ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

ELECTRICAL, ELECTRONIC AND INFORMATION ENGINEERING "GUGLIELMO MARCONI", DEI.

SECOND CYCLE DEGREE

TELECOMMUNICATION ENGINEERING

Optimizing Web Service Performance: A Comparative Analysis of Load Balancing Strategies Using NGINX and HAProxy with StoRM WebDAV Deployment

Dissertation in NETWORK DESIGN M

Supervisor

Prof. CARLA RAFFAELLI

Defended by

MAHSA KAZEMI

Co-Supervisors

FRANCESCO GIACOMINI

ROBERTA MICCOLI

Academic Year 2023/2024

Session III

Introduction

It is imperative to make sure that applications are scalable, dependable, and quick in the quickly changing web service landscape of today. Load balancing is becoming a more vital part of online infrastructure due to the growing need for high-performing web services, which is being driven by more users and more complicated applications. Incoming network traffic is spread among several servers via load balancing, which is essential for avoiding server overload, speeding up response times, and guaranteeing high availability—particularly during moments of heavy demand.

NGINX and HAProxy are two of the most widely used tools for load balancing implementation. Although they both have a solid reputation for efficiently handling massive volumes of online traffic, their features and performance enhancements are different. NGINX was first created as a web server, but it has since developed into a flexible platform that can also be used as a load balancer and reverse proxy. It offers a number of methods, including IP hash, least connections, and round-robin. HAProxy, on the other hand, is renowned for its robustness and cutting-edge features like health checks, sticky sessions, and SSL termination. It is focused on high availability and load balancing for TCP and HTTP applications.

As part of the StoRM (Storage Resource Manager) project, the Italian National Institute for Nuclear Physics (INFN) developed the StoRM WebDAV service, which is intended to satisfy the requirements of large-scale scientific computing environments, especially those connected to grid and cloud infrastructures like the Worldwide LHC Computing Grid (WLCG). Because of its support for the WebDAV protocol, users can effectively handle files on distant servers. Optimizing the performance and scalability of StoRM WebDAV is crucial, considering its significance in data management. The efficiency of StoRM WebDAV can be greatly increased by placing it behind load balancers such as NGINX and HAProxy. However, there hasn't been much research done on how various load balancing techniques affect StoRM WebDAV.

By offering a thorough comparison of NGINX and HAProxy as load balancers for StoRM WebDAV, the thesis aims to close this gap. This study aims to identify the load balancing strategy that provides the best scalability and performance for StoRM WebDAV deployments through practical implementation, performance testing, and analysis. The findings will be insightful for developers and system administrators who oversee large-scale storage environments.

The thesis's first chapter presents the idea of load balancing, describing its main goals and several approaches.

The second chapter explores the system's architecture, including a thorough rundown of NGINX, HAProxy, and StoRM WebDAV. A comprehensive schema of the overall architecture design is also provided in this section, along with a discussion of the difficulties and possible problems that arose during the design process.

The configuration of the system components is covered in detail in Chapter 3, which also covers the integration and deployment of NGINX, HAProxy, and StoRM WebDAV. Examining the effects of various load balancers on the behavior and performance of StoRM WebDAV, it examines how these factors affect the system's efficiency.

The last chapter, which compares the efficacy and performance of NGINX and HAProxy as load balancers, is devoted to testing.

Contents

Introduction 3
Chapter 1: Load balancing 8
1.1 What is load balancing? 8
1.2 Load balancing strategies 8
1.2.1 Load balancers: Hardware or software 8
1.2.2 Load balancers: static and dynamic 9
1.2.3 Load balancers: level 4 and level 7 9
1.3 Load balancing algorithms 10
1.3.1 Round Robin 10
1.3.2 Weighted Round Robin 10
1.3.3 Least Connection 11
1.3.4 IP Hashing 11
1.3.5 Least Response Time 11
1.3.6 Least Bandwidth 12
1.4 Advanced load balancing features 12
Chapter 2: Architecture Components 14
2.1 NGINX overview 14
2.1.1 Core Features of NGINX 14
2.1.2 Advanced HTTP Features 15
2.1.3 Mail and TCP/UDP Proxy Features 16
2.1.4 Architecture and Scalability 17
2.2 HAProxy overview 18
2.2.1 Core Features of HAProxy 18
2.2.2 Advanced Traffic Management 20
2.2.3 High Performance and Scalability 20
2.2.4 Use Cases 21
2.3 StoRM WebDAV overview 22
2.3.1 Key Features of StoRM WebDAV 22
2.3.2 StoRM WebDAV architecture 23
2.3.3 NGINX Role 24
2.4 Architecture design 26

2.5 Design problem ... 27
Chapter 3: Components Configuration ... 28
3.1 StoRM WebDAV Deployment ... 28
3.1.1 StoRM WebDAV Installation ... 29
3.1.2 StoRM WebDAV Configuration ... 29
3.1.3 Puppet installation ... 31
3.2 NGINX Deployment ... 32
3.2.1 NGINX Installation ... 32
3.2.2 NGINX Configuration ... 32
3.3 HAProxy Deployment ... 35
3.3.1 HAproxy Installation ... 35
3.3.2 HAProxy Configuration ... 36
3.4 Client Deployment ... 37
Chapter 4: Performance Testing ... 38
4.1 File download test ... 38
4.1.1 Direct Access to StoRM WebDAV ... 38
4.1.2 Access via NGINX Load Balancer ... 39
4.1.3 Access via HAProxy Load Balancer ... 40
4.1.4 Performance Analysis ... 41
4.2 Stress testing with Vegeta ... 42
4.2.1 Direct access to StoRM WebDAV ... 44
4.2.2 Access via NGINX load balancer ... 52
4.2.3 Access via HAProxy load balancer ... 61
4.2.4 Test results ... 70
4.2.5 Key observations ... 71
4.2.6 Recommendations for optimization ... 71
4.3 Future Work ... 72
Conclusion ... 73
References ... 74

Chapter 1: Load balancing

1.1 What is load balancing?

Load balancing is a technique used in computer and networking that efficiently distributes workloads or incoming requests among several resources, such as servers, CPUs, or network lines. The primary goals of load balancing are to maximize throughput, minimize reaction times, optimize resource use, and ensure high availability and dependability of services or applications.

