

Hellenic Mediterranean University

Department of Electrical and Computer Engineering

Bachelor's Thesis

Design and Implementation of a Scalable and Resilient Web Server Infrastructure: A Comprehensive Study on Load Balancing Strategies with Nginx and High Availability Configurations using Keepalived.

Sideris Fotios

Supervisor: Prof. Grammatikakis Miltiadis

Committee members: Prof. Papadakis Nikos, Prof. Marakakis Manolis

Contents

1. Introduction.....7

1.1 **Networking**.....7

1.1.1 Network Topology.....7

1.1.2 Network Protocols.....9

1.1.3 IP Addressing.....10

1.1.4 Subnetting.....11

1.1.5 Routing.....12

1.1.6 Switching.....13

1.1.7 Network Address Translation.....13

1.1.8 Domain Name System..... 14

1.1.9 Wireless Networking.....14

1.2 **HAProxy**.....15

1.2.1 The role of HAProxy 15

1.2.2 A brief history of HAProxy.....16

1.3 **Nginx** 18

1.3.1 The role of Nginx 17

1.3.2 A brief history of Nginx 18

1.4 **KeepAlived**.....19

1.4.1 The role of keepalived.....19

1.5 **Virtualization** 21

1.5.1 Hardware Virtualization 21

1.5.2 OS Virtualization 22

1.5.3 Network Virtualization.....22

1.5.4 Virtual Box 23

1.6 **Security**.....24

1.6.1 Firewalls.....24

1.6.2 Virtual Private Networks 25

1.6.3 Secure Socket Layer/Transport Layer Security.....26

1.6.4 Network Access Control 27

1.6.5 Selinux.....27

2. Private/Public NATs, Port Forwarding 29

2.1 Introduction to Network Address Translation..... 29

2.2 Types of NAT.....30

2.3 Port Forwarding.....31

2.4 Port Address Translation.....32

3. High performance web-server infrastructure using HAProxy for load balancing.....33

3.1 Basic Architecture.....33

3.2 Health Checking.....34

3.3 Session Persistence.....35

3.4 SSL Termination.....36

3.5 Logging and Monitoring.....36

3.6 High Availability.....37

3.7 Configuration.....38

4. Achieving High Availability with multiple load balancers, configure advanced features (sticky sessions, shared state, failover situations with keepalived, floating ip)……40
4.1 Redundant Load Balancers……40
4.2 Sticky Sessions……41
4.3 Shared State……42
4.4 Failover with keepalived……42
4.5 Floating IP……43
4.6 Testing and Monitoring……44

5. Exeperimental work – Nginx- HAProxy
5.1 Introduction …… 45
5.2 Basic Infrastructure……46
5.3 Assign Load Balancer to listen only host-only adapter ip……51
5.4 Load Balancer with Port Forwarding…… 52
5.5 Add another VM nginx-nd3……54
5.6 Private and Public Nat Network……55
5.7 Implementing HTTPS in Load Balancer Configuration……58
5.8 Implementing High Availability in Load Balancer Configuration……60
5.9 Configuring High Availability Load Balancers with Sticky Sessions and State Sharing……69

6. Summary of Core Network Technologies……73
6.1 Understanding Network Fundamentals……73
6.2 Load Balancing technologies overview……74
6.3 Implementation and Optimization of HAProxy……74
6.4 Advanced configuration and Redundancy techniques……75

7. Summary and future work……78

8. Bibliography……79

LIST OF FIGURES

1.1 Network topology ……..
1.2 HAProxy……..
1.3 Nginx……..
1.4 Keepalived……..
1.5 Virtualization……..
1.6 Security……..
2
2.1 Network Address Translation……
2.2 NATs Types……..
2.3 Port Forwarding……
2.4 Port Address Translation……
3
3.1 Basic Architecture……..
3.2 Load Balancing Algorithms ……..
3.3 Health Checking……..
3.4 Session Persistence……..
3.5 SSL Termination……
3.6 Logging and Monitoring……
3.7 High Availability……..
3.8 Configuration……..
3.9 Scalability……
4
4.1 Redundant Load Balancers ……..
4.2 Sticky Sessions ……..
4.3 Shared State ……..
4.4 Failover with keepalived ……..
4.5 Floating Ip ……..
4.6 Testing and Monitoring ……..

Abstract

The project "Nginx" entails the comprehensive setup and configuration of a network infrastructure utilizing Nginx web servers and load balancers. Spanning across eight tasks, the project meticulously constructs a robust architecture featuring multiple virtual machines interconnected through various network configurations.

Commencing with Task 0(Basic Infrastructure) which establishes the foundational infrastructure, subsequent tasks progressively enhance the network's capabilities. Tasks involve configuring load balancing mechanisms, implementing security measures, and integrating HTTPS protocols. Notably, the project explores diverse load balancing techniques, including round-robin, IP-hash, and sticky sessions, culminating in the deployment of high-availability solutions across multiple load balancers.

Throughout the project, meticulous documentation is maintained, comprising diagrams, proof-of-concept scripts, configuration files, and snapshots to ensure traceability and replicability. Each task is meticulously tested and validated to guarantee the robustness and reliability of the network infrastructure.

The project's significance lies in its practical exploration of networking concepts, security protocols, and load balancing strategies within a controlled environment. Through hands-on experimentation and problem-solving, valuable insights are gained, contributing to a comprehensive understanding of modern network architectures and their practical implementations.

Acknowledgements

I would like to express my sincere gratitude to my supervisor, Dr. Miltos D. Grammatikakis, for his help and guidance over this process and for giving me the opportunity to work with him. I am also very grateful to my family and friends for their continuous support in my life throughout my studies and beyond.

CHAPTER 1

1.0 Introduction

1.1 Networking

1.1.1 Network Topology

In the vast expanse of the digital world, networks serve as the backbone, facilitating communication and connectivity between devices, systems, and users. At the heart of these networks lies a fundamental concept: network topology. Network topology defines the arrangement of nodes, links, and connections within a network, providing a blueprint for its structure and functionality. This essay aims to delve into the essence of network topology, exploring its significance, types, and real-world applications.

Network topology plays a pivotal role in shaping the performance, reliability, and scalability of a network. By defining how devices are interconnected, it determines the efficiency of data transmission, the resilience to failures, and the ease of network management. Understanding network topology is crucial for network administrators, architects, and engineers as it enables them to design, deploy, and optimize networks tailored to specific requirements and objectives.

Network topology can manifest in various forms, each offering distinct advantages and limitations. The most common types of network topology include:

Bus Topology: In a bus topology, all devices are connected to a single central channel, known as the bus. Data transmission occurs sequentially along the bus, with each device receiving and processing relevant information. While simple and cost-effective, bus topology is susceptible to single point failures and limited scalability.

Star Topology: In a star topology, each device is connected to a central hub or switch, forming a star-like structure. All communication between devices passes through the central hub, enhancing reliability and facilitating easier troubleshooting. However, the dependence on the central hub poses a potential bottleneck and single point of failure.

Ring Topology: In a ring topology, each device is connected to two adjacent devices, forming a closed loop. Data travels in one direction around the ring, with each device acting as a repeater to amplify and relay the signal. While ring topology offers uniform performance and fault tolerance, the failure of a single device can disrupt the entire network.

Mesh topology provides additional communication paths, improving reliability and performance. However, mesh topology requires extensive cabling and configuration, making it complex and costly to implement.

Hybrid Topology: Hybrid topology combines elements of multiple topology types to create a customized network architecture that meets specific requirements. By leveraging the strengths of different topologies, hybrid networks can achieve optimal performance, scalability, and resilience.

In conclusion, network topology serves as the foundation of modern communication networks, defining their structure, functionality, and performance characteristics. By understanding the significance of network topology and its various types, stakeholders can design, deploy, and manage networks that meet the evolving demands of today's interconnected world. Whether in enterprise environments, internet infrastructure, or telecommunication networks, network topology remains a cornerstone of digital connectivity, shaping the way we interact, collaborate, and innovate in the digital age.

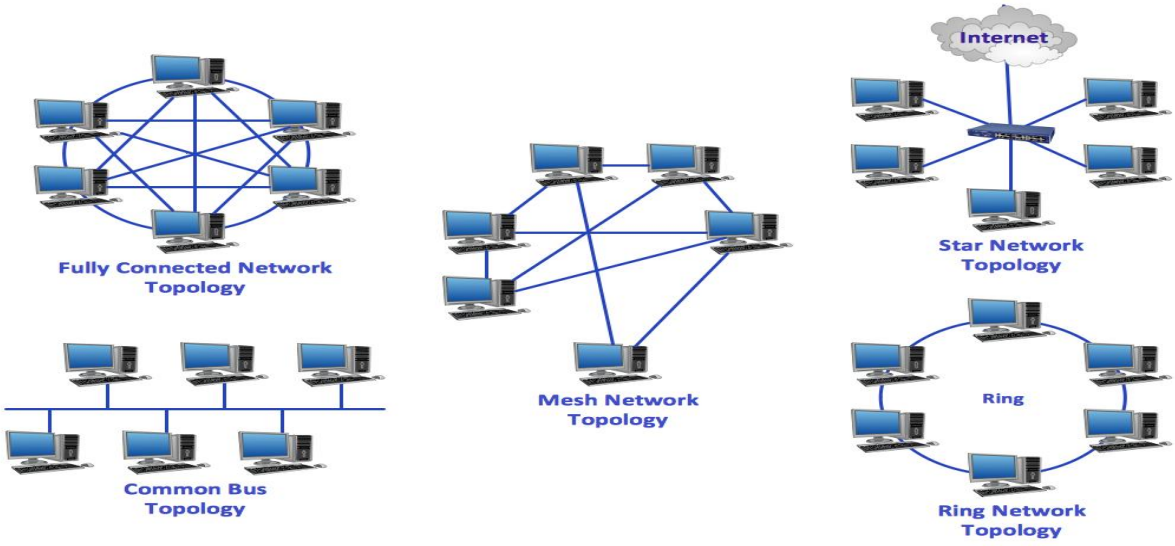


Figure 1.1: Network Topology [1].

1.1.2 Network Protocols

Network protocols are a set of rules and conventions governing how data is transmitted and received over a computer network. They define the format, timing, sequencing, and error control of data transmission between different connected devices, such as computers, servers, routers, and switches. Here are some common network protocols, layered according to the Open Systems Interconnection model (OSI):

Hypertext Transfer Protocol (HTTP): HTTP is the foundation of data communication on the World Wide Web. It defines how web clients (such as browsers) request web pages from web servers and how servers respond to those requests.

Hypertext Transfer Protocol Secure (HTTPS): HTTPS is a secure version of HTTP that uses encryption to protect the confidentiality and integrity of data exchanged between clients and servers.

File Transfer Protocol (FTP): FTP is a standard network protocol used to transfer files between a client and a server on a computer network. It provides commands for performing file operations such as uploading, downloading, renaming, and deleting files.

Simple Mail Transfer Protocol (SMTP): SMTP is a protocol for sending email messages across servers. It is used to transfer outgoing mail from a client to a server or from one server to another**.**

Post Office Protocol (POP): POP is a protocol used by email clients to retrieve messages from a mail server. It allows users to download emails to their local device for offline access.

Internet Message Access Protocol (IMAP): IMAP is another protocol used by email clients to retrieve messages from a mail server. Unlike POP, IMAP allows users to manage emails directly on the server, enabling access to the same mailbox from multiple devices.

Internet Control Message Protocol (ICMP): ICMP is used for diagnostic and error-reporting functions in IP networks. It is commonly used for tasks such as ping (checking if a host is reachable) and traceroute (tracing the route packets take to reach a destination).

Transmission Control Protocol (TCP): TCP is a connection-oriented protocol that provides reliable, ordered, and error-checked delivery of data packets over an IP network. It ensures that data transmitted from one device reach the destination reliably and in the correct order.

User Datagram Protocol (UDP): UDP is a connectionless protocol that provides a simple and lightweight method for transmitting data packets without the overhead of establishing a connection. Unlike TCP, UDP does not guarantee delivery or order of packets.

Internet Protocol (IP): IP is the primary protocol responsible for addressing and routing packets of data so they can travel across networks and reach their intended destinations. It provides the basic framework for packet delivery in the Internet Protocol Suite.

These are just a few examples of the numerous network protocols that exist to facilitate communication and data exchange across computer networks. Each protocol serves specific purposes and operates at a specific layer of the OSI model, which defines a conceptual framework for understanding network communication.

1.1.3 IP addressing

IP addressing is a fundamental aspect of computer networking, defining how devices are uniquely identified and addressed on a network. An IP address is a numerical label assigned to each device participating in a network that uses the Internet Protocol for communication. IP addresses are used for two primary purposes:

Host Identification: An IP address uniquely identifies a device within a network. It allows devices to send and receive data to and from other devices on the same network or across interconnected networks.

Routing: IP addresses are used by routers and other network devices to determine the best path for forwarding data packets from the source to the destination across multiple interconnected networks in a hierarchical way.

IP addresses come in two main versions: IPv4 and IPv6.

IPv4 (Internet Protocol version 4): IPv4 addresses are 32-bit numerical addresses written in a format consisting of four octets separated by periods (e.g., 192.168.0.1). The maximum number of unique IPv4 addresses is limited by its 32-bit length, resulting in approximately 4.3 billion possible addresses. Due to the rapid growth of the Internet with trillions of connected devices, IPv4 addresses are now in short supply.

IPv6 (Internet Protocol version 6): IPv6 was developed to address the limitations of IPv4, primarily the exhaustion of available addresses. IPv6 addresses are 128 bits in length, providing a vastly larger address space than IPv4. IPv6 addresses are typically written as eight groups of hexadecimal digits separated by colons (e.g., 2001:0db8:85a3:0000:0000:8a2e:0370:7334). IPv6 adoption has been increasing to accommodate the growing number of devices connected to the Internet.

CIDR (Classless Inter-Domain Routing) Notation: CIDR notation is a compact representation of an IP address and its associated subnet mask. It uses a slash followed by a number to indicate the number of bits used for the network portion of the address (e.g., 192.168.1.0/24).

Public IP Address: A public IP address is globally unique and routable on the Internet. It is assigned to a device by an Internet Service Provider (ISP) or network administrator and is used for communication over the Internet.

Private IP Address: A private IP address is used only within a local network (such as a home or business network) and is not routable on the Internet. Devices within the same private network can communicate with each other using private IP addresses. Common private IP address ranges include those defined in RFC 1918: 10.0.0.0/8, 172.16.0.0/12, and 192.168.0.0/16.

1.1.4 Subnetting

IP addressing also involves subnetting, which is the process of dividing a single, larger network into smaller, more manageable subnetworks (called subnets). Subnetting involves partitioning a large IP address space by assigning subsets of addresses to different subnetworks. This method allows administrators to efficiently allocate available IP addresses (e.g., by organizing them into logical groups), to optimize network performance, and improve security.

Next, we provide some key concepts and terminology related to subnetting:

- **IP Address**: An IP address is a unique numerical identifier assigned to each device on a network. It consists of two main parts: the network portion and the host portion.
- **Subnet Mask**: A subnet mask is a 32-bit number used to divide an IP address into network and host portions. It indicates which portion of the IP address represents the network and which portion

represents the host. The subnet mask is typically expressed in dotted-decimal notation, like an IP address (e.g., 255.255.255.0).

- **Network Address**: The network address is the portion of an IP address that identifies the specific network to which a device belongs. It is obtained by applying the bitwise AND operation between the IP address and the subnet mask.
- **Broadcast Address**: The broadcast address is a special address used to send data packets to all devices within a specific network. It is obtained by setting all host bits in the subnet portion of the IP address to 1.
- **Host Address Range**: The host address range defines the range of IP addresses that can be assigned to individual devices within a subnet. It includes all IP addresses within a subnet except for the network address and the broadcast address.

The subnetting process involves determining the appropriate subnet mask and dividing the IP address space accordingly. Common subnetting techniques include:

- **Fixed-Length Subnet Mask (FLSM)**: In FLSM, each subnet uses the same subnet mask, resulting in fixed-size subnets. This approach is suitable when the network has homogeneous requirements for each subnet.
- **Variable-Length Subnet Mask (VLSM)**: VLSM allows for the use of different subnet masks for different subnets within the same network. This flexibility enables more efficient allocation of IP addresses, especially in networks with diverse requirements for subnet sizes.

Subnetting is a fundamental skill for network administrators and engineers, as it enables them to design and manage networks effectively, ensuring optimal performance and resource utilization.

1.1.5 Routing

Routing is the process of selecting the best path for data packets to travel from the source to the destination across a network. In computer networking, routers are responsible for performing routing by examining the destination address of incoming packets and determining the next hop or outgoing interface through which the packets should be forwarded.

Here's how routing generally works:

- **Packet Forwarding**: When a device wants to send data to another device on the network, it encapsulates the data into packets and includes the destination IP address in the packet header.
- **Routing Decision**: The sending device sends the packet to its default gateway, typically a router, which examines the destination IP address of the packet.
- **Routing Table Lookup**: The router consults its routing table, which contains information about known networks and the next hops or outgoing interfaces for reaching them.
- **Best Path Selection**: Based on the destination IP address, the router determines the best path to reach the destination network. This decision is often made using routing algorithms that consider factors such as network topology, link costs, and routing metrics.
- **Forwarding the Packet**: Once the best path is determined, the router forwards the packet to the next hop or outgoing interface along the chosen path.
- **Repeat Process**: If the next hop is another router, the process may be repeated until the packet reaches its final destination.

There are different types of routing protocols used to populate routing tables and exchange routing information among routers:

- **Static Routing**: In static routing, network administrators manually configure routing entries in the router's routing table. Static routes remain unchanged unless manually updated by the administrator.
- **Dynamic Routing**: Dynamic routing protocols enable routers to automatically exchange routing information and dynamically update their routing tables based on changes in the network topology. Common dynamic routing protocols include:
    - **RIP (Routing Information Protocol)**: A distance-vector routing protocol that uses hop count as its routing metric.
    - **OSPF (Open Shortest Path First)**: A link-state routing protocol that calculates the shortest path to each destination based on link costs.
    - **BGP (Border Gateway Protocol)**: A path-vector routing protocol used for routing between autonomous systems (ASes) on the Internet.

1.1.6 Switching

Switching refers to the process of forwarding data packets from one network device to another within the same local area network (LAN). A network switch is a hardware device that operates at Layer 2 (Data Link Layer) of the OSI (Open Systems Interconnection) model.

Here's how switching works:

- **MAC Address Learning**: When a switch receives a data packet from a device connected to one of its ports, it examines the MAC (Media Access Control) address of the packet and associates it with the port on which it was received. This process is called MAC address learning, and it builds a MAC address table (also known as a forwarding table) within the switch.
- **Packet Forwarding**: Once the switch has learned the MAC addresses of devices connected to its ports, it uses this information to forward packets to the appropriate destination. When a packet arrives at the switch, it checks the destination MAC address against its MAC address table to determine the outgoing port. If the MAC address is found, the switch forwards the packet only to the port where the destination device is located, reducing unnecessary traffic on the network.
- **Broadcast and Unknown Unicast Forwarding**: If the destination MAC address is not found in the MAC address table, the switch forwards the packet to all ports except the one on which it was received. This behavior ensures that broadcast and unknown unicast traffic reaches all devices on the network.

Switches are essential for building LANs and enabling efficient communication between devices within the same network segment. They provide high-speed, low-latency packet forwarding and facilitate the creation of segmented network architectures.

1.1.7 Network Address Translation (NAT)

Network Address Translation (NAT) is a technique used in computer networking to modify network address information in packet headers while in transit across a router or firewall. NAT allows multiple devices within a private network to share a single public IP address for outbound communication with devices on the Internet.

Here's how NAT works:

- **Private IP Addresses**: Private IP addresses are reserved for use within private networks and are not routable on the Internet. Common private IP address ranges include those defined in RFC 1918: 10.0.0.0/8, 172.16.0.0/12, and 192.168.0.0/16. Devices within a private network are assigned these private IP addresses.
- **Public IP Address**: A public IP address is an address that is routable on the Internet and can be used to communicate with devices outside of the private network. Public IP addresses are assigned by Internet Service Providers (ISPs) or network administrators.
- **Translation**: When a device within the private network initiates communication with a device on the Internet, the source IP address in the packet header is replaced with the public IP address of the NAT device (router or firewall). This process is known as source NAT (SNAT) or outbound NAT.
- **Port Mapping**: In addition to modifying the source IP address, NAT also modifies the source port number of outgoing packets. This port mapping allows multiple devices within the private network to share the same public IP address without port conflicts.
- **Reverse Translation**: When a response packet is received from the destination device on the Internet, the NAT device performs reverse translation, replacing the destination IP address and port number in the packet.

1.1.8 Domain Name System

The Domain Name System (DNS) is a hierarchical and distributed naming system used to translate human-readable domain names into numerical IP addresses and vice versa. It serves as the "phonebook" of the Internet, allowing users to access websites, send emails, and perform other network-related tasks using domain names instead of IP addresses.

Here's how DNS works:

- **Domain Names**: Domain names are human-readable labels used to identify resources on the Internet, such as websites, servers, or services. They consist of a series of labels separated by dots (e.g., example.com).
- **DNS Resolution Process**: When a user enters a domain name (e.g., www.example.com) into a web browser or other network application, the system needs to resolve that domain name to an IP address to establish a connection. The client device first checks its local DNS cache to see if it has a recent record of the IP address corresponding to the domain name. If the information is not cached or has expired, the client proceeds to query DNS servers.
- **DNS Servers**: DNS servers store and manage domain name records and provide responses to DNS queries from client devices.

There are several types of DNS servers:

Recursive DNS Servers: These servers handle DNS queries from client devices and either provide the requested information from their cache or recursively query other DNS servers to resolve the domain name.

Authoritative DNS Servers: These servers store authoritative DNS records for specific domains. They provide responses to DNS queries with authoritative information about the domain names they are responsible for.

Root DNS Servers: These servers form the top level of the DNS hierarchy and are responsible for providing the IP addresses of authoritative DNS servers for top-level domains (TLDs) such as .com, .org, .net, etc.

1.1.9 Wireless Networking

Wireless networking, also known as Wi-Fi (Wireless Fidelity), refers to the technology that enables devices to connect to a local area network (LAN) or the Internet without the need for physical wired connections. Wireless networks utilize radio waves to transmit data between devices, providing flexibility and mobility for users. Here's an overview of wireless networking:

Components of Wireless Networking:

- **Wireless Access Points (APs)**: Wireless access points are devices that create and manage wireless local area networks (WLANs). They act as central points for wireless communication and facilitate connections between wireless devices (such as laptops, smartphones, and tablets) and the wired network infrastructure.
- **Wireless Routers**: A wireless router combines the functions of a traditional router with those of a wireless access point. In addition to routing network traffic between different networks, wireless routers also provide wireless connectivity to devices within a home or office environment.
- **Wireless Clients**: Wireless clients are devices equipped with wireless network adapters (such as Wi-Fi adapters) that enable them to connect to wireless networks. Examples of wireless clients include laptops, smartphones, tablets, and smart home devices.

- **Wireless Network Interface Cards (NICs)**: A wireless network interface card (NIC) is a hardware component installed in a device to enable wireless communication. It allows the device to connect to wireless networks and exchange data wirelessly.
- **Wireless Standards**: Wireless networking standards define the specifications for wireless communication protocols and technologies. Common wireless standards include: 802.11a/b/g/n/ac/ax: These are various iterations of the IEEE 802.11 standard, with each version offering improvements in data rates, range, and performance. Wi-Fi 6 (802.11ax): The latest standard as of my last update, Wi-Fi 6 offers higher data rates, improved network efficiency, and better performance in crowded environments compared to previous standards.

1.2 HAProxy

1.2.1 The role of HAProxy

HAProxy, which stands for High Availability Proxy, is an open-source software solution that functions as a load balancer and proxy server for TCP and HTTP-based applications. It is widely used to improve the performance, scalability, and reliability of web servers and applications by distributing incoming traffic across multiple backend servers.

Here are the key roles and features of HAProxy:

- **Load Balancing**: One of the primary roles of HAProxy is to evenly distribute incoming network traffic (HTTP, HTTPS, TCP, or UDP) across multiple backend servers. It balances the load by intelligently routing requests based on various factors such as server health, server capacity, and predefined algorithms (e.g., round-robin, least connections, source IP hashing).
- **High Availability**: HAProxy improves the availability and fault tolerance of applications by detecting server failures and automatically redirecting traffic to healthy servers. It continuously monitors the health of backend servers using health checks (e.g., TCP checks, HTTP checks) and removes or adds servers to the load-balancing pool based on their status.
- **SSL/TLS Termination**: HAProxy can act as an SSL/TLS termination point, offloading the SSL/TLS encryption and decryption process from backend servers. This reduces the computational overhead on backend servers and simplifies SSL/TLS certificate management.
- **Proxying and Reverse Proxying**: HAProxy serves as a forward proxy or reverse proxy, depending on the deployment scenario. In forward proxy mode, it forwards client requests to backend servers, while in reverse proxy mode, it accepts client requests on behalf of backend servers and forwards responses back to clients. Reverse proxying is commonly used to hide backend infrastructure details, improve security, and optimize resource utilization.
- **Caching**: HAProxy can cache static content (e.g., images, CSS files) in memory to reduce the load on backend servers and improve response times for clients. It supports various caching strategies and configurations to optimize cache utilization and eviction policies.
- **Content-Based Routing**: HAProxy can route incoming requests to different backend servers based on specific content attributes, such as URL paths, HTTP headers, cookies, or request methods. This allows for sophisticated routing and traffic management based on application requirements.
- **Session Persistence**: HAProxy provides session persistence (also known as sticky sessions) by ensuring that requests from the same client are consistently routed to the same backend server. This is essential for maintaining session state and ensuring a seamless user experience in stateful applications.
- **Logging and Monitoring**: HAProxy offers extensive logging and monitoring capabilities, allowing administrators to track request traffic, server health, performance metrics, and error conditions in

real-time. Logs can be customized and integrated with external logging systems for analysis and troubleshooting.



