to run containers on a single machine can use NAT to create an internal network for use by containers. Using NAT allows a host machine to provide internal network connectivity for multiple virtual network hosts without requiring support for address allocation and routing within the external network. The separate internal network address space introduced by NAT gives the virtualization platform the flexibility to manage the internal network addresses as required. The concept of separate internal and external connectivity also fits well with the concept of internal and externally exposed services. Virtual network hosts within the same internal network are able to communicate freely using their internally assigned internal network address. The virtualization platform can provide an integrated mechanism for the configuration of DNAT rules for exposing services on the external network address.

2.3.4 Overlay Networks

NAT allows the use of internal network address space within a single machine without exposing the internal network addresses to the external network. Overlay networks allow the use of internal network address space across multiple machines without exposing the internal network addresses to the external network. Traditional networking methods require a network infrastructure capable of forwarding internally addressed packets between machines, using a link-layer network interconnect capable of forwarding packets with arbitrary internal addresses. Cloud IaaS platforms may provide a separate internal network between virtual machines within

a datacenter, but large cloud platforms typically use custom Software Defined Networking (SDN) implementations. These SDN implementations may either require the use of platform-specified network addresses, or require explicit configuration using a platform-specific method.

Overlay networks form virtual network links between machines using a packet tunneling protocol to encapsulate packets for transport across an external network. When forwarding a packet between internal network addresses on two different machines, a new packet is constructed using the external network addresses of the machines, having the internal packet as the payload. The external network infrastructure forwards the tunnel packet to the destination machine, which de-encapsulates and forwards the internal packet. The external network only inspects the outer packet headers containing external machine addresses, and does not inspect the internal network packet. The tunneling protocol may encrypt the inner packet payload to protect the internal network from packet inspection or spoofing.

Overlay tunnels can be used to forward either link-layer or network-layer packets. Network-layer tunnel are used to route IP packets between machines, forming an IP routing domain connecting virtual IP networks on multiple machines. A network-layer overlay network can allocate a subnet for use by the virtual network hosts on each machine, allowing the use a single routing rule per machine. Link-layer tunnels are used to bridge Ethernet packets between machines, forming larger Ethernet networks connecting virtual Ethernet hosts on multiple machines. A link-layer overlay network can use network addresses within the same IP network across different machines, requiring the use of a forwarding rule per virtual Ethernet host.

Virtual eXtensible LAN (VXLAN) is a link-layer overlay networking protocol using UDP to transport Ethernet packets over IP unicast and multicast. VXLAN is designed for large-scale networking of virtual machines in a physical multi-tenant infrastructure, using IP multicast to implement broadcast flooding and learning similar to a standard Ethernet bridge. The Linux kernel includes an implementation of VXLAN, using either IP multicast for dynamic learning, or a userspace control plane. VXLAN can be used as an alternative to custom tunneling protocols used in userspace implementations of overlay networking, where the in-kernel VXLAN switch offers a reduced packet forwarding overhead. The multicast-based VXLAN control plane can be replaced using a userspace control plane for use with network infrastructure that does not support multicast. Securing the VXLAN tunnels without the use of a dedicated infrastructure network providing strict access control requires the use of IPSec to secure tunneled packets forwarded over untrusted networks.

An overlay network using packet encapsulation must consider the link-layer Maximum Transmission Unit (MTU), which limits the maximum size of a transmitted packet including all headers and payload [15, Section 5.5.7]. The packet payload must fit into the link MTU after prepending any headers, or the payload must be fragmented into multiple smaller packets. Any protocol tunneling packets across an

external network using the same default Ethernet MTU as the internal network will lead to fragmentation, as a sufficiently large internal packet will no longer fit into the external network MTU after adding the encapsulation headers. Tunneling protocols generally avoid the use of fragmented packets, preferring the use of a sufficiently small MTU on the internal network. Determing the correct MTU for the internal network can be difficult if the external network MTU is unknown. For example, the VXLAN protocol does not support the use of fragmented packets, and recommends increasing the external network MTU instead [20, Section 4.3].

2.3.5 IPv6

The IPv6 protocol is designed to address the limitations of IPv4, with a larger 128-bit address space offering new possibilities for address allocation and routing mechanisms [15, Section 5.6.8]. Where IPv4 networks use mechanisms such as NAT to decouple internal network addressing and routing from the external network, IPv6 includes support for new dynamic network address allocation and routing mechanisms such as DHCPv6 Prefix Distribution (PD). A network using DHCPv6 can allocate both globally routable host addresses and networks, allowing hosts to also act as routers. IPv6 address allocation guidelines recommend [21] the allocation of multiple /64 networks for each end site to avoid the requirement for NAT. An IPv6 host could in theory use a dynamic configuration mechanism such as DHCPv6-PD to acquire a sufficiently large routable network of IPv6 addresses for use by local containers.

In practice however, support for the dynamic allocation and routing of multiple /64 networks to end hosts is not universal. Any universal virtual networking architecture must work in network architectures having only a single IPv6 address per host, or lacking any IPv6 addressing at all. While container networking platforms can offer optional support for IPv6, it is unrealistic to design a universal container networking architecture without relying on the use of IPv4 NAT with a separate internal network. The use of NAT with a separate internal network also avoids the need for renumbering the internal network hosts when the host machine roams to a different network. While the IPv6 design includes mechanisms for dynamic renumbering, these are rarely usable in practice.

2.4 Service Discovery

Containers running within a container platform will have an internal network address assigned by the container platform. Container platforms used to deploy network services must support server applications running within containers, allowing each application container to expose services in the form of TCP/UDP ports. Container platforms must allow client applications to connect to services exposed by other

containers, such as an internal database server. In order for a client application to connect to such a service, it must know the network address of the container exposing the service. For services running on dedicated servers, each server may be configured with a static network address, and the clients can simply be configured to connect to this static network address. However, if the service is moved and its network address changes, every client would need to be reconfigured.

A static configuration mechanism is no longer sufficient when deploying horizontally scaled services onto a container platform, as a service may have multiple dynamically allocated network addresses that change over time. Orchestration systems for the automatic scheduling of cluster resources means that we can no longer determine where a particular service will be running ahead of time. A horizontally scaled service will have multiple service instances, with each instance having an individual network address. The set of network addresses will change over time, as new instances are added in order to increase capacity, and failed instances are removed to maintain reliability. Clients will fail to connect to a service if using stale network addresses, requiring a method for clients to dynamically resolve the network address of a service as it changes over time.

A container platform used to deploy service containers must include a service discovery mechanism, registering deployed service instances and providing a protocol for clients to resolve the network addresses of a service. Each service is deployed as a set of application containers, using network addresses assigned by the container platform. A service registration mechanism registers each service instance into a shared service discovery database, updating the set of network addresses for a service as instances are started and stopped. In the case of machine failures within the cluster, any associated service instances must be cleared from the service discovery database.

The Domain Name System (DNS) is the standard mechanim used by client applications to resolve the network address for a service [15, Section 7.1]. A client application connecting to a service is configured with the network address of a DNS resolver, the DNS name of the service, and optionally the port number the service is configured to listen on. Standard operating system libraries provide a method for the client application to resolve DNS names into network addresses, sending DNS `A`/`AAAA` queries to the configured DNS resolver, receiving DNS responses containing network addresses. The client application establishes a connection to the resolved network address, using either a predetermined or explicitly configured port number. A container platform can support the use of DNS for service discovery by configuring the containers to use a dynamic DNS resolver provided by the container platform, using the shared service discovery database to respond to DNS queries for registered services.

Supporting standard client applications using DNS for service discovery requires the container platform to provide each service container with a separate network

address. The standard TCP client-server model involves the use of well-known port numbers for services, where a server providing a service is expected to listen on a predetermined port number. Without the use of container networking namespaces, all of the services running on a machine share the set of available TCP/UDP port numbers, constraining the set of services that can be run on a machine to those configured to listen on distinct ports. Container platforms such as Borg [4] lacking support for network namespaces ultimately require the orchestration system to dynamically allocate ports for use by containers. The use of dynamically allocated ports for services requires client applications to use a service discovery mechanisms capable of dynamically resolving both a service's network address and TCP/UDP port number. While DNS supports the use of `SRV` records to resolve a service name to a set of network addresses and port numbers, standard DNS resolver libraries lack support for `SRV` queries. Using a container platform with a container networking model lacking support for per-container addresses thus requires the use of application-specific service discovery mechanisms for dynamically resolving port numbers. The additional complexity of a container networking model providing per-container network addresses is preferrable to the per-application complexity of alternative service discovery mechanisms [16].

While the use of DNS for service discovery is widely supported by applications, it is dependent on the client application's use of DNS queries to respond to changes in services. DNS uses caching resolvers to optimize scalability and query latency, caching records using a configurable Time-to-Live (TTL) duration to trade performance benefits of caching against the propagation time of updates. DNS caching is less of a problem within a container platform, which may use a local dynamic DNS resolver with low TTLs to quickly propagate updates. If a client application is disconnected from a service, it may reconnect to a different instance of the service by performing a new DNS query. Using DNS for service discovery is problematic in the case of applications that only perform a DNS lookup for the configured service name once at startup, and then continue to use the same resolved address throughout their entire lifetime. An example of this kind of application is the `collectd` agent used for system statistics monitoring, which uses a connectionless UDP protocol, and is unable to notice when a server goes down. Using DNS service discovery for these kinds of applications requires restarting the client processes if the service network address changes. Some applications may not support the use of DNS names, requiring an explicitly configured IP network address.

Implementing service discovery for applications unable to use DNS requires integrating directly with the service discovery database. Service discovery databases generally provide a method to watch for changes to services, following the service state in realtime. One approach for implementing service discovery for applications requiring static configuration is the use of an external tool to manage the application configuration. Such a configuration tool can be used to query the service discovery database, generating an application configuration, reloading the configuration if the service changes. For more advanced needs, the application can also integrate

directly with the service discovery database.

Service discovery only works within a given network domain, where the client is able to connect to the network addresses used by the service. Container network architectures using NAT with a separate internal address space can only use service discovery methods within the internal network. Publishing services to the external network requires the use of DNAT, where connections to a specific port on the machine's external network address are forwarded to an internal service. Such virtual network addresses can also be used as an alternative to service discovery, allowing clients to be configured to connect to the virtual network address, using packet forwarding to redirect each connection to a suitable service instance. The virtual network addresses can be statically allocated by the container platform, implementing the dynamic service discovery mechanisms within the network platform. The use of statically allocated network addresses with dynamic forwarding allows the use of static client configuration mechanisms.

Container platforms using DNAT to publish services on specific network ports encounter similar issues related to port allocation as container platforms that do not use network namespaces. Only one service instance using a well-known port number can be published per machine using DNAT. Dynamically allocating DNAT ports for published services again requires application specific service discovery support for dynamic port numbers. Using virtual network addresses for horizontally scaled services requires a load balancer to forward connections to multiple service instances.

2.5 Load Balancing

Load balancing is a critical component of a container platform used to deploy horizontally scaled services with multiple independent service instances. Load balancing is used to scale the performance of a service by distributing incoming traffic for a service across each instance of the service. Load balancing methods can be used to scale the reliability of a service by rerouting traffic away from any failed instances.

Cloud services use a combination of service discovery and load balancing methods to route traffic for services. Any Internet service intended for access by client applications such as web browsers requires the use of DNS to resolve the service's domain name to a globally routable IP address. This IP address can be provided by a load balancer situated at the edge of the cloud platform, connected to both external and internal networks. The load balancer then associates the incoming traffic with a service, selects a server within the internal network, and forwards the incoming traffic to the selected instance.

Load balancing is closely related to service discovery. Service discovery methods supporting multiple network addresses for a service can be used for client-side load balancing, with each client independently selecting which service instance to use.

Alternatively, a load balancer can be used to provide a virtual network address associated with a service, forwarding the incoming traffic to a service instance. The use of relatively static virtual network addresses for services allows the use of straightforward service discovery mechanisms such as DNS, or even static configuration. The load balancer can be used to implement a more complex service discovery mechanism to dynamically forward traffic to horizontally scaled service instances. The configuration required for a load balancer is similar to the service instance records used for service discovery. A container platform can provide automatic load balancing for services using a dynamic load balancer control plane configured via the shared service discovery database.

The ultimate performance and reliability of a horizontally scaled service is determined by the performance and reliability of the load balancer used to distribute load across the service instances. A scalable service also requires a scalable load balancer. The load balancer data plane can be scaled by combining different load balancing methods across multiple layers of the networking stack:

- DNS load balancing to distribute connections for a service across multiple network addresses
- L3 load balancing to distribute network-layer packets for a network address across multiple network hosts (BGP Anycast, Equal-Cost Multi-Path routing)
- L4 load balancing to distribute transport-layer connections for a TCP/UDP service across multiple servers
- L7 load balancing to distribute application-layer requests for different application-level resources across multiple application servers (HTTP)

2.5.1 Network-layer load balancing

A L3 load balancer uses the network-layer addresses within the packet header for forwarding decisions. Standard network routers can be used for large-scale load-balancing, using standard network-layer routing methods to distribute traffic for a network address across multiple paths to multiple hosts. Routers support the use of routing control protocol for dynamically updated routing rules, and the stateless nature of IP network routing does not require routers to maintain any per-packet state across any topology changes. A stateless L3 load balancer data plane can be efficiently implemented in hardware for greater performance. However, connection-oriented transport protocols such as TCP require connection state at the end hosts, and rerouting TCP packets for an existing connection to a different end host will break any affected connections.

Network routing between AS border routers within the Internet uses the Border Gateway Protocol (BGP) for dynamic routing control [15, Section 5.6.5]. BGP

anycast is a L3 load balancing method, using BGP to advertise the same network address from multiple network locations. Each router within the Internet will use the shortest avilable path for forwarding, causing traffic from clients to be routed to the topologically closest location of the anycast address. If the BGP route for an anycast address is withdrawn from one location, the affected routers will continue forwarding packets using the next-best route for the anycast address, advertised by a different location. BGP anycast is primarily used for geographic load balancing and failover of stateless protocols such as DNS. Using anycast addresses for connection-oriented protocols such as TCP is suspectible to the disruption of connections caused by changes in the network topology, which may cause packets for existing connections to be rerouted to a different end host in a different service location.

Equal-Cost Multi-Path (ECMP) routing provides a highly scalable method for L3 load balancing, allowing incoming packets for a single Virtual IP (VIP) network address to be distributed across multiple paths. A network router having multiple next hops for a route may use Equal-Cost Multi-Path (ECMP) routing to load-balance packets across each path [22]. ECMP routers minimize the disruption of network flows by selecting the forwarding path for a packet using a hash of the packet's network addresses. Stateless ECMP forwarding using hashing can be implemented efficiently in hardware. However, dynamic routing updates that add or remove ECMP forwarding paths will change the output of the hash function used to select the forwarding path, causing packets to be rerouted via different paths.

Using ECMP routing with stateless hashing for L3 load balancing also introduces multiple corner cases for special kinds of traffic. Using transport-layer 5-tuple hashes for ECMP load balancing causes issues with fragmented IP packets, where the first and following fragments of a packet may be forwarded via different paths, complicating any efforts at fragment reassembly [7, Section 4.3]. Incoming ICMP messages returned by routers forwarding outgoing packets from a VIP using L3 load balancing may not be routed back to the same host having sent the outgoing packet, causing issues with path-MTU discovery, particularly for IPv6 connections [23].

2.5.2 Network-level load balancing of Transport-layer connections

Stateless network-layer load balancing methods alone cannot be used for reliable load balancing of stateful transport-layer connections across multiple end hosts. Changes in the network routing topology will cause packets for established connections to be rerouted to different end hosts, leading to broken connections. A network-level L4 load balancer uses both the network addresses and transport-layer port numbers within the packet header for load balancing, forwarding packets for existing connections to the same server host. A L4 load balancer is used to forward traffic for specific TCP/UDP ports on a Virtual IP (VIP) address to a set of backend servers. A L4 load balancer can be used to forward traffic for different TCP/UDP ports on the same VIP to different backend servers, allowing the use of a single external

network address for multiple different services using different port numbers.

A network-level L4 load balancer associates incoming packets with a service configured to accept traffic for the packet's destination network address and TCP/UDP port. For each incoming packet, the L4 load balancer must determine which backend server to forwards the incoming packet to. A L4 load balancer must ensure that every packet for a connection is forwarded to the same backend server. Given a relatively static configuration of backend servers, consistent hashing algorithms can be used to deterministically select a backend server for each incoming packet [7, Section 3]. Connection state tracking is required for reliable backend selection across arbitrary configuration changes. The finite size of any connection tracking table implementation requires the use of connection state tracking and timeouts to evict connection state table entries. Minimizing the impact of load balancer failures within a clustered L4 load balancer performing stateful connection tracking for dynamically configured backends requires the use of explicit connection state synchronization, allowing the migration of connections across load balancers.

There are two main approaches to the forwarding the incoming packets from a L4 load balancer to the backend server. The two approaches differ in whether or not the network addresses within the forwarded packets are modified, either requiring the use of symmetric routing or allowing the use of asymmetric routing for any reply packets from the backend server to the client. While providing symmetric routing for backends behind a single load balancer is straightforward, the requirement for a symmetric return path complicates the design of a scalable L4 load balancer distributing incoming traffic across multiple load balancers.

L4 load balancers using DNAT for forwarding of modified incoming packets replace the destination address of each forwarded packet with the network address of the backend server. This allows the use of unmodified network routing using the locally configured network address of the backend server, without requiring the backend server to be configured with the destination VIP address. However, the stateful nature of NAT requires that any reply packets from the backend server must be processed by the same load balancer having the NAT state formed by the initial incoming packet. Using NAT for load balancing requires a method to provide a symmetric return path from the backend servers via the load balancers. Satisfying the requirement for symmetric routing is trivial in the case of a single load balancer forwarding traffic for backend servers having the single load balancer as their default gateway. Using a cluster of multiple L4 load balancers for load balancing incoming traffic requires additional mechanisms to provide a symmetric return path.

L4 load balancers using Direct Server Return (DSR) forward the unmodified incoming packets to the backend server, preserving the original client source address and destination VIP address. Each backend server having the original source and destination addresses will be able to construct a valid reply packet, allowing the use of an asymmetric return path. The use of asymmetric routing for load balancing means

that packets for the destination VIP will be routed to the load balancer machines, whereas packets with the source VIP will be sent by the backend servers. Using DSR, each backend server may independently route any reply packets using any valid network path, allowing the forwarding of reply packets to be offloaded from the load balancers. However, using DSR for load balancing requires configuration of each backend server to process traffic for each destination VIP address locally. If the load balancer and backend server are connected to the same Ethernet network, the load balancer can use ARP to forward the packet using the destination Ethernet address of the backend server. If the load balancer and backend server are on different networks, a packet tunneling protocol may be used to transport the unmodified internal packet within an external packet using the locally configured network addresses of the load balancer and backend server.

There are pros and cons to both approaches of forwarding incoming load balanced packets. Using DSR simplifies the design of a scalable L4 load balancer, but assumes a network infrastructure allowing the use of asymmetric routing, and requires special configuration of the backend servers. Using NAT allows the use of unmodified backend servers, and provides additional flexibility by allowing the use of port translation to rewrite packets for backend servers listening on a different TCP/UDP port than the client is using to connect. Both methods require additional consideration for internally load balanced services, where the clients, load balancers and servers reside on the same network. Using NAT for a connection from a client within an internal network to a server having a direct route for the client's internal source address requires methods for addressing such NAT hairpinning issues. Using DSR to accept incoming connections for a VIP causes issues if the same VIP is shared between multiple services, and one backend server attempts to connect to a service using the same destination VIP address that is also configured locally.

One design for a scalable network-level load balancer using NAT for forwarding is the use of Full NAT. A cluster of load balancers may each be configured with an unique local internal network address, used for SNAT of load balanced packets forwarded using DNAT. The use of SNAT with the load balancer's local source address ensures a symmetric return path for any reply packets from the backend server. However, the use of Full NAT leads to the complete loss of the original client source network address information by the backend server. Using Full NAT for load balancing also complicates the design of a reliable load balancer allowing failover of connections between load balancers, requiring the new load balancer to use the same source address for SNAT as the initial load balancer. A network-level load balancer using Full NAT for connection forwarding behaves similarly to an application-level load balancer performing transport-level proxying, wherein the failure of any load balancer within a cluster will disrupt some subset of active connections.

2.5.3 Application-level load balancing of Transport-layer connection

An application-level load balancer is implemented using the standard TCP/UDP socket programming API provided by an operating system [15, Section 6.1.3]. An application-level load balancer can be used for load balancing of transport-layer connections by accepting an incoming connection from a client to a service, establishing a new outgoing connection to the backend server, and then proxying traffic between the pair of connections.

A transport-level load balancer is well behaved across a wider variety of network architectures than a network-level load balancer requiring consideration of routing paths for the forward and return paths. Each packet forwarded by a transport-level proxy will use a network address associated with the transport-level load balancer, ensuring the use of symmetric routing paths using normal network host addresses. A transport-level proxy may be used to forward traffic between clients and servers in different networks that do not provide any direct routing between the client and server. A transport-level proxy may also be used to forward connections to local addresses configured on the client, including the `127.0.0.1` host-local address, which poses a challenge for network-level load balancers, particularly those using DSR. A transport-level load balancer is thus more flexible than a network-level proxy, allowing the deployment of load balancers forwarding connections within and between arbitrary external and internal networks.

The disadvantage of a transport-level load balancer over a network-level load balancer is that each connection is tied to the local network addresses used by the transport-level proxy. The server application on the backend server will only see the local network address of the proxy, and not the source network address of the client, compared to a connection forwarded by a network-level load balancer. The use of transport-level connections terminated by the proxy also ties each connection state to the load balancer, making the implementation of connection failover between load balancers impractical. Using a L3 load balancing method such as ECMP to scale a transport-level load balancer will inenvitable lead to broken connections on topology changes, as the stateless hashing of transport-level flows changes and connections are rerouted to different L4 load balancers.

2.5.4 Application-layer load balancing

An application-level load balancer implemented using the TCP/IP stack provided by the operating system may also support application-layer load balancing in addition to generic transport-layer load balancing. Application-level load balancers will use similar transport-layer forwarding methods for proxying incoming application-layer protocol requests to application-layer backend servers. Application-level load balancers for application-layer protocols such as HTTP can use the application-layer protocol semantics to support more flexibility load balancing policies than

a transport-layer proxy. The additional application-layer protocol semantics may also reduce the impact of transport-layer load balancing issues by simplifying the implementation of workarounds such as automatically reconnecting clients.

Many cloud services use the application-layer HyperText Transfer Protocol (HTTP), allowing the use of generic HTTP load balancer implementations for different HTTP services. An application-layer load balancer supporting the load balancing of HTTP requests across HTTP server backends provides more semantics for use in load balancing than a generic transport-layer load balancer. The stateles design of HTTP is well suited for the use of HTTP reverse proxies as load balancers, allowing independent forwarding of each HTTP request. The defined semantics for idempotent HTTP methods such as `GET` allows the transparent implementation of additional HTTP reverse-proxy functionality such as request retry and response caching. Each HTTP/1.1 request will also include a `Host` header specifying the original DNS name used by by the client application to resolve the service, allowing the load balancing of traffic for multiple different HTTP services sharing the same VIP. [24]

2.5.5 DNS

DNS supports multiple address records for a DNS name, allowing clients resolving DNS names with multiple address records to choose which server to connect to. Assuming different clients choose different address records, the load will be distributed across each server returned by the DNS server. DNS can be used for both internal load balancing of services within a cluster, as well as external load balancing of services across the Internet. Any standard DNS server implementation can be used to configure multiple static address records for a DNS name.