Basically, load balancing is dividing up incoming traffic or workload among multiple backend resources in order to keep no resource overworked and to ensure that every resource is utilized to its maximum capacity. To do this, a variety of algorithms and techniques can be applied, such as weighted distribution, least connections, round-robin scheduling, and dynamic adjustments based on real-time variables like network conditions or server health.

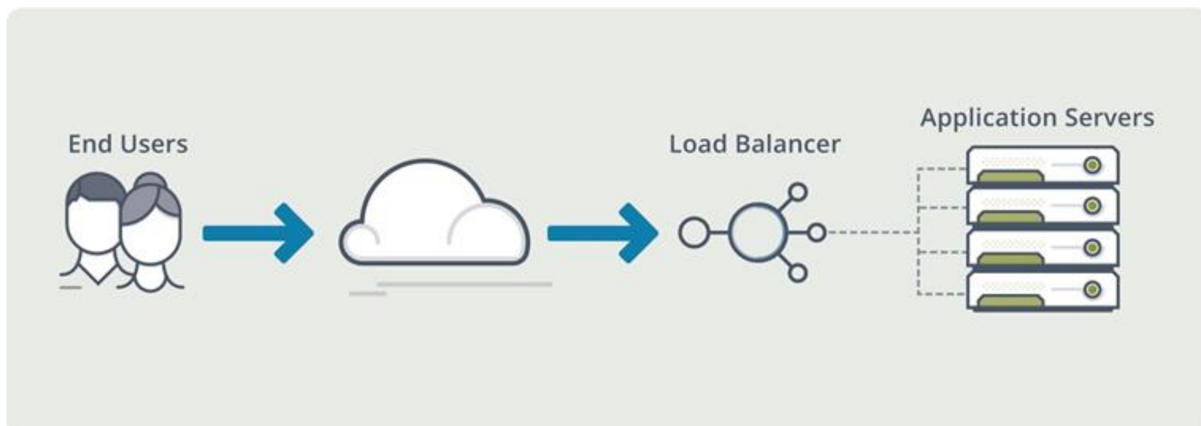

Figure 1.1: Overall Scheme of Load Balancer Architecture

1.2 Load balancing strategies

1.2.1 Load balancers: Hardware or software

Load balancing is a common technique used by web servers, application servers, databases, content delivery networks (CDNs), and other distributed systems to manage high traffic volumes, increase scalability through horizontal resource scaling, and enhance performance. Additionally, load balancing prevents service interruptions by transferring traffic from failing or

underperforming components to functioning ones, which is why it is crucial to fault tolerance and high availability systems.

Load balancing comprises distributing incoming network traffic among several servers or resources to ensure maximum resource utilization, increase throughput, reduce response time, and avoid overwhelming any single resource. It can be implemented using either specialized hardware or software solutions running on conventional hardware or in virtualized environments.

Hardware-based load balancing solutions often use dedicated appliances or specially designed hardware components meant to efficiently manage and disperse network traffic. Conversely, load balancing that is software-based makes use of apps or software components that run on normal servers or virtual machines to perform load balancing functions. These software solutions can be as simple as reverse proxy servers or as sophisticated as software-defined networking systems. They provide flexibility, scalability, and cost-effectiveness in managing and optimizing network traffic flow.

1.2.2 Load balancers: static and dynamic

The behavior of load balancers falls into two categories: dynamic and static. Static load balancers use preset configurations and pre-established policies to distribute traffic among backend servers. They must be manually configured; they cannot be instantly changed in response to variations in the network or server load. Dynamic load balancers, on the other hand, dynamically adjust traffic distribution by constantly observing server health, response times, and network conditions. They distribute traffic as effectively as possible in response to changing conditions by utilizing auto-scaling technology and sophisticated routing algorithms. Load balancers are crucial for maintaining high availability, scalability, and performance in distributed computing environments.

1.2.3 Load balancers: level 4 and level 7

Differentiation of load balancers is made possible by the OSI model layers at which they operate: Layer 4 (L4) and Layer 7 (L7) load balancers. Layer 4 load balancers use network and transport layer protocols, such as IP addresses and TCP/UDP ports, to make routing decisions.

They operate on the transport layer (TCP/UDP) and distribute traffic equitably among backend servers using methods like Network Address Translation (NAT) and connection-based load balancing. In contrast, Layer 7 load balancers operate at the Application Layer (HTTP/HTTPS) and are capable of identifying the most efficient path by utilizing data specific to each application, such as URLs, HTTP headers, and cookies. They provide more sophisticated features including SSL termination, sophisticated load-balancing algorithms, and content-aware routing, making them suitable for applications with complex routing requirements.

1.3 Load balancing algorithms

In distributed computing systems, load balancing algorithms are essential elements that guarantee the effective distribution of workloads among several servers or resources. These algorithms' main goal is to keep all servers from overloading, which improves the system's fault tolerance, availability, and performance. The choice of algorithm typically determines how effective a load balancing solution is because different algorithms have different properties that make them appropriate for different kinds of workloads and system architectures.
The pros and cons of some popular load balancing algorithms are covered in the section below.

1.3.1 Round Robin

The Round Robin algorithm is a simple technique that assigns incoming requests to a pool of servers in a sequential manner. Its simplicity guarantees that each server receives an equal number of requests over time and makes implementation simple. But this approach doesn't take each server's capacity or current load into account. Round Robin might result in inefficiencies in contexts where servers have different capacities or loads since a more powerful server might handle the same number of requests as a less competent one. This disregard for server health could lead to some servers being overworked and others being underutilized.