Figure 1.2: Haproxy[2]

1.2.2 A brief history of HAProxy

HAProxy, or High Availability Proxy, has a rich history that spans several decades, evolving from its origins as a simple load balancer to become one of the most widely used and respected open-source software solutions for load balancing and proxying.

Here's a brief history of HAProxy:

Early Development (2000s): HAProxy was initially developed in 2000 by Willy Tarreau, a software engineer based in France. It was created to address the need for a reliable and high-performance load balancer to manage web traffic for online gaming applications. In its early stages, HAProxy primarily focused on load balancing HTTP and HTTPS traffic, providing basic features such as round-robin and least connections load-balancing algorithms.

Open-Source Release (2001): HAProxy was released as open-source software under the GNU General Public License (GPL) in 2001. This allowed developers and organizations to freely use, modify, and distribute the software, contributing to its widespread adoption.

Feature Expansion (Mid-2000s): Over time, HAProxy evolved to support a broader range of features and protocols, including TCP and UDP load balancing, SSL/TLS termination, health checks, and content-based routing. The development of HAProxy was driven by community feedback and contributions, with new features and enhancements added regularly to meet the growing demands of users.

Stability and Performance Improvements (Late 2000s to Present): HAProxy gained a reputation for its stability, reliability, and high performance, making it a popular choice for load balancing and proxying in production environments.

Widespread Adoption (2010s to Present): HAProxy continued to gain traction in the industry, becoming a cornerstone of many web infrastructures due to its flexibility, scalability, and robust feature set. It is widely used by large enterprises, cloud service providers, content delivery networks (CDNs), and internet companies to distribute traffic, improve application performance, and ensure high availability.**1.3 Nginx**

1.3 Nginx

1.3.1 The role of Nginx

Nginx (pronounced "engine-x") is a powerful open-source web server and reverse proxy server that is known for its high performance, scalability, and reliability. Originally developed to address the scalability challenges of high-traffic websites, Nginx has since evolved into a versatile platform that serves various roles in modern web architectures.

Here are some key roles and features of Nginx:

- **Web Server**: Nginx can serve static content, such as HTML, CSS, JavaScript, and image files, directly to clients without the need for additional server-side processing. Its asynchronous, event-driven architecture allows Nginx to handle many concurrent connections efficiently, making it well-suited for high-traffic websites and applications.
- **Reverse Proxy Server**: Nginx can act as a reverse proxy server, sitting in front of one or more backend servers and distributing incoming client requests to those servers based on various criteria. As a reverse proxy, Nginx can improve security by hiding the internal infrastructure details of backend servers from external clients and providing an additional layer of defense against malicious attacks.

- **Load Balancer**: Nginx can perform load balancing across multiple backend servers to distribute incoming traffic evenly and efficiently. It supports various load-balancing algorithms, including round-robin, least connections, IP hash, and more, allowing administrators to configure load balancing based on their specific requirements.
- **Caching**: Nginx can cache static and dynamic content in memory or on disk to improve performance and reduce the load on backend servers. By caching frequently accessed content, Nginx can serve subsequent requests for the same content more quickly, resulting in faster response times for clients.
- **SSL/TLS Termination**: Nginx can terminate SSL/TLS connections, offloading the encryption and decryption process from backend servers. It supports various SSL/TLS protocols and cipher suites, allowing administrators to configure secure communication between clients and servers.
- **HTTP/2 and HTTP/3 Support**: Nginx supports modern HTTP protocols such as HTTP/2 and HTTP/3, which offer improvements in performance, security, and efficiency over traditional HTTP/1.1. HTTP/2 and HTTP/3 support features such as multiplexing, header compression, and server push, further enhancing the performance of web applications.
- **Dynamic Content Processing**: Nginx can proxy requests to backend application servers (e.g., PHP, Node.js, Python) to process dynamic content and generate responses dynamically. It supports various scripting languages and frameworks, allowing developers to build dynamic web content applications and APIs.
- **High Availability and Scalability**: Nginx is designed for high availability and scalability, with features such as fault tolerance, health checks, and auto-recovery mechanisms. It can be deployed in clustered or distributed configurations to handle large volumes of traffic and ensure uninterrupted service availability.

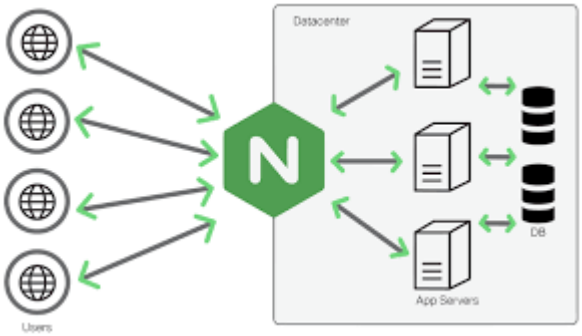


Figure 1.3: Nginx [3]

1.3.1 A brief history of Nginx

Origins and Development (2002-2004): Nginx was initially created by Igor Sysoev, a Russian software engineer, to address the performance limitations of existing web servers when dealing with high-traffic websites and applications. Development of Nginx began in 2002, and the first public release (version 0.1.0) occurred on October 4, 2004.

Introduction to the Public (2005) Nginx gained early attention and adoption for its lightweight and efficient architecture, which allowed it to handle many concurrent connections with low resource usage. The release of Nginx version 0.1.21 in April 2005 marked the first public release to gain widespread recognition in the web server community.

Growth and Adoption (Late 2000s to Early 2010s): As the internet landscape evolved and the demand for high-performance web servers increased, Nginx experienced significant growth in adoption. Many high-profile websites and companies, including Netflix, Dropbox, Airbnb, and WordPress.com, adopted Nginx to improve the performance, scalability, and reliability of their web infrastructure.

Expansion of Features (2010s): Throughout the 2010s, Nginx continued to evolve and expand its feature set to meet the changing needs of web developers and system administrators. Features such as reverse proxying, load balancing, SSL/TLS termination, caching, and support for modern web protocols like HTTP/2 and HTTP/3 were added to Nginx, further enhancing its capabilities.

Establishment of NGINX, Inc. (2011): In 2011, Nginx, Inc. was founded to provide commercial support, services, and additional products around the Nginx software. Nginx, Inc. offers commercial variants of Nginx, such as Nginx Plus, which includes additional features and enterprise-grade support.

Acquisition by F5 Networks (2019): In March 2019, F5 Networks, a leading provider of application delivery and security solutions, announced its acquisition of Nginx, Inc. for approximately $670 million.

1.4 Keepalived

1.4.1 The role of keepalived

Keepalived is an open-source software solution that provides high availability and load balancing for Linux systems. It is primarily used to ensure the continuous availability of network services by implementing redundancy and failover mechanisms. Keepalived is commonly deployed in conjunction with load balancers, web servers, and other critical network services to maintain uptime and reliability.

Here's an overview of the key features and roles of Keepalived:

- **Virtual IP Address Management**: Keepalived manages virtual IP addresses (VIPs) that are shared among multiple nodes in a cluster. The virtual IP address serves as the entry point for clients accessing network services, such as web applications or databases.
- High Availability: Keepalived monitors the health and status of servers (nodes) in a cluster using health checks. If a node becomes unreachable or fails, Keepalived automatically transfers the virtual IP address to another healthy node, ensuring uninterrupted service availability.
- **Load Balancing**: Keepalived can distribute incoming traffic across multiple backend servers (real servers) based on various load-balancing algorithms. It supports algorithms such as round-robin, least connections, source IP hashing, and more, allowing administrators to configure load balancing according to their requirements.
- **Configuration Synchronization**: Keepalived synchronizes configuration information (such as virtual IP addresses and load-balancing rules) across all nodes in a cluster. This ensures consistency and coherence in the cluster configuration, reducing the risk of configuration errors and discrepancies.
  VRRP Protocol Implementation: Keepalived implements the Virtual Router Redundancy Protocol (VRRP), which allows multiple nodes to work together as a single virtual router. VRRP enables automatic failover in the event of node failures and provides a mechanism for electing a master node to handle traffic.
  Health Checking and Monitoring: Keepalived performs health checks on backend servers to verify their availability and responsiveness. Health checks can be configured to use various protocols (such as TCP, HTTP, ICMP) and check specific service endpoints to ensure that servers are functioning properly.
  Event Notification and Logging: Keepalived provides event notification and logging capabilities to alert administrators of significant events, such as failover events or changes in node status. Logs help

administrators monitor the health and performance of the cluster and troubleshoot issues as they arise.

Overall, Keepalived plays a critical role in ensuring high availability and reliability for network services by providing redundancy, failover, and load balancing capabilities. Its lightweight design, simplicity, and robust feature set make it a popular choice for implementing high availability solutions in Linux-based environments.

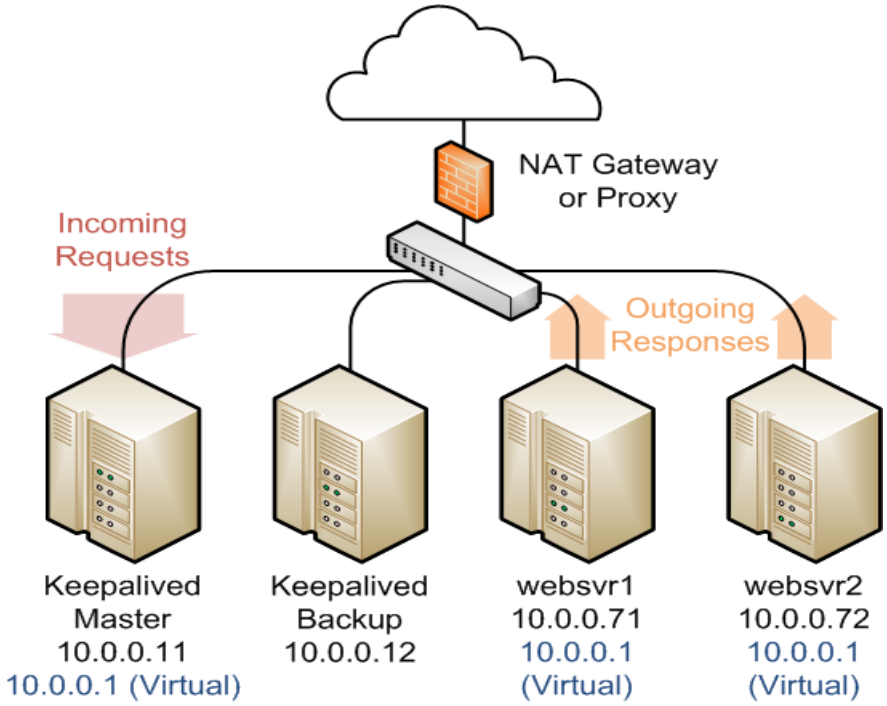


Figure 1.4: Keepalived protocol [3]

1.5 Virtualization

1.5.1 Hardware Virtualization

Hardware virtualization, also known as platform virtualization, refers to the process of creating multiple virtual instances of physical hardware resources, such as servers, storage devices, or networking devices, on a single physical host machine. This technology allows multiple virtual machines (VMs) to run simultaneously on a single physical server, each with its own operating system (OS) and applications, effectively enabling the consolidation of multiple computing environments onto a single hardware platform.

Here's an overview of hardware virtualization:

Virtualization Layers: Hardware virtualization typically involves multiple layers of software that abstract and emulate hardware resources to create virtual environments. The lowest layer is the hardware layer, consisting of the physical CPU, memory, storage, and networking components of the host machine. On top of the hardware layer, a hypervisor (also known as a virtual machine monitor or VMM) is installed. The hypervisor is responsible for managing and controlling the virtualized hardware resources and providing a platform for running multiple guest VMs. Guest VMs run on top of the hypervisor, each with its own virtualized hardware environment, including virtual CPUs, memory, disk, and network interfaces.

Hypervisor Types: Hypervisors can be classified into two main types: Type 1 (bare-metal) hypervisors and Type 2 (hosted) hypervisors.

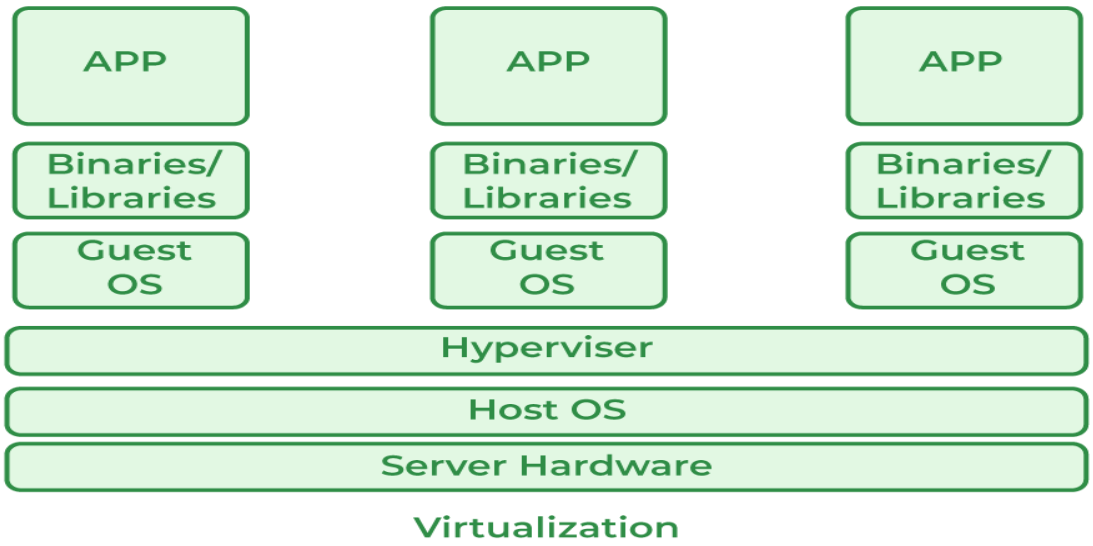


Figure 1.5: Virtualization[4]

1.5.2 OS Virtualization

OS virtualization, also known as operating system-level virtualization or containerization, is a form of virtualization technology that allows multiple isolated user space instances, often referred to as containers, to run on a single host operating system (OS). Unlike hardware virtualization, where each virtual machine (VM) runs its own OS on top of a hypervisor, OS virtualization shares the host OS kernel among multiple containers, enabling lightweight and efficient isolation of applications.

Here's an overview of OS virtualization:

Containerization: In OS virtualization, containers encapsulate an application and its dependencies, including libraries, binaries, and configuration files, into a self-contained package. Each container shares the host OS kernel and resources, such as CPU, memory, and filesystem, with other containers, but maintains its own isolated user space. Containers are lightweight, portable, and fast to deploy, making them ideal for deploying and scaling microservices-based applications, DevOps workflows, and cloud-native environments.

Container Runtimes: Container runtimes, such as Docker, container, and rkt (Rocket), provide the necessary tools and runtime environment for creating, managing, and running containers. Container runtimes leverage OS-level virtualization features, such as namespaces and control groups (cgroups), to isolate and control resource usage for each container.

Kernel-Based Virtual Machine (KVM): Some implementations of OS virtualization, such as KVM, combine elements of hardware virtualization with containerization. KVM allows running multiple VMs, each with its own OS, on a host system, while also supporting containerization through tools like Kata Containers or Firecracker.

Key Features

- **Lightweight**: Containers have minimal overhead compared to traditional VMs since they share the host OS kernel and resources.
- **Fast Startup**: Containers can start up in milliseconds, making them ideal for rapidly deploying and scaling applications.
- **Resource Efficiency**: OS virtualization allows for better resource utilization by sharing resources among containers and optimizing resource allocation.
- **Portability**: Containers can run consistently across different environments, including development, testing, staging, and production, ensuring consistency and reproducibility.

Use Cases: Application Packaging and Deployment: Containers simplify the packaging and deployment of applications, enabling developers to build, test, and deploy applications more efficiently.

1.5.3 Network Virtualization

Network virtualization is a technology that abstracts and decouples network resources from the underlying physical network infrastructure, allowing multiple virtual network instances to run on top of a shared physical network. This enables the creation of isolated and logically separate network environments that can coexist and operate independently, providing flexibility, scalability, and efficiency in network management.

Here's an overview of network virtualization:

Virtual Networks:

In network virtualization, virtual networks are created by abstracting and partitioning physical network resources, such as switches, routers, firewalls, and network links. Each virtual network operates as an independent logical network with its own addressing, routing, security policies, and traffic management.

Key Components

Hypervisor or Network Virtualization Platform: Network virtualization is often implemented using a hypervisor or network virtualization platform that abstracts physical network infrastructure and provides virtual network capabilities.

Virtual Switches and Routers: Virtual switches and routers within the hypervisor or network virtualization platform enable the creation and management of virtual network segments and interconnectivity between virtual machines (VMs) or containers.

Network Overlay Technologies: Network overlay technologies, such as VXLAN (Virtual Extensible LAN), NVGRE (Network Virtualization using Generic Routing Encapsulation), and Geneve, facilitate the creation of virtual networks over existing physical network infrastructure by encapsulating and tunneling virtual network traffic.

Software-defined Networking (SDN) Controllers: SDN controllers provide centralized control and management of virtualized network resources, enabling dynamic provisioning, configuration, and optimization of virtual networks.

1.5.4 VirtualBox

VirtualBox is a free and open-source virtualization software developed by Oracle Corporation. It allows users to create and run virtual machines (VMs) on their host operating system, enabling them to run multiple guest operating systems simultaneously. VirtualBox supports a wide range of guest operating systems, including Windows, Linux, macOS, Solaris, and various versions of BSD.

Here's an overview of VirtualBox's features and capabilities:

Cross-Platform Compatibility: VirtualBox is available for multiple hosts operating systems, including Windows, macOS, Linux, and Solaris, making it a versatile solution for virtualization across different platforms.

Support for Various Guest Operating Systems: VirtualBox supports a wide range of guest operating systems, including Windows (NT 4.0, 2000, XP, Server 2003, Vista, 7, 8, 10), Linux (Ubuntu, Debian, Fedora, CentOS, etc.), macOS (OS X, macOS), Solaris, and various versions of BSD (FreeBSD, OpenBSD, NetBSD).

Flexible Virtual Machine Management: Users can create, configure, and manage virtual machines through an intuitive graphical user interface (GUI) or command-line interface (CLI). VirtualBox allows users to adjust various settings for virtual machines, including CPU and memory allocation, storage configuration, network settings, and display options.

Hardware Virtualization Support: VirtualBox supports hardware virtualization features, such as Intel VT-x and AMD-V, which enhance performance and enable features like nested virtualization. Users can enable hardware virtualization support in the VM settings to improve the performance of guest operating systems.

Snapshot and Clone Functionality: VirtualBox allows users to take snapshots of virtual machines at different points in time, enabling them to revert to previous states or create multiple branches for testing and development. Users can also clone virtual machines to create identical copies for backup, testing, or distribution purposes.

Virtual Networking: VirtualBox provides flexible networking options for virtual machines, including NAT (Network Address Translation), Bridged networking, Host-only networking, and Internal networking. Users can configure networking settings to enable communication between virtual machines and the host system or between virtual machines themselves.

Guest Additions: VirtualBox Guest Additions are software packages that enhance the performance and usability of guest operating systems running inside VirtualBox VMs. Guest Additions provide features such as seamless mouse integration, shared folders, improved display resolutions, and better support for graphics and audio.

USB Device Support: VirtualBox allows users to connect USB devices to virtual machines, enabling them to use USB peripherals such as flash drives, printers, scanners, and external storage devices within the VM environment.

Overall, VirtualBox is a feature-rich and user-friendly virtualization solution suitable for various use cases, including software development, testing, education, and running legacy applications on modern hardware. Its open-source nature and extensive community support make it a popular choice among both individual users and organizations looking for a cost-effective virtualization solution.

1.6 Security

1.6.1 Firewalls

Firewalls are network security devices or software that monitor, and control incoming and outgoing network traffic based on predetermined security rules. They act as a barrier between a trusted internal network and untrusted external networks (such as the internet), filtering traffic to prevent unauthorized access, malicious activities, and potential security threats.

Here's an overview of firewalls:

Packet Filtering Firewalls: Packet filtering firewalls inspect individual packets of data as they pass through the firewall and make decisions based on predefined rules, such as IP addresses, port numbers, and protocols. They can allow or block packets based on criteria such as source and destination IP addresses, source and destination ports, and packet type (e.g., TCP, UDP, ICMP).

Stateful Inspection Firewalls: Stateful inspection firewalls maintain state information about active network connections and analyze the context of traffic flows to make more intelligent filtering decisions. They keep track of the state of network connections (e.g., TCP handshake, established connections) and only allow packets that belong to legitimate, established connections.

Application Layer Firewalls (Proxy Firewalls): Application layer firewalls operate at the application layer (Layer 7) of the OSI model and can inspect traffic at a deeper level, including application-specific protocols and payloads. They act as intermediaries between clients and servers, intercepting and inspecting traffic before forwarding it to the destination. Application layer firewalls can enforce more granular security policies and provide additional features such as content filtering, URL filtering, and application-level access control.

Next-Generation Firewalls (NGFW): Next-generation firewalls combine traditional firewall functionalities with advanced security features, such as intrusion prevention system (IPS), deep packet inspection (DPI), application awareness, and user identity awareness. NGFWs offer more advanced threat detection and prevention capabilities compared to traditional firewalls, allowing organizations to detect and mitigate a wide range of network-based attacks and vulnerabilities.

Unified Threat Management (UTM): Unified threat management (UTM) appliances integrate multiple security features, including firewall, intrusion detection and prevention (IDP), antivirus, antispam, VPN, and content filtering, into a single platform. UTM appliances provide comprehensive security solutions for small and medium-sized businesses (SMBs) and branch offices, offering simplified management and deployment of multiple security functionalities.

Cloud Firewalls: Cloud firewalls are firewall services provided by cloud service providers (CSPs) to protect virtual networks, cloud-based applications, and resources hosted in the cloud. They offer scalable and elastic firewall capabilities for securing cloud-based environments, allowing organizations to define and enforce security policies across distributed and dynamic cloud infrastructures. Firewalls play a critical role in network security by providing a first line of defense against various cyber threats, including unauthorized access, malware, ransomware, phishing attacks, and denial-of-service (DoS) attacks. They are essential components of a layered security approach, working in conjunction with other security solutions such as intrusion detection and prevention systems (IDPS), antivirus software, and security information and event management (SIEM) systems to protect networks and data from cyber threats.

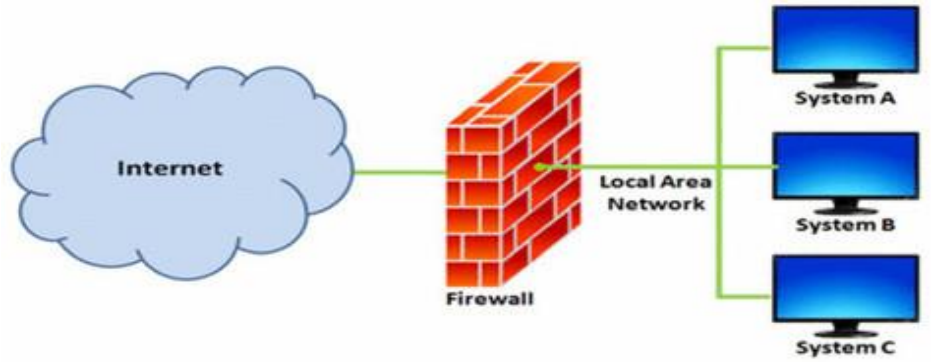


Figure 1.6: Security [5]

1.6.2 Virtual Private Networks

Virtual Private Networks (VPNs) are secure and encrypted connections established over a public network, typically the internet, that allow users to securely access private networks, resources, and services from remote locations. VPNs provide privacy, security, and anonymity by encrypting network traffic and routing it through a VPN server, thereby masking the user's IP address and protecting data from eavesdropping and interception.

Here's an overview of VPNs:

Types of VPNs:

Remote Access VPN: Remote access VPNs allow individual users to connect securely to a private network from remote locations, such as home offices or public Wi-Fi hotspots. They provide secure access to corporate resources, applications, and services for remote employees, contractors, or mobile workers.

Site-to-Site VPN: Site-to-site VPNs establish secure connections between multiple networks or sites over the internet. They enable secure communication and data exchange between geographically dispersed offices, branches, or data centers, creating a virtual network infrastructure that spans multiple physical locations.