A shared service discovery database can be used to implement a dynamic DNS server, such as the SkyDNS server studied in Section 3.2.2. The dynamic DNS server can be used to provide clients with the network addresses of all service instances associated with the queried service name. SkyDNS allows the use of hierarchially defined subdomains of the service name to select a subset of the services to return, allowing clients to be explicitly configured to use a named subset of services. A GeoDNS server uses the origin of the DNS query to automatically select a subset of address records to return. GeoDNS can be used to provide different DNS resolvers with different network addresses for a single DNS service.

GeoDNS can be used to distribute clients across multiple geographically distributed service locations by approximating the geographic source of the DNS query, and returning the network address of the nearest service location. Implementing geographic load balancing using DNS requires a method for mapping the DNS resolver sending the query to the expected location and number of clients using that DNS resolver. Any DNS response returned to a DNS resolver will be used by any client configured to use that DNS resolver. The incoming DNS query will not contain the

network address of the client, unless the DNS resolver supports the use of the DNS `client-subnet` extension. The use of improved client-mapping methods can have a significant positive effect on the network performance of a large scale CDN with a large number of geographically distributed locations [25].

3 A Study of Docker Container Platform Components

This Chapter studies various implementations of the container networking, service discovery and load balancing concepts introduced in the previous chapter. This study focuses on the Docker container platform and related components used in the implementation of the `clusterf` load balancer. Section 3.4 also discusses load balancer implementations designed for specific cloud infrastructure platforms.

3.1 Docker Networking

Docker uses network namespaces to assign each container an individual IP network address. This network isolation between containers serves to avoid conflicts between different applications providing network services using the same well-known TCP/UDP ports [16]. Docker supports a number of mechanisms for networking these containers, in the form of built-in network drivers and external network plugins. Docker is designed to be flexible, allowing applications to be run on a single machine during development, or deployed on a container platform using a cluster of multiple machines. While initial releases of Docker focused on single-machine networking to support application development, Docker has gradually been introducing new features supporting the deployment of horizontally scaled services across clusters of servers.

Designing a container networking architecture usable in such a wide range of networking environments presents interesting challenges. Any networked machine must be able to provide network connectivity for any number of containers acting as virtual network hosts without depending on any explicit support from the external network. Container networking architectures are generally designed around the use of NAT, allowing the use of a separate internal network to provide the necessary flexibility. Containers can use the internal network for communication between containers, using networking mechanisms such as NAT to enable communication with the external network. A scalable container network architecture must also allow containers to be distributed across multiple machines, using different forms of internal and external networking.

Docker's network model allows each container to be run using a separate network namespace, assigning each container an individual network address. Docker's default configuration creates a new network namespace for each container, connecting each container running on a machine to a default network. Docker also allows containers to be run using the native network namespace of the host machine, or sharing one network namespace between multiple containers. Docker allows containers to *expose* ports, allowing other containers within the same network to connect to the exposed

services. [16]

Docker 1.7 [2] introduced a new networking model, allowing the use of multiple separated networks. Docker networks can be created using a variety of network drivers or external plugins, allowing the use of different mechanisms for container networking. Each container can be attached to one or more networks, corresponding to network interfaces within the container's network namespace. Docker containers attached to a network are configured using the same IP subnet, allowing containers within a network to communicate directly using IP addresses within the Docker network's subnet. Multiple networks can be used to limit access to the internal services exposed by a container, limiting connections to those containers attached to the same network. Docker does not route traffic between different Docker networks, requiring the use of proxy applications to forward application traffic between networks. Multiple Docker networks are typically used to split an application stack into multiple tiers, limiting access to a database service to those containers attached to the dedicated database access network.

Docker 1.9 [3] introduced support for multi-host overlay networks. Docker overlay networks can be used to extend the Docker network model from containers running on a single machine to containers running on multiple machines. Docker overlay networks use the VXLAN protocol for Ethernet tunneling, additionally supporting the use of IPSec for encryption.

Docker includes service discovery mechanisms allowing containers to communicate in an internal network environment using dynamically allocated addresses. Containers can be explicitly linked together, allowing the linking container to connect to services exposed by the linked container. Assuming a container named `mysql-test` providing a service on TCP port 3306, a second container can be configured using the `-link db:mysql-test` option. The application running within the linking container can be configured to connect to the MySQL service using either the `db` DNS host name, or environment varibles of the form `DB_PORT=tcp://172.17.0.5:3306` provided by Docker. Internally, Docker container linking is implemented using a dynamically generated `/etc/hosts` file managed by the Docker Engine, which the `libc` resolver uses for DNS resolution.

Docker 1.10 [4] introduced an embedded DNS server for containers attached to user-defined networks. Containers attached to a user-defined network can use DNS to resolve the name or network alias of any container connected to the same user-defined network without the requirement for explicit linking. Containers using the default network must continue to use the legacy explicit linking mechanism, with Docker recommending the use of the dynamically updated `/etc/hosts` DNS names for application configuration. Using environment variables for application configuration

[2] `https://blog.docker.com/2015/06/announcing-docker-1-7`
[3] `https://blog.docker.com/2015/11/docker-1-9`
[4] `https://blog.docker.com/2016/02/docker-1-10/`

does not allow the configuration to be updated if the linked container is moved to a different network address.

Docker allows publishing TCP/UDP ports exposed by containers, using DNAT to provide external access to the service using the machine's external network address. Ports can be published using a dynamic port allocated by Docker, or using a configurable static port. However, only one container may publish a service on a given port. Publishing multiple service instances on a given port requires the use of a load balancer.

Docker 1.12 [5] introduced support for Docker Services, allowing multiple replicated task containers to publish the same TCP/UDP port. Docker 1.12 uses Linux IPVS to load balance incoming connections for a service across the task replica containers.

3.1.1 Single-Host container networking

Docker is designed for ease of application development, and Docker's default network configuration for containers must be compatible with any network environment. Any normal network host should be able to run Docker containers without requiring the local network to be redesigned for container networking support. At the same time, Docker must be able to provide each container with an individual network address. A network host running Docker containers is no longer a simple network host, but a network router that must provide address allocation and routing for virtual network hosts. However, typical networks using mechanisms such as DHCP for configuration only provide each machine with a single network address. Support for mechanisms such as DHCPv6 Prefix Distribution that provide automatic allocation of routable network address space in IPv6 networks is not universal.

One approach would be to connect each container to the local network using Ethernet bridging, as commonly used for servers running virtual machines. This would allow the use of existing networking mechanisms such as DHCP for container networking, but simultaneously also limits the flexibility of container networking. Most networks only have a limited number of available addresses, which would limit the flexibility of container architectures by imposing an arbitrary and unpredictable limit on the number of containers that can be used. Cloud servers use different network configuration mechanisms, and generally only provide each machine with a single network address.

Figure 2 demonstrates the default Docker single-host networking architecture in a typical small home/office (SOHO) network environment. Host A and B are attached to a local network, using a mechanism such as DHCP to configure an interface address within the local network, using the router as the default gateway. Host A is used to develop an application using Docker containers, and is running two

[5] `https://blog.docker.com/2016/06/docker-1-12-built-in-orchestration/`

Docker containers. Each Docker container is a virtual network host within the host machine, attached to the default Docker network within host A.

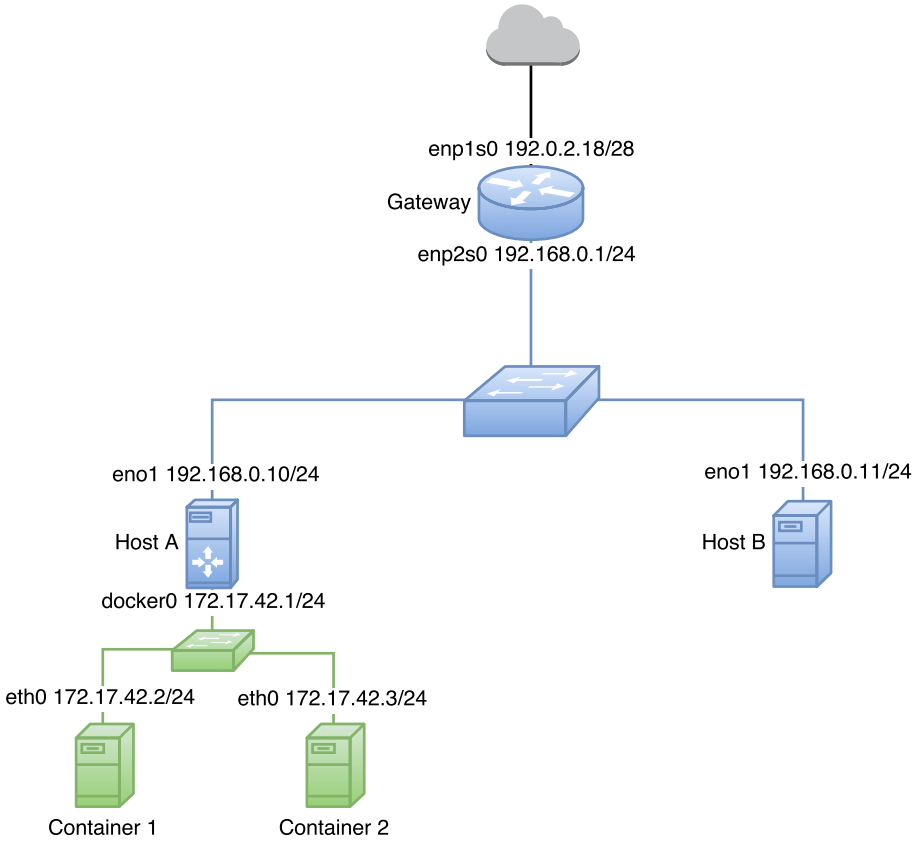

Figure 2: Docker single-host container networking architecture

Docker creates a virtual network bridge (`docker0`), and automatically chooses an internal network of private RFC1918 addresses (`172.17.42.0/24`). The virtual bridge interface on the host machine (`docker0`) is configured using an interface address within the Docker network (`172.17.42.1/24`). For each Docker container, the Docker engine creates a new network namespace, and a pair of virtual Ethernet (`veth`) interfaces. One of the `veth` interfaces is moved into the container's network namespace, and the other `veth` interface is attached to the `docker0` bridge. Docker configures the `veth` interface within the container using an interface address allocated from within Docker's internal address space (`172.17.42.X/24`). The container is configured to use the host machine's Docker network address (`172.17.42.1/24`) as the default gateway. This forms a virtual Ethernet network spanning the host machine and each container, allowing containers to communicate using the internal Docker network. [16]

The resulting routing configuration for each host within the example network is shown in table 1. The network contains two separate Ethernet networks for the host machines (192.168.0.0/24) and host A's Docker containers (172.17.42.0/24).

Hosts within each such network are able to communicate by using ARP to resolve the link-layer Ethernet addresses of neighboring hosts within each such network. Docker configures the operating system networking stack on host A to act as a router, forwarding packets between the two networks.

Table 1: Detailed routing configuration used by the example network in Figure 2

Host	Prefix	Interface	Next-Hop
Gateway	0.0.0.0/0	enp1s0	Router B
	192.0.2.17/30	enp1s0	
	192.168.0.0/24	enp2s0	
	192.168.0.10	enp2s0	Host A
	192.168.0.11	enp2s0	Host B
Host B	0.0.0.0/0	eno1	Gateway
	192.168.0.0/24	eno1	
	192.168.0.10	eno1	Host A
Host A	0.0.0.0/0	eno1	Gateway
	192.168.0.0/24	eno1	
	192.168.0.11	eno1	Host B
	172.17.42.0/24	docker0	
	172.17.42.2	docker0	Container 1
	172.17.42.3	docker0	Container 2
Container 1	0.0.0.0/0	eth0	Host A
	172.17.42.0/24	eth0	
	172.17.42.3	eth0	Container 2
Container 2	0.0.0.0/0	eth0	Host A
	172.17.42.0/24	eth0	
	172.17.42.2	eth0	Container 1

The containers are able to connect to services exposed by other containers on the same host machine using their internal Docker network addresses. Each container has a local interface route for the `172.17.42.0/24` network, and uses a source address within the same network for outgoing connections. The containers will also use their `172.17.42.X` source address for any outgoing connections using their default route, with the host machine acting as a router to forward the packets onto the local network. However, the containers cannot use their internal Docker network addresses to communicate with the other hosts within the local network. While the host machine can forward the packet with a `172.17.42.X` source and `192.168.0.11` destination address to the host B machine, the receiving machine will be unable to route any return packet back to the Docker container. The host B machine does not have a route for `172.17.42.0/24`, and it will forward the reply packet to the Gateway. The Gateway router does not have any route for the Docker network either, and the return packet will leak outside of the local network until hitting a router configured to drop packets for unknown destinations. If the same private network used by the Docker containers is used in a different location within the

network, the return packets may even be forwarded to a completely different host within the network.

Using locally assigned internal network addresses requires the host machine to perform NAT when forwarding packets between the internal Docker network and the local network. Docker configures the host machine as a NAT router by default, using Linux `iptables` firewall rules. SNAT is used for outgoing connections from the internal Docker network, rewriting the source address to the local network address of the host machine. The requirement for symmetric routing when using NAT is trivially satisfied, as the host machine acts as the default gateway for every Docker container. The reliability issues related to the stateful nature of NAT are also irrelevant, as any failure of the machine will also lead to the failure of any containers using the host machine for NAT routing.

Docker allows the use of DNAT for publishing ports exposed by server applications within containers, rewriting the destination address of incoming traffic for a published port to the internal Docker network address of the container. Publishing Docker container ports is commonly used in single-host networking, but quickly becomes limiting when running multiple instances of a service within a machine, requiring each such instance to be published using a different external port. The astute reader may also note the use of private network address space in the example SOHO network shown in Figure 2. The use of NAT within the external network makes it practically impossible for a container platform to publish container services to the Internet without explicit configuration of the external network, which may also be considered a feature by some.

3.1.2 Multi-host container networking

A scalable container platform must be able to run containers on multiple machines without changing the network model seen by the applications. The default Docker networking model for single-host container networking uses a separate internal container network with NAT. Multi-host container networking architectures extend the internal container network to span multiple hosts. Containers running on any machine within the cluster can be attached to the same network, communicating using the same service discovery mechanisms as used within a single machine. Interconnecting the container networks on multiple machines either requires a network infrastructure capable of forwarding traffic using the internal network addresses, or the use of overlay network tunnels.

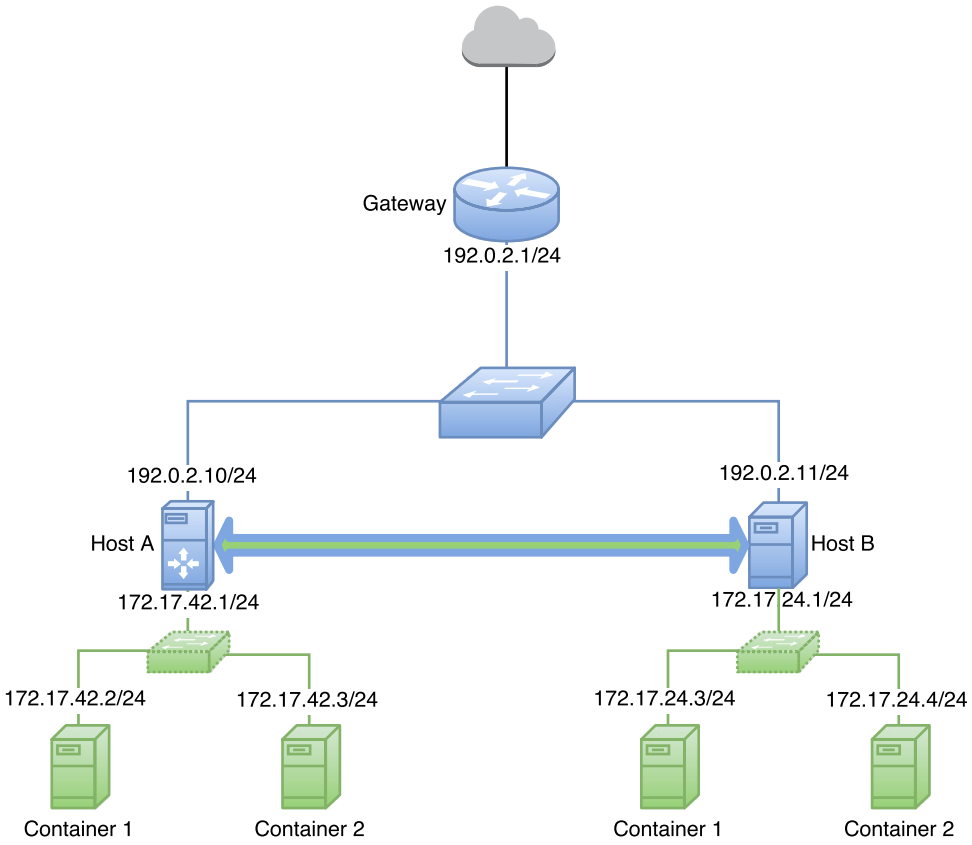

Figure 3: Docker multi-host container networking using a network-layer architecture

Multi-host container networking can be implemented using either network-layer or link-layer architectures. In the network-layer architecture shown in Figure 3, each machine is assigned a unique subnet for use by local containers. The container network infrastructure uses network routing to forward packets for any network address within a container subnet via the container's host machine. In a link-layer architecture, each container Ethernet network can span multiple host machines, with containers running on different machines using the same network subnet. The container network infrastructure uses Ethernet switching to forward packets for each container interface via the container's host machine. Whereas routed network-layer architectures only need a single routing table entry per subnet, bridged link-layer architectures require separate forwarding entries for each container.

The Kubernetes container platform [6] uses a network-layer architecture for multi-host container networking. A Kubernetes Pod is a group of tightly-coupled containers that can be scheduled to run on any machine within a cluster. Kubernetes extends the default Docker networking architecture to give each Pod a unique container network address within the cluster. The default Docker network within each Kubernetes node is configured to use a unique internal subnet within the cluster's network

[6] `http://kubernetes.io/`

address space. Each container pod is configured to use a network address within the node subnet, routing all other traffic via the Kubernetes node. Each Kubernetes node is configured to route container traffic between node subnets. For Kubernetes clusters running in Google Compute Engine, the underlying Andromeda Software Defined Network (SDN) is configured to forward network traffic for each container subnet to the associated node. Flannel can be used to provide a universal network infrastructure for Kubernetes clusters, using overlay networking to tunnel internal container traffic between machines. [16]

The Docker Swarm overlay network driver uses an alternative link-layer architecture for multi-host container networking [26]. The Docker overlay network forms a single Ethernet network spanning all machines within the cluster. Docker containers on multiple machines can be attached to the same overlay network, using the same IP subnet across the cluster. The overlay networking is implemented within the host machine's virtual Ethernet bridge, allowing containers attached to a clustered overlay network to behave identically to containers attached to a local bridge network. Docker supports the use of multiple bridge and overlay networks, allowing containers to be segregated into different networks. Docker's overlay networks provide direct connectivity between containers attached to the same overlay network within a cluster.

A container platform must also allow containers to connect to external services. The typical multi-host container networking model assumes that each machine within a cluster is also connected to an external network using a separate external network address. Docker containers attached to an overlay network are also attached to a special `docker_gwbridge` network, using the host machine as the default route for outgoing connections outside of the overlay network. The host machine uses SNAT to forward the packet using the locally configured external network address. The use of SNAT ensures a symmetric return path, whereby any reply packets will be routed back to the same host machine having the NAT connection state. This property of NAT allows each host machine within a cluster to forward outgoing connections from local containers, assuming that each machine uses a different external network addresses.

A container platform must also allow containers running on different nodes to publish services. Publishing container services within a cluster requires the host machines to use DNAT for forwarding incoming connections from the external network. The symmetric routing requirement for NAT requires any container accepting incoming external traffic to have a default route via the same host machine providing the DNAT forwarding. Publishing ports for a Docker container connected to a Docker overlay network also uses the `docker_gwbridge` network for DNAT. Publishing a port for dynamically scheduled Docker containers requires any incoming connections to use the external network of the specific machine used to run the Docker container. Providing a static virtual network address for ports published by horizontally scaled Docker containers requires the use of a load balancer, such as the service routing

mesh included in Docker 1.12 [7].

3.2 Service Discovery

This Section studies the implementation of various service discovery components designed for use in Docker container platforms, focusing on the components used in the implementation and evaluation of the `clusterf` load balancer in Chapters 4 and 5. The `etcd` key-value store can be used for dynamic configuration and shared service database. The SkyDNS dynamic DNS server provides DNS-based service discovery for hierarchially named services configured in `etcd`. The Gliderlabs `registrator` can be used as a service registration tool for maintaining a dynamically updated inventory of Docker containers in `etcd`. The `confd` tool allows traditional Unix services using static configuration files to be integrated into a container platform using dynamic service discovery.

3.2.1 `etcd`

`etcd` [8] is a highly consistent distributed key-value database. A cluster of `etcd` nodes exchange messages over peer connections, using an implementation of the Raft [27] distributed consensus algorithm to provide leader election and a replicated log. Each of the `etcd` cluster nodes provides a network API, and client applications can establish connections to any available node to send read/write requests. One of the `etcd` cluster nodes is elected as the leader, and the cluster requires a leader to process any write requests. If the currently elected leader node fails, the remaining cluster nodes will automatically elect a new leader, during which time period in-progress writes may fail. A majority of the nodes within an `etcd` cluster are required for leader election, whereby a 5-node `etcd` cluster can tolerate the loss of up to two nodes. A network partition will cause any minority of `etcd` nodes to become unavailable to reject any writes that would lead to a split-brain situation violating the consistency guarantees. If half or more of the cluster nodes are lost, the remaining nodes will be unable to form a strict majority or elect a leader to process writes.

Applications can use `etcd` for a variety of different purposes such as distributed lock services, shared configuration or service discovery. The basic `etcd` application API presents a hierarchial key-value store with atomic get, set and delete operations, including support for consistent create-if-not-exists and check-and-set operations. Keys are filesystem-like paths, starting from the root `/` directory, forming a `/`-delimited path to a directory node, or a key node within a directory. Values are arbitrary strings that are not interpreted by `etcd`, and applications are free to use any syntax such as JSON-encoded configuration objects or plain-text hostname

[7] `https://docs.docker.com/engine/swarm/ingress/`
[8] `https://github.com/coreos/etcd`

and network address strings. Each directory and key-value pair supports automatic expiration using a Time-to-Live (TTL) parameter, allowing the use of periodically refreshed keys to detect failures. Any client can also watch any subset of the tree, with the `etcd` server pushing any updated key-value pairs to the client.

The `etcd` security model is based on a combination of SSL for network security with the recent addition of an auth API for resource-level security. A combination of SSL client and server X.509 certificates can be used to secure both the peer connections between `etcd` nodes and client connections from applications. This can be used to provide strong transport-level security by limiting access to nodes and applications posessing the private keys for X.509 certificates signed by a dedicated CA. The new auth API allows the configuration of users, roles and access control lists granting read/write permissions to specific keys. User authentication requires the use of HTTP Basic Authorization for password authentication of requests.

Security is an important aspect of any `etcd`-based system, particularly for a Docker platform using `etcd` to store critical infrastructure state. Uncontrolled access to the `etcd` datastore by an attacker would have major implications for any platform using `etcd` for shared configuration and service discovery. For example, in the case of Kubernetes [9]:

> Access to the central data store (`etcd`) in Kubernetes allows an attacker to run arbitrary containers on hosts, to gain access to any protected information stored in either volumes or in pods (such as access tokens or shared secrets provided as environment variables), to intercept and redirect traffic from running services by inserting middlemen, or to simply delete the entire history of the cluster.