1.3.2 Weighted Round Robin

With the Weighted Round Robin version, servers with better capacity manage a greater share of the traffic by allocating a weight to each server based on its capacity or other pertinent parameters.

Resource usage is enhanced by this method, particularly in setups with diverse servers and varying capabilities. But determining the right weights can be difficult, especially in dynamic settings where server performance can change. Furthermore, even though Weighted Round Robin takes server capacity into account, it still ignores the strain that servers are under in real time, which could result in a less-than-ideal allocation of requests.

1.3.3 Least Connection

By considering the server load in real-time, the Least Connection algorithm dynamically routes incoming requests to the server with the fewest active connections, assisting in more effective load balancing. Because it makes sure that less busy servers are used more, this technique is especially helpful in settings where connections have variable durations. However, there is extra overhead associated with this approach since it necessitates constant monitoring of the quantity of active connections. Uneven load distribution may also result from the server with fewer connections becoming overloaded if some connections require more resources than others.

1.3.4 IP Hashing

To guarantee that the same client is always sent to the same server, IP hashing divides incoming requests across servers according to the IP address of the client. Because it preserves user sessions without the need for server-side session management, this technique is perfect for applications that need session persistence, such e-commerce websites. On the other hand, IP hashing may result in an uneven load allocation, particularly if the clientele is dispersed unevenly. Furthermore, this method is less adaptable because it struggles to adjust to variations in client distribution or server capacity, which could result in wasteful resource usage.

1.3.5 Least Response Time

Traffic is sent to the server with the fewest active connections and the lowest average response time by the Least Response Time algorithm. This method gives priority to servers that can process requests the fastest, which optimizes both server load and user experience. Although this approach can greatly increase overall performance, it requires real-time server response time monitoring,

which adds complexity. Furthermore, if response times are frequently altered, requests may be allocated unevenly, which could result in instability if not well controlled.

1.3.6 Least Bandwidth

An algorithm known as Least Bandwidth routes traffic to the server that is presently using the least amount of bandwidth. This method works especially well in settings where requests differ greatly in the volume of data they send. The algorithm makes sure that no single server is overloaded with requests with high bandwidth, which maximizes network utilization. But because bandwidth usage must be continuously tracked, it adds overhead. Furthermore, because it just considers bandwidth, it can overlook other important considerations like CPU or memory utilization, which would result in an uneven load among servers.

1.4 Advanced load balancing features

Load balancing has developed to include cutting-edge features that improve resource availability, utilization, and security as digital infrastructure grows more complicated. These functionalities are necessary to handle the demands of contemporary applications.

- **SSL Termination and Offloading**

SSL termination entails centralizing SSL management, decreasing CPU strain on backend servers, and decrypting incoming SSL traffic at the load balancer. While throughput is increased, critical data must be protected during decryption, hence safe implementation is necessary.

- **Application Layer Load Balancing (Layer 7)**

Layer 7 load balancing provides more precise traffic control and improved security by basing routing decisions on content, such as URLs or cookies. Although it can add complexity, it is especially helpful for content-based routing, multi-tenancy, and AB testing.

- **Global Server Load Balancing (GSLB)**

By directing customers to the closest or highest-performing data center, GSLB enhances disaster recovery and lowers latency by distributing traffic across several geographic areas. Although it increases dependability and user experience, it makes management more difficult.

- **Health Monitoring and Failover**

Advanced health monitoring constantly checks the health of backend servers and automatically reroutes traffic from failed servers to ensure high availability. Proactive maintenance lowers downtime and boosts system reliability.

- **Traffic Shaping and Rate Limiting**

Traffic shaping and rate limitation prevent any one user or service from overloading the system by managing the flow of traffic. Specifically, these tactics improve resource efficiency and prevent abuse when it comes to DDoS mitigation and API gateways.

- **Auto-Scaling Integration**

Load balancers with auto-scaling integration can modify the quantity of active servers in response to demand. While cautious preparation is necessary to limit potential latency during scaling, this guarantees cost effectiveness and constant performance during traffic surges.

- **Security Features**

Advanced load balancers offer complete protection at the load balancer level by integrating security features like SSL inspection, DDoS protection, and Web Application Firewalls (WAFs). This can save latency and eliminate the need for separate security equipment.

- **Multi-Cloud and Hybrid Cloud Load Balancing**

Load balancers improve redundancy and performance by distributing traffic among many cloud providers and on-premises data centers in multi-cloud and hybrid cloud settings. Nonetheless, maintaining interoperability in many contexts might be difficult.

Chapter 2: Architecture Components

2.1 NGINX overview

Pronounced "engine x," NGINX is an open-source, very effective HTTP and reverse proxy server. It may also be used as a mail proxy and a general TCP/UDP proxy server. NGINX, which was first developed by Igor Sysoev, is now widely used, especially for managing high traffic levels with the best possible resource efficiency. Numerous popular websites, including well-known Russian platforms like Yandex, Mail.Ru, VK, and Rambler, rely on it, while international businesses like Dropbox, Netflix, and FastMail.FM also utilize it. It is a key component of contemporary web infrastructures due to its capacity to support millions of concurrent connections with little resource usage.

Under a 2-clause BSD-like license, NGINX is released, and F5, Inc. provides commercial support.

2.1.1 Core Features of NGINX

Load Balancing

NGINX's robust load-balancing functionality ensures that no single backend server is overwhelmed by distributing incoming traffic across multiple servers. This improves system performance and reliability by preventing server overload and optimizing resource usage. NGINX offers various load-balancing algorithms, including:

- IP hash
- Least connections
- Round-robin

Additionally, it conducts health checks on backend servers, automatically bypassing any unresponsive servers, thereby improving fault tolerance and availability. This makes NGINX essential in high-traffic environments where scalability and reliability are critical.