Client-to-Site VPN: Client-to-site VPNs, also known as endpoint VPNs or host-to-network VPNs, allow individual client devices, such as laptops, smartphones, or tablets, to connect securely to a private network or VPN server from remote locations. They provide secure access to corporate networks and resources for mobile users and remote workers.

Secure Sockets Layer (SSL) VPN: SSL VPNs use the SSL/TLS protocol to establish secure and encrypted connections between client devices and VPN servers. They are commonly used for remote access VPNs and provide secure access to web-based applications and resources through a web browser.

Internet Protocol Security (IPsec) VPN: IPsec VPNs use the IPsec protocol suite to secure IP communication by encrypting and authenticating IP packets. They are widely used for site-to-site VPNs and client-to-site VPNs, offering robust security and interoperability across different network devices and platforms.

Layer 2 Tunneling Protocol (L2TP)/IPsec: L2TP/IPsec combines the L2TP protocol with IPsec encryption and authentication mechanisms to create secure VPN tunnels. It is commonly used for site-to-site VPNs and client-to-site VPNs, providing strong security and compatibility with various network devices and operating systems.

OpenVPN: OpenVPN is an open-source VPN protocol that uses SSL/TLS encryption for secure communication. It is highly configurable and flexible, supporting various authentication methods and encryption algorithms. OpenVPN is often used for remote access VPNs and client-to-site VPNs.

1.6.3 Secure Socket Layer/Transport Layer Security

Secure Sockets Layer (SSL) is a cryptographic protocol designed to provide secure communication over a computer network, commonly the internet. SSL ensures confidentiality, integrity, and authenticity of data transmitted between clients and servers by encrypting the data and verifying the identities of the communicating parties.

Here's an overview of SSL:

Confidentiality: SSL encrypts data transmitted between a client (such as a web browser) and a server (such as a website) to prevent eavesdropping and unauthorized access to sensitive information. SSL uses asymmetric encryption (public-key cryptography) and symmetric encryption to establish a secure connection between the client and server. During the SSL handshake process, the client and server negotiate encryption algorithms and exchange cryptographic keys to encrypt and decrypt data securely.

Authentication: SSL provides authentication mechanisms to verify the identities of the communicating parties (client and server) and ensure that they are who they claim to be. SSL certificates are used to authenticate servers and establish trust between clients and servers. SSL certificates are issued by trusted certificate authorities (CAs) and contain information about the server's identity, such as its domain name and public key. When a client connects to a server over SSL, the server presents its SSL certificate to the client, allowing the client to verify the server's identity and establish a secure connection.

Integrity: SSL ensures the integrity of data transmitted between the client and server by detecting and preventing data tampering or modification during transmission. SSL uses message authentication codes (MACs) or hash functions to generate checksums or digests of transmitted data, which are then used to verify the integrity of the data upon receipt.

SSL/TLS Protocol: SSL is often used interchangeably with Transport Layer Security (TLS), which is the successor to SSL and provides similar security features. The SSL/TLS protocol operates at the transport layer (Layer 4) of the OSI model and can be used with various application layer protocols, such as HTTP, SMTP, FTP, and IMAP, to secure communication between clients and servers.

SSL Handshake Process: The SSL handshake process is the initial phase of establishing a secure connection between the client and server. During the handshake, the client and server exchange SSL certificates, negotiate encryption algorithms and cryptographic keys, and verify each other's identities.

1.6.4 Network Access Control

Network Access Control (NAC) is a security solution that enforces policies to ensure that only authorized devices and users can access a network. NAC solutions are used to secure wired, wireless, and remote network access by implementing authentication, authorization, and endpoint security measures.

Authentication: NAC solutions authenticate users and devices attempting to connect to a network, verifying their identity and credentials before granting access. Authentication methods may include username/password authentication, digital certificates, biometric authentication, and multifactor authentication (MFA). Devices are often required to authenticate using unique identifiers, such as MAC addresses, IP addresses, or digital certificates.

Authorization: After authentication, NAC solutions enforce access policies to determine the level of access control granted to authenticated users and devices. Access policies define rules and criteria based on factors such as user roles, device types, network locations, time of day, and security posture. Authorized users and devices may be granted access to specific network resources, services, and applications based on their assigned permissions and privileges.

Endpoint Security: NAC solutions assess the security posture of devices seeking network access to ensure they meet predefined security requirements and compliance standards. Endpoint security checks may include antivirus and anti-malware scans, operating system updates, software patches, firewall configurations, and device encryption.

1.6.5 SELinux

SELinux (Security-Enhanced Linux) is a security feature integrated into Linux distributions that provides a mandatory access control (MAC) mechanism to enforce security policies. Unlike traditional discretionary access controls (DAC), which rely on user or group ownership of files and permissions, SELinux defines policies based on rules and contexts associated with subjects (such as processes and users) and objects (such as files, directories, and network ports).

Here's how SELinux policy enforcement works:

Policy Configuration: SELinux policies are defined through configuration files and rules that specify which actions subjects are allowed to perform on objects based on their security contexts. Policies define rules for subjects, objects, and operations, specifying what actions are allowed, denied, or audited.

Security Contexts: Every process, file, directory, and network port in SELinux has a security context associated with it. Security contexts consist of a type, role, user, and optional sensitivity level, providing granular control over access permissions.

Policy Enforcement: When a subject attempts to perform an operation on an object, SELinux checks the security contexts of both the subject and the object against the defined policies. Access is allowed only if the policy permits the action based on the security contexts and rules defined in the policy configuration. If the action violates the policy, SELinux enforces access controls by denying the operation or triggering an audit alert, depending on the policy configuration.

Modes of Operation:

- Enforcing Mode: SELinux actively enforces security policies, denying unauthorized access attempts and generating audit logs for policy violations.
- Permissive Mode: SELinux logs policy violations but does not enforce access controls, allowing administrators to identify and troubleshoot policy issues without blocking operations.

Policy Customization: Administrators can customize SELinux policies to suit their specific security requirements using tools such as semanage, setsebool, and audit2allow. Customizations may involve defining new policies, modifying existing policies, or adjusting boolean settings to control specific behaviors.

Auditing and Logging: SELinux provides auditing capabilities to log security-relevant events, including policy violations, access denials, and policy changes. Audit logs can be analyzed using tools like `auditctl` and `ausearch` to investigate security incidents and monitor system activity.

Integration with Applications: Applications can be designed to interact with SELinux to define security contexts and request access permissions through SELinux-aware APIs. By integrating with SELinux, applications can enforce fine-grained access controls and reduce the attack surface by limiting the privileges granted to processes and users.

In summary, SELinux policy enforcement plays a crucial role in enhancing the security of Linux systems by enforcing mandatory access controls based on defined policies and security contexts. By leveraging SELinux, administrators can strengthen access controls, mitigate security risks, and maintain the integrity and confidentiality of sensitive data and resources.

2.0 Private/Public NATs, Port Forwarding

2.1 Introduction to Network Address Translation (NAT)

Network Address Translation (NAT) is a fundamental networking technology used to enable communication between devices in different IP address realms, such as private and public networks. It serves several purposes, including conserving public IP addresses, enhancing security, and facilitating connectivity within private networks. Here's an introduction to NAT:

Conserving Public IP Addresses: NAT allows multiple devices within a private network to share a single public IP address, reducing the need for unique public IP addresses for each device.

Enhancing Security: NAT acts as a barrier between the private internal network and the public internet, hiding internal IP addresses from external networks and providing a level of security by obscuring the internal network topology.

Enabling Communication: NAT enables devices with private IP addresses to communicate with devices on the public internet by translating their private IP addresses to a public IP address when sending traffic outside the private network.

How NAT Works:

Translation: NAT translates IP addresses and/or port numbers in IP packet headers between the private and public address spaces.

Mapping Table: NAT maintains a mapping table that records the translations between private and public IP addresses, as well as associated port numbers.

Types of NAT:

Static NAT: Maps a fixed private IP address to a corresponding public IP address, typically used for servers or services that need to be accessible from the internet.

Dynamic NAT: Dynamically assigns available public IP addresses from a pool to internal devices as needed, allowing multiple devices to share a smaller pool of public IP addresses.

PAT (Port Address Translation): Also known as NAT Overload, maps multiple private IP addresses to a single public IP address using unique port numbers to distinguish between connections.

Components**:**

NAT Device: The router or firewall that performs the NAT operation, translating IP addresses and managing the mapping table.

Private IP Addresses: IP addresses used within a private network, typically from the ranges defined in RFC 1918 (e.g., 192.168.x.x, 10.x.x.x).

Public IP Addresses: IP addresses routable on the public internet, assigned by Internet Service Providers (ISPs) to enable communication with external networks.

Benefits:

IP Address Conservation: Allows organizations to use private IP address ranges internally while conserving public IP addresses.

Security: Hides internal network structure and addresses from external networks, providing a layer of security against unauthorized access and attacks.

Simplified Network Management: Facilitates network management by reducing the complexity of IP address assignment and routing.

Limitations:

End-to-End Connectivity: NAT can hinder certain applications or protocols that rely on end-to-end connectivity, such as IPsec VPNs or peer-to-peer networking.

Complexity: Large-scale deployments or configurations involving multiple NAT devices can introduce complexity and management overhead.

Performance Impact: Processing NAT translations may introduce latency and overhead, impacting network performance in high-traffic environments.

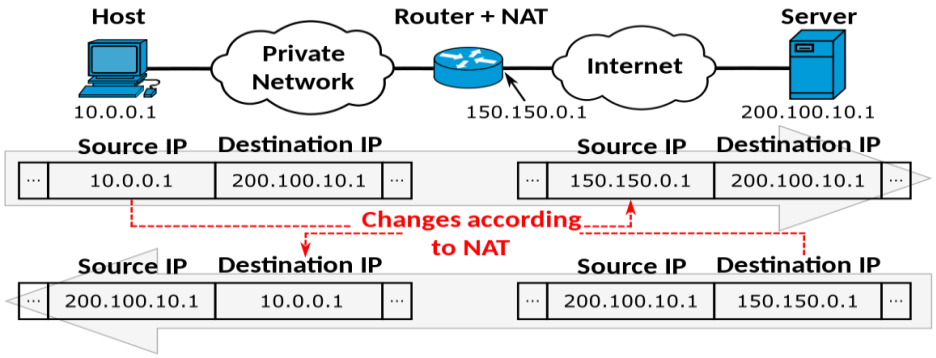


Figure 2.1: NAT [6]

2.2 NAT Types

Static NAT (SNAT): Static NAT maps a specific private IP address to a corresponding public IP address on a one-to-one basis. Typically, it is used for hosting servers or services that need to be accessible from the internet using a fixed public IP address. *Example***:** A web server with a private IP address (e.g., 192.168.1.2) mapped to a public IP address (e.g., 203.0.113.10).

Dynamic NAT: Dynamic NAT dynamically assigns available public IP addresses from a pool to internal devices as needed, allowing multiple devices to share a smaller pool of public IP addresses. It is suitable for environments where a limited number of public IP addresses are available, and devices require intermittent internet access. *Example***:** Multiple devices within a private network accessing the internet through a pool of public IP addresses.

Port Address Translation (PAT) or NAT Overload: PAT, also known as NAT Overload, maps multiple private IP addresses to a single public IP address using unique port numbers to distinguish between connections. It is used in home and small office environments where many devices need to share a single public IP address.

Example: Multiple devices within a home network accessing the internet through a single public IP address, with each connection distinguished by unique port numbers.

Overlapping NAT: Overlapping NAT allows the use of overlapping private IP address ranges within different isolated network segments by translating these addresses to non-overlapping addresses when communicating with external networks. It is useful in complex network architectures where multiple private IP address ranges need to be integrated without conflicts. *Example*: Multiple branch offices using overlapping private IP address ranges connecting to a central network through NAT translation to avoid IP address conflicts.

2.3 Port Forwarding

Port forwarding is a networking technique that allows inbound traffic on a specific port of a router or firewall to be redirected to a device or service on the internal network. It enables external users to access services hosted on devices within a private network, such as web servers, FTP servers, or gaming servers, by mapping a public IP address and port to a private IP address and port.

Here's how port forwarding works and its key aspects:

Configuration: Port forwarding is configured on the router or firewall that serves as the gateway between the internal network and the internet.

Mapping: A specific port on the public IP address of the router is mapped to a corresponding port on the private IP address of the device hosting the service.

Routing: When external traffic arrives at the router's public IP address on the specified port, the router forwards the traffic to the designated device on the internal network based on the port forwarding rule.

Remote Access: Allows users outside the local network to access services hosted on devices within the network, such as accessing a home security camera feed or connecting to a corporate VPN server.

Hosting Services: Enables hosting of servers or services, such as web servers, FTP servers, game servers, or remote desktop services, behind a router or firewall with a single public IP address.

Components:

Public IP Address: The external IP address assigned to the router or firewall by the Internet Service Provider (ISP), which external users use to access services hosted within the private network.

Private IP Address: The internal IP address assigned to the device hosting the service within the private network.

Port Number: A unique identifier assigned to network protocols or services, ranging from 0 to 65535, used to differentiate between multiple services running on the same device.

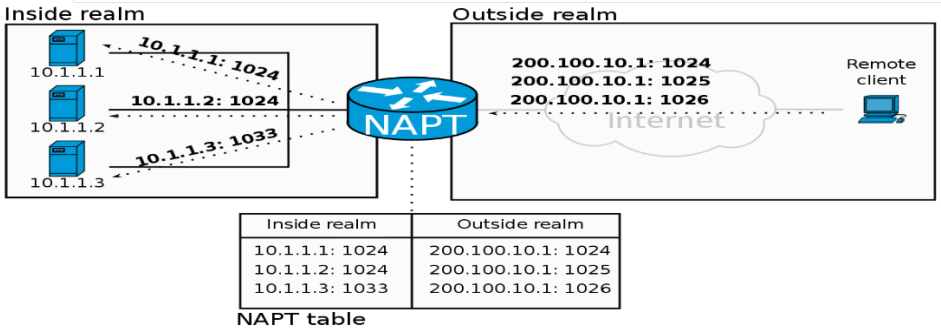


Figure 2.3: Port Forwarding [5]

2.4 Port Address Translation

Port Address Translation (PAT), also known as Port Mapping or NAT Overload, is a variation of Network Address Translation (NAT) that allows multiple devices within a private network to share a single public IP address. Unlike traditional NAT, where each internal device is mapped to a unique public IP address, PAT uses unique port numbers to distinguish between multiple internal devices accessing the internet simultaneously.

Translation Table: The router or firewall performing PAT maintains a translation table that associates each internal device's private IP address and port number with a unique public IP address and port number.

Dynamic Port Allocation: When an internal device initiates outbound traffic to the internet, the router assigns a unique source port number to the traffic.

Outbound Translation: The router translates the source IP address of the internal device to the public IP address and maps the source port number to a unique port number on the public IP address.

Inbound Translation: When the external server responds to the outbound traffic, it sends the response to the public IP address and port number. The router translates the destination port number to the original port number used by the internal device and forwards the response to the appropriate internal device.

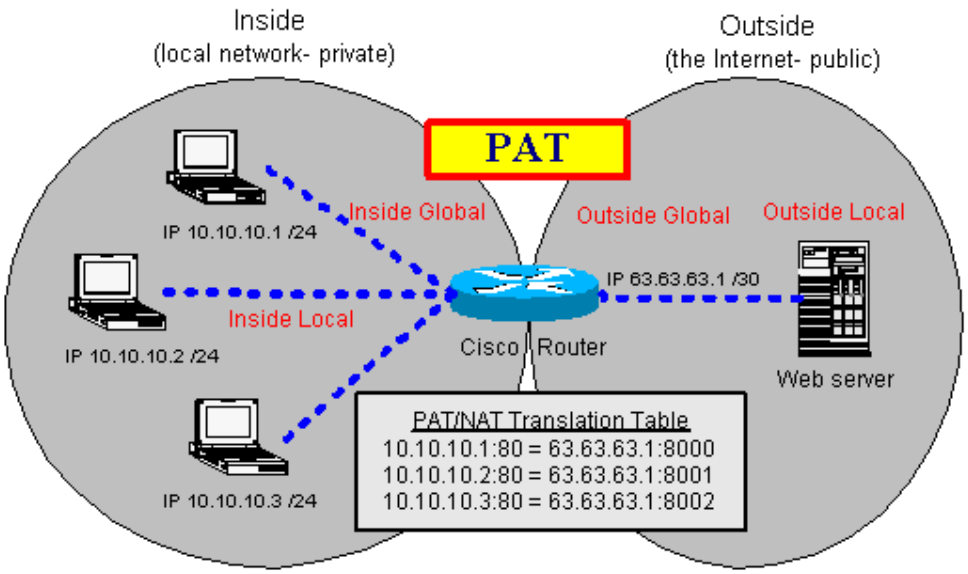


Figure 2.4: PAT [7]

3. High performance webserver using HAProxy for load balancing

3.1 Basic Architecture

Load Balancer (HAProxy):
HAProxy serves as the front-end load balancer, distributing incoming web traffic across multiple backend servers. It performs health checks on backend servers and ensures that requests are routed efficiently. HAProxy can be configured with various algorithms for load balancing such as round-robin, least connections, source IP hashing, etc., depending on specific requirements.

Web Servers:
Multiple web servers (e.g., Apache, Nginx) are deployed as backend servers behind the HAProxy load balancer. Web servers are responsible for serving web content, processing requests, and generating responses. Web servers are configured for high performance and scalability, including optimizations such as caching, compression, and efficient resource utilization.

Database Server:
For database access, a separate database server should be deployed. This server should be optimized for performance and reliability, with appropriate backup and replication mechanisms in place.

Monitoring and Logging:
Monitoring tools such as Prometheus, Grafana, or Nagios can be used to monitor the health and performance of the infrastructure, including server load, response times, and error rates. Centralized logging using tools like ELK stack (Elasticsearch, Logstash, Kibana) or Splunk can collect and analyze logs from HAProxy, web servers, and other components.

Security:
Security measures such as firewalls, intrusion detection/prevention systems (IDS/IPS), and SSL/TLS encryption can be configured to protect your infrastructure and data. The administrator needs to regularly update software and apply security patches to mitigate vulnerabilities.

Scalability and High Availability:
Design your infrastructure for scalability and high availability to handle increased traffic and ensure uptime. Multiple instances of HAProxy allow distributing traffic across them using DNS-based load balancing or anycast routing for resilience. Horizontal scaling for web servers and database servers can be performed by adding more instances or utilizing cloud-based auto-scaling features.

Content Delivery Network (CDN) (Optional):

Using a CDN allows one to cache static content and distribute it closer to end-users, reducing latency and offloading traffic from your servers. Integrating the CDN with the load balancer helps dynamically cache and serve content based on user location and demand.

Automation and Infrastructure as Code:

Automation tools such as Ansible, Terraform, or Kubernetes help automate deployment, configuration, and scaling of your infrastructure. Managing infrastructure as code helps maintain consistency, repeatability, and scalability. By following these principles and deploying a well-designed architecture, you can create a

high-performance web server infrastructure using HAProxy for load balancing, capable of handling large volumes of traffic with reliability and efficiency.

3.2 Load balancing Algorithms

Round Robin: In this algorithm, requests are distributed sequentially to each server in a circular manner. It is simple and ensures that each server receives an equal share of the load. However, it may not consider the server's current load or capacity, leading to uneven distribution if servers have different performance characteristics.

Least Connections: The load balancer routes incoming requests to the server with the fewest active connections. This algorithm dynamically adjusts to the current load on servers, ensuring that requests are distributed to the least loaded server. It is particularly useful when server capacities vary, as it helps to evenly distribute the load based on each server's capacity.

IP Hash: This algorithm calculates a hash value based on the client's IP address and then assigns the request to a server based on the result. Requests from the same client IP address are consistently routed to the same server, which can be useful for maintaining session persistence or caching. However, it may lead to uneven distribution if there are a small number of client IP addresses generating a large portion of the traffic. The load balancer monitors the response times of each server and directs requests to the server with the lowest response time. This algorithm aims to minimize overall response time for clients by sending requests to the fastest server. It requires continuous monitoring of server performance and may not be suitable for all environments, especially if response times fluctuate frequently.

Weighted Round Robin: Each server is assigned a weight based on its capacity or performance. Requests are then distributed to servers in a round-robin fashion, but servers with higher weights receive a proportionally larger share of requests. It allows administrators to fine-tune the load balancing behavior based on server capabilities.

3.2 Health Checking

Health checking is a critical aspect of maintaining reliability and performance of a web server infrastructure, particularly in a load-balanced environment. Health checking involves periodically monitoring the status and availability of backend servers to ensure that they are capable of handling incoming requests. Here's an overview of health checking in the context of load balancing:

Health checking ensures that only healthy servers receive traffic from the load balancer, preventing requests from being sent to servers that are offline, overloaded, or experiencing issues. By continuously monitoring server health, the load balancer can automatically detect failures and remove problematic servers from the pool of available resources, thereby maintaining overall system availability and reliability.

TCP Health Checks: The load balancer establishes a TCP connection to the backend server and checks if it is successful. This is the simplest form of health check but may not detect application-level issues.

HTTP/HTTPS Health Checks: The load balancer sends HTTP/HTTPS requests to specific endpoints on the backend server and verifies the response status code. This allows for more granular health checks, including application-specific checks such as database connectivity or API availability.

Custom Health Checks: In some cases, custom scripts or probes may be used to perform complex health checks tailored to the application's requirements. Health checks should be performed frequently enough to detect issues promptly without imposing excessive load on servers. The timeout for health checks should be set appropriately to avoid false positives (marking a server as unhealthy due to temporary network latency) or false negatives (failing to detect actual server issues).

Failover and Recovery: When a server fails a health check, the load balancer can automatically remove it from the pool of available servers, directing traffic only to healthy servers. Once a failed server recovers and passes subsequent health checks, it can be reintegrated into the pool of available servers, allowing traffic to be distributed across all healthy servers again.

3.3 Session Persistence

Session persistence, also known as session affinity or sticky sessions, is a mechanism used in load-balanced environments to ensure that requests from the same client are consistently routed to the same backend server. This is important for applications that require client-state information to be maintained between multiple requests, such as those using cookies, sessions, or user authentication tokens.

Here's an overview of session persistence:

Source IP Affinity: The load balancer uses the client's IP address to determine session affinity. Requests from the same IP address are routed to the same backend server.

Cookie-Based Affinity: The load balancer inserts a session cookie into the client's response, containing information identifying the assigned backend server. Subsequent requests from the client include the session cookie, allowing the load balancer to route them to the correct server.

Session ID Affinity: Similar to cookie-based affinity, but instead of using cookies, the load balancer examines a session ID provided by the client, typically transmitted as part of the URL or request headers.

SSL Session ID Affinity: In SSL/TLS termination scenarios, the load balancer can use SSL session IDs to maintain session persistence for encrypted connections.

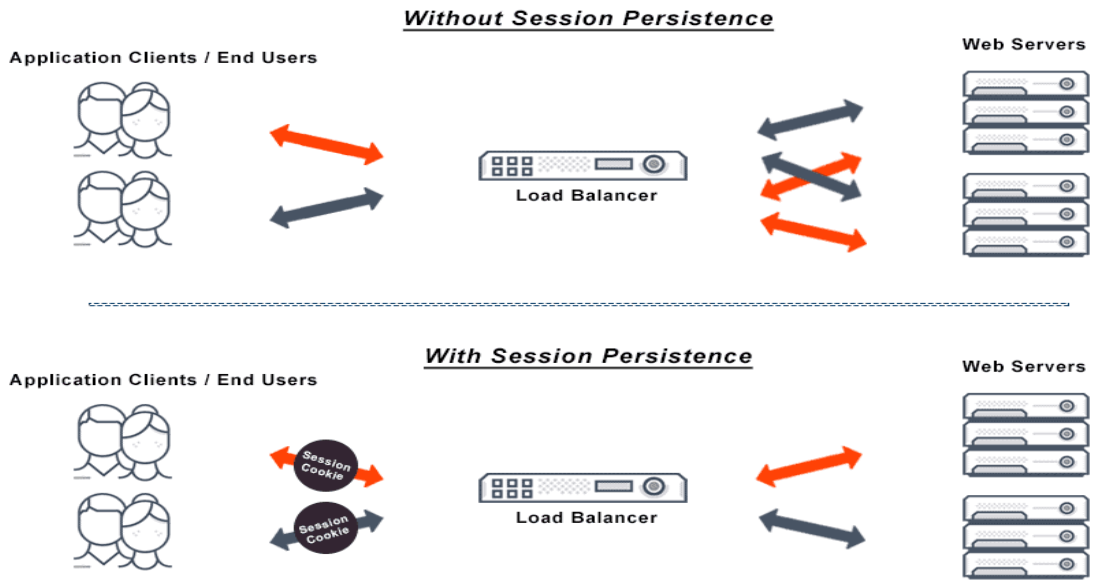


Figure 3.1: Session Persistence [8]

3.4 SSL Termination

SSL termination, also known as SSL offloading or SSL termination proxy, is a technique used to offload the SSL/TLS encryption and decryption process from the backend servers to a dedicated device or server, typically a load balancer or a reverse proxy.