3.2.2 SkyDNS

SkyDNS [10] is a dynamic DNS server used for DNS-based service discovery, using dynamically configured host records stored in the distributed `etcd` database. DNS query names are translated into an `etcd` path, returning DNS host records retrieved from `etcd`. SkyDNS uses `etcd` keys of the form `/skydns/local/skydns/service-a/instance-a` with JSON-encoded values of the form `{"host":"10.0.1.125"}`. Configuring a client host to use the SkyDNS resolver allows applications to resolve the `instance-1.service-a.skydns.local` DNS name, connecting to the `10.0.1.125` network address configured in `etcd`. The SkyDNS servers and the `etcd` database can be distributed across multiple machines, ensuring reliable service across possible machine failures. The implementation of DNS service discovery in the Kubernetes container platform is based on SkyDNS [11].

[9] `https://github.com/kubernetes/kubernetes/blob/master/docs/design/security.md`
[10] `https://github.com/skynetservices/skydns`
[11] `http://kubernetes.io/docs/admin/dns/`

SkyDNS also supports the use of DNS for application-based load balancing across multiple service instances. A DNS query for `service-a.skydns.local` will return multiple DNS host records for any values under the `/skydns/local/skydns/service-a/` key prefix. Any such hierarchial structure can be used to further classify service instances, allowing a client to be configured to connect to some subset of the available service instances.

3.2.3 Gliderlabs `registrator`

Allowing clients to query running services from a shared service database such as `etcd` requires each running service to be registered into the shared service database. The service discovery records must also be removed if the service instance is stopped, or the host machine fails. For a schemaless key-value database such as `etcd`, the services must be registered using a key-value schema defined by the service discovery implementation.

Gliderlabs `registrator` [12] is a service registration tool for Docker containers. The `registrator` tool includes support for the SkyDNS schema, registering the internal Docker container network addresses used for multi-host Docker networking. The `registrator` tool uses the Docker API to synchronize running Docker containers into the service discovery database. Using the Docker Events API allows immediately synchronizing containers as soon as they are started or stopped, without requiring the use of periodic polling to refresh container state. The service registration records stored in `etcd` are periodically refreshed, using `etcd` TTLs to automatically expire service records if the host machine fails and stops refreshing the records.

3.2.4 `confd`

`confd` [13] is a configuration management tool used to integrate services requiring static configuration into a platform using dynamic service discovery. `confd` uses user-defined templates to generate static configuration files from records stored in a configuration database. `confd` watches the configuration database for changes, generating a new configuration file and reloading the statically configured service process.

Assuming a container platform registering running containers for a servce into `etcd`, `confd` can be used to synchronize the configuration of a horizontally scaled load balancer using proxy servers running on multiple machines. Each proxy machine uses a `confd` template to enumerate the service backends stored in `etcd`, generating a proxy configuration file to route traffic for those services to the running backends. If

[12] `https://github.com/gliderlabs/registrator`
[13] `https://github.com/kelseyhightower/confd`

a new service backend is started, or an existing one is stopped, the service registrator updates the backends in `etcd`. `etcd` propagtes the update within the cluster, and notifies each `confd` process watching the updated keys. The `confd` process on each machine generates an updated configuration file and reloads the proxy, allowing traffic to use the updated service instances.

3.3 Networking

Section 3.1 studied common Docker networking architectures, and briefly covered the multi-host overlay networking implementations included in Docker itself. This Section studies two specific implementations of multi-host Docker networking in more detail. Given the rapid pace of Docker development, the Flannel and Weave implementations of multi-host networking for Docker clusters were both originally designed to extend the default Docker single-host networking architecture, before Docker itself added support for multi-host networking using globally scoped network plugins.

The Flannel implementation of network-layer multi-host container networking was originally designed for use with the Kubernetes platform, using `etcd` as a control plane. Flannel provides multiple alternative implementations of routed connectivity between multiple machines having unique local container network subnets. Flannel is only used to configure the default Docker network on each node, and does not provide a Docker network plugin implementation. The Weave implementation of link-layer multi-host container networking provides integrated control and data planes for overlay mesh networking, providing a single Ethernet network spanning containers running on any machines within a cluster. Weave provides multiple approaches for Docker integration, including multiple Docker network plugin implementations.

3.3.1 Flannel

Flannel [14] implements a network-layer architecture for multi-host Docker networking, designed for use with the Kubernetes platform. Flannel can be used to provide an internally routable network address for each Docker container running on a machine, allocating a unique internal network subnet for each machine. Each machine in the cluster runs the `flanneld` agent, using the distributed `etcd` database for configuration and control. Given an internal network address space (`10.0.0.0/8`), each Flannel agent will automatically allocate a unique internal network subnet (`10.0.10.0/24`) for the local container network. The Flannel agent configures Docker to use a local bridge with the allocated subnet for the local container network. Traffic between local containers uses the normal Docker bridge network mechanism. For traffic outside of the host machine, each Docker container is configured to use the

[14] `https://github.com/coreos/flannel`

host machine's bridge interface address as the default gateway. Flannel configures the Linux kernel's network stack to route traffic for the internal Flannel network across different machines, using a packet forwarding dataplane provided by one of the Flannel backends. Each Flannel backend provides a dataplane for forwarding packets between internal container addresses on different machines, using the dynamic `etcd` configuration for automatic configuration. The shared `etcd` database contains the internal subnets and external network addresses of each machine within the cluster.

The `udp` backend routes internal network packets via a virtual Linux `tun` interface to a userspace proxy, which uses UDP encapsulation to tunnel packets for each configured Flannel subnet via the external network address of the remote machine. The `vxlan` backend uses the Linux kernel implementation of VXLAN tunneling, routing packets for internal container addresses via a virtual L2 address configured to tunnel packets using the external network address of the remote machine. The `host-gw` backend configures normal Linux kernel routes to forward traffic for each internal container subnet via the directly connected address of the remote machine, using ARP for routing across an Ethernet network connecting each host machine. The `aws-vpc` and `gce` backends provide direct integration with the Software Defined Network (SDN) implementations used for the Amazon Web Services (AWS) Virtual Private Cloud (VPC) and Google Compute Engine (GCE) IaaS platforms, offloading the inter-machine routing for the Flannel subnets into the cloud network infrastructure. All of the Flannel backends assume a full connectivity mesh between machines, lacking support for any multi-hop forwarding within Flannel itself.

3.3.2 Weave

Weave [15] implements a link-layer architecture for multi-host Docker networking using overlay networking. Weave provides a single Ethernet network spanning multiple Docker machines, using overlay networking to allow direct communication between Docker containers connected to the virtual `weave` bridge on different machines. The Weave Mesh [16] control plane implements a gossip protocol for peer discovery and configuration. Weave supports a combination of two different dataplanes for Ethernet forwarding, providing both a userspace and kernel datapath. The `sleeve` tunneling protocol implemented in user space, using a UDP-based protocol providing built-in support for encryption. The alternative Fast Data Path (`fastdp`) mechanism uses the Open vSwitch and VXLAN modules within the Linux kernel. Weave allows the simultaneous use of both dataplane mechanisms, only using the `fastdp` mechanism between nodes on a configurable "trusted subnet", due to the lack of encryption support in the `fastdp` implementation. Weave does not require direct connectivity between all peer nodes, and the topology discovered by the Weave

[15] `https://github.com/weaveworks/weave/`
[16] `https://github.com/weaveworks/mesh`

Mesh control plane can be used for multi-hop mesh routing, forwarding Ethernet packets via a chain of multiple nodes. Weave also provides additional infrastructure services implemented on top of Weave Mesh, including Weave DNS [17]. Weave DNS can be used for DNS-based service discovery of dynamic service endpoints within the Weave overlay network.

Using Weave for overlay networking between Docker containers can have a significant performance impact on network performance, depending on the virtual machine and Weave datapath implementation used. For a pair of smaller single-CPU virtual machines, using Weave for overlay networking between Docker containers can reduce total transfer rates by up to 70-80 % compared to the reference performance of the virtual machine network without container networking [28]. Using larger virtual machines with multiple CPUs reduces the contention between the Weave dataplane and application service, providing improved network performance with Weave [29]. The newer Weave `fastdp` dataplane provides improved network performance, with Weave's own performance tests using `fastdp` reaching near-native performance when using a sufficiently large MTU for the overlay tunnels [18].

3.4 Load balancing

This Section studies various load balancer implementations used in a variety of environments. The HAProxy and `nginx` application-level proxies are widely used for both TCP and HTTP load balancing. The Vulcand and `traefik.io` application-level proxies are designed for use with container platforms, supporting dynamic configuration for load balancing HTTP services. Google Maglev and Ananta are proprietary, highly scalable network-level load balancers designed for specific cloud infrastructure platforms. The network-level load balancer implementations use the NAT and Direct Server Return (DSR) forwarding methods introduced in Section 2.5.2. The final part of this Section discusses the Linux IPVS load balancer used in the implementation of the `clusterf` load balancer.

3.4.1 HAProxy and `nginx`

HAProxy [19] and `nginx` [20] are open-source application-layer proxies. Both HAProxy and `nginx` support application-layer HTTP forwarding, providing a wide range of features to route and handle HTTP traffic. At the transport layer, HAProxy also supports TCP forwarding, and `nginx` supports both TCP and UDP forwarding. At the application layer, `nginx` provides a wider range of features for HTTP traffic, in-

[17] `https://www.weave.works/have-you-met-weavedns/`
[18] `https://www.weave.works/weave-docker-networking-performance-fast-data-path/`
[19] `http://www.haproxy.org/`
[20] `http://nginx.org/`

cluding response caching. `nginx` also supports a variety of other protocols including SMTP, IMAP and POP for email services.

Traditional Unix services like `nginx` and HAProxy are configured using a local configuration file, designed for manual editing. The running process can be signalled to reload the configuration file, gracefully applying any changes to the configuration. The use of configuration files provides a simple method for static configuration, but a load balancer for a container platform must support dynamic configuration when deploying horizontally scaled services within a cluster, where the number and location of the service backends can change quickly. Both `nginx` and HAProxy also provide a command API for runtime control over their load-balancer configuration, but this is either proprietary or limited in terms of the configuration operations supported. The general solution for using such proxy servers in a container platform is the use of a configuration management tool such as `confd` to dynamically generate and reload the configuration file.

3.4.2 Vulcand and `traefik.io`

Vulcand [21] and `traefik.io` [22] are open-source application-layer proxies designed for use with container platforms. Rather than using a static configuration file, a dynamic application-layer proxy uses a shared database such as `etcd` for configuration, automatically applying any configuration updates. Both Vulcand and `traefik.io` support the use of `etcd` for configuration, load balancing HTTP/HTTPS requests across multiple backend servers. `traefik.io` also supports a wider variety of application protocols including HTTP WebSockets and HTTP/2. `traefik.io` also supports the use of multiple configuration backends, including integration with the Docker API for direct registration of Docker services without using a separate service discovery mechanism. Using a dynamic application-layer proxy can be compared to a combination of a traditional application-layer proxy such as HAProxy with a dynamic configuration management tool such as `confd`.

3.4.3 Google Maglev

Google Maglev [7] is the network-layer L4 load balancer used in the Google cloud infrastructure. Maglev is also used as the load balancing service for Google's IaaS cloud (Google Cloud Platform). Compared to traditional hardware load balancers deployed in pairs for redundancy, Maglev is designed for horizontal scalability using commodity server machines. Google services use GeoDNS to load balance services using separate Virtual IP (VIP) addresses routed to different frontend locations, based on user geolocation and current frontend loads. Multiple Maglev load bal-

[21] `https://github.com/vulcand/vulcand`

[22] `https://github.com/containous/traefik`

ancers are deployed at each such location, using BGP to control the routing of traffic across different. Maglev machines. Maglev scales using a combination of ECMP routing for a single VIP across a group of Maglevs, and dividing multiple VIPs between different groups of Maglevs. Maglev uses globally consistent hashing with local connection tracking to maintain reliable connections across most operational conditions, including the addition and remove of service backends or Maglev machines.

The packet forwarding dataplane and connection scheduler is implemented using Linux userspace networking to optimize the performance of each individual Maglev machine. The Maglev forwarder bypasses the Linux kernel's network stack for low-latency processing of small packets at 10 Gbps line rate, handling up to 9.06 Mbps of small 100 byte packets with a maximum latency of 300 µs. Each Maglev forwarder associates packets for a configured VIP with an existing connection state, or creates a new connection state using a consistent hashing algorithm to select a backend. The incoming packets are forwarded unmodified using Generic Routing Encapsulation (GRE) for tunneling, using the backend server's address for the outer packet header. The backend server accepts the de-encapsulated packet and forms a response packet using the incoming packet's client and VIP addresses, which can be routed using Direct Server Return (DSR), bypassing the Maglev machines. [7]

Each service VIP can be scaled using ECMP routing for L3 load balancing within the network, distributing traffic for a single VIP across multiple Maglev machines. The Maglev control plane manages the ECMP routing within the network, acting as a BGP client to announce VIPs to the network routers, and withdraw individual routes on machine failures. Reliable load balancing of connection-oriented protocols requires that all packets for a given 5-tuple are forwarded to the same backend server to maintain the end-to-end connection state. The use of ECMP routing complicates the reliable load balancing of TCP connections, as changes to the ECMP routing will alter the distribution of traffic across multiple load balancers. [7]

Maglev relies on a combination of local state tracking with globally consistent hashing to maintain TCP connections between clients and backend servers in the case of backend configuration or ECMP routing changes. Changes to the ECMP routing configuration are handled using a consistent hashing algorithm such that given a consistent backend configuration, each Maglev will independently schedule new connections to the same backend. Each Maglev machine maintains a local table of connection states to handle cases where changes to the backend configuration change the results of the consistent hashing algorithm for existing connections. However, these connection state tables are not synchronized between Maglev machines, and concurrent changes to ECMP routing and backend configuration will lead to broken connections. [7]

3.4.4 Ananta

Ananta [8] is a horizontally scalable L4 load balancer implemented in software, using distributed control and data planes for scalable NAT forwarding. Ananta's design rquirements consider a datacenter having 40k servers, handling 400Gbps of external traffic and 100Tbps of internal traffic, where up to 44Tbps of traffic uses virtual service network addresses and requires load-balancing. Ananta uses a distributed control plane and a three-layer data plane architecture to provide the necessary scalability to meet these design requirements economically while maintaining flexibility and reliability. Utilizing hardware ECMP routing capabilities within the network infrastructure allows the horizontal scaling of the load balancers themselves, where a pool of Ananta load-balancers can handle 100Gbps of combined traffic for a single virtual network address. Utilizing a custom virtual software networking component allows the server machines to directly route both intra-datacenter and outgoing external traffic, bypassing the dedicated load-balancer machines, which are only required to handle the 20% design ratio of incoming external traffic. Ananta provides the network load balancer infrastructure used in the Windows Azure public cloud platform, with 1Tbps of combined capacity deployed between 2011 and 2013 [8].

The Ananta cluster infrastructure consists of hardware network routers connecting physical server machines, which act as Ananta Mux nodes or Host nodes. The Host nodes are used to run the network services being load-balanced as either virtual machines or native services, and each of these service machines is assigned an internal network Direct IP (DIP). Traffic between servers within a service can use the internal DIP addresses, and does not require load-balancing. All external network traffic to/from the Internet uses an external Virtual IP (VIP), as well as all internal network traffic between different services, and must thus be load-balanced by Ananta using NAT. The network Ananta model thus includes six distinct cases of packet forwarding, including the forward and reverse paths for incoming external traffic (DNAT) for services, internal intra-cluster traffic between services (SNAT+DNAT) and outgoing external traffic (SNAT) from services. Ananta's dataplane uses a three-layer architecture involving the network routers, dedicated Ananta Mux nodes, and a Host Agent on each of the physical servers to efficiently process each class of traffic. Ananta's scalability comes from the ability to offload the majority of the DNAT/SNAT packet forwarding workload to the Host Agent on each server within the datacenter, bypassing the Ananta Mux nodes for everything except incoming external packets for the forward external DNAT and reverse external SNAT paths. [8]

Incoming external packets are forwarded by the network routers to the Mux nodes announcing their configured VIPs via BGP, using ECMP routing to distribute packets for each L3 VIP address across multiple Ananta Mux nodes. The Ananta Mux nodes forward the incoming packets using IP-in-IP tunneling to a server DIP included in the service's configured backends, using a consistent hashing algorithm and connection state table to distribute connections for each L4 TCP/UDP end-

point across multiple backend servers. The Ananta Host agent on the physical server hosting the service intercepts and de-encapsulates the tunneled packets, and performs DNAT forwarding to the backend server, rewriting the inner packet's external VIP address to the outer packet's internal DIP address using the configured NAT rules. Finally, the backend server receives the incoming packet using its internal DIP address and optionally rewritten TCP/UDP port number. [8]

The return path for incoming external connections uses the Host Agent's local DIP connection state to rewrite the reply packets from the backend server, translating the internal DIP source address to the original external VIP address. The translated reply packets can be forwarded directly by the host node via the network, bypassing the Mux nodes for the return path. A similar mechanism is also used for SNAT forwarding of locally originated outgoing external connections from the backend servers using the shared VIP address, where the Ananta Manager allocates a pool of ephemeral ports for use by each Host Agent on demand. For each outgoing connection, the Host Agent chooses a source port allocated for its use, and forwards the packet via the network using the translated VIP address, bypassing the Mux nodes. For the return path, the Ananta Manager configures each Mux Node for the VIP to forward packets for the allocated ephemeral ports to the correct Host Node. The Ananta architecture provides highly scalable NAT forwarding for a single VIP by distributing the NAT forwarding across each Host Node, allowing the outgoing packets to bypass the Mux nodes. However, this comes at the cost of significant complexity for the Mux node configuration, requiring the dynamic configuration of allocated ports for the outgoing SNAT reverse path. [8]

For intra-DC traffic between different services, Ananta supports a fastpath technique, bypassing the Mux nodes in both directions by forwarding packets directly between Host Agents. Traffic between two services uses the source and destination service VIP addresses, involving both SNAT and DNAT: the originating host connects to the destination VIP using its source DIP, the source address is rewritten to use the source service VIP, the destination address is rewritten to use the destination service DIP, and the destination host sees the connection from the source VIP to the destination DIP. The connection is initially established using the combination of SNAT and DNAT data paths described above, involving both the source host, destination mux and destination host for the forward path, and the destination host, source mux and source host for the return path. The destination Mux selects a DIP for the incoming connection as normal, but once the connection is established, it is treated as an internal fastpath connection between two Ananta services. The destination Mux signals the source Mux with the selected destination DIP, and the source mux signals both the source DIP and destination DIP using a redirect message. The source and destination Host Agents use the signalled DIP addresses to set up a direct forwarding state between the host nodes for the connection, bypassing both Mux nodes. [8]

3.4.5 Linux IP Virtual Server

Linux IP Virtual Server (LVS, IPVS) is the network-level L4 load balancer included in the Linux kernel. Similarly to Google Maglev and Ananta, Linux IPVS implements a network-level load balancer in software. Compared to Google Maglev, the Linux IPVS load balancer does not support the use of global consistent hashing. The Linux IPVS load balancer implements local connection state tracking with optional connection state synchronization for failover. Large internet sites such as Facebook have used Linux IPVS for load balancing [23], contributing support for IPv4-in-IPv6 `ipip` forwarding [30].

IPVS is configured using a Linux `netlink` interface, used to implement userspace tools such as the `ipvsadm` command-line. The IPVS configuration consists of a set of Services, defining the virtual network address and TCP/UDP port used for incoming traffic to the service. Each IPVS Service is configured with a set of Destinations, defining the real network address and TCP/UDP port used for forwarding traffic to a backend server. The IPVS module maintains a separate connection state table, used to forward packets for each TCP/UDP connection to the same backend server. A configurable scheduler [24] is used to select the destination for each incoming connection to a service, distributing connections between destinations using a configurable weight parameter per destination.

IPVS allows each Destination to be configured using one of three different forwarding methods. The `droute` and `tun` forwarding methods preserve the service's virtual destination address for forwarded packets, whereas the `masq` forwarding method rewrites the forwarded packets to use the destination server's network address. The `droute` and `tun` forwarding methods require each backend server to be configured to accept incoming connections for the service's virtual network address. The `masq` forwarding method does not require any special configuration on the destination servers, but requires all reply packets from the destination server to be routed via the IPVS load balancer to restore the external network address and TCP/UDP port used by the client. The `droute` and `tun` forwarding methods allow the use of Direct Server Return (DSR) techniques, where each backend server can route any reply packets directly, bypassing the IPVS load balancer for the return path. The `droute` method can be used to forward packets to destination servers on a directly connected Ethernet network, using ARP to resolve the configured network address for Ethernet forwarding. The `tun` method uses the IP-in-IP tunneling protocol to forward packets, allowing the use of IP network routing between the IPVS load balancer and destination servers. Both `droute` and `tun` methods also allow the server application on the destination to inspect the original network source address of the client.

IPVS allows each Service and Destination to be individually configured with a given

[23] `https://www.isc.org/blogs/how-facebook-is-using-kea-in-the-datacenter/`
[24] `http://www.linuxvirtualserver.org/docs/scheduling.html`

TCP/UDP port. Using `droute` and `tun` forwarding requires the use of the same TCP/UDP port for both the virtual service address and the destination servers, ignoring the TCP/UDP port configured for the destination server. IPVS will not rewrite the destination TCP/UDP port when using `droute` or `tun` forwarding, as this would cause the destination server to reply using the wrong source port, breaking DSR. Only the `masq` forwarding method supports the configuration of a destination server TCP/UDP port, rewriting return packets to use the original TCP/UDP source port chosen by the client.

The IPVS load balancer supports the use of connection state synchronization to maintain client connections to destination servers in the case of failover or ECMP traffic redistribution between different IPVS load balancer. IPVS implements connection state synchronization using UDP messages with IP multicast. Each IPVS server can be joined to the multicast group in either master or backup modes. An IPVS server with the master mode enabled will send multicast messages for each updated connection state entry, and an IPVS server with the backup mode enabled will receive multicast messages, updating the local connection state table. Each IPVS server may also be configured to enable both master and backup connection state synchronization for peer-to-peer synchronization, assuming each node has the same configuration state.

4 The `clusterf` Load Balancer

The focus of this thesis is on my design and implementation of a scalable network-level load balancer for horizontally scalable services within a Docker container platform. This Chapter presents the design of a distributed network-level load balancer data plane using Linux IPVS, and the implementation of the `clusterf` distributed control plane for dynamic IPVS configuration within a Docker platform. The resulting `clusterf` load balancer architecture is shown in Figure 4.