HTTP Server Capabilities:

Known for its exceptional performance as a web server, NGINX excels in managing dynamic

applications using the uwsgi, SCGI, and FastCGI protocols while also serving static content efficiently. Key features include:

Static File Serving and Caching:
NGINX enhances performance for serving static files by leveraging an open file descriptor cache.

Reverse Proxy with Caching:
It acts as an efficient reverse proxy, enabling fault tolerance and distributing traffic across multiple servers.

Modular Architecture:
NGINX's modular design allows for the implementation of various filters, such as: XSLT transformations, Byte-range serving, Gzipping

Additionally, it supports parallel processing for tasks managed by FastCGI/uwsgi or proxied servers.

SSL/TLS and HTTP/2/3 Support:
NGINX natively supports modern protocols, enhancing resource prioritization, security, and overall performance.

2.1.2 Advanced HTTP Features

- **Virtual servers:**

NGINX supports name-based and IP-based virtual hosting.

- **Connection management:**

It supports keep-alive and pipelined connections, ensuring efficient use of network resources.

- **Logging and monitoring:**

NGINX offers extensive logging capabilities, with support for different log formats, buffering, fast log rotation, and syslog integration.

- **Security and access control:**

NGINX provides robust access control based on IP addresses, password protection, and HTTP Basic authentication, with validation of HTTP referrers.

- **Streaming and rate limiting:**

It supports streaming for FLV and MP4 formats and allows for response rate limiting and connection limiting to mitigate DDoS attacks.

2.1.3 Mail and TCP/UDP Proxy Features

In addition, NGINX serves as a powerful mail proxy that supports SMTP, POP3, and IMAP protocols with multiple authentication options. Support for STARTTLS, SSL/TLS, and internal server redirection depending on external HTTP authentication are all provided.

NGINX offers IP-based geolocation, load balancing, fault tolerance, generic proxying, access control, and logging tools for TCP/UDP traffic.

2.1.4 Architecture and Scalability

Using a master-worker process approach, where each worker process manages requests under an unprivileged user, NGINX is built for high concurrency and scalability. With this architecture, upgrades and reconfigurations may be completed without affecting services. NGINX's capacity to manage thousands of inactive keep-alive connections with low memory consumption is one of its most important features, which makes it perfect for managing heavy traffic volumes.

Figure 2.1: Overview of NGINX Process Model

To better understand this design, you need to understand how NGINX runs. NGINX has a master process (which performs the privileged operations such as reading configuration and binding to ports) and several worker and helper processes.

2.2 HAProxy overview

By splitting up incoming network traffic among several servers, HAProxy (High Availability Proxy) is an open-source, incredibly dependable load balancer and proxy server that is frequently used to increase the speed, availability, and scalability of applications. HAProxy is a well-liked option for environments needing high availability, including those hosting web applications, databases, or microservices. It is renowned for its reliability, low latency, and resource economy. High-traffic situations benefit greatly from HAProxy's ability to manage numerous concurrent connections while consuming low resources. Because it can function at both the TCP (Layer 4) and HTTP (Layer 7) levels, it can effectively route traffic according to many kinds of data, including IP addresses, hostnames, and HTTP headers.

2.2.1 Core Features of HAProxy

Load Balancing

The fundamental function of HAProxy's load-balancing features is to divide traffic equally among several backend servers. By preventing a single server from receiving too many requests, this improves reliability and performance. Numerous load-balancing algorithms are supported by HAProxy, including:

- **Round Robin:** Requests are distributed evenly across servers in a circular fashion.
- **Least Connections:** Requests are routed to the server with the fewest active connections.
- **Source:** A client's IP address determines the server that will handle the request, ensuring that the client is directed to the same server for subsequent requests.

Additionally, session persistence—which is especially helpful for stateful applications or preserving user sessions in web applications—allows specific users to connect to the same server on a regular basis via HAProxy.

Health Checking

HAProxy uses configurable health checks to continuously monitor the state of backend servers. High availability is ensured by HAProxy, which automatically distributes traffic to other servers that are available in case a server becomes unresponsive or sick. Proactive monitoring improves system resilience and decreases downtime.

Fault Tolerance and Failover

The fault tolerance features of HAProxy make sure that, in the event of a server failure, incoming traffic is smoothly routed to other servers that are operational, preventing service interruptions. The available failover options are active-passive and active-active, contingent on the required redundancy level.

SSL Termination and TCP/HTTP Proxying

By managing incoming SSL/TLS encrypted traffic and offloading the decryption process from backend servers, HAProxy can function as an SSL terminator. By enabling backend servers to handle unencrypted traffic, this enhances performance. Furthermore, HAProxy can function as a proxy on the HTTP and TCP levels:

- **TCP Proxying (Layer 4):** routes data efficiently for database traffic and other non-HTTP services by using low-level network information, such as IP addresses and ports.
- **HTTP Proxying (Layer 7):** makes it possible for HAProxy to decide how to route traffic based on URLs, HTTP headers, and cookies. This is especially helpful for web apps and API-based services.

2.2.2 Advanced Traffic Management

HAProxy offers a range of advanced traffic management features, including:

- **Connection limits:** In order to help prevent distributed denial-of-service (DDoS) attacks and server overload, HAProxy has the ability to limit the number of concurrent connections per server or per client.
- **Request queuing**: HAProxy has the ability to queue incoming requests and distribute them when server resources become available in the event of a server overload.
- **Request redirection and rewriting**: Flexible URL rewriting and redirection depending on predefined criteria, such as user-agent, request headers, or client location, is supported for request redirection and rewriting.
- **Logging and Monitoring:** Comprehensive request and response logging is offered by HAProxy, and it may be tailored to include crucial performance indicators like connection numbers, response times, and status codes. In order to identify irregularities and make sure the system is operating effectively, these logs can be connected with outside monitoring systems.