Here's an overview of SSL termination.

SSL termination improves the performance and scalability of web servers by relieving them of the resource-intensive task of SSL/TLS encryption and decryption. It allows the load balancer or reverse proxy to handle SSL/TLS negotiation with clients, decrypting incoming HTTPS requests, and then forwarding unencrypted requests to backend servers. Thus, backend servers can focus on processing application logic and serving content without the overhead of SSL/TLS operations, leading to improved response times and resource utilization.

Components:

Load Balancer or Reverse Proxy: The device or server responsible for terminating SSL connections and forwarding unencrypted traffic to backend servers. This can be hardware-based or software-based, such as HAProxy, Nginx, or F5 BIG-IP.

SSL Certificate: A valid SSL/TLS certificate is required for the load balancer or reverse proxy to negotiate SSL/TLS connections with clients. This certificate is installed on the load balancer or reverse proxy.

Backend Servers: The servers that host the web applications or services. They receive unencrypted HTTP traffic from the load balancer or reverse proxy.

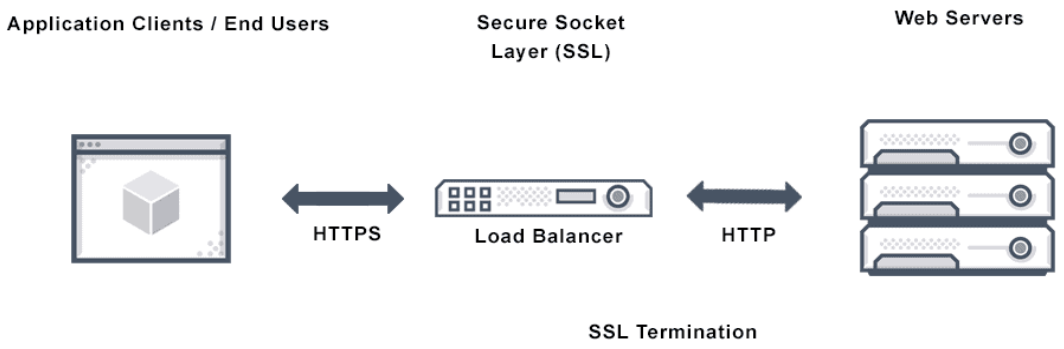


Figure 3.2: SSL Termination [6]

3.5 Logging and Monitoring

We distinguish between several types of logs.

Access Logs: These logs record details about incoming requests to the web server, including the source IP address, requested URL, response status code, user-agent, and timestamps. Access logs provide valuable information for troubleshooting, performance analysis, and auditing.

Error Logs: Error logs capture information about server errors, warnings, and exceptions encountered during request processing. They help diagnose and troubleshoot issues such as misconfigurations, application bugs, or resource exhaustion.

Security Logs: Security logs document security-related events such as failed login attempts, potential attacks (e.g., SQL injection, XSS), and unauthorized access attempts. Monitoring security logs is crucial for detecting and responding to security threats promptly.

Application Logs: For web applications, application-specific logs provide insights into application behavior, including business logic execution, database interactions, and custom events. Application logs aid in debugging, performance optimization, and business analytics.
Syslog Integration: Centralized logging solutions such as syslog enable aggregation, storage, and analysis of logs from multiple servers and applications. Syslog integration simplifies log management, enhances scalability, and facilitates compliance with regulatory requirements.

The logs can provide different metrics.

- **Server Metrics**: Monitoring server metrics such as CPU usage, memory utilization, disk I/O, network traffic, and system load provides visibility into server performance and resource utilization. Monitoring tools such as Prometheus, Nagios, or Zabbix can collect and visualize server metrics in real-time.
- **Application Metrics**: Monitoring application-specific metrics such as request throughput, response times, error rates, and database query performance helps identify performance bottlenecks, optimize resource allocation, and ensure optimal user experience.
- **Alerting**: Configuring alerting rules based on predefined thresholds or anomalies enables proactive notification of critical issues or performance degradation. Alerting mechanisms can notify administrators via email, SMS, or integration with collaboration platforms like Slack or Microsoft Teams.
- **Health Checks**: Implementing automated health checks verifies the availability and responsiveness of critical services, ensuring high availability and reliability. Health checks monitor endpoints, databases, dependencies, and external integrations to detect failures or performance degradation.
- **Log Monitoring**: Log monitoring solutions analyze log data in real-time to identify patterns, anomalies, or security incidents. Log monitoring tools such as Elasticsearch, Splunk, or Graylog offer advanced search capabilities, visualization, and correlation to facilitate troubleshooting and incident response.
- **Performance Monitoring**: Performance monitoring involves tracking key performance indicators (KPIs) such as response time, latency, throughput, and error rates. Performance monitoring tools provide insights into system performance, user experience, and capacity planning.
-

3.6 High Availability

High Availability (HA) is a crucial aspect of any web server infrastructure, ensuring continuous availability of services and minimizing downtime.

Here's how to achieve high availability in a web server environment:

Redundancy
Deploy redundant components, including servers, networking equipment, and storage systems, to eliminate single points of failure. Redundancy ensures that if one component fails, there is another available to take over seamlessly.

Load Balancing:
Use a load balancer to distribute incoming traffic across multiple servers. Load balancers monitor the health of servers and automatically route traffic away from failed or overloaded servers to healthy ones, ensuring uninterrupted service availability.

Server Clustering:
Implement server clustering or replication to maintain identical copies of data and configurations across multiple servers. This allows for failovers in case of server failures, ensuring uninterrupted service availability and data integrity.

Data Replication and Backup:
Utilize data replication mechanisms such as database replication or distributed file systems to replicate data across multiple nodes or data centers. Regularly backup data to secondary storage locations to ensure data availability in case of disasters or data corruption.

Automatic Failover:
Implement automatic failover mechanisms that detect failures and automatically switch to redundant components without manual intervention. This reduces downtime and minimizes the impact on users.

Monitoring and Alerting:
Implement robust monitoring and alerting systems to continuously monitor the health and performance of servers, network devices, and applications. Set up alerts to notify administrators of potential issues or anomalies that require attention.

Regular Maintenance and Testing:
Conduct regular maintenance activities, including software updates, patches, and hardware upgrades, to prevent failures and ensure optimal performance. Test failover procedures regularly to verify their effectiveness and identify any potential issues proactively.

Scalability:
Design the infrastructure to scale horizontally by adding more servers or resources dynamically to handle increased traffic or workload demands. Scalable architectures ensure that the system can accommodate growth without sacrificing availability or performance.

Disaster Recovery Planning:
Develop a comprehensive disaster recovery plan that outlines procedures for restoring services in the event of major disruptions or disasters. Test the disaster recovery plan periodically to validate its effectiveness and identify areas for improvement.

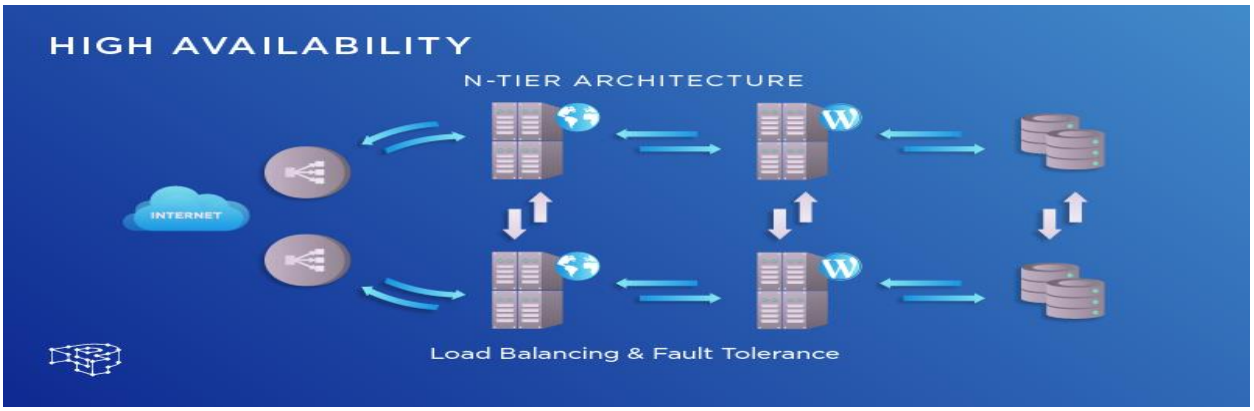


Figure 3.3: High Availability [7]

3.7 Configuration

Configuring a highly available web server infrastructure involves setting up several components, such as load balancers, web servers, databases, and networking equipment.

Here's a basic configuration guide:

Network Configuration:

- Configure network interfaces, IP addresses, and DNS settings for all servers and networking devices.
- Ensure proper routing and firewall rules to allow traffic between components and from external clients.

Load Balancer Configuration:

- Install and configure a load balancer (e.g., HAProxy, Nginx) on dedicated servers or virtual machines.
- Define backend server pools and configure health checks to monitor server availability.
- Set up load balancing algorithms, SSL termination, and session persistence as needed.
- Configure virtual IPs (VIPs) or DNS records to point to the load balancer for incoming traffic.

Web Server Configuration:

- Install and configure web server software (e.g., Apache, Nginx) on multiple servers behind the load balancer.
- Configure virtual hosts or server blocks to host multiple websites or applications.
- Enable SSL/TLS for secure communication and configure certificates.
- Optimize web server settings for performance, including connection handling, caching, and compression.
- Implement security measures such as firewalls, intrusion detection/prevention, and regular software updates.

Database Configuration:

- Set up a highly available database cluster (e.g., MySQL Cluster, PostgreSQL with streaming replication) for storing application data.
- Configure database replication, clustering, or sharding to ensure data redundancy and availability.
- Secure database access with strong authentication and authorization mechanisms.
- Tune database parameters for performance and scalability based on workload characteristics.

Testing and Validation:

- Test the entire infrastructure setup in a staging or testing environment to validate configuration changes, failover procedures, and disaster recovery plans.
- Conduct performance testing and load testing to ensure that the infrastructure can handle expected traffic volumes and workload spikes.
  Review security configurations and conduct vulnerability assessments to identify and remediate potential security risks.

4. Implementing High Availability with Multiple Load Balancers

4.1 Redundant Load Balancers

Redundant load balancers are an essential component of highly available and resilient systems, particularly in the realm of web services, cloud computing, and networking infrastructure. Load balancers distribute incoming network traffic across multiple servers or resources to ensure optimal resource utilization, minimize downtime, and enhance performance. Redundant load balancers provide an additional layer of fault tolerance by replicating load balancing functionality across multiple instances.

Fault Tolerance: Redundant load balancers mitigate the risk of single points of failure in the network architecture. If one load balancer instance experiences issues or becomes unavailable, another instance can seamlessly take over the load balancing responsibilities, thereby maintaining service availability.

Traffic Distribution Algorithms: Load balancers employ various algorithms to distribute incoming traffic among backend servers efficiently. Redundant load balancers must synchronize their configurations and session states to ensure consistent traffic distribution across all instances.

Health Monitoring: Redundant load balancers continuously monitor the health and availability of backend servers or resources. If a server becomes unhealthy or unresponsive, the load balancer can automatically remove it from the pool of available resources, thereby improving overall system reliability.

Failover Mechanisms: In active-passive configurations, redundant load balancers use failover mechanisms to ensure seamless transition of traffic from a failed or degraded instance to a healthy one. This failover process typically involves detecting failures, initiating a failover event, and redirecting traffic to the standby instance.

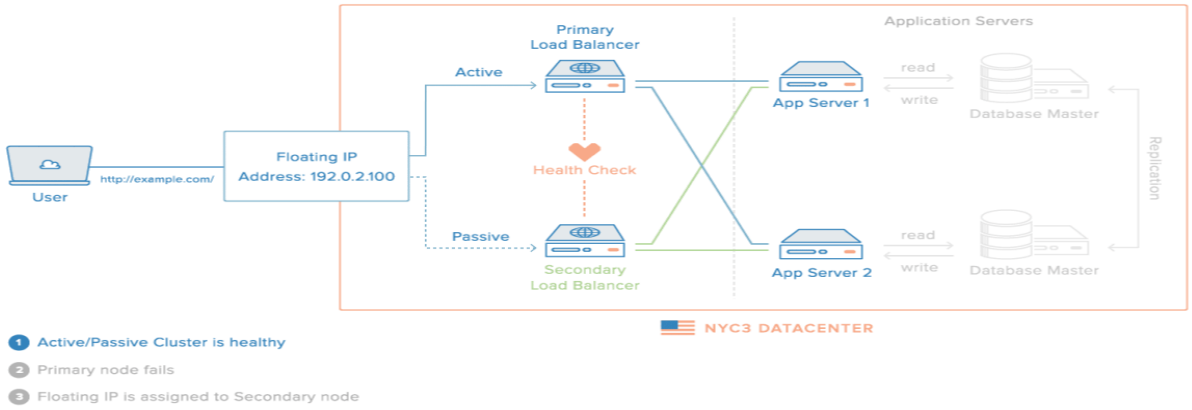


Figure 4.1: Redundant Load Balancers [7]

4.2 Sticky Sessions

Sticky sessions, also known as session affinity or session persistence, are a mechanism used in load balancing to ensure that once a client establishes a session with a server, all subsequent requests from that client are routed to the same server. This contrasts with traditional load balancing methods where each request from a client is independently routed to any available server, potentially causing issues with session management, especially in stateful applications.

Sticky sessions allow a load balancer to associate a client's session with a specific server for the duration of that session. This ensures that all subsequent requests from the same client are directed to the same server, maintaining session state and continuity.

Stateful Applications: Sticky sessions are particularly useful for applications that maintain session state on the server side, such as e-commerce platforms, webmail services, and online gaming websites. Without sticky sessions, clients might experience issues such as lost shopping carts, incomplete transactions, or disrupted gameplay due to session data being spread across multiple servers.

Session Tracking: Load balancers use various methods to implement sticky sessions, including source IP hashing, cookie-based affinity, and URL rewriting. These methods involve tracking client identifiers (such as IP addresses or session cookies) and consistently routing requests from the same client to the appropriate server.

Load Balancer Configuration: Configuring sticky sessions typically involves specifying a session timeout period, after which the association between a client and a server expires. This ensures that if a client's session becomes inactive or terminates, subsequent requests may be directed to a different server based on the load balancer's algorithm. High Availability Considerations: When deploying sticky sessions in a redundant load balancer setup, it's crucial to ensure that session state is synchronized across all load balancer instances. This allows for seamless failover and session continuity in the event of a load balancer failure or maintenance.

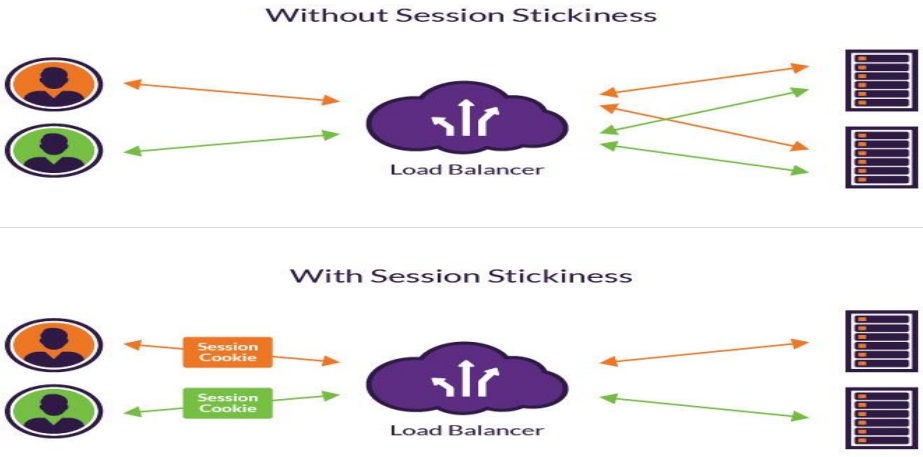


Figure 4.2: Sticky Sessions [8]

4.3 Shared State

Shared state refers to data or information that is accessible and modifiable by multiple components or processes within a system. In a computing context, shared state often refers to shared memory or shared resources that can be accessed and modified by multiple threads, processes, or instances of an application.

Concurrency: Shared state becomes relevant in multi-threaded or distributed systems where multiple components concurrently access and update the same data. In such scenarios, proper synchronization mechanisms are essential to prevent race conditions and ensure data consistency.

Shared Memory: In multi-threaded programming, shared state often involves accessing shared memory regions where variables or data structures are stored. Without proper synchronization mechanisms such as locks, mutexes, or atomic operations, concurrent access to shared memory can lead to data corruption or inconsistent behavior.

Distributed Systems: In distributed computing environments, shared state may involve shared resources or data stored across multiple nodes or servers. Distributed systems often rely on distributed databases, distributed file systems, or coordination services like Apache ZooKeeper or Consul to manage shared state across multiple nodes.

Communication Overhead: Accessing shared state across multiple components can introduce communication overhead and synchronization delays, especially in distributed systems where components may be located on different machines or network nodes. Efficient communication protocols and data replication strategies are crucial for managing shared state in distributed environments.

4.4 Failover with keepalived

Keepalived is a popular open-source software tool used for implementing high availability solutions in Linux-based systems. It primarily focuses on providing failover and load balancing for network services by monitoring the health of servers and automatically shifting traffic to healthy servers when failures occur. Here's how failover with Keepalived typically works:

- **Configuration Setup**: Keepalived is configured on multiple servers (nodes) that provide the same service. These nodes can be physical servers or virtual machines.
- **Health Checking**: Keepalived continuously monitors the health of the services running on the nodes using health checks. It can perform various types of health checks, including simple TCP checks, HTTP checks, or custom scripts to verify service availability.
- **Virtual IP (VIP)**: Keepalived assigns a virtual IP address (VIP) to the group of nodes. This VIP is used by clients to access the service. Keepalived ensures that the VIP is always available on at least one healthy node.
- **Master/Backup Election**: Among the nodes running Keepalived, one is designated as the "master" and the others as "backup" nodes. The master node actively handles client requests and manages the VIP, while the backup nodes remain in standby mode, ready to take over if the master node fails.
- **Heartbeat Mechanism**: Keepalived nodes communicate with each other using a heartbeat mechanism to detect failures and synchronize state information. If the master node becomes unresponsive or fails its health checks, the backup nodes detect this condition and initiate failover procedures.
- **Failover Handling**: When a failure is detected, one of the backup nodes is promoted to become the new master. This node takes over the VIP and begins serving client requests. The failover process is transparent to clients, as they continue to access the service using the same VIP.
- **Recovery and Resynchronization**: Once the failed node becomes healthy again, Keepalived can optionally perform a reverse failover to restore it to its original role as the master node. The system then returns to its normal operating state with the original master node serving client requests.

- **Configuration Persistence**: Keepalived maintains its configuration persistently, ensuring that failover settings and VIP assignments are preserved across system reboots and restarts.

4.5 Floating IP

A floating IP (also known as a virtual IP or VIP) is an IP address that can be dynamically assigned to different hosts or nodes within a network. It's a technique commonly used in high availability (HA) setups to provide seamless failover and load balancing for network services.

Here's how floating IPs work:

1. **Shared IP Address**: A floating IP is assigned to a group of nodes that provide the same service. This IP address acts as a single point of access for clients to reach the service.
2. **Dynamic Assignment**: Unlike a static IP address, a floating IP can be dynamically reassigned to different nodes within the group based on certain conditions, such as node availability or health status.
3. **Health Monitoring**: Typically, a monitoring system (like Keepalived, as mentioned earlier) continuously checks the health and availability of the nodes associated with the floating IP. If a node fails or becomes unresponsive, the monitoring system triggers a failover event.
4. **Failover Handling**: When a failure is detected, the floating IP is automatically reassigned to another healthy node within the group. This ensures that the service remains accessible to clients, even in the event of node failures.
5. **Load Balancing**: In addition to providing failover capabilities, floating IPs can also be used for load balancing purposes. By dynamically distributing client requests across multiple nodes, floating IPs help optimize resource utilization and improve service performance.
6. **Transparent Failover**: From the perspective of clients accessing the service, the failover process is transparent. Clients continue to connect to the same IP address (the floating IP) regardless of which node is currently hosting the service.
7. **Configuration Persistence**: Floating IP configurations are typically persisted across system reboots and restarts to ensure that failover settings and VIP assignments are maintained consistently.

4.6 Testing and Monitoring

We distinguish between different types of testing.

- **Unit Testing**: Tests individual components or modules of software in isolation.
- **Integration Testing**: Validates the interaction between different components or modules.
- **System Testing**: Tests the entire system to ensure it meets specified requirements.
- **Performance Testing**: Evaluates the speed, scalability, and responsiveness of applications under various conditions.
- **Security Testing**: Identifies vulnerabilities and weaknesses in software and infrastructure to prevent security breaches.
- **Regression Testing**: Ensures that new changes or updates do not introduce bugs or regressions into existing functionality.

There are also several types of monitoring.

- **Infrastructure Monitoring**: Tracks the health and performance of servers, networks, and other infrastructure components.
- **Application Monitoring**: Monitors the behavior and performance of software applications, including response times, error rates, and resource utilization.
- **Logging and Log Monitoring**: Collects and analyzes log data to identify issues, troubleshoot problems, and gain insights into system behavior.
- **Security Monitoring**: Detects and responds to security threats and suspicious activities within the network or application.

Monitoring systems can be configured to send alerts and notifications when predefined thresholds are exceeded or when anomalies are detected. Integration with communication channels like email, Slack, PagerDuty, or SMS enables timely incident response and resolution.

5. Experimental Work: Nginx-HAProxy

5.1 Introduction

In today's digital world, where websites and online services are crucial for businesses and individuals alike, ensuring smooth and uninterrupted access is paramount. This is where load balancing technology solutions, like Nginx and HAProxy come into play. Our project, aptly named Nginx-HAProxy, aims to explore and implement these technologies to enhance web service performance and reliability.

At its core, our project revolves around setting up a robust infrastructure using virtual machines (VMs) and configuring Nginx as the web server and HAProxy as the load balancer. The primary goal is to distribute incoming web traffic across multiple servers efficiently, ensuring optimal resource utilization and preventing any single point of failure.

To achieve this, we begin by laying the groundwork for our infrastructure, setting up networking configurations, ensuring security measures are in place, and updating all systems to their latest versions. We then proceed to install and configure Nginx on multiple VMs, each serving unique content.

Next comes the crucial part: configuring HAProxy to act as the central traffic manager, intelligently distributing requests among the Nginx servers. We explore different load balancing algorithms to understand their impact on performance and user experience.

As we progress through the tasks, we encounter various challenges, from basic configurations to more complex setups involving private and public networks, HTTPS encryption, and high availability. Each task builds upon the previous one, allowing us to incrementally enhance our infrastructure's capabilities.

By the end of the project, we not only achieve a functional Nginx-HAProxy setup but also gain valuable insights into load balancing concepts, network security practices, and system administration techniques. Our documented journey, complete with diagrams, configurations, and proof-of-concepts, serves as a valuable resource for understanding and implementing similar solutions in real-world scenarios.

In summary, Project Nginx-HAProxy is a practical exploration of load balancing technologies, aimed at improving web service reliability, performance, and scalability. Through systematic experimentation and documentation, we aim to empower individuals and organizations to harness the power of these tools for their own applications.

5.2 Basic Infrastructure

Description:

To begin, we will create a new NAT (Network Address Translation) network. This network will allow the virtual machines (VMs) to communicate with each other and with the host system. We'll assign an IP range to this network and enable DHCP (Dynamic Host Configuration Protocol) to automatically assign IP addresses to the VMs. Next, we will assign NAT network virtual Network Interface Cards (vNICs) to all three VMs. This will enable them to connect to the NAT network and communicate with each other and the host system.

One of the VMs will be designated to perform load balancing. To achieve this, we will assign a Host-Only NIC (Network Interface Card) to only this VM. This NIC will allow the load balancer VM to communicate exclusively with the host system. We will utilize the first VM as a jump host to access the other two VMs via SSH (Secure Shell). This means that we will establish SSH connections to the first VM and then SSH into the other VMs from there. To simplify the SSH connections, we can configure SSH tunnels, SSH gateways, or utilize SSH jump functionality in tools like MobaXterm. This will allow us to create SSH sessions for all systems, making it easier to manage and access the VMs securely.

To ensure security, all VMs must have SELinux enforcing and firewalld enabled and running. Each VM will be assigned a name to identify its role within the system. Before proceeding with further configurations, all three systems will be updated to ensure they have the latest security patches and software updates installed.

Both NGINX web servers (nginx-nd1 and nginx-nd2) will have NGINX installed. nginx-nd1 will serve the message "This is for nginx-nd1", while nginx-nd2 will serve the message "This is for nginx-nd2". Both servers will utilize port 80 (HTTP) for communication. Simple commands will be used to test if the NGINX servers are functioning correctly using nginx-lb1. The NGINX load balancer (nginx-lb1) will have HAProxy installed to manage load balancing. Round-robin load balancing is examined to distribute incoming requests evenly across the NGINX web servers (nginx-nd1 and nginx-nd2). This will optimize performance and ensure high availability of the web application.