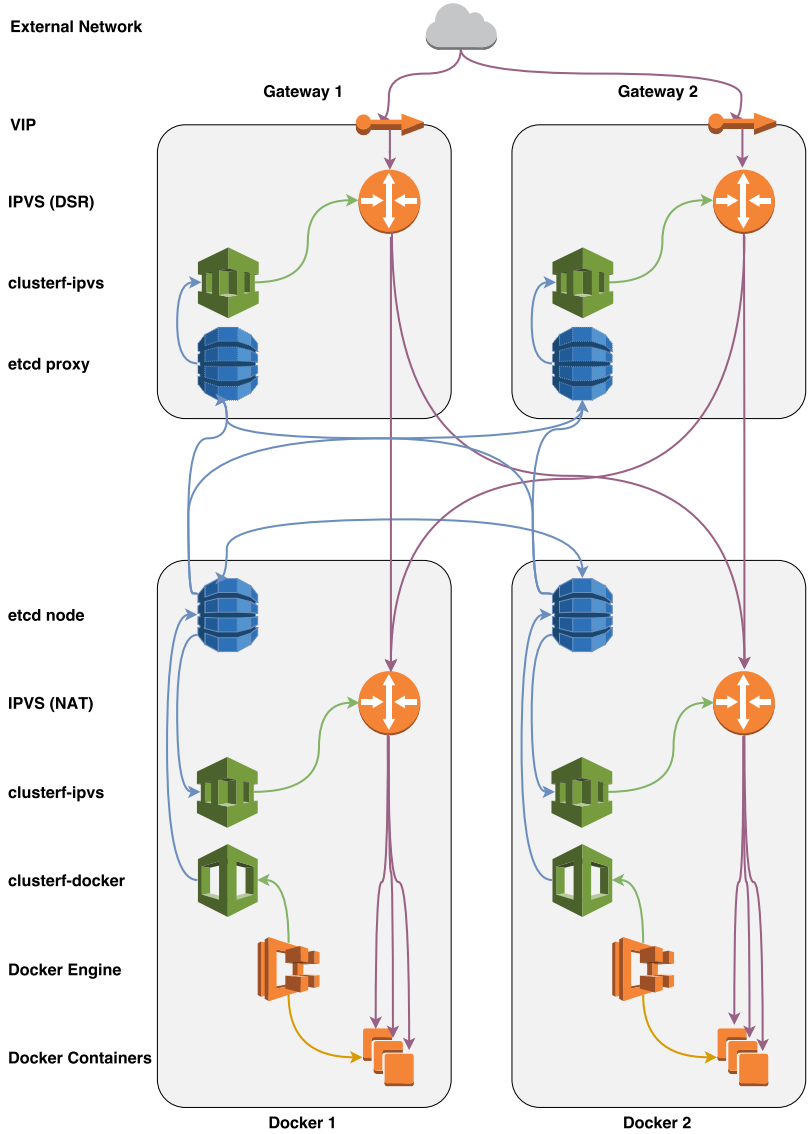

Figure 4: The architecture of the `clusterf` load-balancer, including the Docker platform components in orange, the `clusterf` control plane components in green, the shared `etcd` configuration database in blue, and the IPVS data plane network flows for incoming packets in purple.

Section 4.1 discusses the design rationale for the `clusterf` network-level load balancer architecture, considering the study of load balancing methods and implementations in Sections 2.5 and 3.4. Section 4.2 discusses the network architecture of a network-level load balancer within a container network, forwarding packets for incoming connections from external network clients to container services within the internal container network. Section 4.3 presents the two-layer load balancing architecture used to allow the use of both DSR for load balancer scalability and the use of NAT for application compatibility. Section 4.4 presents the implementation of the control plane, used to provide dynamic configuration of clustered IPVS load balancers within a Docker platform. Section 4.5 discusses the configuration model used by the `clusterf` control plane in more detail. The final Section 4.6 discusses some specific challenges with the current implementation of the load balancer design, in addition to other possibilities for future work.

4.1 Design Rationale

The highly dynamic nature of container networking for horizontally scaled services within a container platform requires the integration of an automated load balancer control plane for optimal load balancing as service containers are added and removed. The control plane must allow the use of a scalable load balancer data plane, synchronizing the configuration of multiple clustered load balancers for a given service. The control and data planes must be fault tolerant, allowing individual machines and service instances to be dropped from the cluster while minimizing the impact on any active connections.

The ultimate scalability of a horizontally scaled service is limited by the scalability of the load balancer data plane. If a single load balancer does not offer sufficient performance or relability, the load balancer design must support scaling to a cluster of load balancers. A cluster of load balancers requires a control plane capable of synchronizing the configuration of each load balancer, a mechanism for reliable connection forwarding across failures and configuration changes, and a method for distributing traffic between multiple load balancers. Different load balancing methods can be used to distribute traffic across multiple load balancers, from the use of DNS with multiple service addresses to the use of a dynamic routing protocol with ECMP forwarding for a single service address. The use of multipath routing for a VIP is sufficient for an load balancer using DSR, allowing each destination server to independently determine the return path. Different methods can be used for maintaining reliable connections, from the use of consistent hashing in Maglev [7] to the use of multicast connection synchronization in Linux IPVS.

Scaling a load balancer using NAT for forwarding requires additional complexity to maintain the symmetric routing required for NAT. The DNAT connection state formed by the incoming packet is required for rewriting any reply packets to restore the original external network address and TCP/UDP port used by the client. Any

reply packets from the backend server must be routed via the same load balancer machine that forwarded the incoming packet. This is straightforward for the trivial case of a single load balancer acting as the default gateway for each destination server.

One method used to cluster a load balancer using NAT forwarding is the use of Full NAT. A clustered load balancer using Full NAT uses both DNAT and SNAT for each forwarded packet. Packets are forwarded to the destination server using the load balancer's local source address and the server's destination address. The use of SNAT with a local address for each load balancer ensures that all return packets from the backend server are first routed back to the load balancer for rewriting. The use of Full NAT does not allow the server application on the destination server to see the network source address of the connection client. Implementing connection failover between load balancers using Full NAT would also require failover of the local source address on the load balancer used for SNAT forwarding.

The requirement for a symmetric return path complicates the design of a clustered load balancer using NAT forwarding. A load balancer using DSR with asymmetric return paths allows each backend server to independently form a return path using the unmodified packet source and destination addresses, simplifying the implementation of horizontal scaling with connection failover between load balancers. However, destination servers or applications unable to accept connections using the service's incoming external destination network address or TCP/UDP port require the use of NAT for forwarding. Using Full NAT to implement such a clustered load balancer results in the loss of connection source address information and connection failover capabilities. Implementing a scalable load balancer supporting the use of NAT thus requires a more complex load balancer architecture as seen in Ananta [8].

4.2 Network Architecture

The architecture of the `clusterf` load balancer consists of the Gateway and Docker machines shown in Figure 4. The detailed network architecture of the cluster machines and networks is shown in Figure 9. The Docker cluster consists of a number of Docker machines, which are each assigned internally unique `/24` networks for their local Docker network. A container orchestration system such as Docker Swarm is used to run service containers on any Docker machine within the cluster. Each service instance is a Docker container having a unique, internally routable network address. The Docker machines are connected via an internal network, using IP network routing to provide multi-host networking between Docker containers on different machines. The standard network-layer multi-host networking model is extended to include a set of Gateway machines used for load balancing, capable of routing traffic to internal container network addresses. Each of the Gateway machines and Docker machines uses the Linux IPVS load balancer to forwarding incoming connections to virtual service addresses to the Docker containers.

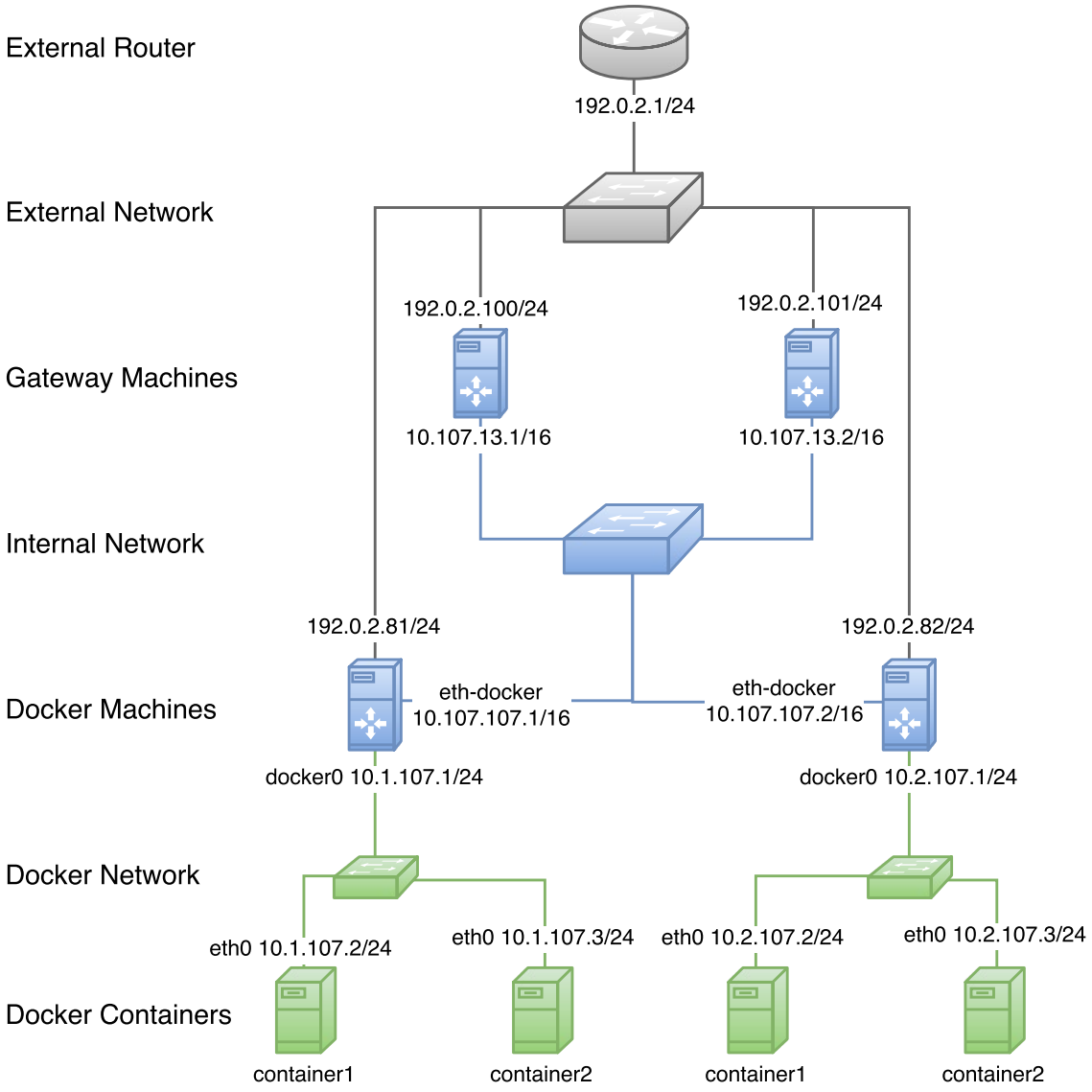

Figure 5: Network architecture used in the `clusterf` load balancer

The internal network in Figure 5 is represented as a standard Ethernet network. The internal network uses private `10.0.0.0/8` network address space, using the `10.107.0.0/16` subnet for the internal network between machines, and distinct `10.X.107.0/24` subnets for each machine's local Docker network. The cluster network uses OSPF [15, Section 5.6.4] for dynamic routing, using the *bird* [25] routing daemon managing the Linux kernel routing table. Each Docker machine acts as an OSPF router advertising an OSPF stubnet for the local Docker network, installing Linux kernel routes for each Docker network advertised by the other machines.

The Gateway machines are used for load-balancing of incoming connections to con-

[25] `http://bird.network.cz/`

tainer services within the cluster. Each Gateway machine also acts as an OSPF router, having a route for each Docker network. The Gateway machines are configured to accept traffic for the VIPs used for load balanced services. The Gateway machines accept incoming traffic for services using TCP/UDP ports on any of these VIPs, using Linux IPVS to forward the incoming packets via the Docker machines to the container services. The `clusterf` control plane manages the IPVS service configuration on the Gateway machines. The Gateway machines use IPVS connection synchronization multicast within the internal network to allow for connection failover between Gateway machines.

The specific mechanisms used for the allocation and routing of the Gateway VIPs are outside the scope of `clusterf`. For internally load balanced services, the Gateway machines may be configured to advertise the internal VIP within the cluster using OSPF. For external load balanced services, any mechanism supported by the external network may be used to provide external routing for any external VIPs. The same `bird` routing daemon used on the Gateway machines may also be used to announce VIPs to the external router using a routing protocol such as OSPF or BGP. An external network having multiple routers may use ECMP to distribute traffic for each VIP across multiple Gateway machines announcing the same VIP.

Compared to the design for Ananta [8], `clusterf` does not include any mechanism for SNAT forwarding of outgoing external network connections from the Docker containers. The `clusterf` design follows the standard Docker multi-host networking architecture, and allows the use of separate external network connectivity for each Docker machine. Each Docker Machine may use its locally configured external network address to SNAT outgoing connections from the Docker containers. The Docker machines may also use the Gateway machines for outgoing SNAT connectivity, but such a design is outside the scope of `clusterf`.

Extending the `clusterf` network architecture for use in different cloud networking environments is a subject for further study. The current `clusterf` implementation assumes the use of standard Ethernet networking for both the internal and external networks. Assuming a standard Ethernet network for the internal network allows the use of OSPF routing and IPVS `droute` forwarding. Alternative implementations of the internal network could use similar mechanisms as provided by Flannel, including the use of routing over overlay network tunnels. Assuming a standard Ethernet network for the external network allows the use of DSR for the return path from the Docker machines, and the use of mechanisms such as VRRP for VIP failover between Gateway machines. The use of DSR is not possible within an external network limiting the use of source addresses across different machines. Alternative implementations of the external network could use a different VIP per Gateway machine. The Docker machines would use policy routing to return packets sourced from a specific external network address via the same Gateway machine owning that address. A mechanism such as DNS would be used to distribute client connections between each external address.

4.3 IPVS Data Plane

The data plane architecture of the `clusterf` load balancer is shown in Figure 4, using the Linux IPVS network-level load balancer on both Gateway and Docker machines. The `clusterf` load balancer uses a hybrid two-level load balancing scheme combining the use of both DSR and NAT network-level load balancing techniques for achieving scalability without breaking compatibility with unmodified Docker containers and applications. Both Gateway and Docker machines are configured to process incoming traffic for load-balanced VIPs using IPVS, but only the Gateway machines need external routing for the VIPs. The Gateway machines accept incoming traffic for the VIP, and use IPVS to forward the unmodified packets to a Docker machine having a local backend for the service. The Docker machine accepts the forwarded VIP traffic, and uses IPVS to forward the packets to a local Docker container using DNAT. This load balancing scheme is similar to the Ananta [8] design for load balancing of incoming traffic.

Limiting the use of NAT to packets within the Docker machine avoids the issues related to the use of NAT for load balancing, and allows the use of the default Docker networking model using internal network addresses and ports within the Docker containers. Using IPVS `masq` forwarding is required to translate between different external TCP/UDP ports used by the service and internal TCP/UDP ports used by the server application. Each Docker container is configured with a default route via the host machine, ensuring a symmetric return path for any outgoing packets requiring reverse NAT. The failure of any Docker machine will only lose the NAT state for connections to containers within the same machine, which will inevitably fail anyways. More importantly, each Docker machine may use DSR for the return path. This both offloads any return traffic from the Gateway machines, and simplifies the horizontal scaling of the Gateway machines. Avoiding the use of Full NAT preserves the original network source address of the client for the connection seen by the server application within the container.

The Gateway machines for a given VIP must all have the full set of IPVS services for that VIP configured. The Docker machines only configure those IPVS services having local container backends. Most importantly, the Docker machines only configure IPVS destinations for local Docker containers. If the Docker machines configured IPVS destinations for remote Docker containers, this would lead to routing loops, where packets forwarded by a Gateway machine would be forwarded from Docker machine to Docker machine. Using a strict hierarchy of Gateway machines with the full set of IPVS services and destinations forwarding traffic to Docker machines having only local IPVS services and destinations ensures the correct forwarding of incoming traffic.

As discussed in Section 4.1, the use of IPVS `droute`/`tun` forwarding on the Gateway machines provides the best options for scaling the Gateway machines. Having the Gateway machines forward incoming packets using the original source and destina-

tion addresses allows the use of DSR, where each Docker machine may independently form a return path for any reply packets. Allowing each Docker machine to independently form the return path simplifies the distribution of incoming traffic across multiple Gateway machines, allowing the use of techniques such as ECMP forwarding with IPVS connection synchronization.

Using internal load balanced services within the cluster requires additional consideration when configuring the service VIPs on each Docker machine. A client application within a container attempting to connect to an internal VIP used for load balanced services will forward the outgoing packet to the host machine. Because the Docker machine will have each internal service VIP configured as a local address in order to accept incoming connections forwarded by the Gateway machines for IPVS forwarding, it will also attempt to process any outgoing packets to a VIP from any local container. If the Docker machine does not have any local IPVS service or backends configured for the packet's destination, it will drop the packet, causing the connection to fail.

```
$ ip link add dummy-ipvs type dummy
$ ip rule add iif eth-terom-dev lookup clusterf
$ ip route add local 10.0.107.0/24 dev dummy-ipvs
   table clusterf
$ ip link set dummy-ipvs up
```

Figure 6: Example Linux policy routing configuration for asymmetric IPVS forwarding of incoming packets to any `10.0.107.X` VIP on the `eth-terom-dev` interface

The `clusterf` implementation uses the Linux routing policy shown in Figure 6 to work around this issue. The local processing of routed packets within Linux uses special `local` routes, which are implicitly created when configuring a local interface address. The `clusterf` implementation uses a combination of Linux IP policy routing rules and `local` routes to limit the use of local IPVS forwarding to incoming packets forwarded by the Gateway machines. A policy routing rule is created to lookup routes for packets received on the internal network interface from a separate routing table. The VIPs used for IPVS forwarding are configured using `local` routes within this separate routing table. This use of routing policy rules allows outgoing connections to use the default IP routing table, presumably using the internal OSPF route for the VIP advertised by the Gateway machines.

For traffic to such internally load balanced services, the Gateway machine uses the IPVS configuration to either route the packet back to the same Docker machine, or to a different Docker machine having a local backend for the service. In the later case, the remote Docker machine receives the incoming packet with the source network address of the client container, and routes any reply packets directly to the originating machine using the internal network routes. However, the return path for

the former case is broken in the current implementation of `clusterf` due to NAT hairpinning issues discussed further in Section 4.6.

Extending the strictly hierarchial two-layer forwarding architecture to allow a mesh of load balanced connections routed directly between Docker machines, as seen in Ananta [8], is the subject of further study. Two alternative options would involve tunneling traffic directly to the Docker containers, configuring each Docker container to perform IP-in-IP packet de-encapsulation for the VIPs. This would allow each machine within the network to perform independent IPVS forwarding, using a full set of IPVS services and destinations without triggering forwarding loops. Alternatively, the Docker machines could also configure IPVS services that do not have any local destinations to independently forward traffic to the remote Docker machines. This would require the Docker machine to disable any such remote IPVS destinations as soon as any local IPVS destination is added, as the Gateway machines will begin forwarding incoming traffic. The `clusterf` control plane could be used to reconfigure the disabled IPVS destinations using a zero weight, ensuring that existing connections could continue, while all new connections would be routed to the local backends.

4.4 Control Plane

The `clusterf` control plane is designed for horizontally scaled services running as Docker containers within a cluster of machines using the Linux IPVS network-level load balancer. The `clusterf` control plane synchronizes the IPVS load balancer configuration across a cluster of load balancers, providing dynamic load balancer configuration for any service containers within the Docker cluster. The `clusterf` control plane allows starting and stopping Docker containers associated with a `clusterf` service on any machine within the cluster, automatically distributing incoming connections across each active service container. The `clusterf` control plane scales with the number of Docker machines, services and containers, while allowing the Linux IPVS data plane to scale with the service's network load. The `clusterf` control plane components are implemented in the Go [26] programming language, and are available on GitHub [27] under the MIT license.

The `clusterf` control plane components within the Gateway and Docker machines are shown in Figure 4. The `clusterf` load balancer control plane consists of the `clusterf-docker` and `clusterf-ipvs` components, using the distributed `etcd` database for dynamic configuration. The `clusterf-docker` component runs on each Docker machine and implements a service registrator, using the Docker API to register running containers associated with `clusterf` services into `etcd`. The `clusterf-ipvs` component runs on all Gateway and Docker machines and imple-

[26] `https://golang.org/`
[27] `https://github.com/qmsk/clusterf`

ments a dynamic IPVS configuration mechanism, using the Linux IPVS `netlink` control interface to apply any updates to the `clusterf` configuration within `etcd`.

The distributed `etcd` database is used to store the `clusterf` configuration state used by each machine within the cluster. The `clusterf` configuration state consists of `clusterf Route`s, `ServiceFrontend`s and `ServiceBackend`s. The `clusterf Route`s describe the cluster network topology used for IPVS forwarding. The `clusterf ServiceFrontend`s are used to configure IPVS services, describing the VIP and TCP/UDP ports used on the Gateway machines. The `clusterf ServiceBackend`s are used to configure IPVS destinations, describing the internal network addresses and TCP/UDP ports within the Docker containers. The `clusterf` configuration model is discussed in more detail in Section 4.5.

The `clusterf-docker` component runs on each Docker machine, implementing a service discovery mechanism for maintaining a dynamically updated inventory of `clusterf Route`s and `ServiceBackend`s in `etcd`. The Docker Engine API is used to observe the state of local Docker networks and containers, constructing an internal `clusterf` configuration state which is compiled into configuration objects and written into `etcd`. Docker networks are used to configure `clusterf Route`s, using the Docker network subnet and configurable `clusterf Route` parameters. Docker containers are used to configure `clusterf ServiceBackend`s, using the container's internal network address, exposed ports and `net.qmsk.clusterf.*` labels. The `clusterf-docker` component uses the Docker Events API to dynamically update the `clusterf` configuration as Docker containers and networks are created, started, stopped and removed. The `clusterf-docker` component periodically refreshes each `etcd` configuration node, using `etcd` TTLs to automatically expire any `clusterf` configuration for local Docker containers on machine failure.

The `clusterf-ipvs` component runs on each Docker and Gateway machine, with the Docker machines using a local `clusterf` Route configuration to restrict the IPVS configuration to local containers backends. The `cluster-ipvs` component implements a load-balancer control plane for dynamic configuration of the Linux kernel IPVS load balancer, using the global `clusterf` configuration state stored in `etcd`. During the initial startup, the current runtime IPVS configuration within the kernel is examined, forming the initial internal IPVS configuration. A recursive `etcd get` operation is used to read the `clusterf` configuration state, constructing a new internal IPVS configuration state. The IPVS configuration update mechanism runs any necessary IPVS configuration commands to transform the runtime IPVS configuration from the previous state into the new state. A recursive `etcd watch` operation is used to consistently follow any updates to the configuration objects in `etcd`, applying them to the internal `clusterf` configuration state. The updated `clusterf` configuration is used to generate a new internal IPVS configuration state, using the same IPVS configuration update mechanism to run any necessary IPVS configuration commands.

The runtime IPVS configuration mechanism is designed to minimize any disruption to existing IPVS connections upon `clusterf` control plane restarts or reconfigurations, as opposed to an approach where the entire IPVS configuration is first cleared before loading the initial configuration. When the `clusterf` configuration state is modified, any new IPVS services and destinations will be added before updating or removing the old IPVS services and destinations, minimizing any disruption to services. Docker service backends can be stopped gracefully, configuring a zero IPVS destination weight to drain connections until the container exits and `clusterf` removes the IPVS destination.

4.5 Configuration

The `clusterf` configuration is stored as a tree of JSON-encoded configuration objects, using the `etcd` key schema shown in Figure 7. A `clusterf Service` associates a number of `ServiceBackend` network endpoints with a `ServiceFrontend` network endpoint provided by IPVS. The `ServiceFrontend` and `ServiceBackend` network endpoints can be IPv4 or IPv6 addresses with TCP or UDP ports, and any combination thereof. Connections to the `ServiceFrontend` network endpoint will be load-balanced across the associated `ServiceBackend` endpoints. A `clusterf Route` further associates a set of `ServiceBackends` within the network with an IPVS forwarding configuration. The `ServiceBackend` and `Route` configuration objects can be automatically managed by `clusterf-docker`, based on active Docker networks and containers within the cluster. The `ServiceFrontend` configuration is determined by the external network addressing and routing methods used, which are outside the scope of `clusterf`. The `clusterf ServiceFrontends` must be configured externally.

```
/clusterf/routes/docker1
        {"Prefix":"10.1.107.0/24", "Gateway":"10.107.107.1",
            "IPVSMethod":"droute"}
/clusterf/routes/docker2
        {"Prefix":"10.2.107.0/24", "Gateway":"10.107.107.2",
            "IPVSMethod":"droute"}
/clusterf/services/example/frontend
        {"IPv4":"192.0.2.99", "TCP":1337}
/clusterf/services/example/backends/84dd43bc...
        {"IPv4":"10.1.107.2", "TCP":1337}
/clusterf/services/example/backends/443b85a3...
        {"IPv4":"10.2.107.3", "TCP":1337}
```

Figure 7: Resulting `clusterf` configuration state for the example network in Figure 5. The `etcd` key paths are aligned to the left, with the JSON-encoded value indented.