2.2.3 High Performance and Scalability

- **High Performance and Low Latency:** HAProxy is intended for settings where performance with minimal latency is essential. By effectively managing traffic and making the best use of system resources, it reduces latency. It is appropriate for large-scale installations since it can manage millions of concurrent connections.
- **SSL/TLS Encryption:** Data transfer between clients and servers can be done securely thanks to HAProxy's support for SSL/TLS encryption. It can manage SSL certificates and decrypt them, relieving the backend servers of these responsibilities and enhancing overall performance.
- **Scalability:** By adding additional servers to the pool of backend servers, HAProxy may be readily scaled. It can adjust to increasing traffic needs by dynamically reconfiguring load balancing without experiencing any downtime, guaranteeing the system's responsiveness and dependability as it grows horizontally.

2.2.4 Use Cases

- **Web application load balancing:** dividing up website traffic over several web servers to guarantee reliable performance and high availability.
- **Database load balancing:** distributing queries among several database replicas in order to improve performance and balance the load.
- **API Gateway:** serving as an API gateway for microservice architectures, enabling traffic to be redirected to various services according to headers, URL routes, and other specifications.

2.3 StoRM WebDAV overview

StoRM WebDAV is a critical component of the StoRM (Storage Resource Manager) suite, designed to provide efficient data transfer and management through the WebDAV (Web Distributed Authoring and Versioning) protocol. StoRM WebDAV enables users to manage files stored on distributed storage systems, such as IBM GPFS and Lustre, by offering a lightweight management layer over a POSIX-compliant file system. It is widely used in grid and cloud computing environments, where large volumes of data need to be stored, transferred, and managed across distributed infrastructures.

2.3.1 Key Features of StoRM WebDAV

WebDAV Protocol Integration

The HTTP protocol has an extension called WebDAV that lets users add, remove, move, and manage resources on a web server. By utilizing this protocol, StoRM WebDAV makes browser-based data management possible, streamlining communications with intricate storage systems. Researchers, scientists, and engineers working in contexts requiring effective data management and storage across multiple locations will find this especially helpful.

Flexible Authentication and Authorization

StoRM WebDAV supports multiple authentication methods to ensure secure access to storage resources, including:

- X.509 Certificates
- JWT tokens from OAuth/OpenID Connect (OIDC)
- VOMS (Virtual Organization Membership Service) proxies
- File access rights are enforced via POSIX Access Control Lists (ACLs), ensuring compliance with. POSIX standards

Third Party Copy (TPC) Support

Support for Third Party Copies (TPC), which enables effective bulk data transfers between two distant storage endpoints, is a special feature of StoRM WebDAV. By extending the WebDAV COPY verb, this feature allows for seamless data replication or migration between dispersed systems without the need for direct user engagement.

Integration with Hierarchical Storage Management (HSM) Systems

StoRM WebDAV integrates with GEMSS Hierarchical Storage Management (HSM) technology, allowing efficient management of disk- and tape-based storage. Key integrations include:

- IBM GPFS
- IBM Tivoli Storage Manager (TSM)
- StoRM backend

This comprehensive solution supports high-performance disk storage and cost-effective tape archives.

2.3.2 StoRM WebDAV architecture

StoRM WebDAV's architecture design combines a number of essential parts with outside services to provide great performance, scalability, and manageability. The design maintains flexibility and strong security measures while simplifying the system as a whole by utilizing well-known third-party components like NGINX.

The architecture of StoRM WebDAV was designed with the following motivations:

- **Simplification of the Codebase:** StoRM WebDAV minimizes coding complexity by externalizing common functionality including authentication (AuthN), TLS termination, and maybe authorization (AuthZ). This makes it possible for the team to concentrate on key functions while relying on outside components to complete jobs that are better served by well-established solutions.

- **Reusability of Deployment Models:** Reusability was taken into consideration while designing the architecture. The team's other solutions, such INDIGO IAM and StoRM Tape, can use the same concept, which guarantees a scalable and uniform approach across many services.
- **Performance Enhancement:** The system depends on external components—like NGINX—that are made expressly to handle HTTP GET and PUT requests in an effective manner in order to increase the speed of data transfers.
- **Scalability:** With the help of readily scalable external services like NGINX, the architecture makes sure that StoRM WebDAV can expand to accommodate growing demand without requiring a major overhaul of the underlying system.

2.3.3 NGINX Role

The open-source HTTP server and reverse proxy NGINX is a key component of the architecture of StoRM WebDAV. Because NGINX is well-known for its great performance, reliability, ease of configuration, and minimal resource usage, it was selected as the perfect component for processing HTTP requests on a large scale.

Key responsibilities of NGINX in the architecture include:

- **HTTP Request Handling:** NGINX efficiently manages incoming HTTP requests, including GET and PUT operations, which are critical for data transfer in StoRM WebDAV.
- **TLS Termination:** NGINX handles the termination of TLS connections, offloading this resource-intensive process from StoRM WebDAV and enhancing overall performance.
- **Flexible Authentication:** NGINX provides flexible authentication via custom modules, including the **ngx_http_voms_module** developed specifically for this architecture.

A dedicated NGINX module called **ngx_http_voms_module** has been developed to facilitate client-side authentication using X.509 certificates and VOMS (Virtual Organization Membership Service) proxy certificates. The module:

- **Enables client authentication** based on X.509 and VOMS proxy certificates, ensuring secure access to resources.

- **Validates VOMS proxies** and extracts important attributes from the VOMS Attribute Certificate (AC), such as Fully Qualified Attribute Names (FQANs), which are then embedded into NGINX variables.
- **AuthZ delegation:** These VOMS attributes are forwarded to the WebDAV component, which remains responsible for making authorization (AuthZ) decisions.

This modular approach ensures secure, fine-grained access control, while leveraging NGINX's performance capabilities for handling the authentication process.

This architecture's main objective is to give NGINX as much of the data transfer duty as possible, freeing up StoRM WebDAV to concentrate on essential features like authorization (AuthZ) decision-making.

In the current implementation, NGINX handles HTTP GET request processing for StoRM WebDAV.