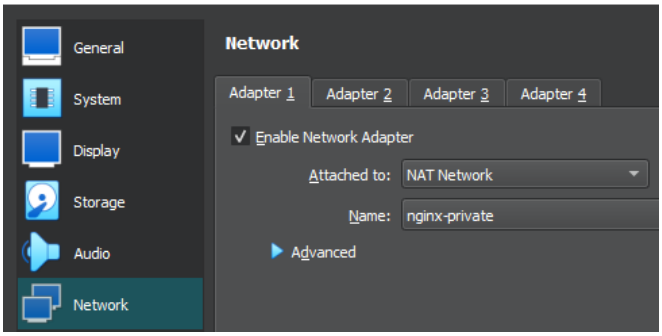
Tasks:

1.Create a new Nat Network. Assign an ip range and enable DHCP. (Oracle VM VirtualBox Manager->Tools->NatNetwork->General Options)

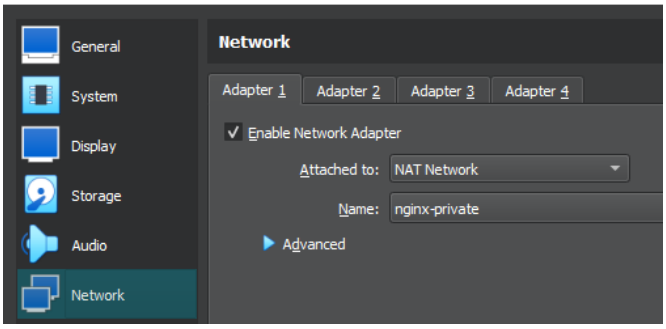


2. Assign Nat Network vNICs to all 3 VMs and host-only NIC to only the VM that will do load balancing.

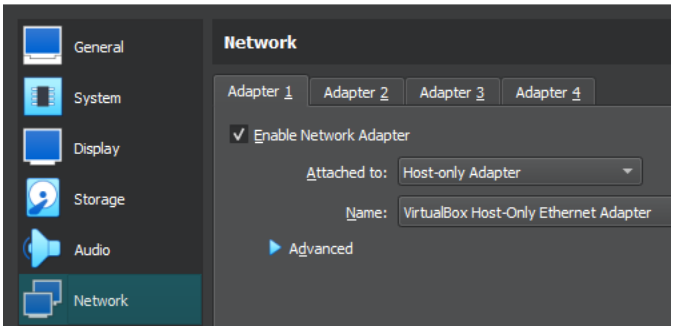
nginx-nd1 - Settings



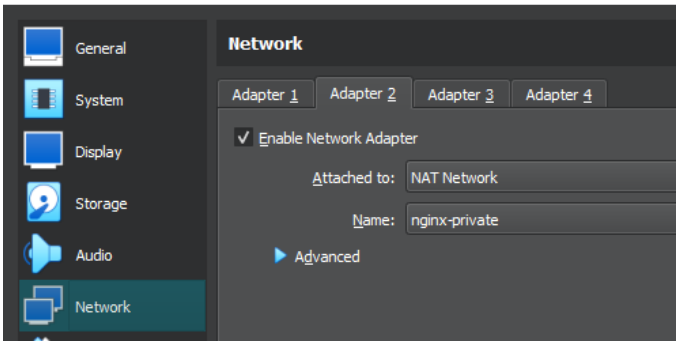
nginx-nd2 - Settings



nginx-lb1 - Settings



nginx-lb1 - Settings



3.Install and configure the NGINX Web Servers

Nginx-nd1.sh

```
# UPDATE SYSTEM
sudo dnf update -y

#Change Hostname
sudo hostnamectl set-hostname nginx-nd1

#Configure Networking - Nat Net Adapters
sudo nmcli connection modify enp0s3 ipv4.method manual ipv4.addresses 192.168.100.201/24 ipv4.gateway 192.168.100.1 ipv4.dns 8.8.8.8  #nd1

# Reload Adapters
sudo nmcli connection up enp0s3

#Reload network manager
sudo systemctl restart NetworkManager

# Confirm Network Settings
ip -c a s
ip -c r s

# Configrm Network Connectivity
ping 192.168.100.201

# Reboot to confirm and clear old configuration
sudo reboot

sudo systemctl restart NetworkManager

# Install Nginx

sudo dnf install nginx -y

# disable apache httpd
sudo systemctl stop httpd
sudo systemctl disable httpd

# allow http on firewall
sudo firewall-cmd --zone=public --add-service=http --permanent
sudo firewall-cmd --reload

# backup index.html
sudo cp /usr/share/nginx/html/index.html /usr/share/nginx/html/index.html.bak

# set index.html
echo "This is fotis $(hostname) " | sudo tee /usr/share/nginx/html/index.html

# start,enable nginx
sudo systemctl enable --now nginx
```

Nginx-nd2.sh

```
# UPDATE SYSTEM
sudo dnf update -y

#Change Hostname for each system
sudo hostnamectl set-hostname nginx-nd2

#Configure Networking - Nat Net Adapters
sudo nmcli connection modify enp0s3 ipv4.method manual ipv4.addresses 192.168.100.202/24 ipv4.gateway 192.168.100.1 ipv4.dns 8.8.8.8  #nd2

# Reload Adapters
sudo nmcli connection up enp0s3   #lb1,nd1,nd2

#Reload network manager
sudo systemctl restart NetworkManager

# Confirm Network Settings
ip -c a s
ip -c r s

# Configrm Network Connectivity
ping 192.168.100.202

# Reboot to confirm and clear old configuration
sudo reboot

# Install Nginx

sudo dnf install nginx -y

# disable apache httpd
sudo systemctl stop httpd
sudo systemctl disable httpd

# allow http on firewall
sudo firewall-cmd --zone=public --add-service=http --permanent
sudo firewall-cmd --reload

# backup index.html
sudo cp /usr/share/nginx/html/index.html /usr/share/nginx/html/index.html.bak

# set index.html
echo "This is fotis $(hostname) " | sudo tee /usr/share/nginx/html/index.html

# start,enable nginx
sudo systemctl enable --now nginx
```

4.Set and Configure the Load Balancer

```
# UPDATE SYSTEM
sudo dnf update -y

#Change Hostname for each system
sudo hostnamectl set-hostname nginx-lb1

#Configure Networking - Nat Net Adapters
sudo nmcli connection modify "Wired connection 1" ipv4.method manual ipv4.addresses 192.168.100.101/24
ipv4.gateway 192.168.100.1  ipv4.dns 8.8.8.8  #lb1

#Configure Networking - Host Only Adapters
sudo nmcli connection modify enp0s3 ipv4.method manual ipv4.addresses 192.168.56.99/24

# Reload Adapters
sudo nmcli connection up "Wired connection 1"   #lb1  (nat)     or "enp0s8(depends)"
sudo nmcli connection up enp0s3   #lb1,nd1,nd2

#Reload network manager
sudo systemctl restart NetworkManager

# Confirm Network Settings
ip -c a s
ip -c r s

# Configrm Network Connectivity
ping 192.168.100.101

# Reboot to confirm and clear old configuration
sudo reboot
```

```
# disable apache httpd
sudo systemctl stop httpd
sudo systemctl disable httpd

# allow http on firewall
sudo firewall-cmd --zone=public --add-service=http --permanent
sudo firewall-cmd --reload

# Install HAProxy on VM1, the jump host: You can use a load balancer such as HAProxy to distribute incoming requests to VM2 and VM3.
sudo dnf install haproxy -y

# Edit the HAProxy configuration file: (round-robin load balancing setup)
cd /etc/haproxy/
sudo chmod 777 haproxy.cfg
vi haproxy.cfg


frontend http-in
    bind *:80
    mode http
    default_backend nginx-backend

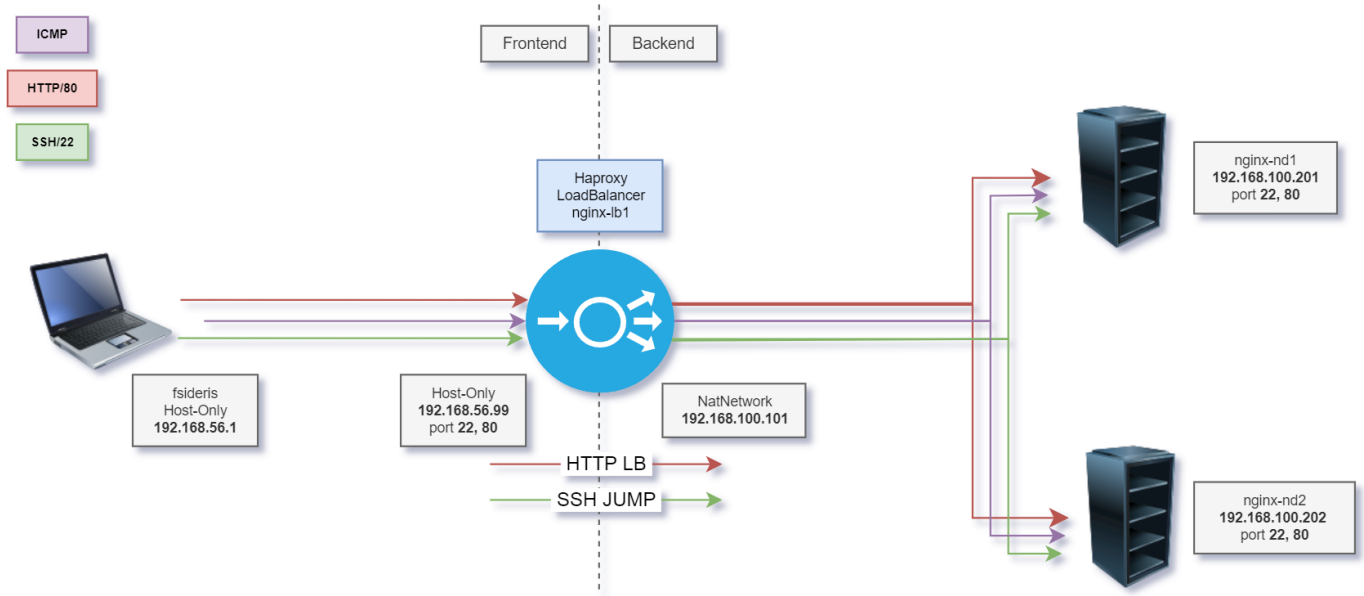
backend nginx-backend
    balance roundrobin
    server nginx-nd1 192.168.100.201:80 check
    server nginx-nd2 192.168.100.202:80 check


sudo systemctl restart haproxy
sudo systemctl status haproxy
```

Proof of Concept:

```
[fsideris@nginx-lb1 ~]$ ip -c a s
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: enp0s3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP group default qlen 1000
    link/ether 08:00:27:e2:7a:5f brd ff:ff:ff:ff:ff:ff
    inet 192.168.56.99/24 brd 192.168.56.255 scope global noprefixroute enp0s3
       valid_lft forever preferred_lft forever
    inet6 fe80::a00:27ff:fee2:7a5f/64 scope link noprefixroute
       valid_lft forever preferred_lft forever
3: enp0s8: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP group default qlen 1000
    link/ether 08:00:27:fd:d1:4d brd ff:ff:ff:ff:ff:ff
    inet 192.168.100.101/24 brd 192.168.100.255 scope global noprefixroute enp0s8
       valid_lft forever preferred_lft forever
    inet6 fe80::2db0:a2fd:58af:67eb/64 scope link noprefixroute
       valid_lft forever preferred_lft forever
[fsideris@nginx-lb1 ~]$ curl localhost
This is fotis nginx-nd2
[fsideris@nginx-lb1 ~]$ curl localhost
This is fotis nginx-nd1
[fsideris@nginx-lb1 ~]$ curl 192.168.56.99
This is fotis nginx-nd2
[fsideris@nginx-lb1 ~]$ curl 192.168.56.99
This is fotis nginx-nd1
[fsideris@nginx-lb1 ~]$ curl 192.168.100.101
This is fotis nginx-nd2
[fsideris@nginx-lb1 ~]$ curl 192.168.100.101
This is fotis nginx-nd1
[fsideris@nginx-lb1 ~]$
```

Overview Diagram:



5.2 Assign LoadBalancer to listen only to the host-only adapter IP

Description:

This task focuses on *setting up a basic infrastructure for load balancing using a host-only adapter*. The primary objective is to configure the load balancer (nginx-lb1) to listen exclusively on the IP address provided by the host-only adapter, ensuring that it is accessible only within the virtual environment. This isolation helps in testing and debugging the load balancer's functionality before exposing it to external networks.

Once the load balancer is configured, the next step involves confirming and fine-tuning its operation. This includes verifying that requests from the host/laptop browser are correctly directed to the load balancer's host-only IP address and port 80. By setting up a continuous loop, it becomes possible to monitor the load balancer's behavior and ensure that it serves content from both nginx-nd1 and nginx-nd2 web servers.

Additionally, it's essential to test the load balancer's failover capability. By shutting down one of the nginx web servers (e.g., nginx-nd2), the load balancer should continue to serve content from the operational web server (e.g., nginx-nd1) without errors. This test helps validate the resilience of the load balancing setup and ensures uninterrupted service even in the event of node failures.

Tasks:

1.Configure HAProxy to listen only the host-only adapter ip(nginx-lb1.sh)

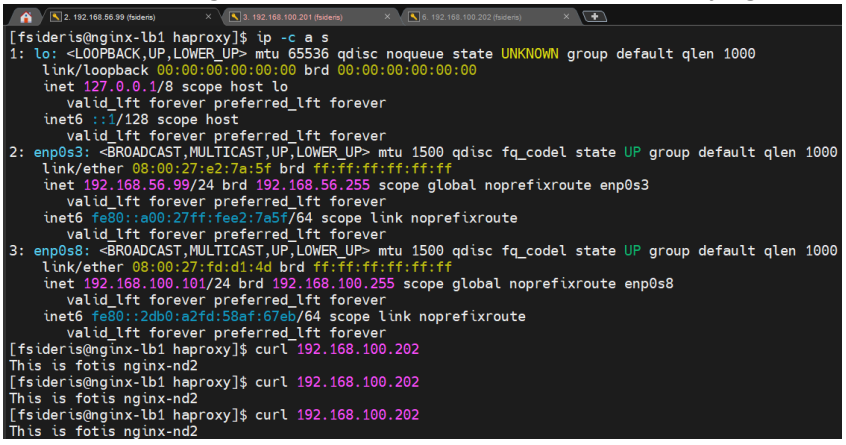
```
#VI haproxy.cfg

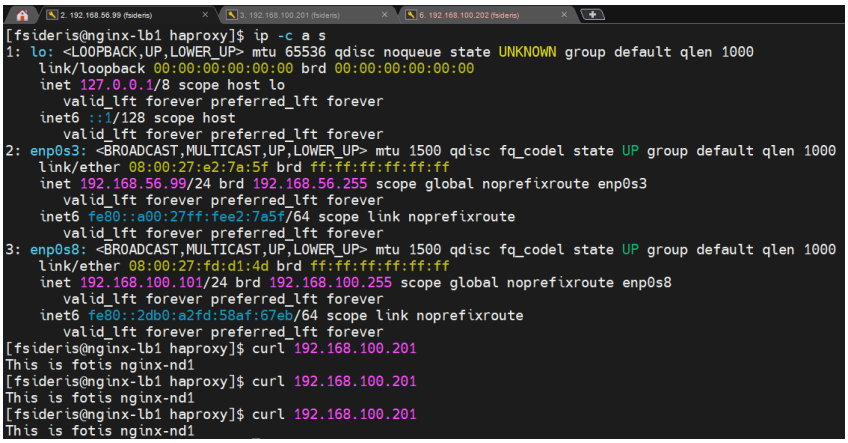
frontend http-in
    bind 192.168.56.99:80          #bind to the host only adapter
    mode http
    default_backend nginx-backend

backend nginx-backend
    balance roundrobin
    server nginx-nd1 192.168.100.201:80 check
    server nginx-nd2 192.168.100.202:80 check


sudo systemctl restart haproxy
sudo systemctl status haproxy
```

POC: Confirm if nginx-nd2 is shutdown then the LB serves only nginx-nd1 without errors (and the reverse)





5.3 LoadBalancer with port-forwarding

Description:

The objective is to *enhance the accessibility of the load balancer (nginx-lb1) by configuring it to listen on the IP address provided by the NAT network (NatNetwork) and enabling port forwarding* to facilitate external access. This task builds upon the groundwork laid in Task 1, where the load balancer was initially set up to listen only on the host-only adapter IP.

To begin, the network configuration is adjusted to designate the load balancer to listen exclusively on the IP address allocated by the NAT network. This configuration change broadens the scope of accessibility, allowing devices within the NAT network to interact with the load balancer. Moreover, it sets the stage for enabling external access to the load balancer from the host system or other external devices. Port forwarding is then implemented to establish a communication pathway between the host machine and the load balancer hosted within the virtual environment. Port forwarding involves the redirection of network traffic from a specific port on the host machine to the corresponding port on the load balancer within the NAT network.

Tasks:

1.Assign LB to listen to only the NatNetwork IP **(**nginx-lb1.sh**)**

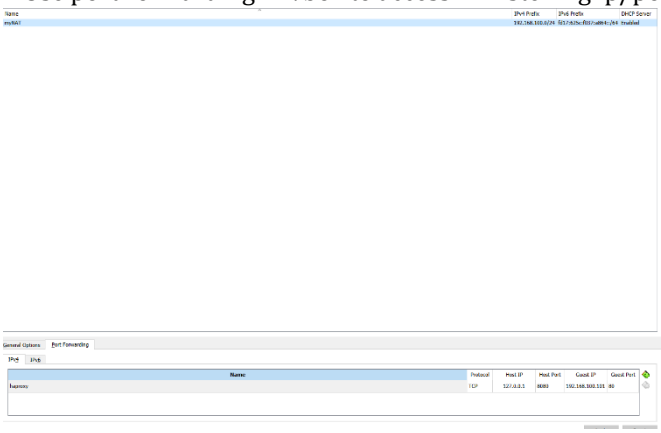
```
#VI haproxy.cfg

frontend http-in
    bind 192.168.100.101:80   # assign haproxy LB to listen on the NatNetwork ip , accept incoming connections on the IP address 192.168.100.101,
    mode http
    default_backend nginx-backend

backend nginx-backend
    balance roundrobin
    server nginx-nd1 192.168.100.201:80 check
    server nginx-nd2 192.168.100.202:80 check


sudo systemctl restart haproxy
sudo systemctl status haproxy
```

2.Use port-forwarding in vbox to access LB listening ip/port via localhost



Proof Of Concept: Confirm operation via port-forwarding, on public listening, from Host Browser

Command Prompt

```
Microsoft Windows [Version 10.0.19045.3570]
(c) Microsoft Corporation. All rights reserved.

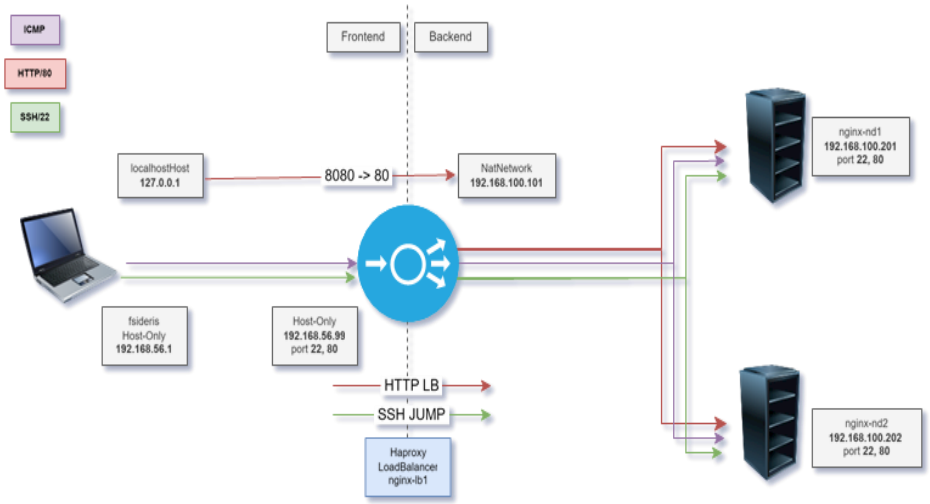
C:\Users\fsider>curl 127.0.0.1:8080
This is fotis nginx-nd1

C:\Users\fsider>curl 127.0.0.1:8080
This is fotis nginx-nd2

C:\Users\fsider>curl 127.0.0.1:8080
This is fotis nginx-nd1

C:\Users\fsider>curl 127.0.0.1:8080
This is fotis nginx-nd2
```

Overview Diagram:



5.4 Add another VM nginx-nd3

nginx-nd3.sh

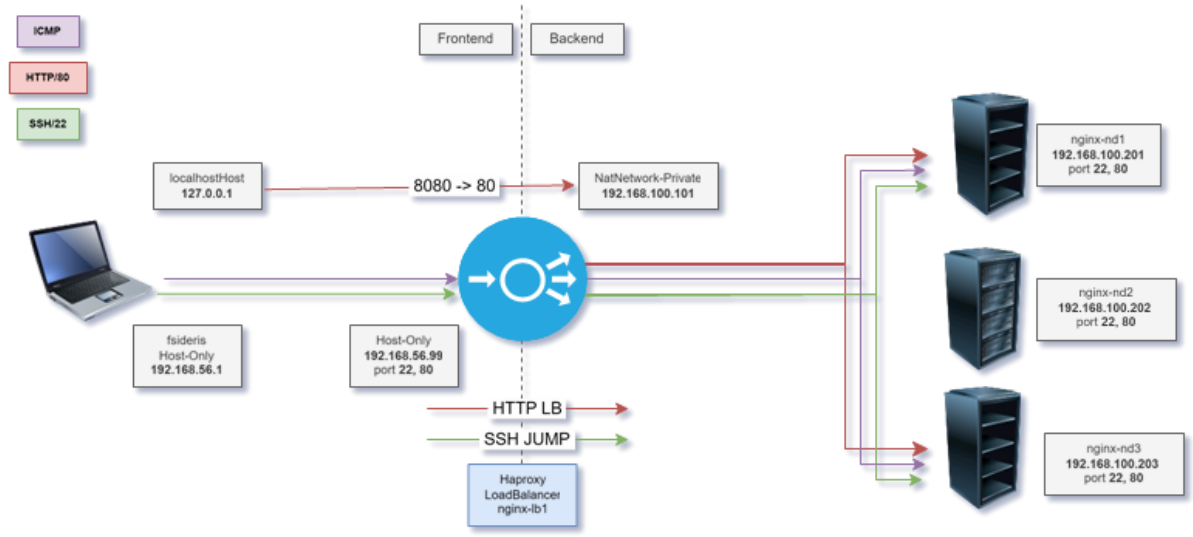
```
#Change Hostname for each system
sudo hostnamectl set-hostname nginx-nd3

#Configure Networking - Nat Net Adapters
sudo nmcli connection modify enp0s3 ipv4.method manual ipv4.addresses 192.168.100.203/24

# Reload Adapters
sudo nmcli connection up enp0s3

#Reload network manager
sudo systemctl restart NetworkManager

# set index.html
echo "This is fotis $(hostname) " | sudo tee /usr/share/nginx/html/index.html
```



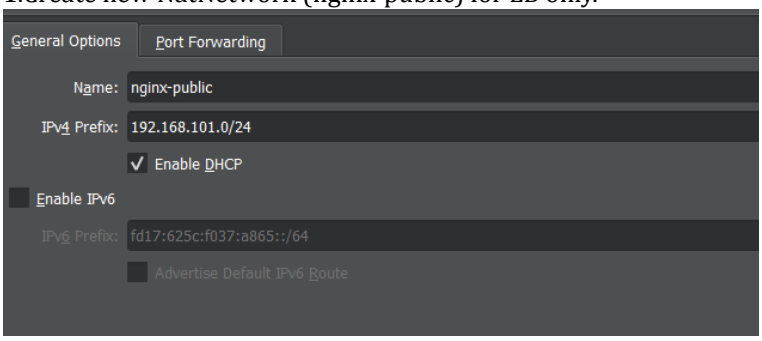
5.5 Private and Public NAT Network

Description:

Creating a more complex infrastructure with private and public NAT networks, as well as *implementing load balancing and isolation between different components*. Aims to expand the infrastructure by introducing additional network layers, implementing load balancing techniques, and ensuring the overall reliability and performance of the system.

Tasks:

1.Create new NatNetwork (nginx-public) for LB only.