The configuration schema is designed to allow atomic updates of the individual configuration objects without leaving the configuration in an invalid state between individual operations. Storing a network address and port as separate configuration nodes would lead to invalid intermediate configuration states, as the `etcd` database does not provide atomic updates of multiple key-value nodes. For example, changing a service from the `ipv4=192.0.1.1 tcp=80` endpoint to the `ipv4=192.0.1.2 tcp=8080` endpoint would involve an intermediate configuration state of either `ipv4=192.0.1.2 tcp=80` or `ipv4=192.0.1.1 tcp=8080`, both of which are invalid configuration states.

The `clusterf Route`s and `ServiceBackend`s within a Docker cluster can be automatically configured by `clusterf-docker` based on the networks and containers managed by the Docker Engine. The Docker API is used to maintain an internal representation of the current state of each Docker network and container. The internal representation of the local Docker state is used to generate a local `clusterf` configuration. The `clusterf ConfigWriter` implementation is used to update and refresh the local `clusterf` configuration into `etcd`. Each local `clusterf` configuration object is compiled into `etcd` configuration nodes, which is created or updated as necessary using a `etcd` set operation. Any `etcd` configuration nodes written by the same `ConfigWriter` that are no longer present in the new configuration object are deleted from `etcd`. Each `etcd` node is written using a configurable TTL, and the `ConfigWriter` periodically issues `etcd` refresh operations to keep the configuration nodes alive. If the `clusterf-docker` process fails, the refresh operations will cease, and `etcd` will eventually expire each configuration node managed by the `clusterf-docker` instance that failed. The same mechanism is also used to handle `clusterf-docker` restarts, where the new `ConfigWriter` instance does not know about the existing configuration nodes.

Each Docker container can use the `net.qmsk.clusterf.service` label to associate itself with a named `clusterf Service`. A `clusterf ServiceBackend` will be generated using the Docker container's IPv4 and IPv6 network addresses within the default Docker bridge network. The `net.qmsk.clusterf.backend.tcp` and/or `net.qmsk.clusterf.backend.udp` labels must be used to select the exposed port to configure. A container can also be attached to multiple services using a space-separated list of service names. Different services can use different TCP/UDP ports configured using multiple `net.qmsk.clusterf.backend:$service.*` labels. The `clusterf-docker` component also allows a `clusterf Route` to be configured for the default Docker network on each machine, generated using the Docker network's IPv4 and IPv6 subnets. The `clusterf Route` is exported using a configureable IPVS forwarding method and optional `clusterf Route Gateway` address.

The `clusterf` configuration in `etcd` is used by `clusterf-ipvs` to automatically manage the runtime IPVS configuration. The `ServiceFrontend` objects are translated into IPVS services, and the corresponding `ServiceBackend` objects are used configure the IPVS destinations. Both the `ServiceFrontend` and `ServiceBackend`

JSON objects use a combination of `IPv4`, `IPv6`, `TCP` and `UDP` keys to represent a set of network endpoints. For each `clusterf` service, the `ServiceFrontend` configuration is used to configure up to four separate IPVS services, one for each valid combination of `IPv4 TCP`, `IPv4 UDP`, `IPv6 TCP` and `IPv6 UDP` addresses and ports. For each generated IPVS service, a single IPVS destination is generated for each of the `clusterf` service's `ServiceBackend`s that have a matching `IPv4` or `IPv6` address and `TCP` or `UDP` port configured. The `ServiceBackend` JSON object also contains a `Weight` key, which is used to configure IPVS destination's weight for use by the IPVS scheduler.

The `clusterf Route` objects are used to generate the IPVS destination configuration for each `clusterf ServiceBackend`. The Route `Prefix` is an IPv4 or IPv6 CIDR subnet that is matched against each `ServiceBackend IPv4` or `IPv6` address. The resulting IPVS destination for each `ServiceBackend` is configured using the most-specific matching `Route`. The `Route IPVSMethod` is used to configure the IPVS destination forwarding method, and must be either `"masq"`, `"droute"` or `"tunnel"`. If the `Route IPVSMethod` is omitted or an empty string, no IPVS destination will be configured for the `ServiceBackend`. This mechanism can be used to limit the configured IPVS destinations to those matching specific `clusterf Route`s by configuring a default `Route` with an empty `IPVSMethod`.

The two-level load balancing scheme used in the `clusterf` load balancer is based on the extension of the `Route` object to include an optional `Gateway` address. If the `Route` has a `Gateway` configured, then the IPVS destination is generated using the matching `Route Gateway` address instead of the `ServiceBackend` address. The IPVS destination is generated using the original `ServiceFrontend TCP` or `UDP` port to enable the chaining of IPVS load balancers. Since IPVS does not allow multiple identical destinations for a service, multiple `ServiceBackend` objects matching the same `Route` are merged into a single IPVS destination. This merging preserves the `ServiceBackend` weights by configuring the merged IPVS destination with the sum of the merged `ServiceBackend` weights.

IPVS supports the use of a zero destination weight value for graceful shutdown, where packets for existing connections will continue to be forwarded, but no new connections will be scheduled for the destination. Using the Docker Engine Events API, `clusterf-docker` can detect when a container has been commanded to stop, but has not yet exited. The `clusterf ServiceBackend`s for such a container will immediately be reconfigured using a `Weight` of zero, allowing the service to gracefully complete any pending connections. Once the Docker container exits, the `ServiceBackend` is removed.

The `clusterf` configuration schema can be used to conveniently represent services using a combination of IPv4 and IPv6 networking or TCP and UDP transport protocols. For example, a dual-stack DNS service could be configured as a single `clusterf` service, supporting the use of both IPv4 and IPv6 for DNS queries over

TCP or UDP transports. Not all of the `ServiceBackend`s for a `clusterf` service are necessarily required to use the same set of network families and transport protocols. The `ServiceBackend` endpoints can use different `TCP` and `UDP` ports than the `ServiceFrontend` ports when using the IPVS `masq` forwarding method. The IPVS `droute` and `tun` forwarding methods requires the backend to use the same TCP/UDP ports as the `ServiceFrontend`. A `clusterf` service that does not have any `ServiceFrontend` configured does not result in any IPVS configuration.

The two-level forwarding method requires the Docker and Gateway machines to use different IPVS configurations. The `clusterf ConfigReader` implementation can be used to read and merge multiple `clusterf` configuration trees from different configuration sources. The `ConfigReader` supports the use of multiple global `etcd` configuration sources or local filesystem configuration sources. The local configuration source can be used to configure local `Route` objects that override all `Route` objects in the global `etcd` configuration. The Gateway machines use the global `etcd Route` objects to generate IPVS configurations for all available `clusterf` services in `etcd`. The Docker machines use a locally configured `Route` corresponding to the local Docker container network to only generate IPVS configurations for `clusterf` services having local container backends.

4.6 Implementation Challenges and Future Work

As discussed in Section 4.2, the current `clusterf` implementation assumes the use of standard Ethernet networking, allowing the straightforward use of DSR techniques for load balancing. Any form of stateful firewalling or reverse path filtering based on source addresses within the external network will limit the use of DSR techniques for scaling the load balancer, requiring the use of symmetric routing paths for load balanced traffic. The `clusterf` design allows the use of stateful firewalling and NAT on the Docker machines, but the Gateway machines must use stateless firewall rules for VIPs to allow IPVS connection failover. Extending the `clusterf` network model to work within cloud networking environments that assume the use of a single network address per machine is the subject of future work.

As discussed in Section 4.1, one method to provide symmetric paths for load balancing is the use of both SNAT and DNAT for forwarded packets. The Linux IPVS implementation supports the use of Full NAT for network-level load balancing, allowing the use of a local source address on each load balancer to provide a symmetric return path. The use of Full NAT for network-level load balancing closely resembles the behavior of an application-level proxy, while potentially offering better performance. However, the use of Full NAT does not allow the server application to see the true network address of the client, and greatly complicates the implementation of connection failover when using source addresses local to each load balancer. An

alternative approach would be the use of multiple VIPs for each service [28], one for each Gateway machine. The use of DSR forwarding within the internal network preserves the individual destination VIP within each incoming connection, allowing the use of source routing policy rules on each Docker machine to provide a symmetric return path for each VIP via the internal network address of the corresponding Gateway machine.

As discussed in Section 4.3, the current design of `masq` forwarding within the Docker machines suffers from a serious issue related to NAT hairpinning within each local Docker network. The return path for a containerized client application connecting to an internally load balanced service is broken when the connection is load balanced to a backend container on the same local Docker network. The incoming packets forwarded by the Docker machine to the service container using DNAT will have source and destination addresses both within the local Docker network. The service backend container will have a direct route across the local Docker network for the incoming packet's source address, and the reply packets bypass the reverse NAT within the host machine. The client machine will be unable to associate the reply packets with the source address of the service backend, rather than the virtual network address of the load balanced service.

Possible solutions to this NAT hairpinning issue would be the use of separate local container networks for different containers, the use of Full NAT for connections, or the use of Linux `ebtables BROUTING` to force bridged packets on the virtual Docker network bridge to be processed as routed packets. An alternative approach would be the implemention of a Docker network plugin that configures the Docker container network namespace to accept unmodified packets for the virtual service network address. Assuming the use of matching service frontend and backend TCP/UDP ports, this would allow the use of `droute` forwarding on the Docker machines, using DSR for load balancing within the local Docker network, The implementation of a solution for this flaw is tracked in a GitHub project issue [29].

While the `clusterf` control plane implementation, the Linux IPVS data plane and Docker all include support for IPv6, extending the network model for IPv6 support and fixing any IPv6-related implementation issues is the subject of further work. The current implementation of `clusterf-docker` is at least partially broken for the case of a dual-stack Docker network having both IPv4 and IPv6 subnets [30]. Using external IPv6 addresses routable by the external network would avoid the need for NAT on the Docker machines. Linux IPVS also supports the use of IPv4-in-IPv6 tunneling [30], which could be used to allow the implementation of IPv6-only container networking, using separate dual-stack load balancers to tunnel incoming IPv4 connections to the containers.

[28] `https://github.com/qmsk/clusterf/issues/15`
[29] `https://github.com/qmsk/clusterf/issues/10`
[30] `https://github.com/qmsk/clusterf/issues/20`

A network-level load balancer processing individual network packets must also consider the case of fragmented IP packets. While the Linux IPVS implementation includes support for IP fragment reassembly within a single machine, the Google Maglev design also discusses further issues related to IP fragmentation and ECMP [7, Section 4.3]. Handling such differences in ECMP hashing for first and trailing fragments would require the implementation of additional fragment synchronization mechanisms between IPVS load balancers.

5 Evaluating the `clusterf` Load Balancer

The design and implementation of the `clusterf` load balancer is evaluated using the CATCP testbed. A cluster of physical server machines is used to construct a virtual network topology including two Docker clusters, and two sets of Gateway machines used for routing and load balancing. Docker containers running the `iperf` network measurement tool are used to generate TCP traffic within the test network, evaluating the `clusterf` load balancer in different networking scenarios. A centralized monitoring system is used to measure and record performance metrics across the test cluster.

5.1 Physical Infrastructure

The physical infrastructure used for the testbed network is shown in Figure 8. The physical infrastructure consists of seven identical "CATCP" machines and an eighth "obelix" machine. Each of the machines is connected to both an external network and a dedicated internal network switch. The CATCP machines are used to form a virtualization cluster that is used to run the virtual testbed network. The CATCP machines are not dedicated for the `clusterf` measurements, so some level of interference from other virtual machines must be accounted for.

A HP ProCurve 2810-48G switch is used for the internal network infrastructure, using quad-gigabit Link Aggregation Control Protocol (LACP) trunks with VLAN tagging to form the virtual network infrastructure used by the CATCP virtual machines. The ProCurve switch distributes traffic across the LACP trunk ports based on a hash of the L2 Ethernet source/destination address pairs. The CATCP machines are configured to use OVS `bond_mode=balance-tcp` to distribute traffic across the LACP trunk ports based on L4 Ethernet + IP + TCP/UDP source/destination address pairs. The maximum capacity between any two Ethernet hosts is thus limited to 1 Gbit/s by the ProCurve switch's LACP transmit balancing, up to a maximum of 4 Gbit/s per physical machine distributed across multiple targets.

Each CATCP machine is a Dell R410 server with an Intel E5640 quad-core CPU and 12GB of memory, connected to the CATCP switch with a quad-gigabit LACP trunk. Each physical CATCP machine has 4 logical CPU cores, for a total of 8 hyperthreads. The physical network interfaces are Intel gigabit NICs, using eight queues to distribute the packet processing workload across all CPU cores. The CATCP machines run the Ubuntu Linux 3.13 kernel included in Ubuntu 14.0 . The CATCP machines use Open vSwitch (OVS) for networking, using VLAN tagging to form multiple virtual networks over the LACP trunks to the internal CATCP network switch. The CATCP machines form a virtualization cluster, running QEMU KVM virtual machines. Each virtual machine may have multiple `virtio` virtual network interfaces attached to different virtual networks via OVS.

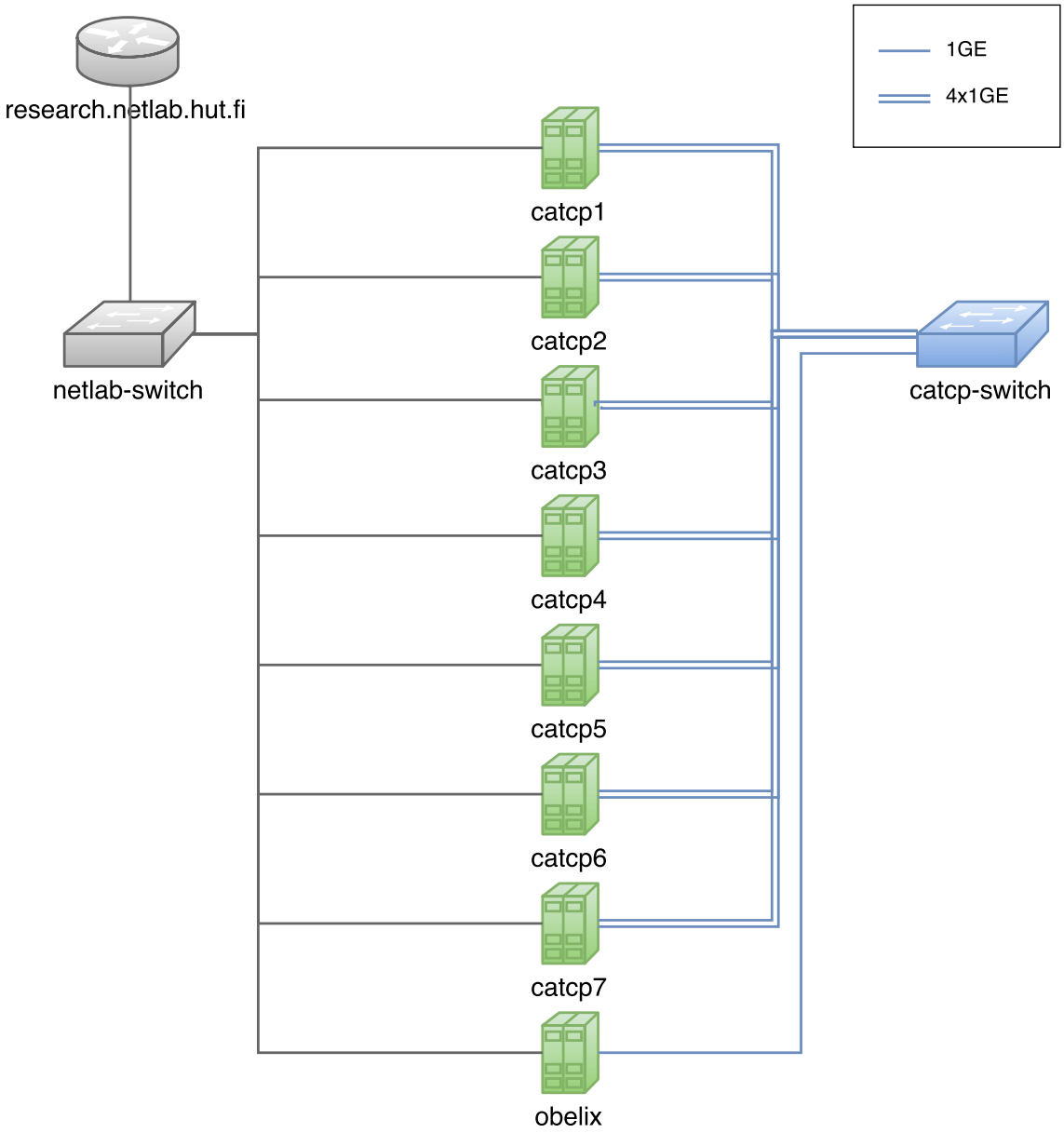

Figure 8: The physical testbed machines and network topology

The CATCP machines are used to run the virtual machines used to form the evaluation testbed. Each virtual machine used for the measurement infrastructure has 2 virtual CPUs. The virtual network interfaces are virtio devices, using a single queue to process all packets on the first CPU core. The virtual machines use Linux Receive Packet Steering (RPS) to distribute as much of the kernel packet processing workload across each CPU as possible. Without RPS, the network performance of a gateway machines is limited by the saturation of the first CPU core at full `softirq` utilization. Tuning the Linux networking stack using `/sys/class/net/eth-*/queues/rx-*/rps_cpus=3` provides improved network performance with more even CPU utilization.

The Obelix machine is a Dell R410 server with dual Intel X5570 quad-core CPUs and 64GB of memory, connected to the CATCP switch with a single gigabit link. The Obelix machine is used to run the central monitoring system used for network performance measurements. The Obelix machine runs multiple LXC containers having VLAN network interfaces connected to the same virtual networks used by the CATCP machines. The Obelix machine is not used directly for any network measurements.

5.2 Virtual Infrastructure

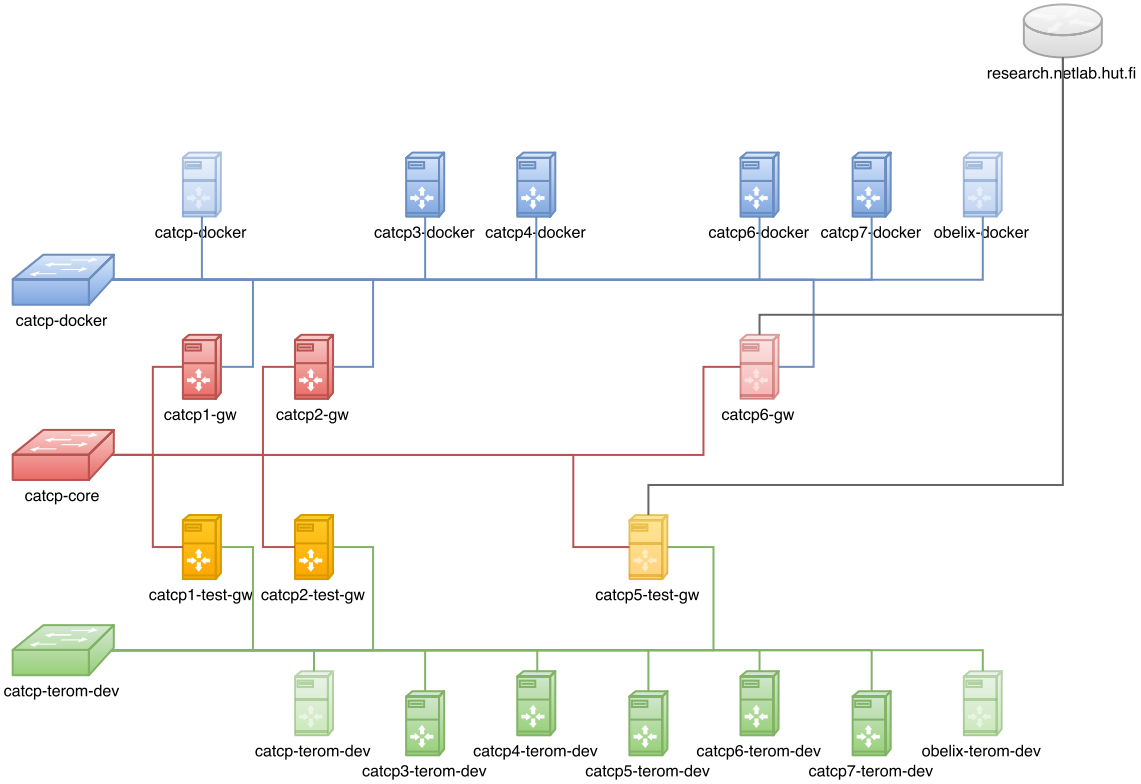

Figure 9: The virtual testbed machines and network topology

The testbed environment used to evaluate the `clusterf` load balancer consists of multiple Docker and Gateway machines, running as virtual machines on the physical CATCP machines. The virtual machines are connected to a number of virtual Ethernet networks to form the testbed network topology shown in Figure 9. Each virtual machine is named per the physical machine they are running on, as arranged into vertical columns in Figure 9. The numbered **catcpX-*** virtual machine are fixed to the corresponding **catcpX** physical machine, using local disk storage. The unnumbered **catcp-*** virtual machines are free to move within the CATCP virtualization cluster, using clustered Storage Area Network (SAN) disks that allows for live migration across physical machines. The numbered **catcpX-*** virtual machines are designed for horizontally scalable applications, whereas the clustered **catcp-*** virtual machines are designed for other applications that lack application-level fault tolerance.

The majority of the virtual machines runs the Linux 4.4 kernel included in Ubuntu 16.04. The **catcp-terom-dev** virtual machines mostly run the Linux 3.16 kernel included in Debian `jessie`, apart from the **catcp7-terom-dev** virtual machine, which runs the same Linux 4.4 kernel included in Ubuntu 16.04. All of the machines

use the Linux `iptables` firewall to enforce network access policies, including the use of connection tracking for stateful firewall rules. All of the machines use Puppet [31] for configuration management, used to ensure a consistent system configuration for each machine.

The virtual testbed contains a total of three virtual Ethernet networks and two Docker clusters. The **catcp-docker** and **catcp-terom-dev** Docker machines form two separate Docker clusters. Each cluster of Docker machines is connected to a separate internal Ethernet network, using internal `10.109.0.0/16` (**catcp-docker**) and `10.107.0.0/16` (**catcp-terom-dev**) network addresses. Each Docker machine is assigned a unique internal network subnet (**catcpX-docker** `10.X.109.0/24`, **catcpX-terom-dev** `10.X.107.0/24`) for use by Docker containers within the local Docker bridge. The **catcp-gw** and **catcp-test-gw** Gateway machines act as routers, interconnecting the Docker cluster networks via a common core network, using internal `10.255.0.0/16` network addresses.