The HTTP X-Accel-Redirect response header is used in conjunction with an internal redirect mechanism to do this. After processing a file request, StoRM WebDAV sends an internal redirect to NGINX, which gets the file and sends it to the client.

- Example: X-Accel-Redirect: /internal/{file-path}

The system's scalability and performance are enhanced by this architecture, which enables NGINX to handle the actual data transport. While StoRM WebDAV transfers the resource-intensive activities to NGINX, it maintains control over important authorization decisions.

2.4 Architecture design

Two instances of StoRM WebDAV are set up behind a load balancer. These StoRM WebDAV instances handle HTTP requests such as GET and PUT for distributed storage systems. The load balancer is introduced to distribute the incoming traffic evenly across both StoRM WebDAV instances, ensuring optimal resource utilization and preventing any single instance from becoming overwhelmed.

- First Configuration (NGINX as Load Balancer): In the first step, NGINX is used as the load balancer. NGINX provides a variety of load balancing algorithms, such as round-robin and least connections, which are configured to handle the incoming traffic. NGINX is known for its high performance and flexibility, making it a suitable choice for balancing traffic between the StoRM WebDAV instances.

- Second Configuration (HAProxy as Load Balancer): After testing with NGINX, HAProxy is introduced as the load balancer. HAProxy is a specialized tool designed specifically for load balancing with advanced traffic management features such as session persistence, health checks, and fine-grained control over traffic distribution.

By comparing HAProxy to NGINX in this basic deployment, the performance, scalability, and traffic management capabilities of each tool are evaluated.

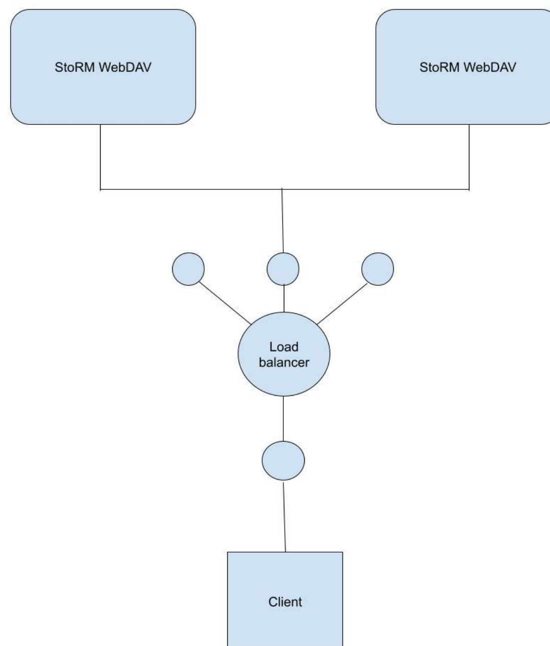

Figure2.2: comprehensive scheme of the overall architecture design

2.5 Design problem

One of the primary issues in the current deployment architecture is that all network traffic passes through the load balancer, whether it's NGINX or HAProxy. While this can introduce a potential bottleneck - limiting overall network bandwidth to the capacity of the load balancer - there are significant benefits to this setup. The load balancer efficiently distributes traffic, ensuring that each server is optimally utilized, improving fault tolerance, and enabling automatic failover in case of server failure. Centralized management simplifies scaling and enhances resilience.

It is possible to use DNS, which is the system used to translate domain names into IP addresses, to distribute traffic between different servers. Multiple IP addresses can be assigned to the same DNS record by permuting the sequence each time a DNS request is received. This solution is very simple to set up and does not require specialized hardware or software. It has, however, several limitations. In fact, DNS responses are often cached, so subsequent requests receive the same IP instead of different IPs. It is also a static algorithm that ignores server load, assumes that all requests are computationally equivalent and that all servers have equivalent capacities.

In contrast, using a DNS round-robin approach, the client connects directly to individual servers, and the network bandwidth becomes the sum of the capacities of all the servers. Bypassing the centralized load balancer, clients can take advantage of the full combined bandwidth of the servers, improving raw performance in terms of data throughput. However, this approach comes with notable drawbacks. Traffic distribution is less efficient, manual failover handling is required, and there is no automated monitoring of server health, making the system less reliable and harder to scale.

In our case, the load balancer approach would be the best option, as it ensures efficient resource usage, better system resilience, and simplified growth, even if it requires careful attention to the capacity of the load balancer to avoid potential bottlenecks.

Chapter 3: Components Configuration

The objective of this work is to set up a system architecture using OpenStack to deploy NGINX or HAproxy load balancers along with StoRM WebDAV instances. This deployment is part of an ongoing comparison between NGINX and HAProxy in a distributed system architecture. The deployment involves multiple virtual machines (VMs) hosted on OpenStack and connected through private and public networks.

3.1 StoRM WebDAV Deployment

SSH Key Generation

The first step involved creating an SSH key pair to facilitate secure access to the VMs. This was achieved using the following command:

```
ssh-keygen -t rsa -b 4096
```

VM Creation

Two VMs were created for StoRM WebDAV. Each VM is assigned a private IP and a floating IP from the tenant's network in OpenStack. Floating IPs assigned are:

- **StoRM WebDAV1:** 131.154.99.230
- **StoRM WebDAV2:** 131.154.98.59

Network Configuration

- **SSH Security Group:** Configured to allow inbound connections on port 22.
- **HTTP and HTTPS Security Groups:**
  - **HTTP (Port 8085):** Allows unencrypted web traffic.
  - **HTTPS (Port 8443):** Allows secure, encrypted web traffic.

Volume Creation and Attachment

Volumes were created and attached to each VM for persistent storage.

Volume Formatting and Mounting

After attaching, the volumes were formatted using ext4 and mounted to the /mnt directory in each VM using following commands:

```
ssh ubuntu@<floating_ip>
mkfs.ext4 /dev/vdb
sudo mount /dev/vdb /mnt
```

SSL/TLS Setup

Two SSL certificates, issued by a trusted CA, were created for each StoRM WebDAV instance to enable secure HTTPS connections.