2. Assign LB to listen to only the Nat-Network-Public IP (**nginx-lb1.sh**)

```
#Configure Networking - Nat Net Adapters
sudo nmcli connection modify "Wired connection 2" ipv4.method manual ipv4.addresses 192.168.101.9/24

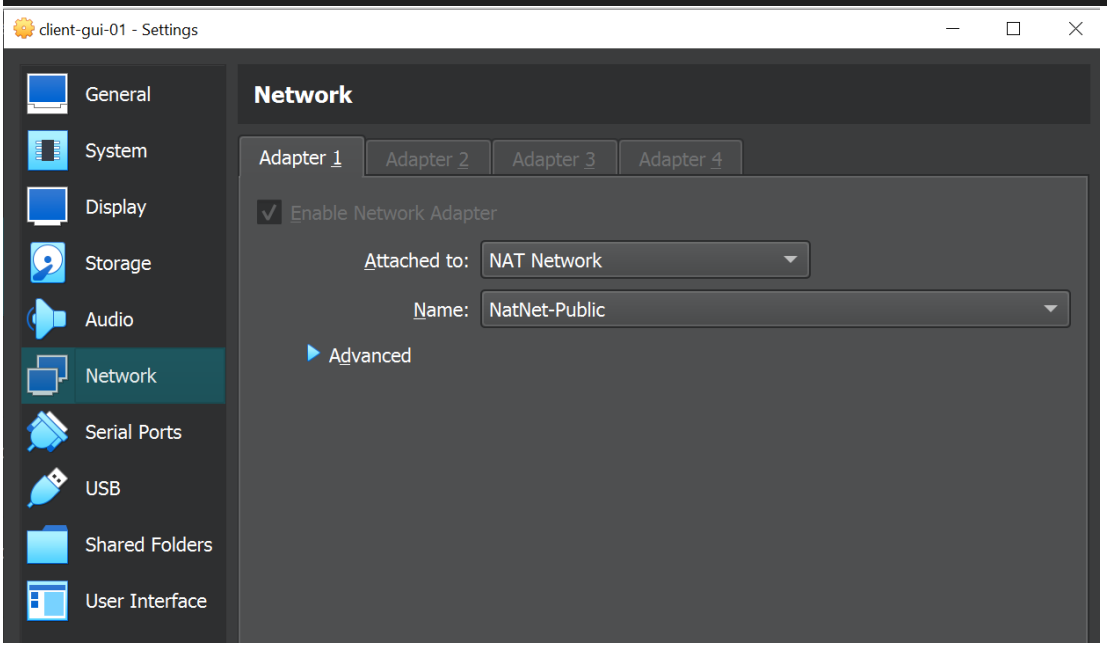

# Reload Adapters
sudo nmcli connection up "Wired connection 2"   #lb1  (nat2) or "enp0s9(depends)"

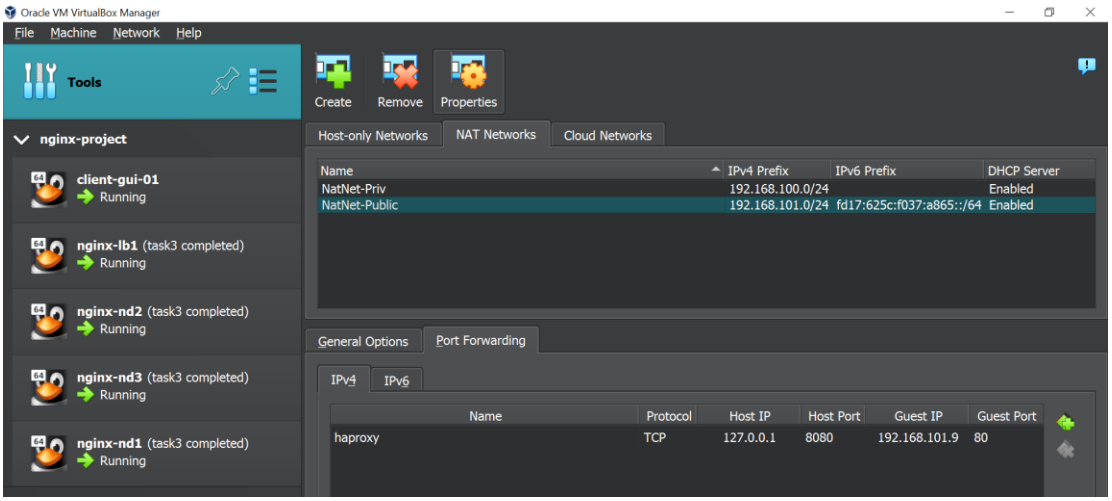

#Reload network manager
sudo systemctl restart NetworkManager


#change the configuration
vi /etc/haproxyhaproxy.cfg
#frontend
bind 192.168.101.9:80    #listen at this ip
#backend
server client-gui-01 192.168.101.10:80 check     #serve at this ip
server nginx-nd1 192.168.100.201:80 check
server nginx-nd2 192.168.100.202:80 check
server nginx-nd2 192.168.100.203:80 check
```

3. Add a new GUI VM (e.g., OL9/Xubuntu/Mint/) in NatNetwork-Public with name: client-gui-01. (**Client-gui-01.sh)**

```
sudo nmcli connection modify enp0s3 ipv4.method manual ipv4.addresses 192.168.101.10/24 ipv4.gateway 192.168.101.5  ipv4.dns 8.8.8.8
sudo nmcli connection up enp0s3   #nat
sudo systemctl restart NetworkManager
```



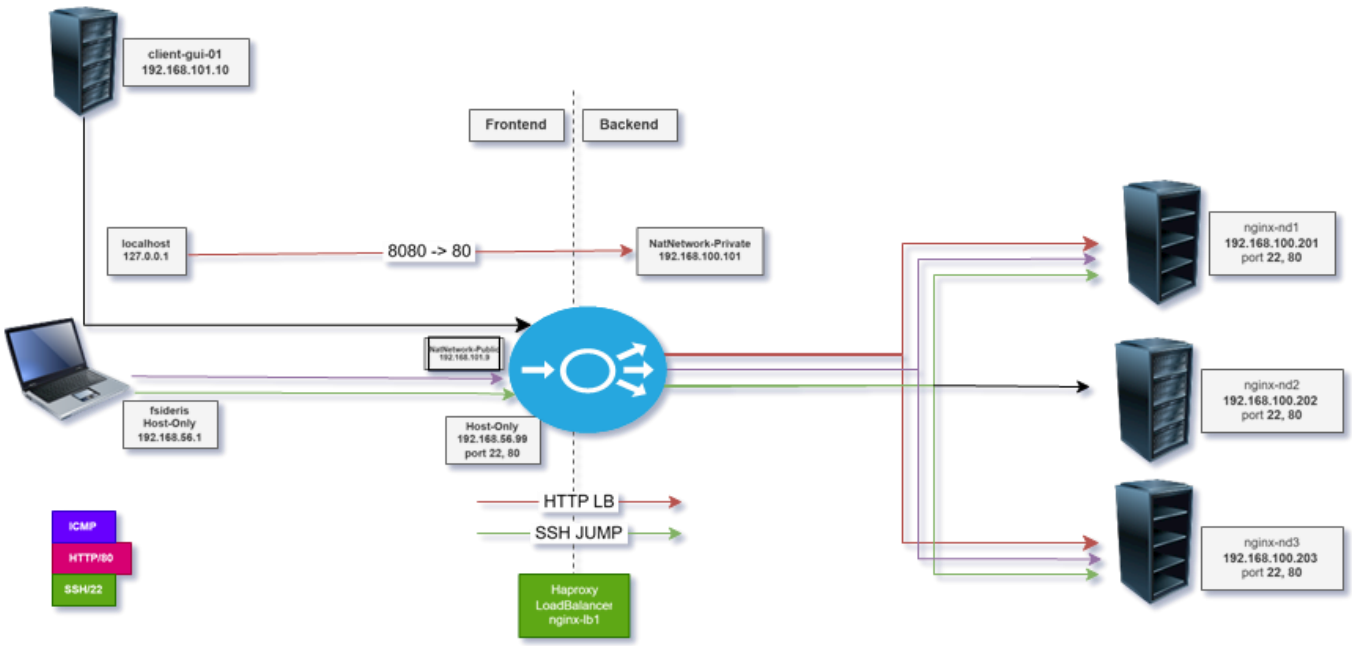


Proof Of Concept: Confirm load balancing and isolation from Private Network.

```
[fsideris@nginx-lb1 ~]$ ip -c a s
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group defaul
t qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: enp0s3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP gr
oup default qlen 1000
    link/ether 08:00:27:e2:7a:5f brd ff:ff:ff:ff:ff:ff
    inet 192.168.56.99/24 brd 192.168.56.255 scope global noprefixroute enp0s3
       valid_lft forever preferred_lft forever
    inet6 fe80::a00:27ff:fee2:7a5f/64 scope link noprefixroute
       valid_lft forever preferred_lft forever
3: enp0s8: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP gr
oup default qlen 1000
    link/ether 08:00:27:fd:d1:4d brd ff:ff:ff:ff:ff:ff
    inet 192.168.100.101/24 brd 192.168.100.255 scope global noprefixroute enp0s
8
       valid_lft forever preferred_lft forever
    inet6 fe80::2db0:a2fd:58af:67eb/64 scope link noprefixroute
       valid_lft forever preferred_lft forever
4: enp0s9: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP gr
oup default qlen 1000
    link/ether 08:00:27:6b:8a:40 brd ff:ff:ff:ff:ff:ff
    inet 192.168.101.9/24 brd 192.168.101.255 scope global noprefixroute enp0s9
       valid_lft forever preferred_lft forever
    inet6 fe80::95f1:717c:d30f:a9d4/64 scope link noprefixroute
       valid_lft forever preferred_lft forever
[fsideris@nginx-lb1 ~]$ curl 192.168.101.9
This is fotis nginx-nd1
[fsideris@nginx-lb1 ~]$ curl 192.168.101.9
This is fotis nginx-nd3
[fsideris@nginx-lb1 ~]$ curl 192.168.101.9
This is fotis nginx-nd2
```

```
[fsideris@client-gui-01 ~]$ ip -c a s
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group defau
lt qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: enp0s3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP g
roup default qlen 1000
    link/ether 08:00:27:6c:07:05 brd ff:ff:ff:ff:ff:ff
    inet 192.168.101.10/24 brd 192.168.101.255 scope global noprefixroute enp0s
3
       valid_lft forever preferred_lft forever
    inet6 fe80::a00:27ff:fe6c:705/64 scope link noprefixroute
       valid_lft forever preferred_lft forever
[fsideris@client-gui-01 ~]$ curl 192.168.101.9
This is fotis nginx-nd1
[fsideris@client-gui-01 ~]$ curl 192.168.101.9
This is fotis nginx-nd3
[fsideris@client-gui-01 ~]$ curl 192.168.101.9
This is fotis nginx-nd2
[fsideris@client-gui-01 ~]$
```

Overview Diagram:



5.6 Implementing HTTPS in Load Balancer Configuration

Description:

An enhancement of the infrastructure setup and configuration tasks, focusing on security and functionality of the load balancer (LB) by implementing HTTPS for external traffic while maintaining HTTP for internal communication. The primary goal is to configure the LB to serve web content over HTTPS to external clients while still using HTTP for communication between the LB and the backend web servers. This ensures that data transmitted over the public internet is encrypted and secure, mitigating risks associated with data interception and manipulation.

Tasks:

1.Configuration of HTTPS (The LB needs to be configured to handle HTTPS requests. This involves generating or obtaining a self-signed SSL/TLS certificate to encrypt the communication between the client and the LB. Additionally, the LB's web server software (such as Nginx) needs to be configured to listen for HTTPS requests on port 443.) **Nginx-lb1.sh**

```
# assign your certificate
openssl req -x509 -noenc -newkey rsa:4096 -keyout /etc/ssl/certs/haproxy.pem -out  /etc/ssl/certs/haproxy.pem -days 365

vi /etc/haproxy/haproxy.cfg

frontend http_front
    bind 192.168.101.9:80
    mode http
    default_backend nginx-backend-http

frontend https_front
    bind 192.168.101.9:443  ssl crt /etc/ssl/certs/haproxy.pem
    mode http
    default_backend nginx-backend-http


backend nginx-backend-http
    balance     roundrobin
    server nginx-nd1 192.168.100.201:80 check
    server nginx-nd1 192.168.100.202:80 check
    server nginx-nd1 192.168.100.203:80 check

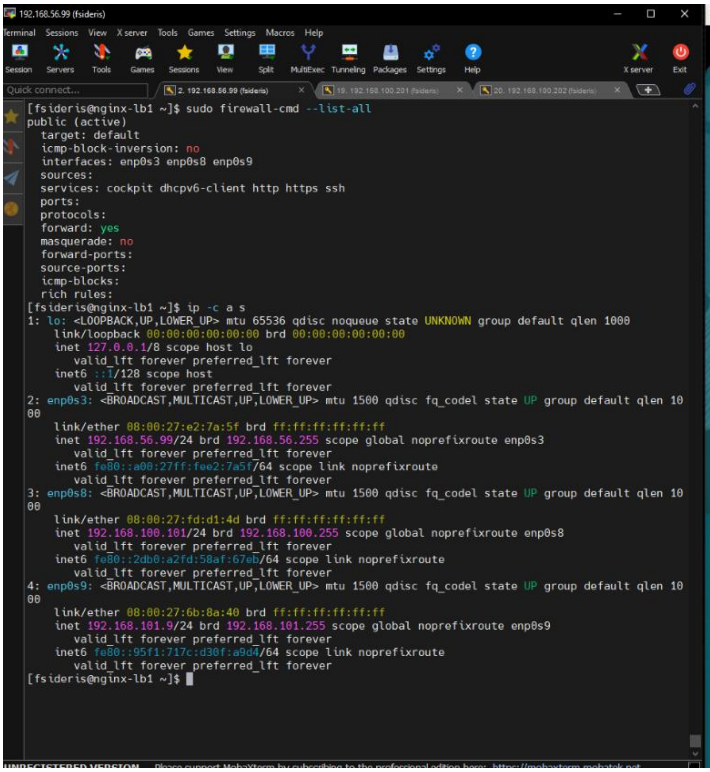

sudo haproxy -f /etc/haproxy/haproxy.cfg
sudo systemctl restart haproxy
sudo firewall-cmd --add-service=https  #firewall rules
curl https://192.168.101.9 -k
curl https://192.168.101.9 -kI  #show cookie
sudo firewall-cmd --list-all # list firewall rules
```

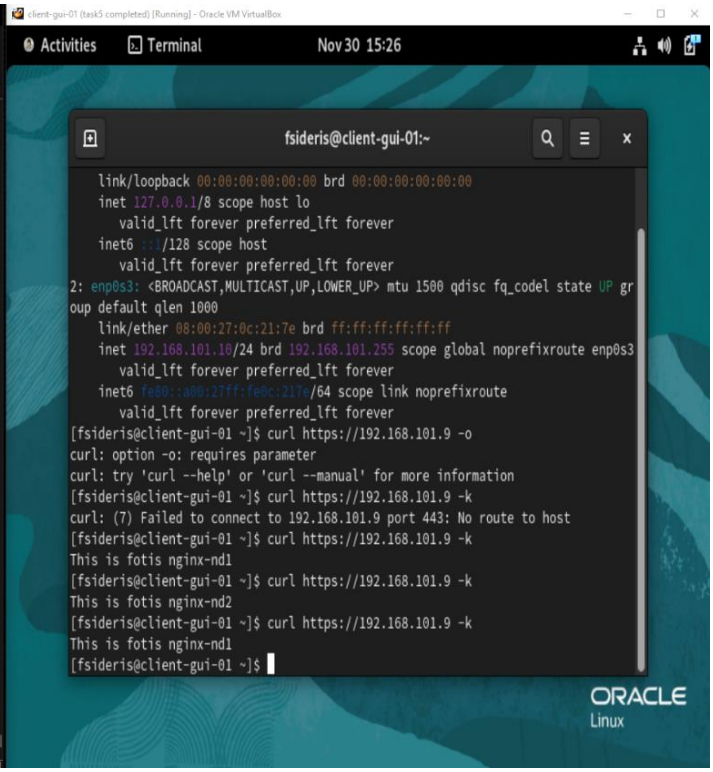
```
#Two frontends are defined: http_front and https_front.
# The http_front frontend listens on port 80
#https_front frontend listens on port 443 with SSL termination usingyour SSL certificate (your_certificate.pem).

#The http_back and https_back backends are defined.
#The balance roundrobin option ensures that requests are distributed in a round-robin fashion to the backend servers.
#Replace the IP addresses and ports with the actual addresses of your backend servers.

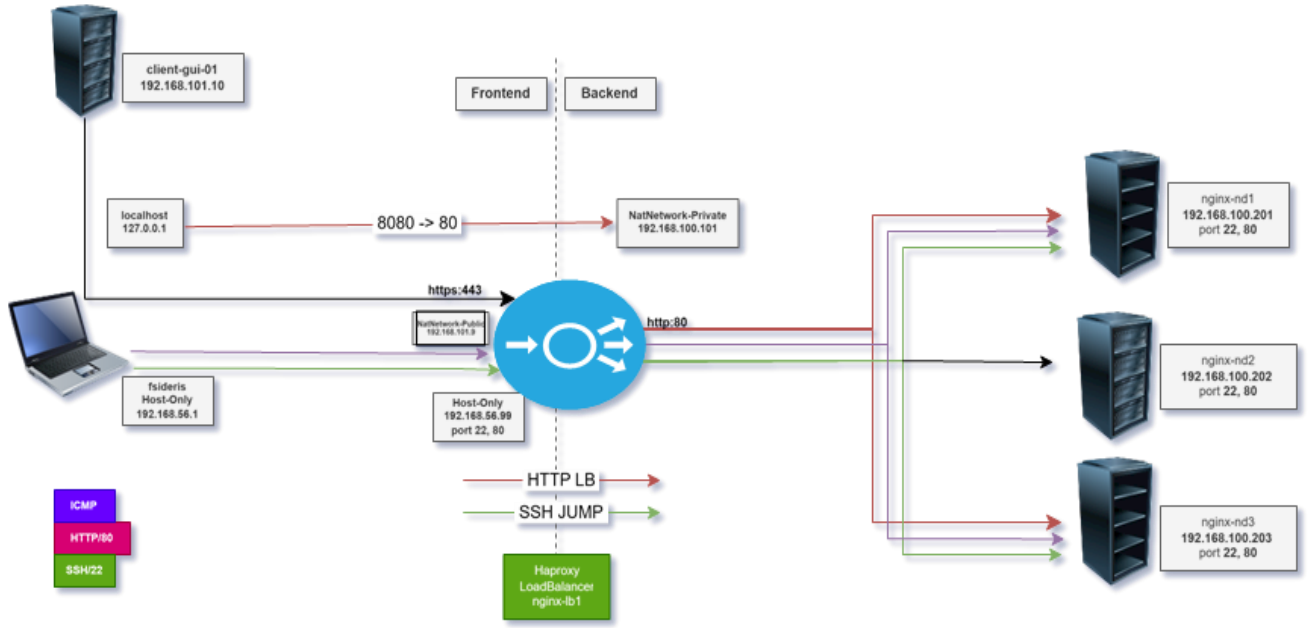
#For HTTPS traffic, you should set check ssl verify none to disable SSL certificate verification for the
#backend servers since HAProxy will terminate the SSL connection.
```

Proof Of Concept: (External clients can access the web servers securely via HTTPS and that internal communication between the LB and the backend servers remains unaffected).





Overview Diagram:



5.7 Implementing High Availability in Load Balancer

Description:

This task represents a significant advancement by *introducing high availability to the load balancer (LB) configuration* in the infrastructure setup. The primary objective is to ensure uninterrupted service by deploying multiple Load Balancer instances and configuring them to operate in a highly available manner. To begin, new LB VMs, such as nginx-lb2 and nginx-lb3, are created and connected to both the private and public networks. This allows each LB instance to communicate with the backend web servers and handle incoming client requests.

High availability is achieved by configuring the LB instances to work in tandem, ensuring that if one LB instance fails, others can seamlessly take over its responsibilities. Various methods can be employed to implement high availability, such as *active-passive or active-active configurations*. In an active-passive setup, one LB instance serves traffic while the others remain on standby. If the active LB instance fails, a standby instance automatically takes over. Conversely, in an active-active setup, all LB instances actively serve traffic simultaneously, distributing the load evenly among them.

To facilitate failover and ensure continuous service, a floating IP address can be utilized. This IP address dynamically moves between the LB instances, allowing clients to connect to the active LB instance regardless of its physical location. This ensures uninterrupted service even in the event of LB instance failures. Testing the high availability configuration involves simulating failures by closing individual LB instances and verifying that traffic is seamlessly redirected to the remaining instances. Additionally, the use of only the floating IP address in the public network ensures that clients always connect to the active LB instance, further enhancing the reliability of the infrastructure.

Tasks:

1. Create nginx-lb2(install and configure Keepalived and HAProxy)

```
#CLONE nginx-lb1

#Change Hostname for each system
sudo hostnamectl set-hostname nginx-lb2

#Configure Networking - Nat Net Adapters
sudo nmcli connection modify enp0s3 ipv4.method manual ipv4.addresses 192.168.56.98/24
sudo nmcli connection modify "Wired connection 1" ipv4.method manual ipv4.addresses 192.168.100.102/24 ipv4.gateway 192.168.100.1 ipv4.dns 8.8.8.8
sudo nmcli connection modify "Wired connection 2" ipv4.method manual ipv4.addresses 192.168.101.10/24 ipv4.gateway 192.168.101.1 ipv4.dns 8.8.8.8

# Reload Adapters
sudo nmcli connection up enp0s3
sudo nmcli connection up "Wired connection 1"
sudo nmcli connection up "Wired connection 2"   #lb1  (nat2)

#Reload network managers
sudo systemctl restart NetworkManager
```

```
#Install the keepalived package on each server:
sudo dnf install keepalived

#Before you can set up load balancing by using Keepalived, you must install and configure the feature

/etc/sysctl.conf:
net.ipv4.ip_forward = 1
Verify that the IP forwarding has been applied:
sudo sysctl -p


#Enable and start the keepalived service on each server:
sudo systemctl enable --now keepalived
#If you change the Keepalived configuration, reload the keepalived service:
sudo systemctl reload keepalived


sudo nmcli con mod "Wired connection 1" connection.id "net-private"
sudo nmcli con mod "Wired connection 2" connection.id "net-public"
sudo nmcli con mod net-private ipv6.method disabled
sudo nmcli con mod net-public ipv6.method disabled
sudo nmcli con mod enp0s3 ipv6.method disabled
sudo systemctl restart NetworkManager


#Add firewall rules to accept VRRP communication by using the multicast IP address
#224.0.0.18 and the VRRP protocol (112) on each network interface that Keepalived
#controls, for example:

sudo firewall-cmd --direct --permanent --add-rule ipv4 filter INPUT 0 \
 --in-interface enp0s9 --destination 224.0.0.18 --protocol vrrp -j ACCEPT
sudo firewall-cmd --direct --permanent --add-rule ipv4 filter OUTPUT 0 \
 --out-interface enp0s9 --destination 224.0.0.18 --protocol vrrp -j ACCEPT
sudo firewall-cmd --reload
```

```
# HAPROXY CONFIGURATION

frontend http_front
    bind 192.168.101.13:80
    http-request track-sc0 src nginx-backend-http    # track haproxy3 stick table  sc=stickycounter
    mode http
    default_backend nginx-backend-http

frontend https_front
    bind 192.168.101.13:443 ssl crt /etc/ssl/certs/haproxy.pem
    http-request track-sc0 src nginx-backend-http    # track haproxy3 stick table  sc=stickycounter
    mode http
    default_backend nginx-backend-http


backend nginx-backend-http
     stick-table type ip size 1m expire 10s store http_req_rate(10s)      #collect and store data from haproxy1 for the next one
     mode http
     balance     roundrobin
     option httpchk
     option forwardfor
     cookie nginx-lb1 insert indirect nocache
     server  nginx-nd1 192.168.100.201:80 check cookie nginx-nd1
     server  nginx-nd2 192.168.100.202:80 check cookie nginx-nd2
     server  nginx-nd3 192.168.100.203:80 check cookie nginx-nd3
```

```
### KEEPALIVED CONFIGURATION ###


vrrp_script chk_haproxy {
  script "killall -0 haproxy"                          # check the haproxy process
  interval 2                                           # every 2 seconds
  weight 2                                             # add 2 points if OK
}
vrrp_instance VI_1 {
    state BACKUP
    virtual_router_id 51
    priority 100                                  # PRIORITY SHOULD BE LOWER THAN MASTER VALUE
    advert_int 1
    authentication {
        auth_type PASS
        auth_pass 1111
    }
    unicast_src_ip 192.168.100.102
    unicast_peer {
        192.168.100.101 dev enp0s8
        192.168.100.103 dev enp0s8
   }
    virtual_ipaddress {
        192.168.101.13/24 dev enp0s9
    }
track_script {
    chk_haproxy
  }
}
```