Per the `clusterf` network model discussed in Section 4.2, the Open Shortest Path First (OSPF) routing protocol is used to provide network routing within the testbed, including each machine attached to the internal networks and any local Docker container networks. Each Docker and Gateway machine runs the `bird` [32] routing daemon, configured for OSPF routing within the OSPF `0.0.0.0` backbone area. Each Docker machine's `docker0` bridge is configured as an OSPF stub interface in *bird*, and is thus advertised as an OSPF stubnet to the other machines. Network traffic between Docker containers within each Docker cluster is routed directly across the virtual Ethernet network between the Docker machines. Network traffic between Docker containers in different clusters is routed via the Gateway machines across the core network.

Using OSPF routing within the testbed network allows the use of Equal-Cost Multi-Path (ECMP) routing for inter-cluster traffic. The `bird` OSPF routing daemon on each machine is configured to enable ECMP routing, installing multiple Linux next-hops for each matching set of OSPF routes. With multiple Gateway machines providing alternate paths across the network, each machine will have multiple equal-cost routes for each subnet, allowing the use of L3 load-balancing for traffic across the testbed network. While the use of L2 load balancing within the physical network infrastructure limits the total throughput for each symmetric routing path across the cluster to 1Gbit/s, using L3 to distribute traffic across multiple L2 flows across the same network allows achieving a higher total throughput of up to 4Gbit/s per machine.

The **catcp-test-gw** machines act as load balancers for the test services on the **catcp-terom-dev** machines. The **catcp1-test-gw** and **catcp2-test-gw** machines use internal VIPs within `10.0.107.0/24` for loadbalancing, announced via OSPF.

[31] `https://puppet.com/`
[32] `http://bird.network.cz/`

Each of the two gateway machines announces the `10.0.107.0` VIP with the same OSPF cost. Incoming traffic for the shared `10.0.107.0` VIP is distributed evenly across the two gateway machines using ECMP routing on the **catcp-gw** machines. The `10.0.107.1` and `10.0.107.2` VIPs are announced by each of the two gateway machines using a different weight. The **catcp1-test-gw** machine acts as the primary router for the `10.0.107.1` VIP, and the **catcp2-test-gw** machine acts as the primary router for the `10.0.107.2`. In the case of gateway machine failure, the remaining gateway machine will be used for all three VIPs.

The **catcp-test-gw** machines also act as network routers for the return path from the **catcp-terom-dev** machines. The **catcp-gw** machines act as external network routers, handling traffic for the **catcp-docker** test clients. The **catcp5-test-gw** and **catcp6-gw** machines are used for infrastructure purposes, and have external network connectivity. These gateway machines are used as NAT gateways for outgoing external traffic from the Docker machines, as well as load balancing traffic for the centralized measurement services running on the **obelix-*** containers. The **catcp5-test-gw** and **catcp6-gw** machines are configured with higher OSPF costs on the **catcp-terom-dev** and **catcp-docker** networks to avoid routing any inter-cluster traffic.

The baseline performance measurements in Section 6.3 show that each CATCP virtual machine is capable of handling a minimum of 10Gbit/s of `iperf` traffic within the local Docker network, and a minimum of 1Gbit/s of `iperf` traffic across the **catcp-terom-dev** network.

5.3 Docker Infrastructure

Each Docker cluster uses Docker Swarm for orchestration, using the numbered **catcpX-*** virtual machines as Swarm nodes, and the clustered **catcp-*** virtual machine as the Swarm Manager. Each Docker machine runs Docker 1.11. Due to the use of customized `iptables` policy, the Docker `iptables` configuration mechanism and `docker-proxy` functionality are disabled, meaning that the standard Docker port publishing mechanism cannot be used.

Each Docker cluster uses `etcd` for shared configuration, and SkyDNS for service discovery within the internal `*.catcp` domain. The Docker machines within each Docker cluster are used as `etcd` nodes, running a five-node `etcd` cluster. Each Docker machine runs a modified version of Gliderlabs `registrator` [33] as well as `clusterf-docker` to maintain a service discovery database of running Docker services within `etcd`. Both Docker clusters include SkyDNS services providing dynamic `*.docker.test.catcp` and `*.docker.catcp` DNS names for each Docker container and service within each Docker cluster. The internal SkyDNS domains are delegated

[33] `https://github.com/qmsk/registrator`

within the internal `*.catcp` DNS hierarchy, allowing the use of all such DNS names across both Docker clusters.

The **catcp5-test-gw** and **catcp6-gw** machines within each cluster provide `clusterf` load balancing for internal infrastructure services. Each Docker cluster also runs the Vulcand HTTP load balancer, using both internal DNS service discovery and the `clusterf` load balancer to expose internal and external HTTP services. Such HTTP services include a private Docker Registry and the Grafana service used for monitoring.

5.4 Measurement Tools

A simple `clusterf-test` client-server application is used to test the routing of connections within the testbed network using different load balancing methods. A Docker container is used to run the `clusterf-test` server, listening on a configurable TCP port, registered as a `clusterf` service backend. The `cluserf-test` client connects to a configurable destination DNS or network address and TCP port, where the `clusterf-test` server returns a string describing the remote and local network addresses seen by the server application. The `clusterf-test` client application prints both the outgoing TCP connection's local and remote network addresses, and the corresponding addresses sent by the server. An example of the output printed by the `clusterf-test` client application is shown in Figure 10.

```
client:        10.8.109.4:51158 -> 10.0.107.1:1337
server:        10.8.109.4:51158 -> 10.8.107.4:1338
```

Figure 10: Example `clusterf-test` output for a connection from the `10.8.109.4` client container to TCP port 1337 on the `10.0.107.1` VIP, using `clusterf` to load balance the connection to the `10.8.107.4` server container listening on TCP port 1338

The **iperf** network measurement tool is used to generate network traffic for each experiment, measuring both the baseline throughput of the network, and each load balancing scenario tested. The **catcp-docker** cluster is used to run multiple `iperf` clients, connecting to a horizontally scaled `iperf` service consisting of `iperf` servers on the **catcp-terom-dev** cluster. The `iperf` clients generate upload traffic to the `iperf` clients, resulting in high network transmission rates from the **catcp-docker** machines, via the **catcp-gw** machines to the **catcp-test-gw** machines, and on to the **catcp-terom-dev** machines. The `iperf` server on the **catcp-terom-dev** machines only returns TCP ACK packets.

5.5 Load Balancing

Three different techniques are used to load balance connections from the `iperf` clients to the `iperf` servers. Each tested load balancing technique uses `etcd` for dynamic configuration, using dynamic service discovery information registered for each running `iperf` server container. The load balancer data plane components are run on the **catcp-test-gw** machines.

DNS

The `SkyDNS` support for hierarchially classified services is used to resolve the container network address of the `iperf` server running on each `catcp-terom-dev` machine. The container network addreses on the **catcp-terom-dev** machines are also routable from the **catcp-docker** machines. The baseline measurements in section 6.3 use a full mesh of `iperf` clients, with each client machine connectiong to each server machine.

HAProxy + `confd`

The application-level `HAProxy` load balancer is used in combination with the `confd` dynamic configuration mechanism to provide transport-layer load balancing of TCP connections. The `confd` configuration template shown in Figure 11 is used to provide dynamic HAProxy configuration for services registered in `etcd` by `clusterf-docker`. The use of the `clusterf` configuration state for HAProxy ensures an identical load balancing configuration as for `clusterf`.

`clusterf`

The network-level `clusterf` load balancer is used to provide L4 load balancing of TCP connections using the Linux IPVS load balancer. Each Docker machine runs the `clusterf-docker` service discovery agent, managing the `clusterf` configuration state in `etcd` for each running Docker container associated with a `clusterf` service. Each Gateway machine and Docker machine runs the `clusterf-ipvs` agent, manging the local IPVS configuration. The design and implementation of `clusterf` is discussed in Chapter 4.

Each of the **catcp1-test-gw** and **catcp2-test-gw** machines can be configured to use either HAProxy or `clusterf` for load balancing. Switching between the two involves stopping or starting the respective systemd `haproxy.service` and `clusterf-ipvs.service` services. The `clusterf-ipvs` service is configured to flush

```
{{ range $service := lsdir "/services"}}
# {{$service}}
{{- $frontend := json (getv (printf "/services/%s/frontend"
   $service) "null") -}}
{{- if $frontend.tcp }}
listen clusterf-{{$service}}-tcp
    mode tcp

    {{ if $frontend.ipv4 -}}
    bind {{$frontend.ipv4}}:{{$frontend.tcp}}
    {{- end -}}
    {{- if $frontend.ipv6 -}}
    bind {{$frontend.ipv6}}:{{$frontend.tcp}}
    {{- end }}

    {{ range $node := gets (printf "/services/%s/backends/*"
        $service) }}
    {{-  $backendName := base $node.Key -}}
    {{-  $backend := json $node.Value -}}
    # {{$backendName}}
    {{-  if and $backend.ipv4 $backend.tcp }}
    server {{$backendName}} {{$backend.ipv4}}:{{$backend.tcp
       }} weight {{$backend.weight}}
    {{   end }}
    {{-  if and $backend.ipv6 $backend.tcp }}
    server {{$backendName}} {{$backend.ipv6}}:{{$backend.tcp
       }} weight {{$backend.weight}}
    {{   end }}
    {{ end -}}
{{- end }}
{{ end }}
```

Figure 11: `confd` configuration template for HAProxy using the `clusterf etcd` schema

the IPVS configuration when stopping, allowing HAProxy to pick up the connections. The use of the correct load balancing method is tested by monitoring the internal performance metrics reported by each load balancer.

5.6 Load Measurements

A centralized monitoring system is used to monitor various performance metrics across the entire testbed network, including the physical CATCP machines and all virtual machines. The use of a centralized monitoring system allows the correlation of multiple performance metrics across multiple machines in real time, providing a better understanding of how the system behaves under testing. The centralied monitoring system can be used to detect any resource contention bottlenecks within the underlying physical testbed.

Analyzing the performance of a complex system being load tested requires analyzing the resource utilization, saturation and error rates across each of the underlying components in order to identify the underlying reasons for any resulting performance bottlenecks [31]. Verifying the correct behaviour of the system is also important for determining the accuracy of the results. For example, initial tests of the load balancers exhausted the default Linux `conntrack` table limits on the **catcp-test-gw** machines, causing the Linux firewall to drop the TCP SYN packets. Increasing the `conntrack` table limits significantly improved the results of the affected measurements. Verifying the accuracy of the measurements requires monitoring the utilization of the `conntrack` table to detect any dropped connection errors when saturated.

The Telegraf [34] monitoring agent runs on each virtual machine, and periodically samples various system and application performance counters. The Collectd agent also runs on every machine within the measurement infrastructure, including the physical CATCP machines. Both the `telegraf` and `collectd` agents are configured to sample each performance metric at 10-second intervals. The collected metrics are sent to a central InfluxDB [35] time-series database on the **obelix** server for storage and aggregation. During the operation of the system, the gathered metrics are queried and visualized using the Grafana [36] web application.

Various `telegraf` and `collectd` plugins are used to gather relevant performance metrics, using a Grafana dashboard to monitor test runs, and using InfluxDB queries to generate the results shown in the following Section 6.

[34] `https://influxdata.com/time-series-platform/telegraf/`
[35] `https://influxdata.com/time-series-platform/influxdb/`
[36] `http://grafana.org/`

CPU

The `telegraf cpu` input plugin is used to gather CPU utilization percentages per CPU state for each CPU core.

The following InfluxDB query structure is used to report the CPU utilization in percentages:

```
SELECT 100.0 - usage_idle - usage_steal AS util FROM cpu
    WHERE host ~= /catcp[0-9]+-terom-dev/ AND cpu != 'cpu-
    total' AND time >= $from AND time <= $to GROUP BY host,
    cpu
```

Network Interfaces

The `telegraf` *net* input plugin is used to gather network interface counters for each network interface.

The following InfluxDB query structure is used to report the interface utilization rates in bits/s:

```
SELECT non_negative_derivative(bytes_sent, 1s) * 8 AS tx_bps
    , non_negative_derivative(bytes_recv, 1s) * 8 AS rx_bps
    FROM net WHERE host =~ /catcp[0-9]+-terom-dev/ AND
    interface = 'eth-terom-dev' AND time >= $from AND time <=
     $to GROUP BY host, interface
```

Docker Containers

The `telegraf docker` input plugin is used to gather the CPU and network utilization of labeled Docker Containers.

The following InfluxDB query structure is used to report the Docker network interface utilization rates in bits/s:

```
SELECT non_negative_derivative(rx_bytes, 1s) * 8 AS rx_bps
    FROM docker_container_net WHERE iperf = 'server' AND
    network = 'eth0' AND time >= $from AND time <= $to GROUP
    BY host, container_name
```

Background Signals

The `collectd ipvs` plugin is used to verify that connections are being load balanced by IPVS as expected.

The `telegraf haproxy` input plugin is used to verify that connections are being load balanced by HAProxy as expected.

The `collectd conntrack` plugins is used to monitor the size of the Linux conntrack table, verifying that no connections are being dropped due to the saturation of the `conntrack` table used for stateful firewalling.

6 Results and Analysis

The primary purpose of the experiments is to verify the design of the `clusterf` load balancer, evaluating the behavior of the `clusterf` load balancer under different load balancing scenarios including load balancer failures. The secondary purpose of the experiments is to compare the performance of the Linux IPVS implementation of a network-level load balancer used in the `clusterf` load balancer with the HAProxy implementation of an application-level load balancer. The experiment design does not include any evaluation of the performance of the `clusterf` load balancer control plane itself.

The following hypotheses are used to design the set of experiments performed and analyzed:

1. The `clusterf` network-level load balancer is able to preserve the original network source address of the client application for load balanced connections, which is not possible when using a application-level load balancer.

2. The use of local NAT forwarding within the `clusterf` load balancer allows the use of backend servers listening on a different port number than the port number used by the client to connect to the service.

3. The `clusterf` load balancer is capable of distributing connections across multiple backends.

4. A single instance of the Linux IPVS network-level load balancer provides better performance than a single instance of the HAProxy application-level load balancer.

5. The use of distributed DSR forwarding within the `clusterf` load balancer allows the use of ECMP routing across multiple load balancers to scale the throughput for a single VIP.

6. The use of the network-level IPVS load balancer allows the use of connection state synchronization to scale the reliability of a single VIP by migrating active connections away from a failed load balancer.

The baseline performance provided by the tested infrastructure itself is first evaluated using a series of measurements, including local performance within each machine, intra-cluster performance across the machines within a network, and inter-cluster performance between networks. We compare implementations of network-level and application-level load balancing using a symmetric routing scenario, with a single gateway machine used to route all network packets for both the incoming path and the return path. We evaluate the use of ECMP routing for horizontal scaling, using multiple IPVS load balancers to provide additional throughput while allowing for connection failover.

The figures and tables included in this section are rendered from the gathered metrics stored in the InfluxDB database. A Python script is used to run InfluxDB queries using the specific time range of each experiment's test run, rendering the included plots using `matplotlib`.

6.1 Challenges

The use of shared virtual machine and network infrastructure for performance measurements is challenging at best and misleading at worst. The use of multiple virtual machines contending for the same physical machine resources does not provide perfect performance isolation between different virtual machines. Saturating the packet forwarding capacity of a virtual machine will partially saturate the physical CPU resoures of the physical machine, which may affect the performance of a neighboring virtual machine.

Careful analysis of the measurement results either requires an experimental design avoiding any such resource contention, or requires measurement results to be qualified with any such bottlenecks identified. Segregating the different virtual machines across diffrent physical machines provides a reasonable level of performance isolation for most experiments. The physical CATCP1 and CATCP2 machines were selected to run the virtual Gateway machines used for performance measurements, as they have less background load than the other physical CATCP machines. The use of a centralized measurement platform including both physical and virtual machines allows us to identify any bottlenecks formed by the underlying physical infrastructure.

The experimental design must account for the use of multiple levels of load balancing within the testbed network:

- Docker Swarm's dynamic scheduling of Docker containers across differnet Dockermachines
- L2 load-balancing of traffic between each pair of virtual machines within each virtual ethernet network across multiple LACP trunk links
- L3 load-balancing of traffic across multiple Gateway machines using ECMP routing
- L4 load-balancing of connections across multiple backends using IPVS/HAProxy

Each experiment uses statically assigned machines to run each iperf client, avoiding the use of Docker Swarm for dynamic scheduling. The use of L2 load balancing across LACP trunks of gigabit links limits the maximum throughput achievable for any given measurement. The inter-cluster symmetric routing tests are by design

limited to a maximum 1Gbps by the use of a single L2 flow between the pair of Gateway machines. Congestion between multiple L2 flows hashing onto the same trunk port may potentially interfere with the results. The distribution of L3 flows across gateway machines and L4 flows across server machines is included in each set of results.

6.2 Connection Routing

The `clusterf-test` client-server application is used to test the behavior of TCP connections routed through the testbed network. For each of the three VIPs used for load balancing, one instance of the `clusterf-test` server is run on each of the **catcp-terom-dev** machines, for a total of $3 * 5$ server instances. Connections to the `10.0.107.1` VIP are routed to the **catcp1-test-gw** machine, which uses the `clusterf` IPVS load balancer to forward each connection to one of the five **catcpX-terom-dev** backends having an internal IP address of the form **10.X.107.Y**. Connections to the `10.0.107.2` VIP are routed to the **catcp2-test-gw** machine, which uses the HAProxy load balancer to likewise proxy each connection to one of the five **catcp-terom-dev** backends.

6.2.1 DNS Service Discovery

The correct operation of the SkyDNS service used for service discovery within the cluster is verified by running a single `clusterf-test` client on the **catcp-docker** machine, using DNS names within the `clusterf-test.docker.test.catcp` domain to connect to the `clusterf-test` servers on the **catcp-terom-dev** machines.

```
client: 10.9.109.3:43774 -> 10.5.107.3:1337
server: 10.9.109.3:43774 -> 10.5.107.3:1337

client: 10.9.109.3:39176 -> 10.3.107.4:1337
server: 10.9.109.3:39176 -> 10.3.107.4:1337
```

Figure 12: The `clusterf-test` client output when using the `clusterf-test.docker.test.catcp` DNS name with the network address of each `clusterf-test` server

Figure 12 shows the results of multiple client runs using the `clusterf-test.docker.test.catcp` DNS name. A different server instance may handle each connection, using application-level DNS load balancing across each of the `clusterf-test` servers. The use of standard IP network routing without any NAT means that both the client and server see the same network source and destination addresses.

```
client: 10.9.109.3:39166 -> 10.3.107.4:1337
server: 10.9.109.3:39166 -> 10.3.107.4:1337

client: 10.9.109.3:53788 -> 10.4.107.3:1337
server: 10.9.109.3:53788 -> 10.4.107.3:1337
```

Figure 13: The `clusterf-test` client output when using the `catcpX-terom-dev.clusterf-test.docker.test.catcp` DNS name with the network address of each `clusterf-test` server on the **catcpX-terom-dev** machine

Figure 13 shows the results of two consecutive client runs using the `catcp3-terom-dev.clusterf-test.docker.test.catcp` and `catcp4-terom-dev.clusterf-test.docker.test.catcp` DNS names, respectively. The name will always resolve to a server instance running on the named **catcpX-terom-dev** machine, having a corresponding **10.X.107.0/24** container network addresses.

6.2.2 Application-level proxying

```
client:  10.9.109.3:34504 -> 10.0.107.2:1337
server: 10.107.13.2:43996 -> 10.5.107.4:1337

client:  10.9.109.3:34508 -> 10.0.107.2:1337
server: 10.107.13.2:57446 -> 10.3.107.5:1337
```

Figure 14: The `clusterf-test` client output when using the `10.0.107.2` VIP routed to **catcp2-test-gw** using the HAProxy application-level load balancer

Figure 14 shows the results of multiple client runs using the `10.0.107.2` VIP routed to the HAProxy load balancer on **catcp2-test-gw**, proxying connections across the server instances on the **catcp-terom-dev** machines. The network source and destination addresses seen by the client and server applications are different. The client uses the destination address `10.0.107.2` routed to the proxy server, and the server sees the internal source address of the proxy server `10.107.13.2` within the **catcp-terom-dev** network. The server application is unable to see the true network address of the client application.

6.2.3 Network-level forwarding

```
client: 10.9.109.3:38314 -> 10.0.107.1:1337
server: 10.9.109.3:38314 -> 10.7.107.5:1337

client: 10.9.109.3:38330 -> 10.0.107.1:1337
server: 10.9.109.3:38330 -> 10.6.107.5:1337
```

Figure 15: The `clusterf-test` client output when using the `10.0.107.1` VIP routed to **catcp1-test-gw** using the IPVS network-level load balancer

Figure 15 shows the results of multiple client runs using the `10.0.107.1` VIP routed to the **catcp1-test-gw** machine using the `clusterf` network-level load balancer. The `clusterf` load balancer uses the two-level forwarding scheme presented in Section 4.3 to load balance the connections across the server instances listening on the same TCP port on each **catcp-terom-dev** machine. The network source addresses seen by the client and server applications are the same, but the destination addresses are different. The packets for each test connection are routed to the **catcp1-test-gw** machine, which forwards them unmodified to the **catcp-terom-dev** machines, which then forwards them to the server container using DNAT. The server application can see the true network address of the client application.

6.2.4 Network-level forwarding with port translation

For this test scenario, the TCP port used by each `clusterf-test` server is changed, configuring each **catcpX-terom-dev** instance to listen on TCP port $1330+X$. The `clusterf ServiceFrontend` still uses the same TCP port 1337 as before, but each `clusterf ServiceBackend` uses a different TCP port.

```
client: 10.9.109.3:38358 -> 10.0.107.1:1337
server: 10.9.109.3:38358 -> 10.6.107.5:1336

client: 10.9.109.3:38360 -> 10.0.107.1:1337
server: 10.9.109.3:38360 -> 10.5.107.5:1335

client: 10.9.109.3:38356 -> 10.0.107.1:1337
server: 10.9.109.3:38356 -> 10.7.107.5:1337
```

Figure 16: The `clusterf-test` client output when using the `10.0.107.1` VIP routed to **catcp1-test-gw** using the IPVS network-level load balancer with different server ports

Figure 16 shows the results of multiple client runs using the `10.0.107.1` VIP routed to the **catcp1-test-gw** machine using the `clusterf` network-level load balancer. The `clusterf` load balancer uses the two-level forwarding scheme presented in Section 4.3 to load balance the connections across the server instances listening on a different TCP port on each **catcp-terom-dev** machine. The network source addresses seen by the client and server applications are the same, but the destination addresses and ports are different. Each **catcp-terom-dev** machine uses NAT forwarding for each connection to the `clusterf-test` service on TCP port 1337, allowing the use of different TCP ports within the server containers. Even with the use of heterogeneous server ports, the server application can see the true network address of the client application.

6.3 Baseline Measurements

Baseline measurements are used to determine the performance limitations of the testbed network environment. This includes the **catcp-terom-dev** machines running the `iperf` Docker containers, the **catcp-terom-dev** network interconnecting the Docker machines, and the **catcp-test-gw** machines interconnecting the networks.

6.3.1 Local container network

To measure the baseline performance of the local Docker network within each **catcp-terom-dev** machine, we run a single instance of the `iperf` server on each **catcp-terom-dev** machine. For each **catcp-terom-dev** machine, we run a single `iperf` client connecting to the local `iperf` server, using the `catcpX-terom-dev.iperf.docker.test.catcp` DNS name provided by SkyDNS to resolve the address of the local `iperf` server. The resulting traffic within each **catcp-terom-dev** machine is switched across the internal `br-docker` bridge, with the Linux kernel bridging traffic between neighboring Docker containers. The resulting traffic is limited by the CPU processing resources available on each **catcp-terom-dev** machine.