3.1.1 StoRM WebDAV Installation

Since StoRM WebDAV relies on Java, the next step involved installing Java on both StoRM WebDAV VMs. The following command was used to install OpenJDK 11 and wget on both machines:

```
sudo dnf install -y java-11-openjdk wget
```

This installed the necessary Java runtime environment, enabling StoRM WebDAV to function properly.

After installing Java, the StoRM WebDAV repository was added to the system. This was achieved by running the following command to install the StoRM WebDAV repository:

```
sudo dnf install https://repo.cloud.cnaf.infn.it/repository/storm-rpm-stable/redhat9/storm-webdav-1.4.2-1.el9.noarch.rpm
```

Once the repository was added, StoRM WebDAV was installed with the following command:

```
sudo dnf install storm-webdav
```

3.1.2 StoRM WebDAV Configuration

- **StoRM WebDAV1 Configuration File:** This file defines essential service settings for the StoRM WebDAV instance, including network bindings, SSL certificate paths, and performance parameters. Details of the configuration file located in /etc/systemd/system/storm-webdav.service.d/storm-webdav.conf:

```
[Service]

Environment="STORM_WEBDAV_USER=storm"

Environment="STORM_WEBDAV_JVM_OPTS=-Xms1024m -Xmx1024m"


# Bind address and hostname

Environment="STORM_WEBDAV_SERVER_ADDRESS=0.0.0.0"

Environment="STORM_WEBDAV_HOSTNAME_0=vm-131-154-99-
230.cloud.cnaf.infn.it"


# Ports for HTTPS and HTTP

Environment="STORM_WEBDAV_HTTPS_PORT=8443"

Environment="STORM_WEBDAV_HTTP_PORT=8085"


# SSL certificate and key paths

Environment="STORM_WEBDAV_CERTIFICATE_PATH=/etc/grid-secu-
rity/storm-webdav/hostcert.pem"

Environment="STORM_WEBDAV_PRIVATE_KEY_PATH=/etc/grid-secu-
rity/storm-webdav/hostkey.pem"


# Storage area configuration

Environment="STORM_WEBDAV_SA_CONFIG_DIR=/etc/storm/webdav/sa.d"


# Should StoRM WebDAV always require a valid client certificate
on the HTTPS endpoint?


# Set to 'false' if you want token-based authentication to work
(and thus third-party copy)
  Environment="STORM_WEBDAV_REQUIRE_CLIENT_CERT=false"
```

- **StoRM WebDAV2 Configuration File:** Similar to StoRM WebDAV1, except for differences in the hostname configuration:

```
Environment="STORM_WEBDAV_HOSTNAME_0=vm-131-154-
Y.Y.cloud.cnaf.infn.it"
```

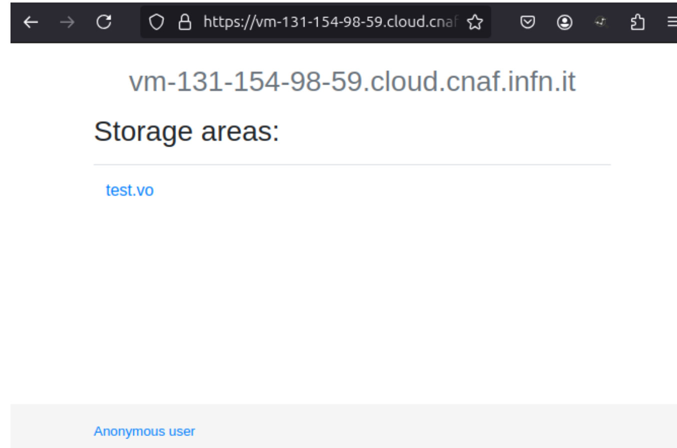

Figure 3.1: picture of StoRM WebDAV page

3.1.3 Puppet installation

Puppet was used to manage certificate configurations for StoRM WebDAV, ensuring consistent and automated setup on both virtual machines. After installing Puppet and the necessary modules for certificate management and user configurations, a manifest file (manifest.pp) was created on each VM. This file included definitions for setting up certificate authority repositories, configuring user groups, and ensuring the latest packages were installed. The manifest was applied to both VMs using Puppet, streamlining the process and ensuring proper certification configuration for StoRM WebDAV in a secure and efficient manner.

3.2 NGINX Deployment

VM Creation

One VM created for NGINX with floating IP: 131.154.99.77

Network Configuration

SSH (port 22), HTTP (port 80), and HTTPS (port 443) security groups were applied.

Volume Creation, Attachment, Formatting, and Mounting

Same as for StoRM WebDAV VMs.

SSL/TLS Setup

A self-signed SSL certificate was generated for testing purposes using the command:

```
openssl req -x509 -nodes -days 365 -newkey rsa:2048 -keyout nginx.key
-out nginx.crt
```

3.2.1 NGINX Installation

First of all the package index is up to date :

```
sudo apt update

sudo apt upgrade -y
```

Installing NGINX using following command and in the end checking it is up and running:

```
sudo dnf install nginx -y

sudo systemctl start nginx

sudo systemctl enable nginx

sudo systemctl status nginx
```

3.2.2 NGINX Configuration

Configuration Files Overview

http.conf (in etc/nginx/conf.d/http.conf):

This file defines the **upstream block**, which lists the StoRM WebDAV instances. It enables NGINX to act as a load balancer.

```
upstream stormwebdav-backend {
    server vm-131-154-99-230.cloud.cnaf.infn.it:8443;
    server vm-131-154-98-59.cloud.cnaf.infn.it:8443;
}
server {
    listen 80;
    server_name nginx.com;

    location / {
        return 301 https://$host$request_uri; # Redirect HTTP to HTTPS
    }
}
```

nginx.com.conf (in /etc/nginx/sites-available/nginx.com.conf):