2. Create nginx-lb3 (Install and configure Keepalived and HAproxy)

```
#CLONE nginx-lb1

#Change Hostname for each system
sudo hostnamectl set-hostname nginx-lb3

#Configure Networking - Nat Net Adapters
sudo nmcli connection modify enp0s3 ipv4.method manual ipv4.addresses 192.168.56.97/24
sudo nmcli connection modify "Wired connection 1" ipv4.method manual ipv4.addresses 192.168.100.103/24 ipv4.gateway 192.168.100.1 ipv4.dns 8.8.8.8
sudo nmcli connection modify "Wired connection 2" ipv4.method manual ipv4.addresses 192.168.101.11/24 ipv4.gateway 192.168.101.1 ipv4.dns 8.8.8.8

# Reload Adapters
sudo nmcli connection up enp0s3
sudo nmcli connection up "Wired connection 1"
sudo nmcli connection up "Wired connection 2"   #lb1  (nat2)

#Reload network managers
sudo systemctl restart NetworkManager
```

```
#Install the keepalived package on each server:
sudo dnf install keepalived

Before you can set up load balancing by using Keepalived, you must install and configure the feature.

/etc/sysctl.conf:
net.ipv4.ip_forward = 1
Verify that the IP forwarding has been applied:
sudo sysctl -p


#Enable and start the keepalived service on each server:
sudo systemctl enable --now keepalived
#If you change the Keepalived configuration, reload the keepalived service:
sudo systemctl reload keepalived

sudo nmcli con mod "Wired connection 1" connection.id "net-private"
sudo nmcli con mod "Wired connection 2" connection.id "net-public"
sudo nmcli con mod net-private ipv6.method disabled
sudo nmcli con mod net-public ipv6.method disabled
sudo nmcli con mod enp0s3 ipv6.method disabled
sudo systemctl restart NetworkManager

#Add firewall rules to accept VRRP communication by using the multicast IP address
#224.0.0.18 and the VRRP protocol (112) on each network interface that Keepalived
#controls, for example:

sudo firewall-cmd --direct --permanent --add-rule ipv4 filter INPUT 0 \
 --in-interface enp0s9 --destination 224.0.0.18 --protocol vrrp -j ACCEPT
sudo firewall-cmd --direct --permanent --add-rule ipv4 filter OUTPUT 0 \
 --out-interface enp0s9 --destination 224.0.0.18 --protocol vrrp -j ACCEPT
sudo firewall-cmd --reload
```

```
# HAPROXY CONFIGURATION

frontend http_front
    bind 192.168.101.13:80
    http-request track-sc0 src nginx-backend-http    # track haproxy3 stick table  sc=stickycounter
    mode http
    default_backend nginx-backend-http

frontend https_front
    bind 192.168.101.13:443 ssl crt /etc/ssl/certs/haproxy.pem
    http-request track-sc0 src nginx-backend-http    # track haproxy3 stick table  sc=stickycounter
    mode http
    default_backend nginx-backend-http


backend nginx-backend-http
    stick-table type ip size 1m expire 10s store http_req_rate(10s)      #collect and store data from haproxy1 for the next one
    mode http
    balance     roundrobin
    option httpchk
    option forwardfor
    cookie nginx-lb1 insert indirect nocache
    server  nginx-nd1 192.168.100.201:80 check cookie nginx-nd1
    server  nginx-nd2 192.168.100.202:80 check cookie nginx-nd2
    server  nginx-nd3 192.168.100.203:80 check cookie nginx-nd3
```

```
#KEEPALIVED CONFIGURATION

vrrp_script chk_haproxy {
  script "killall -0 haproxy"                    # check the haproxy process
  interval 2                                     # every 2 seconds
  weight 2                                       # add 2 points if OK
}
vrrp_instance VI_1 {
    state BACKUP
    virtual_router_id 51
    priority 99                                  # PRIORITY SHOULD BE LOWER THAN MASTER VALUE
    advert_int 1
    authentication {
        auth_type PASS
        auth_pass 1111
    }
    unicast_src_ip 192.168.100.103
    unicast_peer {
        192.168.100.101 dev enp0s8
        192.168.100.102 dev enp0s8

   }
    virtual_ipaddress {
        192.168.101.13/24 dev enp0s9
    }
track_script {
    chk_haproxy
  }
}
```

3.Install and configure Keepalived on nginx-lb1

```
#Install the keepalived package on each server:
sudo dnf install keepalived

#Before you can set up load balancing by using Keepalived, you must install and configure the feature.

/etc/sysctl.conf:
net.ipv4.ip_forward = 1
Verify that the IP forwarding has been applied:
sudo sysctl -p

#Add firewall rules to accept VRRP communication by using the multicast IP address
#224.0.0.18 and the VRRP protocol (112) on each network interface that Keepalived
#controls, for example:



#Enable and start the keepalived service on each server:
sudo systemctl enable --now keepalived
#If you change the Keepalived configuration, reload the keepalived service:
sudo systemctl reload keepalived



sudo nmcli con mod "Wired connection 1" connection.id "net-private"
sudo nmcli con mod "Wired connection 2" connection.id "net-public"
sudo nmcli con mod net-private ipv6.method disabled
sudo nmcli con mod net-public ipv6.method disabled
sudo nmcli con mod enp0s3 ipv6.method disabled
sudo systemctl restart NetworkManager

#Add firewall rules to accept VRRP communication by using the multicast IP address
#224.0.0.18 and the VRRP protocol (112) on each network interface that Keepalived
#controls, for example:

sudo firewall-cmd --direct --permanent --add-rule ipv4 filter INPUT 0 \
 --in-interface enp0s9 --destination 224.0.0.18 --protocol vrrp -j ACCEPT
sudo firewall-cmd --direct --permanent --add-rule ipv4 filter OUTPUT 0 \
 --out-interface enp0s9 --destination 224.0.0.18 --protocol vrrp -j ACCEPT
sudo firewall-cmd --reload
```

```
#KEEPALIVED CONFIGURATION


vrrp_script chk_haproxy {
  script "killall -0 haproxy"             # check the haproxy process
  interval 2                              # every 2 seconds
  weight 2                                # add 2 points if OK
}


vrrp_instance VI_1 {
    state MASTER
    virtual_router_id 51
    priority 101                          # 101 for MASTER AND 100 for SLAVE
    advert_int 1
    authentication {
        auth_type PASS
        auth_pass 1111
    }
    unicast_src_ip 192.168.100.101
    unicast_peer {
        192.168.100.102 dev enp0s8
        192.168.100.103 dev enp0s8
    }
    virtual_ipaddress {
        192.168.101.13/24  dev enp0s9
    }
    track_script {
    chk_haproxy
  }
}
```

```
# HAPROXY CONFIGURATION

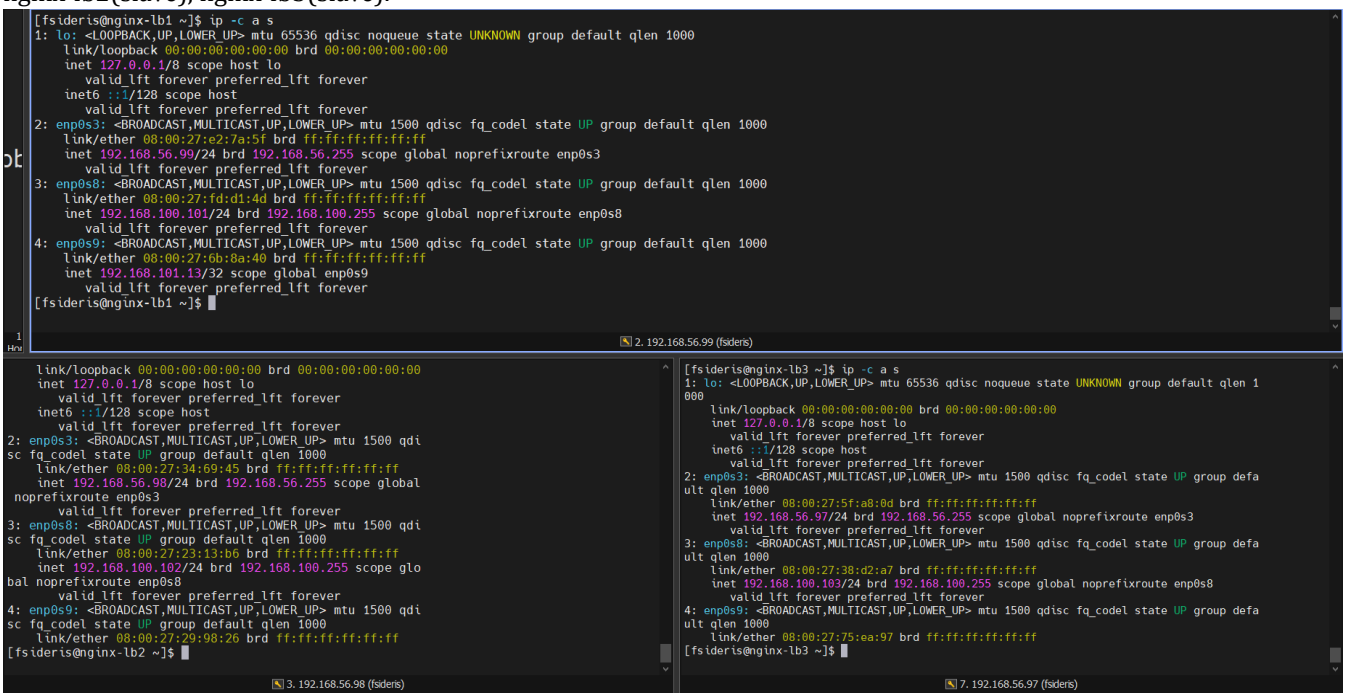
frontend http_front
    bind 192.168.101.13:80
    http-request track-sc0 src nginx-backend-http    # track haproxy3 stick table  sc=stickycounter
    mode http
    default_backend nginx-backend-http

frontend https_front
    bind 192.168.101.13:443 ssl crt /etc/ssl/certs/haproxy.pem
    http-request track-sc0 src nginx-backend-http    # track haproxy3 stick table  sc=stickycounter
    mode http
    default_backend nginx-backend-http

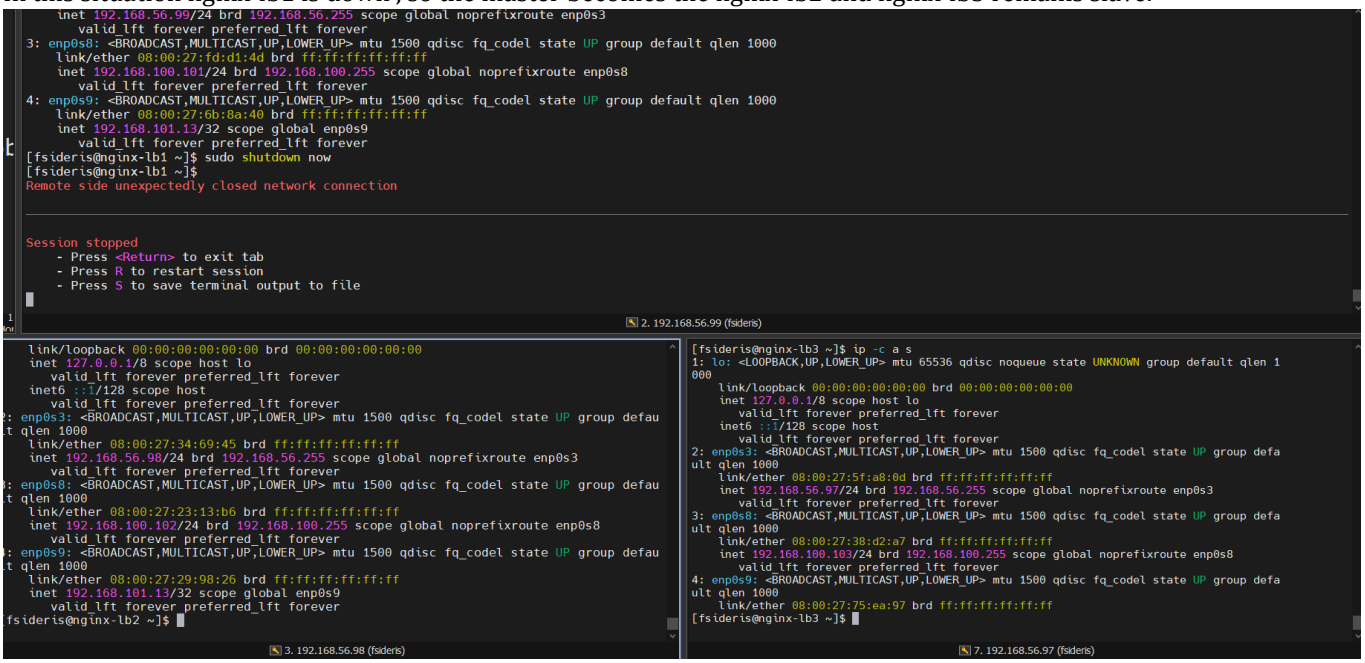

backend nginx-backend-http
    stick-table type ip size 1m expire 10s store http_req_rate(10s)      #collect and store data from haproxy1 for the next one
    mode http
    balance     roundrobin
    option httpchk
    option forwardfor
    cookie nginx-lb1 insert indirect nocache
    server  nginx-nd1 192.168.100.201:80 check cookie nginx-nd1
    server  nginx-nd2 192.168.100.202:80 check cookie nginx-nd2
    server  nginx-nd3 192.168.100.203:80 check cookie nginx-nd3
```

POC: Confirm operation by closing lb1 or lb2, or lb3.

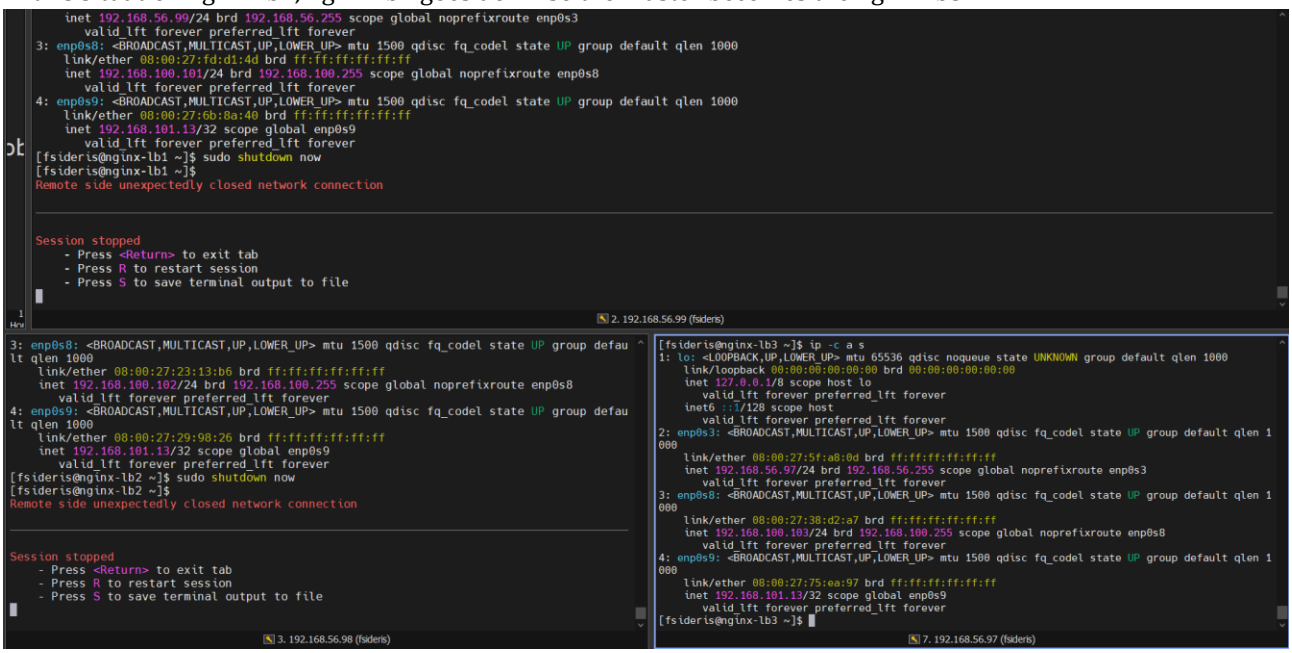
In this situation everything is up and the floating ip 192.168.101.13 functions at the nginx-lb1(master) nginx-lb2(slave), nginx-lb3(slave).



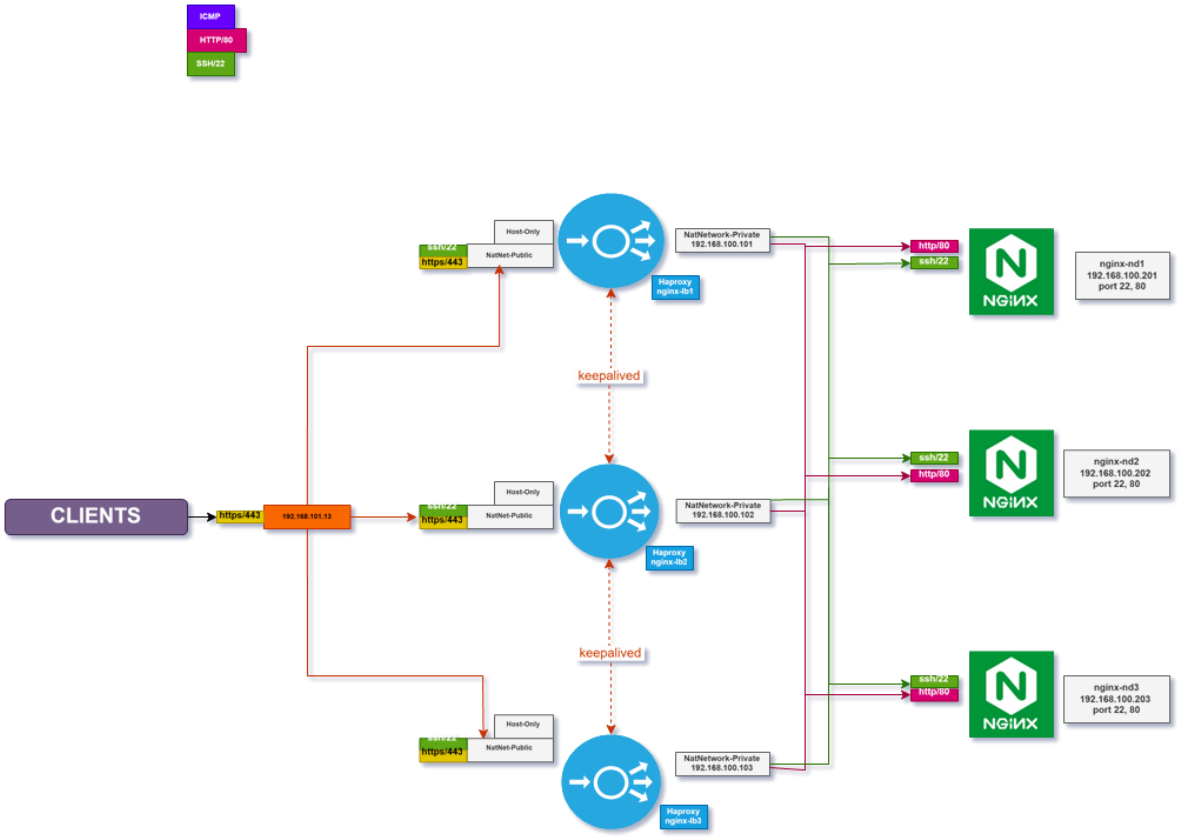
In this situation nginx-lb1 is down , so the master becomes the nginx-lb2 and nginx-lb3 remains slave.



In this situation nginx-lb1, nginx-lb2 goes down so the master becomes the nginx-lb3.



Overview Diagram:



5.8 High-Availability Load Balancers: Sticky Sessions and State Sharing

Description:

That task represents the pinnacle of complexity in the infrastructure setup, focusing on *achieving high availability, secure communication, and session persistence in a load balancer (LB) configuration*. The primary objective is to configure the LB and load balancer instances to maintain continuous service even in the face of failures and to ensure that clients maintain their session state during failover events. To accomplish this, the LB configuration is extended to include features such as session stickiness and state sharing among LB instances.

Session stickiness ensures that once a client establishes a session with a particular backend server, subsequent requests from the same client are directed to the same server to maintain session continuity. This is crucial for applications that rely on session data, such as e-commerce platforms or web applications with user authentication.

Additionally, state sharing among LB instances ensures that critical information, such as session state and LB status, is synchronized across all LB instances. This allows LB instances to coordinate and make informed decisions, such as determining which LB instance should serve as the master LB and prioritizing failover events accordingly. Configuring session stickiness and state sharing involves implementing specialized algorithms and protocols to synchronize data and manage session persistence effectively. For example, the LB instances may use techniques such as IP hashing or session cookies to associate clients with specific backend servers and maintain session affinity. Testing the configuration involves simulating failures by closing individual LB instances and verifying that session state is preserved and requests are seamlessly redirected to the remaining

LB instances. Additionally, verifying that the designated master LB instance resumes control after a failover event ensures that the infrastructure behaves as expected under various failure scenarios.

Tasks:

1. Put this configuration on every HAProxy to share state on sticky mode

```
# HAPROXY CONFIGURATION


peers mypeers
  peer nginx-lb1 192.168.100.101:80
  peer nginx-lb2 192.168.100.102:80
  peer nginx-lb3 192.168.100.103:80

frontend http_front
    bind 192.168.101.13:80
    mode http
    default_backend nginx-backend-http
    option forwardfor                                                #This command
    http-request set-header X-Forwarded-Port %[dst_port]
    http-request add-header X-Forwarded-Proto https if { ssl_fc }

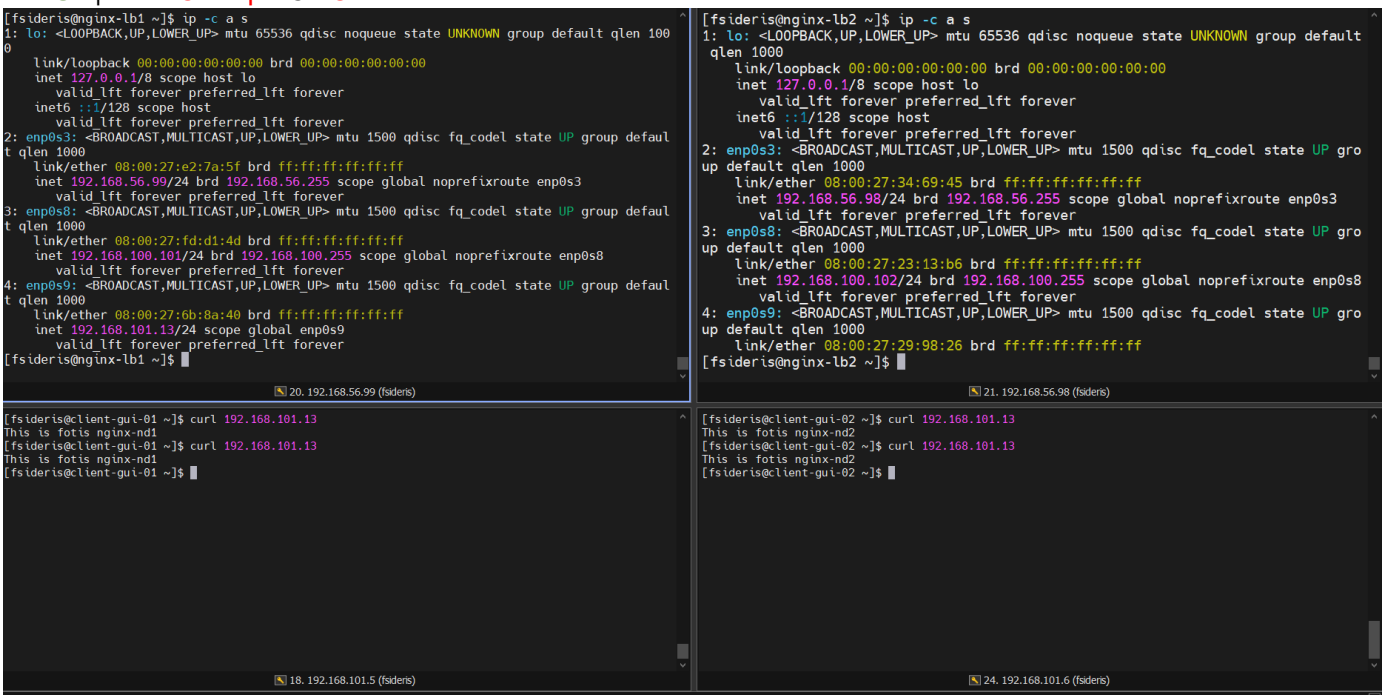
frontend https_front
    bind 192.168.101.13:443 ssl crt /etc/ssl/certs/haproxy.pem
    mode http
    default_backend nginx-backend-http
    option forwardfor
    http-request set-header X-Forwarded-Port %[dst_port]
    http-request add-header X-Forwarded-Proto https if { ssl_fc }



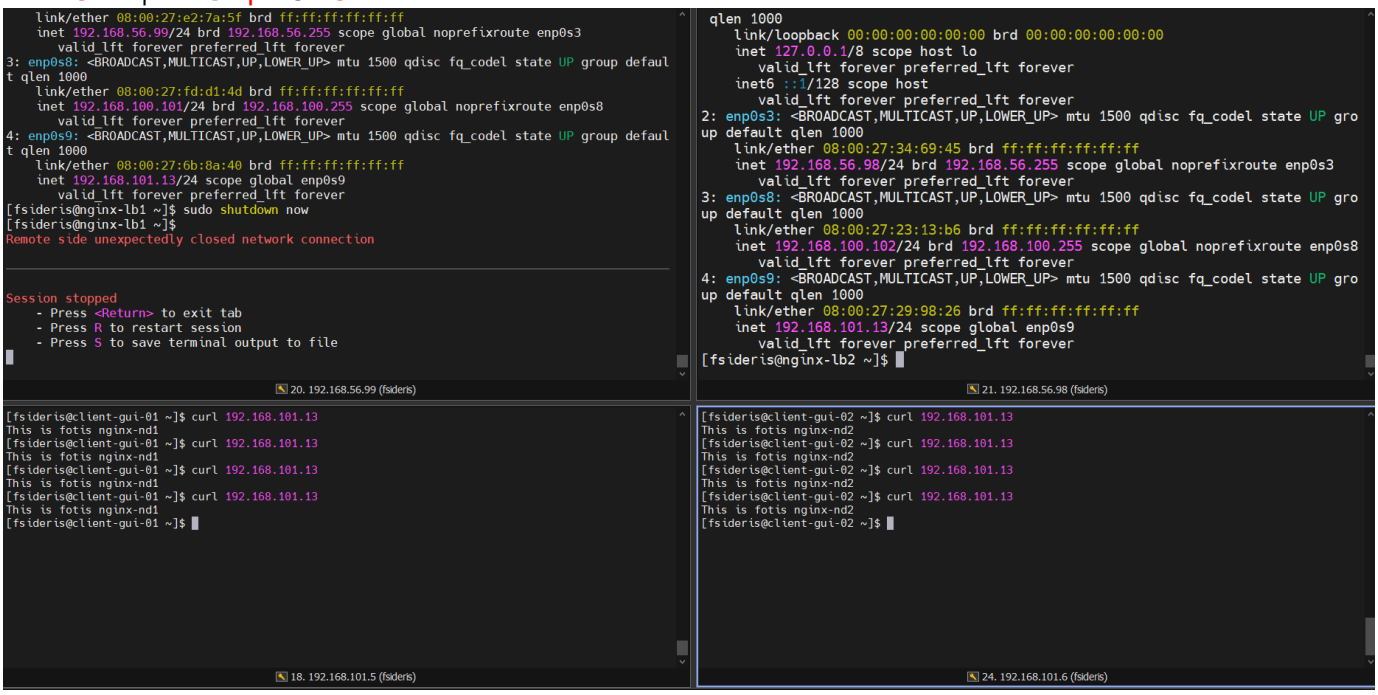
backend nginx-backend-http
     mode http
     balance roundrobin
     stick-table  type ip  size 1m  expire 1h  store http_req_rate(10s) peers mypeers
     stick on src                              # This command tells HAProxy to use th
     server nginx-nd1 192.168.100.201:80
     server nginx-nd2 192.168.100.202:80
     server nginx-nd3 192.168.100.203:80
```