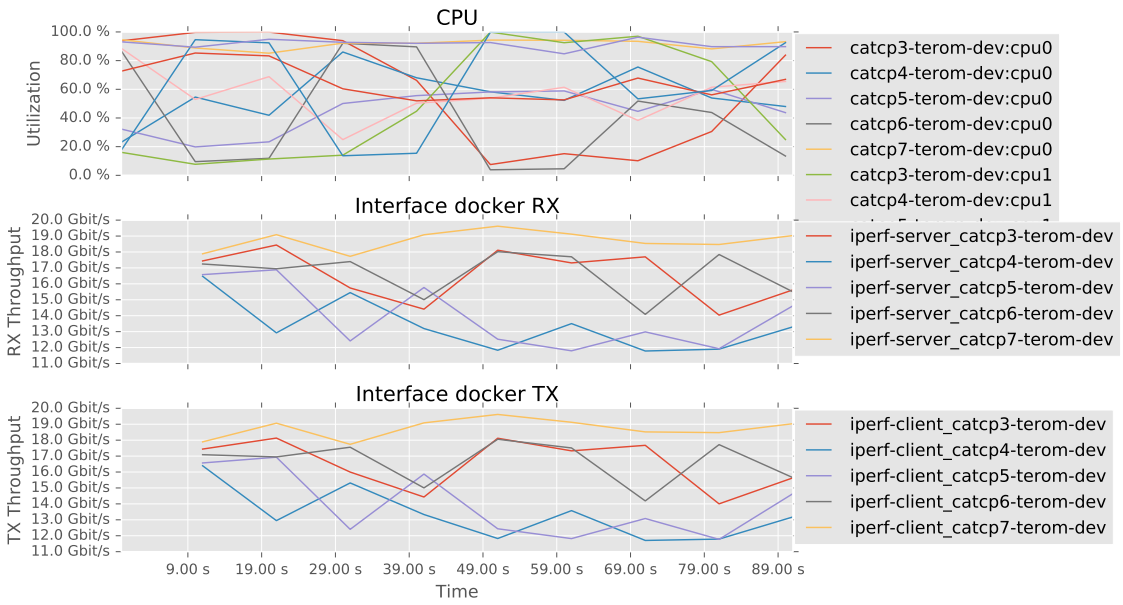

Figure 17: `iperf` measurement results for the baseline performance of local bridging within each local **catcp-terom-dev** Docker network

Figure 17 shows the achieved `iperf` upload throughput and resulting CPU utilization for the iperf upload test, with each iperf client connecting directly to the local iperf server (`iperf -c catcpX-terom-dev.iperf.docker.test.catcp -t 120`). The results show a higly varying level of CPU utilization for each **catcp-terom-dev** machine, handling both the `iperf` client and server processes, as well as packet switching of both upload traffic and return TCP ACK packets. The different machines achieve varying upload rates between 11 - 20 Gbit/s.

While the results vary between servers, the results indicate that each Docker machine is capable of handling at least 10Gbit/s of `iperf` traffic.

6.3.2 Intra-cluster

To measure the baseline performance of the internal cluster network, we run a single instance of the `iperf` service on each **catcp-terom-dev** machine. For each **catcp-terom-dev** machine, we run multiple `iperf` clients, each connecting to a remote `iperf` server, using the `catcpX-terom-dev.iperf.docker.test.catcp` DNS name provided by SkyDNS to resolve the address of each remote `iperf` server. This results in a total of $n * (n - 1)$ `iperf` connections, one for each pair of **catcp-terom-dev** machines.

The resulting traffic between **catcp-terom-dev** machines is routed across the local

cluster network infrastructure, with the Linux kernel routing traffic to the neighboring **catcp-terom-dev** machines using routes provided by OSPF. The resulting traffic is limited by both the CPU processing resources available on each **catcp-terom-dev** machine, and the L2 hashing used for the LACP trunks. The L2 flows between the **catcp-terom-dev** machines are distributed across three of the four LACP trunk ports, limiting the maximum incoming bandwidth for each machine to 3Gbit/s.

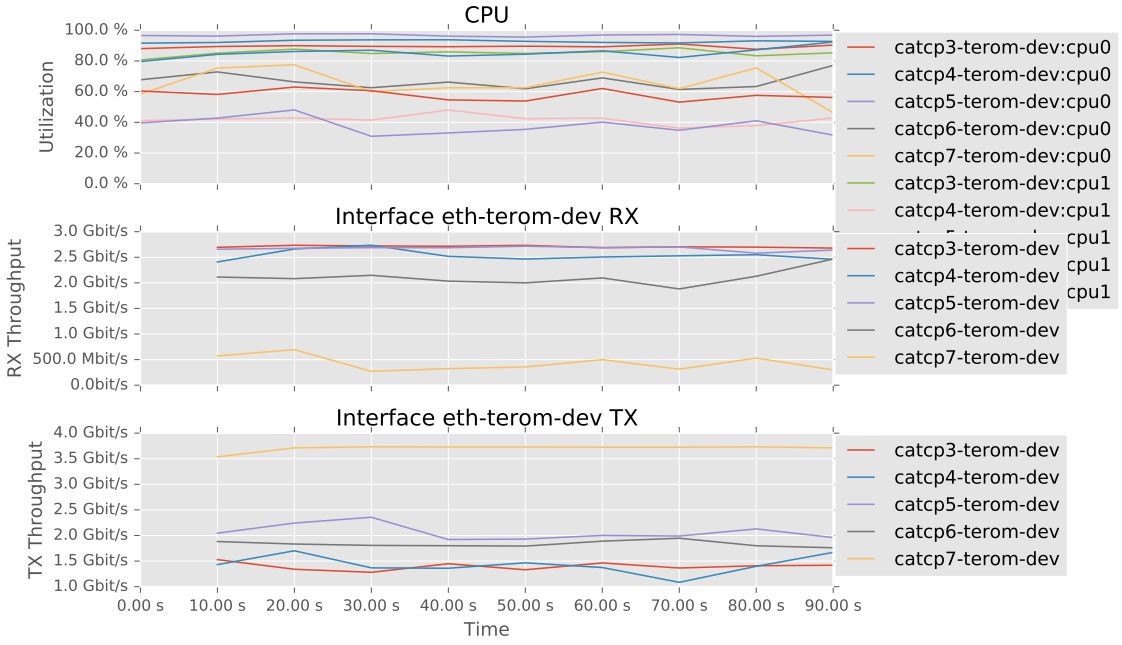

Figure 18: `iperf` measurement results for the baseline performance of intra-cluster routing between **catcp-terom-dev** machines

Table 2: Measurements for intra-cluster `iperf` between **catcp-terom-dev** machines

Interface eth-terom-dev RX	**Min**	**Max**	**95'th percentile**
catcp3-terom-dev	2.7 Gbit/s	2.7 Gbit/s	2.7 Gbit/s
catcp4-terom-dev	2.4 Gbit/s	2.7 Gbit/s	2.7 Gbit/s
catcp5-terom-dev	2.6 Gbit/s	2.7 Gbit/s	2.7 Gbit/s
catcp6-terom-dev	1.9 Gbit/s	2.5 Gbit/s	2.3 Gbit/s
catcp7-terom-dev	270.6 Mbit/s	694.3 Mbit/s	645.2 Mbit/s
Total	270.6 Mbit/s	2.7 Gbit/s	11.1 Gbit/s
Interface eth-terom-dev TX	**Min**	**Max**	**95'th percentile**
catcp3-terom-dev	1.3 Gbit/s	1.5 Gbit/s	1.5 Gbit/s
catcp4-terom-dev	1.1 Gbit/s	1.7 Gbit/s	1.7 Gbit/s
catcp5-terom-dev	1.9 Gbit/s	2.4 Gbit/s	2.3 Gbit/s
catcp6-terom-dev	1.8 Gbit/s	1.9 Gbit/s	1.9 Gbit/s
catcp7-terom-dev	3.5 Gbit/s	3.7 Gbit/s	3.7 Gbit/s
Total	1.1 Gbit/s	3.7 Gbit/s	11.2 Gbit/s

Figure 18 shows the achieved `iperf` upload throughput and resulting CPU utilization for the iperf upload test, with each iperf client connecting directly to each remote iperf server (`iperf -c catcpX-terom-dev.iperf.docker.test.catcp -t 300`). The results show a high level of CPU utilization for each **catcp-terom-dev** machine, handling both the `iperf` client and server processes, as well as routing of both upload traffic and return TCP ACK packets. The different machines achieve varying upload rates between 1.1 - 3.7 Gbit/s, for a total combined 95th percentile throughput of approximately 11Gbit/s as shown in table 2.

The behaviour of the **catcp7-terom-dev** machine is an outlier, with the `iperf` clients on the machine sending significant more traffic than on other machines, and the `iperf` server receiving significantly less traffic than on other machines. One possible explanation for this behaviour is the use of a different Linux kernel on the **catcp7-terom-dev** machine (Linux 4.4) than on the other **catcp-terom-dev** machines (Linux 3.16). The newer Linux kernel may prioritze the `iperf` client processes sending traffic over the local `iperf` server process receiving traffic.

The results show that packet routing across the internal network is more CPU-intensive than simply switching traffic between local Docker containers, with a higher CPU utilization for a lower throughput. The results are also highly suspectible to the effects of the L2 hashing algorithm used for LACP trunking, which determine the maximum achieivable throughput in quantiles of the gigabit Ethernet links used for trunking.

6.3.3 Inter-cluster symmetric routing

To measure the baseline performance of the inter-cluster network, we run an instance of the `iperf` service on each of the **catcp-terom-dev** machines. For each **catcp-docker** machine, we run multiple `iperf` clients connecting to each remote `iperf` server, using the `catcpX-terom-dev.iperf.docker.test.catcp` DNS name provided by SkyDNS to resolve the address of each remote `iperf` server. This results in a total of $N * M$ `iperf` clients, one for each pair of **catcp-terom-dev** and **catcp-docker** machines.

The resulting traffic between the **catcp-docker** and **catcp-terom-dev** machines is routed by the **catcp1-gw** and **catcp2-test-gw** machines across the core network. For the symmetric routing scenario, the other **catcp-gw** and **catcp-test-gw** routers are disabled (configured with a higher OSPF cost). The resulting inter-cluster traffic is limited to 1Gbit/s by the use of L2 hashing for the single L2 flow between the two gateway machines, as all packets for the **catcp2-test-gw** machine are hashed to the same LACP trunk port on the CATCP2 machine.

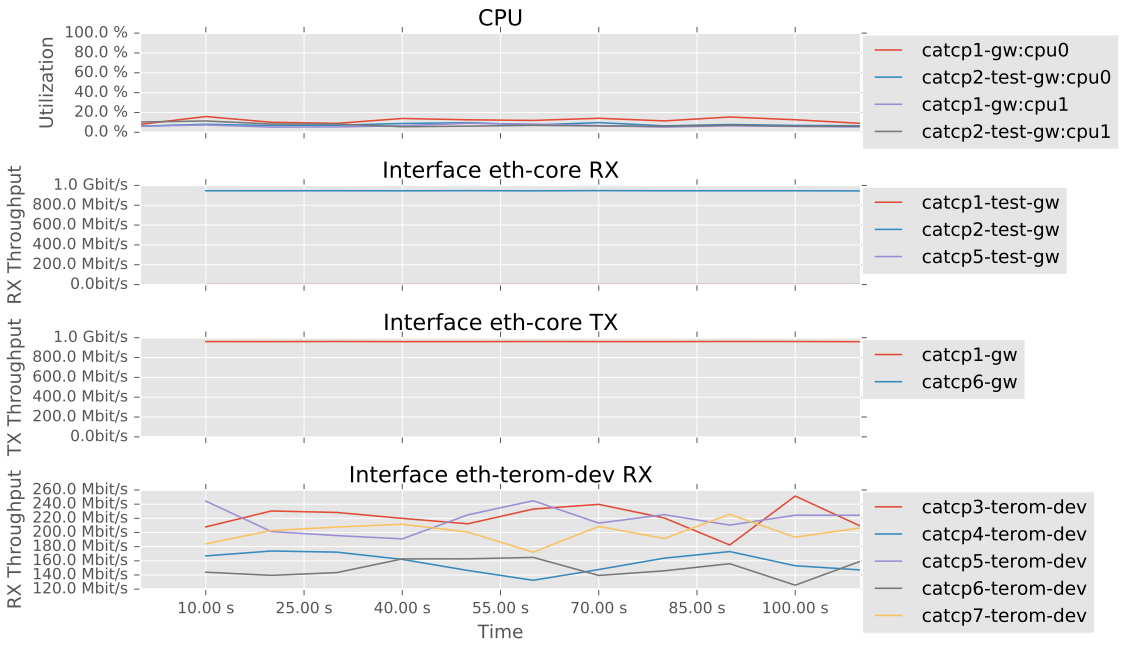

Figure 19: `iperf` measurement results for the baseline performance of inter-cluster symmetric routing

Figure 19 shows the achieved `iperf` upload throughput and resulting gateway machine CPU utilization when using symmetric routing for packets between the two Docker cluster networks, with each iperf client connecting directly to each remote iperf server (`iperf -c catcpX-terom-dev.iperf.docker.test.catcp -t 120`). The

results show a low level of CPU utilization for the **catcp1-gw** and **catcp2-test-gw** machines forwarding both the upload traffic and return TCP ACK packets, achieving a stable 1Gbit/s upload rate with somewhat unevenly distributed traffic across the server machines. Both **catcp1-gw** and **catcp2-test-gw** are routing approximately 200k packets/s in each direction, for a total of approximately 400k packet/s per machine.

6.3.4 Inter-cluster multipath routing

To expand the capacity of the inter-cluser network, we configure two pairs of Gateway machines with the same OSPF costs, providing multiple equal-cost paths used for L3 load balancing across the core network. Each of the **catcp-terom-dev** and **catcp-docker** machines uses OSPF ECMP routing for L3 load-balancing of traffic across each pair of Gateway machines. The resulting inter-cluster traffic uses a total of four L2 flows between the two pairs of Gateway machines on the core network, limiting the total attainable throughput to 2-4Gbit/s depending on the resulting distribution of L2 flow hashes. Traffic from multiple different machines is hashed to the same LACP trunk interface on **catcp1**, causing the **catcp1-test-gw** machine to receive less traffic than the **catcp5-test-gw** machine.

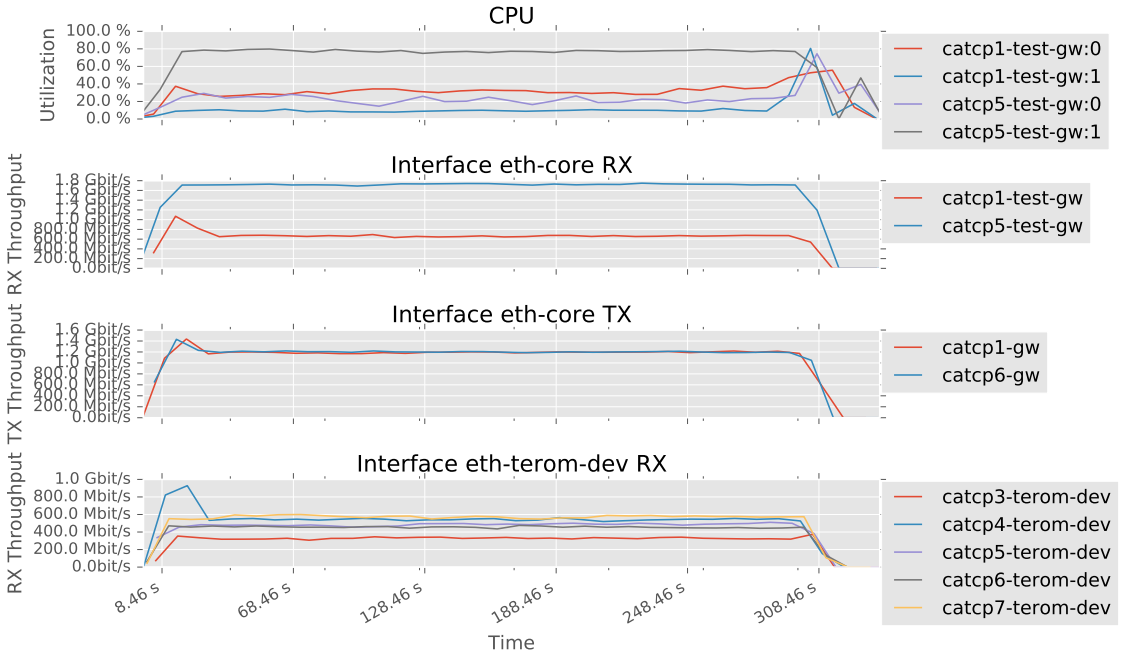

Figure 20: `iperf` measurement results for the baseline performance of inter-cluster multipath routing

Figure 20 shows the achieved `iperf` upload throughput and resulting Gateway CPU

utilization for the iperf upload test when using ECMP routing between two pairs of gateway machines, with each iperf client connecting directly to the iperf server Docker containers (`iperf -c catcpX-terom-dev.iperf.docker.test.catcp -t 300`). The results show a moderate level of CPU utilization for the **catcp-test-gw** machines forwarding both the upload traffic and return TCP ACK packets, achieving a combined 2.5Gbit/s total upload rate with traffic evenly distributed across the server machines.

The L2 flows in this scenario saturate LACP trunk interfaces on **catcp1** (1 interface), **catcp6** (1 interface) and **catcp5** (2 interfaces). The lower `eth-core` RX throughput for the **catcp1-test-gw** machine is presumably explained by upload traffic from **catcp-docker** machines to the **catcp1-gw** machine colliding with upload traffic routed by the **catcp6-gw** machine.

6.4 Comparing Implementation Performance

To compare the performance of a single Linux IPVS network-level load balancer with the performance of a single HAProxy application-level load balancer, we use the symmetric routing scenario shown in Section 6.3.3. One instance of the `iperf` service is run on each of the **catcp-terom-dev** machines, using `clusterf-docker` to provide a `clusterf ServiceBackend` for each service instance within `etcd`. For each **catcp-docker** machine, we run a single `iperf` client, connecting to the load-balanced `iperf` service using the `10.0.107.2` VIP announced by **catcp2-test-gw**. All of the `iperf` upload traffic and returning TCP ACK packets are routed via the same `catcp2-test-gw` machine. The **catcp2-test-gw** machine is configured to either run the `clusterf` or HAProxy load balancers, using the same `clusterf` service configuration provided by `clusterf-docker` for both load balancers. This configuration has the gateway machine as a single point of failure with no options for failover.

6.4.1 The HAProxy application-level load balancer

The **catcp2-test-gw** machine is configured to run a combination of HAProxy and `confd` to load balance incoming TCP connections for the `iperf` service across each of the **catcp-terom-dev** `iperf` server containers.

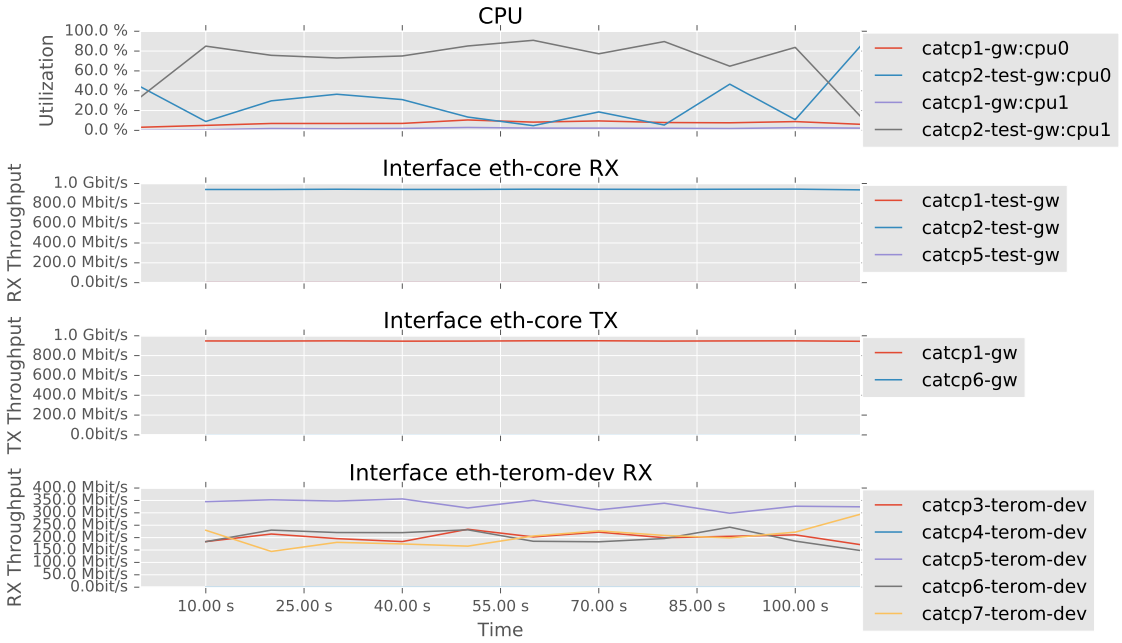

Figure 21: `iperf` measurement results for HAProxy load balancing with symmetric routing

Figure 21 shows the achieved `iperf` upload throughput and resulting Gateway CPU utilization for the iperf upload test using the HAProxy load-balancer (`iperf -c 10.0.107.2 -t 120`). The results show a high level of CPU utilization for the **catcp2-test-gw** machine using HAProxy to forward the TCP connections, achieving a stable 1Gbit/s upload rate with traffic evenly distributed across the `iperf` servers. Both **catcp1-gw** and **catcp2-test-gw** are routing approximately 100k packets/s in each direction, for a total of 200k packet/s for each machine.

This result shows that using the HAProxy application-level proxy for load-blanacing has a significant CPU overhead compared to the symmetric routing configuration without any load balancing in Section 6.3.3.

6.4.2 The IPVS network-level load balancer

The **catcp2-test-gw** machine is configured to run the `clusterf-ipvs` control plane, using Linux IPVS to load balance packets for incoming TCP connections to the `iperf` service across each of the **catcp-terom-dev** `iperf` server containers.

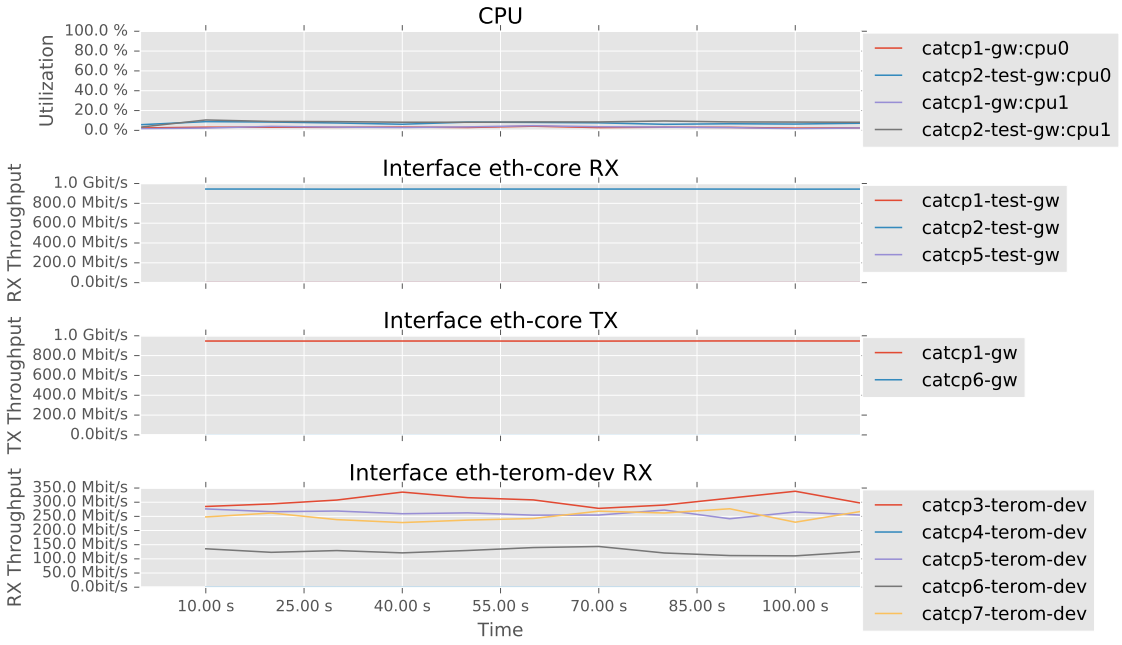

Figure 22: `iperf` measurement results for IPVS load balancing with symmetric routing

Figure 22 shows the achieved `iperf` upload throughput and resulting Gateway CPU utilization for the iperf upload test using the IPVS load balancer (`iperf -c 10.0.107.2 -t 120`). The results show low CPU utilization for the **catcp2-test-gw** machine forwarding both the upload traffic and return TCP ACK packets, achieving a stable 1Gbit/s upload rate with traffic evenly distributed across the `iperf` servers. Both **catcp1-gw** and **catcp2-test-gw** are routing approximately 100k packets/s in each direction, for a total of 200k packet/s for each machine.