Proof Of Concept:

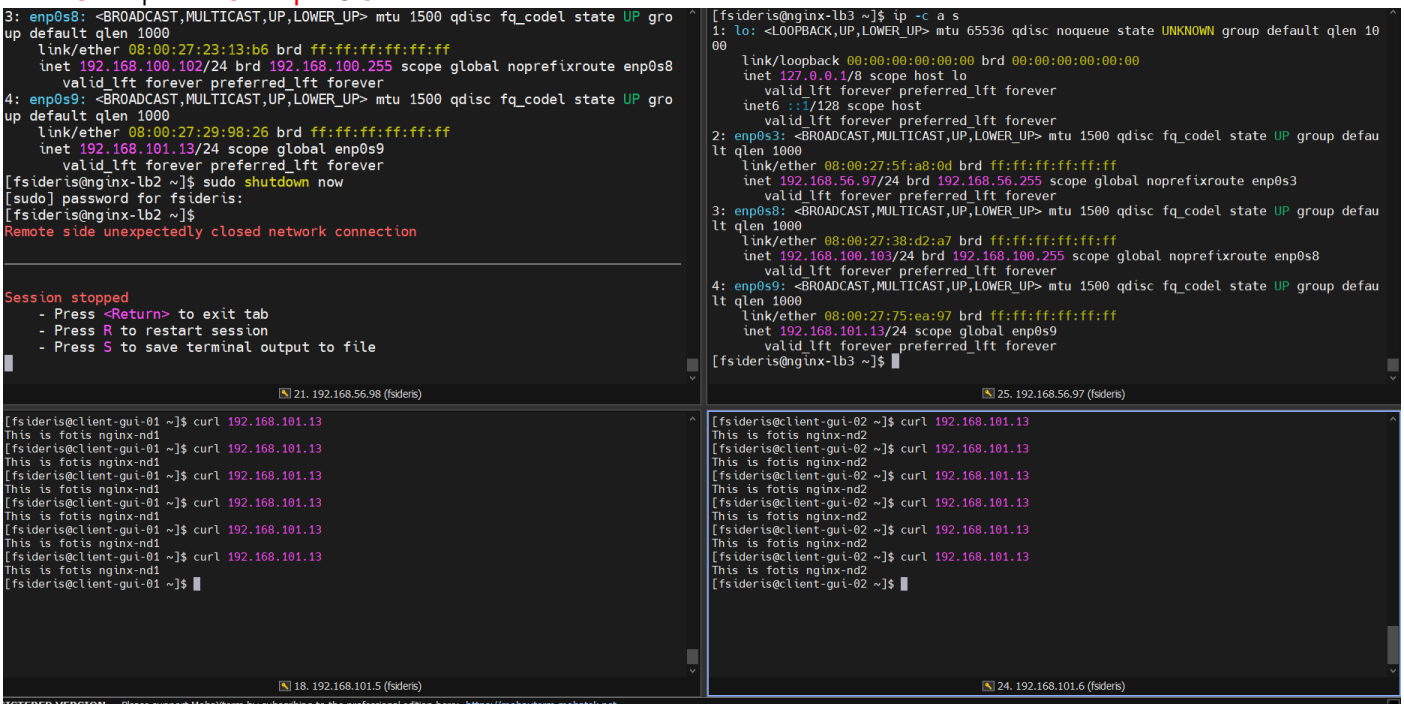
LB1 **UP** | LB2 **DOWN** | LB3 **DOWN**



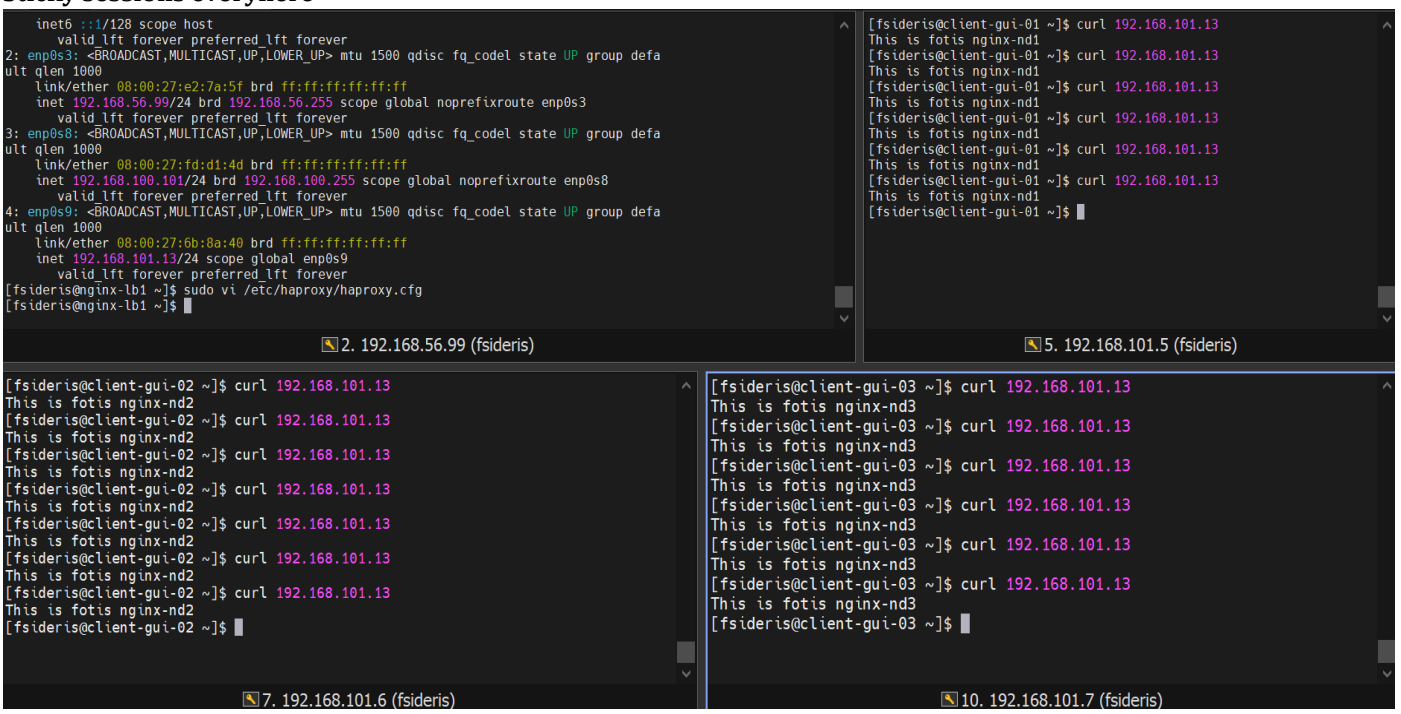
LB1 **DOWN** | LB2 **UP** | LB3 **DOWN**

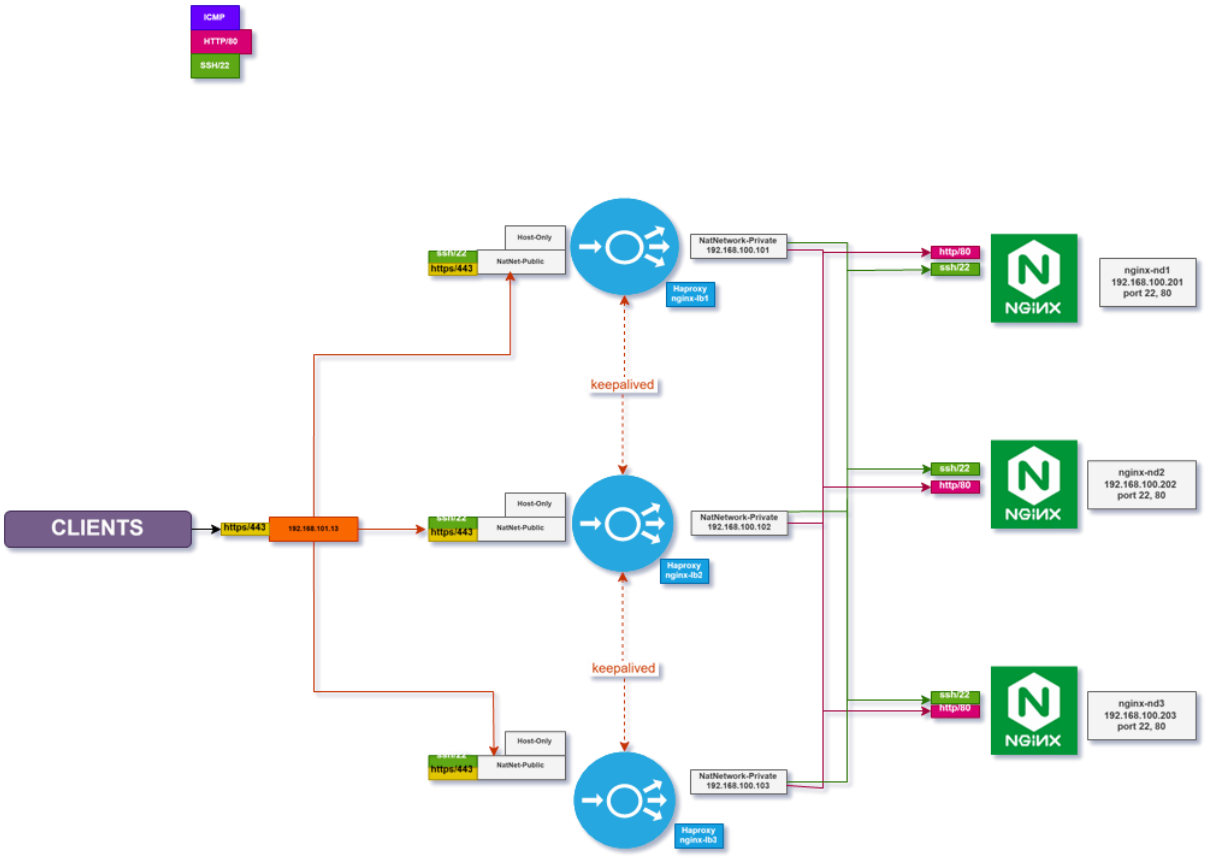


LB1 **DOWN** | LB2 **DOWN** | LB3 **UP**



Sticky sessions everyhere





This table describes a network configuration involving multiple servers and their roles within a system that uses Nginx and HAProxy. Here's a breakdown of the information:

	host only	nginx-private	nginx-public	services
nginx-nd1	-	192.168.100.201	-	Nginx
nginx-nd2	-	192.168.100.202	-	Nginx
nginx-nd3	-	192.168.100.203	-	Nginx
nginx-lb1	192.168.56.99	192.168.100.101	192.168.101.13	HAProxy
nginx-lb2	192.168.56.98	192.168.100.102	192.168.101.13	HAProxy
nginx-lb3	192.168.56.97	192.168.100.103	192.168.101.13	HAProxy
client-gui-01	-	-	192.168.101.5	-

6. Summary of Core Network Technologies

This work pursues the design and implementation of a scalable and resilient web server infrastructure using several networking concepts, including load balancing and high availability technologies. In this Section, we provide a summary of these core network technologies.

6.1 Understanding Network Technologies

Network Fundamentals

1. **Communication Protocols**: Networks rely on protocols such as TCP/IP, HTTP, DNS, and SMTP to facilitate communication between devices.
2. **OSI Model**: The OSI (Open Systems Interconnection) model defines the seven layers of network communication, from physical transmission to application interaction.
3. **IP Addressing**: IP addresses are unique identifiers assigned to devices on a network, enabling data routing and delivery.
4. **Subnetting and CIDR**: Subnetting divides a network into smaller subnetworks, while CIDR (Classless Inter-Domain Routing) notation specifies network prefixes and subnet masks.
5. **MAC Addresses**: MAC (Media Access Control) addresses are hardware addresses assigned to network interfaces, facilitating data link layer communication.

Network Devices

1. **Routers:** Routers connect multiple networks and forward packets between them based on IP addresses.
2. **Switches**: Switches connect devices within a local area network (LAN) and use MAC addresses to forward data packets to the appropriate destination.
3. **Firewalls:** Firewalls enforce network security policies by filtering incoming and outgoing traffic based on predefined rules.
4. **Load Balancers:** Load balancers distribute incoming network traffic across multiple servers to improve performance, reliability, and scalability.

Network Services

1. **DNS (Domain Name System):** Translates domain names into IP addresses, enabling users to access websites using human-readable names.
2. **DHCP (Dynamic Host Configuration Protocol)**: Automatically assigns IP addresses and network configuration parameters to devices on a network.
3. **NAT (Network Address Translation)**: Maps private IP addresses to public IP addresses, allowing devices within a private network to access the internet.
4. **VPN (Virtual Private Network):** Establishes secure, encrypted connections over a public network, enabling remote access and private communication.

Network Security

1. **Encryption**: Protects data confidentiality by encoding information to make it unreadable without the appropriate decryption key.
2. **Authentication**: Verifies the identity of users or devices accessing a network or system.

3. **Access Control**: Restricts user access to network resources based on predefined policies and permissions.
4. **Intrusion Detection and Prevention**: Monitors network traffic for suspicious activity and takes actions to prevent or mitigate security threats.

6.2 Load Balancing Technologies & Algorithms – High Availability

Layer 4 Load Balancing:

TCP Load Balancing operates at the transport layer (Layer 4) of the OSI model and distributes incoming TCP traffic based on factors such as IP addresses and port numbers. UDP Load Balancing: It is like TCP load balancing, but specifically designed for UDP (User Datagram Protocol) traffic. It is commonly used for real-time streaming and VoIP applications.

Layer 7 Load Balancing:

HTTP Load Balancing operates at the application layer (Layer 7) and makes load balancing decisions based on HTTP header information, such as URL paths, cookies, and request methods.

HTTPS Load Balancing provides secure load balancing for HTTPS traffic by terminating SSL/TLS connections at the load balancer and decrypting traffic before forwarding it to backend servers.

Content-Based Routing directs traffic to specific servers based on the content of the request, allowing for more granular control and customization of load balancing rules.

Load Balancing Algorithms

- **Round Robin**: Distributes traffic evenly across backend servers in a sequential manner.
- **Least Connections**: Routes traffic to the server with the fewest active connections, helping to evenly distribute load across servers.
- **IP Hashing**: Hashes the client's IP address to determine which server should handle the request, ensuring that requests from the same client are consistently routed to the same server.

High Availability and Failover

- **Active-Active**: All load balancer instances actively distribute traffic, providing redundancy and scalability over failures.
- **Active-Passive**: One load balancer instance handles traffic while the others remain in standby mode, ready to take over in case of failure.
- **Health Checks**: Periodically monitor the health and availability of backend servers and remove or add servers to the load balancing pool based on their status.

Different Approaches:

Round Robin:

- **Advantages:** Simple and fair, evenly distributes traffic sequentially.
- **Limitations:** May not account for varying server loads or capacities , potentially leading to inefficiencies

Least Connections:

- **Advantages:** Efficiently distributes traffic based on the number of active connections, balancing the load dynamically.
- **Limitations:** Requires continuous monitoring of connection counts , which can be resource-intensive.

IP Hashing:

- **Advantages:** Ensures session persistence by consistently routing requests from the same client to the same server.
- **Limitations:** May lead to uneven load distribution if client IP distribution is not uniform.

6.3 Implementation and Optimization of HAProxy

Implementation of HAProxy involves several key steps aimed at seamlessly integrating it into the existing infrastructure and configuring it to efficiently manage incoming traffic. Firstly, a thorough assessment of the network topology and traffic patterns must be conducted to determine the optimal placement of HAProxy instances. Typically deployed in a reverse proxy mode, HAProxy sits between clients and backend servers, effectively distributing requests based on predefined rules and algorithms.

Upon determining the deployment strategy, the next step involves the installation and configuration of HAProxy on designated servers. HAProxy offers a myriad of configuration options, enabling administrators to tailor its behavior to specific requirements. Configuration parameters encompass load balancing algorithms, health checks, SSL termination, session persistence, and request routing rules, among others.

Optimizing HAProxy entails fine-tuning its configuration settings and leveraging advanced features to maximize performance, scalability, and reliability. One fundamental optimization technique revolves around selecting appropriate load balancing algorithms based on the characteristics of the workload. Options such as round-robin, least connections, and weighted least connections offer varying degrees of efficiency and fairness in distributing traffic across backend servers.

Furthermore, optimizing backend server health checks and timeouts ensures prompt detection and mitigation of server failures, thereby enhancing fault tolerance and availability. HAProxy's ability to perform active and passive health checks enables proactive monitoring of server status and automatic removal of unhealthy servers from the rotation.

Another pivotal aspect of HAProxy optimization lies in optimizing SSL termination and encryption. By offloading SSL decryption and encryption to HAProxy, backend servers are relieved of this computational overhead, resulting in improved performance and scalability. Additionally, enabling features like HTTP/2 support and connection multiplexing further enhances throughput and responsiveness, particularly in high-traffic environments.

In conclusion, the implementation and optimization of HAProxy form pivotal endeavors in achieving scalable, high-performance, and reliable web architectures. By leveraging HAProxy's robust features and flexible configuration options, organizations can effectively manage incoming traffic, distribute workloads, and mitigate potential points of failure. Through meticulous planning, deployment, and optimization, HAProxy emerges as a cornerstone solution in modernizing and optimizing web infrastructures to meet the demands of today's digital landscape.

6.4 Advanced Configuration and Redundancy Techniques

High Availability (HA) Clustering

HA clustering involves grouping multiple servers or nodes together to provide redundancy and failover capabilities. In the event of a hardware or software failure on one node, services are automatically migrated to another node within the cluster, ensuring uninterrupted operation.

Advanced clustering solutions like Pacemaker with Corosync (or Kubernetes) enable automatic failover, load balancing, and resource management across clustered nodes. They monitor the health and status of nodes and services, orchestrating failover actions seamlessly.

Active-Active Redundancy

Active-active redundancy involves distributing workload across multiple redundant components simultaneously, rather than relying on a passive standby. This approach maximizes resource utilization and scalability while ensuring redundancy and fault tolerance.

Load balancers like HAProxy and software-defined networking (SDN) solutions can be configured for active-active redundancy, distributing incoming traffic across multiple backend servers or network paths in real-time.

Fault-Tolerant Architectures

Implementing fault-tolerant architectures involves designing systems with redundancy at every level to minimize the impact of failures. Therefore, it involves hardware, software, and information fault tolerance techniques.

Examples include redundant power supplies, storage arrays, network paths, and communication links. Technologies like hardware or software RAID (Redundant Array of Independent Disks) and network bonding enable redundant storage and network configurations, respectively, ensuring data integrity and network resilience in the face of hardware failures.

Container Orchestration and Microservices

Container orchestration platforms like Kubernetes and Docker Swarm enable the deployment, scaling, and management of containerized applications across clusters of nodes.

By leveraging microservices architecture and container orchestration, organizations can achieve redundancy and fault tolerance at the application level, with individual services resilient to failures and capable of horizontal scaling.

7. Summary and Future Work

The thesis presents a comprehensive approach to designing and implementing a scalable and resilient web server ecosystem that leverages advanced network technologies. This ecosystem is engineered to support scalability through the use of various load balancing strategies and to ensure high availability, thereby providing efficient and reliable web services for a company or organization. The proposed solution harnesses the power of open-source, industrial-strength technologies: Nginx serves as the web server, HAProxy functions as the load balancer, and Keepalived provides reliability and failover capabilities.

The development of the Nginx-HAProxy-Keepalived ecosystem follows a methodical and incremental approach. Starting from a basic infrastructure, the system is built up step-by-step using multiple layers of Linux scripts. These scripts provide a well-defined, gradual path for system and network modifications, which can be adjusted depending on the available system resources. This modular approach ensures that the ecosystem can be scaled and adapted to different environments and requirements.

In the initial phase, the system was tested in a controlled environment using Virtual Machines. The test setup included a small number of server nodes, typically two or three, to simulate a realistic yet manageable scenario. The preliminary measurements focused on continuous connections to assess the system's performance under load. The results showed that the developed load balancing and fault-tolerant solution offers significant scalability while maintaining a low latency overhead. Specifically, the overhead was found to be only around 30% compared to a completely unbalanced, non-reliable solution that relies on a single point of failure. This demonstrates the effectiveness of the proposed ecosystem in distributing load and ensuring high availability.

Future work will focus on extending the application of the solution to real-world scenarios and enhancing its management capabilities. One area of improvement is the development of tools or services that can validate the consistency of existing security policies. This will ensure that the system remains secure and compliant with organizational standards. Additionally, implementing network monitoring tools will facilitate automated reconfiguration and orchestration based on resource availability. This dynamic management approach will allow the system to adapt to changing conditions and optimize performance.

Another important direction for future research is to examine the performance and scalability of different load balancing strategies in more detail. This involves comparing the proposed Nginx-HAProxy-Keepalived solution to other load balancing and fault-tolerant technologies. Potential comparisons include custom software load balancing solutions, distributed fault-tolerant filesystems, and container-based technologies such as Kubernetes scheduling and Docker Swarm. These comparisons will provide valuable insights into the strengths and weaknesses of each approach and help identify the most effective strategies for different use cases.

In summary, this thesis lays the groundwork for a robust and adaptable web server ecosystem that can meet the demands of modern web services. By leveraging established technologies and following a structured development approach, the proposed solution offers a scalable and reliable framework for load balancing and high availability. Future enhancements and comparisons with alternative technologies will further refine this ecosystem, ensuring it remains at the forefront of web server management solutions.

Bibliography

[1] Tanenbaum, A. S., & Wetherall, D. J. (2011). Computer Networks (5th ed.). Pearson Education. Ch1

[2] Mladenovic, M. (2019). Mastering HAProxy: Advanced Techniques to Load Balance and Proxy HTTP Services (1st ed.). Packt Publishing.

[3] Garrett, R. (2019). Nginx HTTP Server: Adopt Nginx for your web applications to make the most of your infrastructure and serve pages faster than ever (3rd ed.). Packt Publishing.

[4] Gomes, R. (2018). Keepalived Network Management: A practical guide to configuring, administering, and securing Keepalived (1st ed.). Packt Publishing.

[5] Rosenblum, M., & Garfinkel, T. (2005). Virtual Machine Monitors: Current Technology and Future Trends. IEEE Computer Society.

[6] Cheswick, W. R., Bellovin, S. M., & Rubin, A. D. (2003). Firewalls and Internet Security: Repelling the Wily Hacker (2nd ed.). Addison-Wesley.

[7] Borowiec, J. (2017). HAProxy High Availability: Ensure high availability and optimize performance of your HAProxy using techniques such as timeouts, health checks, and backup servers (1st ed.). Packt Publishing.

[8] Gomes, R. (2018). Keepalived Network Management: A practical guide to configuring, administering, and securing Keepalived (1st ed.). Packt Publishing.