This result shows that using the IPVS network-layer load-balancer for load balancing has similar CPU overhead to the symmetric routing configuration without any load balancing in Section 6.3.3. However, the two scenarios differ in the number of `iperf` clients, with the use of fewer `iperf` clients resulting in a smaller packets/s rate for this experiment, and thus a lower level of CPU utilization for packet routing. However, the number of `iperf` clients and resulting packets/s rate for this experiment is the same as in the equivalent HAProxy experiment in Section 6.4.1.

6.5 Scaling the `clusterf` Load Balancer

To evaluate the horizontal scaling capabilities of the IPVS load balancer, we use the multipath routing scenario described in Section 6.3.4. The **catcp-terom-dev** machines are used to run 12 `iperf` server instances of the `iperf` service. Both **catcp1-**

test-gw and **catcp5-test-gw** machines are configured using the same OSPF costs, allowing traffic in both directions to be distributed between both machines. Both **catcp1-test-gw** and **catcp5-test-gw** machines are configured to announce the same `10.0.107.0` VIP with equal OSPF costs, causing traffic for the `10.0.107.0` VIP to be distributed across both load balancers. This configuration provides options for testing failover, as traffic for each VIP can be routed to either of the **catcp-test-gw** machines.

6.5.1 ECMP routing for a single VIP

The ability to add more load balancers to scale the total load balancing capacity available for a service is demonstrated using two IPVS load balancers sharing a single VIP with ECMP routing. The **catcp-docker** machines are used to run 8 `iperf` client instances, connecting to the `10.0.107.0` VIP announced by both load balancer machines.

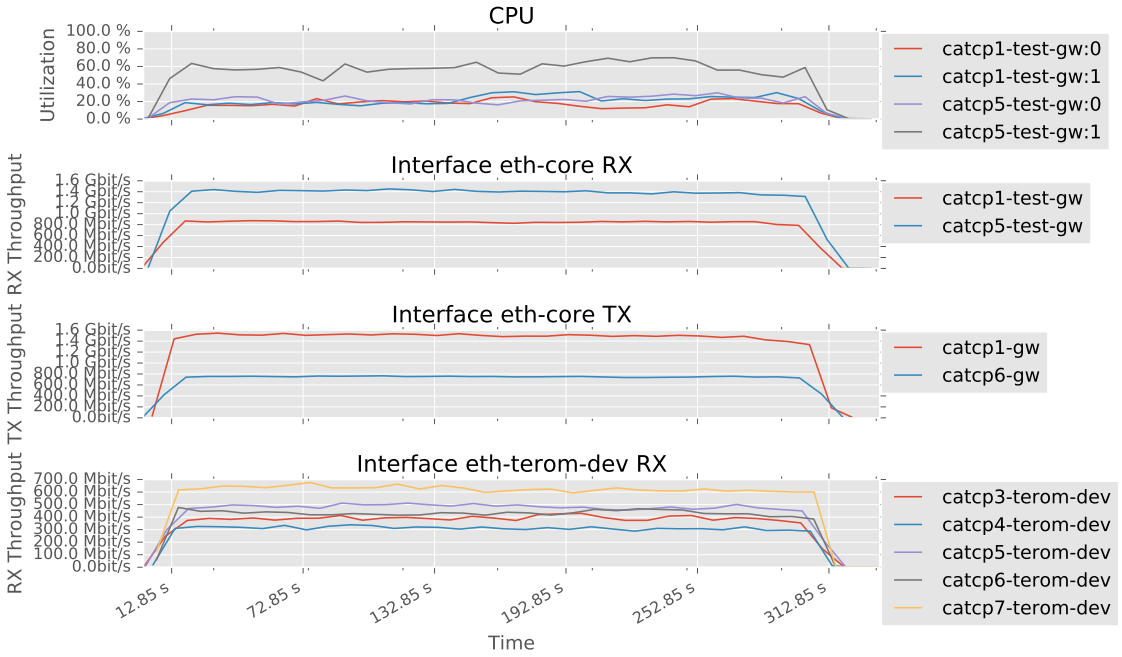

Figure 23: `iperf` measurement results for IPVS load balancing with EMCP routing for a single VIP

Figure 23 shows achieved network upload throughput and resulting gateway CPU utilization for the iperf upload test using the IPVS load-balancer, connecting to the ECMP VIP address (`iperf -c 10.0.107.0 -P 2 -t 300`). The results show a moderate level of CPU utilization for the **catcp-test-gw** machines forwarding both the upload traffic and return TCP ACK packets, achieving an approximately

2.5Gbit/s total upload rate with traffic moderately distributed across the server machines.

This result demonstrates that the `clusterf` design is capable of horizontal scaling of traffic for a single VIP across multiple IPVS network-level load balancers using ECMP routing. Using ECMP routing to distribute traffic for a single VIP across multiple IPVS load balancers providing more throughput (2.5Gbit/s) than when using a single IPVS load balancer (1Gbit/s). The 2.5x increase in performance is an artifact of the use of L2 load balancing for the underlying physical network infrastructure, resulting from the use of 4 distinct L2 flows between two pairs of routers compared to the use of a single L2 flow between a single pair of routers. The imbalance of throughput levels between the two gateways is a result of the L2 flow hashing collisions discussed in Section 6.3.4.

6.5.2 Connection failover for a single VIP with ECMP routing

To test the use of IPVS connection synchronization for fault tolerance, we use a single `iperf` client on a randomly selected **catcp-docker** machine, connecting to the `10.0.107.0` VIP announced by both IPVS load balancer machines. The **catcp1-test-gw** and **catcp5-test-gw** machines use IPVS connection synchronization to share the load-balanced connection states.

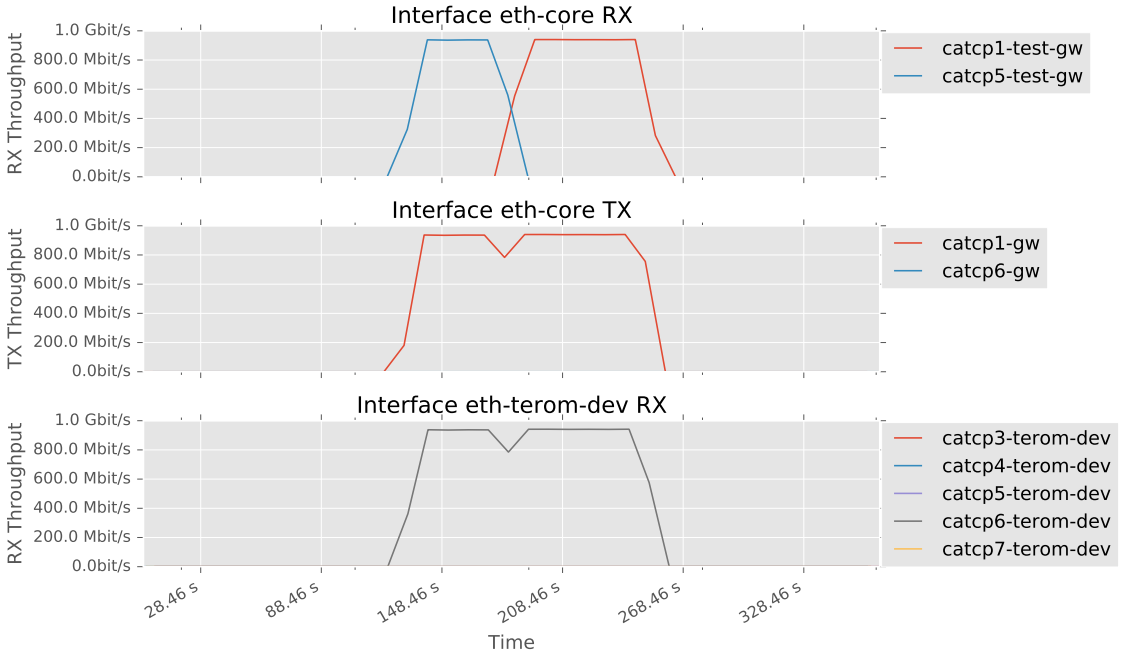

Figure 24: `iperf` measurement results for IPVS failover with ECMP routing for a single VIP

In Figure 24, a single `iperf -c 10.0.107.0 -t 120` client is run on the **catcp-docker** nodes, which establishes a single TCP connection to the VIP at $t = 121s$, and transmits data for a total of 120 seconds. The packets for this connection are initially routed using ECMP to the **catcp5-test-gw** machine, as shown by the 1Gbit/s of RX throughput on the **catcp5-test-gw** machine's `eth-core` interface. At $t = 180s$, the **catcp5-test-gw** machine fails, withrawing the OSPF route for the VIP. The **catcp1-gw** machine updates its routing table to forward all packets for the VIP to **catcp1-test-gw**. Using the IPVS connection state synchronized from the failed load-balancer node, the IPVS load balancer on **catcp1-test-gw** is able to resume forwarding the packets for the existing `iperf` connection. The `iperf` client succesfully completes its test at $t = 260s$ after a brief hiccup, despite the failure of the initial load balancer used for the connection and resulting ECMP rerouting of packets to a different load balancer.

This result demonstrates that the `clusterf` design provides fault tolerance for multiple IPVS load balancers using connection synchronization and ECMP routing for a single VIP. The use of IPVS `masq` for NAT forwarding within the **catcp6-terom-dev** Docker machine is not disrupted by the migration of incoming traffic across different Gateway machines.

6.6 Analysis

The `clusterf` load balancer behaves as designed in each of the load balancing scenarios tested. The `clusterf` load balancer preserves the original network source address of the client, while allowing the use of varying port numbers for the listening server. The `clusterf` load balancer is capable of distributing incoming traffic across multiple Docker container backends, allowing the horizontal scaling of Docker machines and service containers. The `clusterf` load balancer is capable of distributing incoming traffic across multiple Gateway machines, allowing the horizontal scaling of Gateway machines and VIPs. The failure of a Gateway machine does not cause the failure of active connections when using IPVS connection state synchronization.

Using IPVS for load balancing involves minimal overhead compared to the equivalent routing of packets within the Linux kernel. Using HAProxy for load balancing involves significnat processing overhead compared to the equivalent routing of packets within the Linux kernel. The quantitative analysis of any pair of network-level and application-level load balancer implemenations cannot itself be used to draw any conclusions about the performance of application-level and network-level load balancers in general. It is entirely possible for a well implemented application-level load balancer to have better performance than a poorly implemented network-level load balancer, and vice-versa.

The study of existing network-level load balancer implementations in Section 3.4 shows that network-level load balancer implementations are capable of achiving

very high levels of performance. A single Maglev load balancer with 8 CPU cores is capable of achiving packet processing rates beyond the 12 M packets/s line rate limit of 10-gigabit Ethernet by implementing a userspace networking stack, where the Linux kernel network implementation was only able to handle less than 4 M packets/s in the same configuration [7]. Any application-level load balancer implementation will be limited by the performance of the underlying Linux kernel network implementation.

7 Conclusions

A container platform based on Docker can be used for the development and deployment of horizontally scalable services on both local machines and within cloud server infrastructures. Such a container platform will include various components, including a container engine, an orchestration system, an internal network infrastructure, a dynamic service discovery mechanism, and a load balancer. The load balancer plays a critical role in the scalability of any cloud service, and thus the scalability of the load balancer itself must also be considered when designing such a container platform. A container platform supporting dynamic scaling of services must also have a dynamic load balancer control plane capable of automatically distributing incoming traffic as service instances are added and removed.

Based on a study of the background networking and load balancing theory in Sections 2.3 and 2.5 and existing implementations of load balancers in Section 3.4, I create a design for a scalable network-level load balancer. The `clusterf` design uses the Linux IPVS network-level load balancer implementation to provide a scalable load balancer data plane, using a two-level load balancing scheme with both DSR and NAT forwarding for both scalability and compatibility with existing Docker containers and applications. I implement the `clusterf` control plane for the automatic load balancing of Docker services within a cluster. The design and implementation of the `clusterf` load balancer is evaluated within a testbed network infrastructure using standard Ethernet networking with ECMP routing capabilities for network-layer load balancing.

The results in Chapter 6 show that the `clusterf` load balancer provides excellent performance and reliability for load balancing horizontally scaled services within a container platform as studied in Chapter 3. The results in Section 6.4 show that the Linux IPVS network-level load balancer implementation provides better performance than the HAProxy application-level load balancer implementation. The results in Section 6.5 show that the `clusterf` design is capable of scaling in terms of both performance and reliability using standard network-layer ECMP routing. However, the current implementation of the `clusterf` load balancer assumes the use of standard Ethernet networking allowing the use of asymmetric routing for Direct Server Return (DSR). Extending the `clusterf` design for load balancing within arbitrary cloud network infrastructures that restrict the use of asymmetric routing is the subject of further study.

IETF BCP 38 [32] requires source address filtering within networks to prevent the abuse of source address spoofing for Denial of Service (DoS) attacks. Depending on the exact implementation of such Reverse Path Filtering (RPF), complying with BCP 38 for ingress filtering of spoofed source addresses may preclude the use of asymmetric routing for DSR load balancing within a network. The author can only speculate as to the various implementations of RPF within cloud network infrastructures, but the generally valid assumption for a container platform designed for

deployment across arbitrary network infrastructures is that each host only has a single external network address. Resolving this design conflict is the subject of further research, involving either extending the `clusterf` design to use multiple separate VIPs with DNS load balancing, or exploring the possibilities for the use of asymmetric routing and DSR within various cloud infrastructure platforms.

In the meantime, network-level load balancer implementations are more likely to be found within specific cloud infrastructure platforms. The providers of physical cloud platforms have both more control over the specific network infrastructure implementation and configuration, allowing the use of network-level techniques such as ECMP and DSR for load balancing, and operate at a sufficiently large scale for the scalability differences between application-level and network-level load balancers to become apparent. Examples of such cloud provider specific network-level load balancer implemenations include the Amazon Web Services (AWS) Elastic Load Balancer (ELB), the Microsoft Azure Ananta [8] load balancer, and the Google Compute Engine (GCE) Maglev [7] load balancers.

As a corollary, portable container platforms designed for deployment across different network infrastructure environments are more likely to use application-level load balancing methods. The use of application-level load balancing for proxying transport-layer connections allows the use of symmetrically routed connections between and within arbitrary networks. However, scaling such an application-level load balancer either requires the use of DNS within the Internet for load balancing client traffic across multiple VIPs, or the use of a scalable network-level load balancer provided by the cloud infrastructure platform. Designing a load balancer control plane for a container platform capable of integrating with either global DNS load balancing or various platform-specific load balancer infrastructures to avoid cloud vendor lock-in [10] is also a subject for future research.

Detailed study of the recent Docker 1.12 service routing mesh using the same network-level Linux IPVS load balancer implementation as used in `clusterf` is also the subject of future research. The author speculates that the Docker 1.12 service load balancer design is based on the use of Full NAT for forwarding, thus resembling more of a high-performance application-level load balancer implementation than a network-level load balancer design. Based on the background networking theory and study, the author hypothesies that the Docker 1.12 implementation of network-level load balancing is unable to support network source address transparency or connection failover between machines. Without the use of external load balancing routing configured to route traffic to specific Docker Swarm nodes, the loss of any Docker Swarm node acting as a service load balancer may lead to the failure of active connections routed to the remaining healthy nodes. Testing this hypothesis is outside the scope of this thesis, and the author disclaims the possibility for theoretical misunderstandings and any future extensions of the Docker Swarm implementation that may negate this hypothesis.

The `clusterf` load balancer as currently implemented is thus generally unsuitable for use within a container platform intended to be portable across different networking environments. With further research of the limitations and proposals explored throughout Chapter 4, the author believes that the use of a truly scalable network-level load balancer for horizontally scalable Docker services deployed across multiple cloud platforms may be possible. Until then, users of container platforms will likely be using application-level load balancer designs that are capable of providing satisfactory levels of performance and reliability as well as additional flexibility for application-layer load balancing.

After all, there are numerous other reasons for client connections to fail, and assuming the existence of client support for reconnecting on connection loss, do we really need to pay the price for the complexity of a network-level load balancer design including complete support for connection failover on load balancer failures? Horizontal scalability is about the design of abstract distributed computing layers while navigating the minefield of inenvitable practical implementation issues. The answer to the ultimate question of load balancers, the implementation of dynamic service discovery and application container platforms may lie in the role of VIPs in the implementation of dynamic service discovery for client applications, a topic not fully explored in this thesis.

References

[1] Peter Mell and Tim Grance. "The NIST definition of cloud computing". In: (2011).

[2] Abhishek Verma et al. "Large-scale cluster management at Google with Borg". In: *Proceedings of the Tenth European Conference on Computer Systems*. ACM. 2015, p. 18.

[3] David Lo et al. "Heracles: improving resource efficiency at scale". In: *ACM SIGARCH Computer Architecture News*. Vol. 43. 3. ACM. 2015, pp. 450–462.

[4] Brendan Burns et al. "Borg, Omega, and Kubernetes". In: *ACM Queue* 14.1 (2016), p. 10.

[5] *Evolution of the Modern Software Supply Chain*. Tech. rep. Docker, Inc, Apr. 2016.

[6] ClearPath Strategies. *Hope Versus Reality: Containers In 2016*. Tech. rep. Cloud Foundry Foundation, June 2016.

[7] Daniel E Eisenbud et al. "Maglev: A Fast and Reliable Software Network Load Balancer". In: *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. 2016, pp. 523–535.

[8] Parveen Patel et al. "Ananta: cloud scale load balancing". In: *ACM SIGCOMM Computer Communication Review*. Vol. 43. 4. ACM. 2013, pp. 207–218.

[9] Jürgen Cito et al. "The making of cloud applications: An empirical study on software development for the cloud". In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM. 2015, pp. 393–403.

[10] Nane Kratzke. "A lightweight virtualization cluster reference architecture derived from open source paas platforms". In: *Open J. Mob. Comput. Cloud Comput* 1 (2014), pp. 17–30.

[11] Dirk Merkel. "Docker: lightweight linux containers for consistent development and deployment". In: *Linux Journal* 2014.239 (2014), p. 2.

[12] *cgroups(7) Linux Programmer's Manual*. 4.06.

[13] *namespaces(7) Linux Programmer's Manual*. 4.06.

[14] Benjamin Hindman et al. "Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center." In: *NSDI*. Vol. 11. 2011, pp. 22–22.

[15] Andrew S Tanenbaum. "Computer Networks 4th Edition". In: (2003).

[16] Victor Marmol, Rohit Jnagal, and Tim Hockin. "Networking in Containers and Container Clusters". In: *Proceedings of netdev 0.1* (Feb. 2015).

[17] Joris Claassen, Ralph Koning, and Paola Grosso. "Linux containers networking: Performance and scalability of kernel modules". In: *NOMS 2016-2016 IEEE/IFIP Network Operations and Management Symposium*. IEEE. 2016, pp. 713–717.

[18] Y. Rekhter et al. *Address Allocation for Private Internets*. RFC 1918 (Best Current Practice). Updated by RFC 6761. Internet Engineering Task Force, Feb. 1996. URL: http://www.ietf.org/rfc/rfc1918.txt.

[19] P. Srisuresh and M. Holdrege. *IP Network Address Translator (NAT) Terminology and Considerations*. RFC 2663 (Informational). Internet Engineering Task Force, Aug. 1999. URL: http://www.ietf.org/rfc/rfc2663.txt.

[20] M. Mahalingam et al. *Virtual eXtensible Local Area Network (VXLAN): A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks*. RFC 7348 (Informational). Internet Engineering Task Force, Aug. 2014. URL: http://www.ietf.org/rfc/rfc7348.txt.

[21] T. Narten, G. Huston, and L. Roberts. *IPv6 Address Assignment to End Sites*. RFC 6177 (Best Current Practice). Internet Engineering Task Force, Mar. 2011. URL: http://www.ietf.org/rfc/rfc6177.txt.

[22] D. Thaler and C. Hopps. *Multipath Issues in Unicast and Multicast Next-Hop Selection*. RFC 2991 (Informational). Internet Engineering Task Force, Nov. 2000. URL: http://www.ietf.org/rfc/rfc2991.txt.

[23] M. Byerly, M. Hite, and J. Jaeggli. *Close Encounters of the ICMP Type 2 Kind (Near Misses with ICMPv6 Packet Too Big (PTB))*. RFC 7690 (Informational). Internet Engineering Task Force, Jan. 2016. URL: http://www.ietf.org/rfc/rfc7690.txt.

[24] R. Fielding et al. *Hypertext Transfer Protocol – HTTP/1.1*. RFC 2616 (Draft Standard). Obsoleted by RFCs 7230, 7231, 7232, 7233, 7234, 7235, updated by RFCs 2817, 5785, 6266, 6585. Internet Engineering Task Force, June 1999. URL: http://www.ietf.org/rfc/rfc2616.txt.

[25] Fangfei Chen, Ramesh K Sitaraman, and Marcelo Torres. "End-user mapping: Next generation request routing for content delivery". In: *ACM SIGCOMM Computer Communication Review*. Vol. 45. 4. ACM. 2015, pp. 167–181.

[26] Don Mills. *Docker Multi-Host Networking: Overlays to the Rescue*. Tech. rep. SingleStone. URL: https://www.singlestoneconsulting.com/~/media/files/whitepapers/dockernetworking2.pdf.

[27] Diego Ongaro and John Ousterhout. "In search of an understandable consensus algorithm". In: *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. 2014, pp. 305–319.

[28] Nane Kratzke. "About microservices, containers and their underestimated impact on network performance". In: *Proceedings of CLOUD COMPUTING* 2015 (2015).

[29] Nane Kratzke and Peter-Christian Quint. "How to Operate Container Clusters more Efficiently?" In: *International Journal On Advances in Networks and Services* 8.3&4 (2015), pp. 203–214.

[30] Alex Gartrell. *[PATCH ipvs 0/7] Support v6 real servers in v4 pools and vice versa*. July 2014. URL: `http://archive.linuxvirtualserver.org/html/lvs-devel/2014-07/msg00047.html`.

[31] Brendan Gregg. "Thinking methodically about performance". In: *Communications of the ACM* 56.2 (2013), pp. 45–51.

[32] P. Ferguson and D. Senie. *Network Ingress Filtering: Defeating Denial of Service Attacks which employ IP Source Address Spoofing*. RFC 2827 (Best Current Practice). Updated by RFC 3704. Internet Engineering Task Force, May 2000. URL: `http://www.ietf.org/rfc/rfc2827.txt`.