Configures HTTPS with a self-signed certificate and sets up proxy settings for secure backend communication.

```
server {
    listen 443 ssl;
    server_name nginx.com;

    # SSL Configuration
    ssl_certificate /home/ubuntu/cert.crt;
    ssl_certificate_key /home/ubuntu/cert.key;

    client_max_body_size 100M; # Set maximum upload size

    location / {
        proxy_pass https://stormwebdav-backend; # Forward requests to
the upstream block
        proxy_ssl_verify off; # Disable SSL verification for self-
signed certs
```

```
        proxy_set_header Host $host;
        proxy_set_header X-Forwarded-Proto https;
        proxy_set_header X-Real-IP $remote_addr; # Pass client IP to
the backend
        proxy_read_timeout 60;
        proxy_connect_timeout 60;

        # Add a custom header to track backend server responses
        add_header X-Upstream-Address $upstream_addr;
    }
}
```

nginx.conf (in /etc/nginx/nginx.conf):

This is the main NGINX configuration file. Following best practices, it includes the custom configuration files (http.conf and nginx.com.conf) without modifying the main structure.

```
http {
    include /etc/nginx/conf.d/*.conf;
    include /etc/nginx/sites-enabled/*;
}
```

3.3 HAProxy Deployment

VM Creation

One VM created for HAProxy with floating IP: 131.154.98.99.

Network Configuration

SSH, HTTP, and HTTPS security groups were applied as for NGINX.

Volume Creation, Attachment, Formatting, and Mounting

Same as for StoRM WebDAV VMs.

SSL/TLS Setup

A self-signed SSL certificate was generated for testing purposes using the command:

```
openssl req -x509 -nodes -days 365 -newkey rsa:2048 -keyout nginx.key
-out nginx.crt
```

3.3.1 HAproxy Installation

Once the virtual machine is ready, establish a connection to it via SSH and install HAProxy using the following commands:

```
Sudo dnf update

sudo dnf install haproxy
```

During the HAProxy setup, certain adjustments were necessary to ensure proper functionality due to security and configuration constraints. SELinux was in "enforcing" mode, which restricted HAProxy's network connections. To change it to "permissive" mode permanently, the SELinux configuration file was updated:

```
sudo vim /etc/selinux/config
```

Update it to:

SELINUX=permissive

Then, add Port 80 to the Firewall:

sudo firewall-cmd --add-port=80/tcp --permanent

sudo firewall-cmd --reload

3.3.2 HAProxy Configuration

This configuration ensures a functional HAProxy setup for load balancing with secure SSL/TLS communication.

Global Settings: Sets up basic HAProxy environment, logging, and SSL policies.

```
global
    log 127.0.0.1 local0
    chroot /var/lib/haproxy
    pidfile /var/run/haproxy.pid
    maxconn 4000
    user haproxy
    group haproxy
    daemon
    stats socket /var/lib/haproxy/stats
    ssl-default-bind-ciphers PROFILE=SYSTEM
    ssl-default-server-ciphers PROFILE=SYSTEM
```

Default Settings: Applies common defaults for all frontend and backend sections.

```
defaults
    mode http
    log global
    option httplog
    option forwardfor
    timeout connect 10s
    timeout client 1m
    timeout server 1m
    maxconn 3000
```

Frontend Configuration: Handles incoming traffic on port 443 (SSL-enabled) and forwards it to the backend.

```
frontend haproxy
    bind *:443 ssl crt /etc/ssl/haproxy.pem
    default_backend stormwebdav
    stats uri /haproxy?stats
```

Backend Configuration: Distributes traffic to StoRM WebDAV instances using the roundrobin algorithm.

```
backend stormwebdav
    balance roundrobin
    mode http
    option httpchk GET /actuator/health
    server stormwebdav1 131.154.99.230:8443 check ssl verify none
    server stormwebdav2 131.154.98.59:8443 check ssl verify none
```

3.4 Client Deployment

Purpose of the Client VM

- Used to generate requests and test the performance of StoRM WebDAV instances behind NGINX and HAProxy.
- No floating IP assigned; communication is internal within the private network.

Network Configuration

- Assigned a private IP within the tenant's network.
- Security groups configured to allow necessary traffic between client and StoRM WebDAV instances.

Chapter 4: Performance Testing

4.1 File download test

To benchmark the performance of direct and load-balanced file downloads, a file named *pippo.txt* was created. This 1GB file serves as a standard test file for evaluating download speeds and resource utilization across different configurations, including direct access to StoRM WebDAV, access via NGINX, and access via HAProxy. By using a large, consistent file, we can observe variations in data transfer rates and resource demands on StoRM WebDAV in each scenario.

The purpose of this test is to compare the performance of direct access to StoRM WebDAV versus access through load balancers (NGINX and HAProxy) in a distributed environment. Key metrics include download speed, CPU, and memory usage on StoRM WebDAV instances.

4.1.1 Direct Access to StoRM WebDAV

We created a file in StoRM WebDAV of 1GB.

```
dd if=/dev/random of=pippo.txt count=1000 bs=1000
```

Download Command:

```
curl -O -k -w "Time: %{time_total}s\n"
https://131.154.99.230:8443/test.vo/pippo.txt
```

Top command to measure the StoRM WebDAV CPU and memory:

```
top -b -d 1 -p 138003 >> stormwebdav_ usage.log
```

the table summarizing the Direct Access to StoRM WebDAV performance and resource usage:

Metric	Details
Download Speed	~324 MB/s
Time Taken	3.87 seconds
CPU Usage (%)	us: 18.8, sy: 9.4, id: 66.1
Memory (MiB)	Total: 3659.6, Free: 302.4, Used: 2007.3, Buff/Cache: 1659.5
PID 138003 (storm)	CPU: 62.4%, MEM: 39.7%, VIRT: 3827832 KiB, RES: 1.4 GiB, SHR: 30364 KiB, TIME: 6:20.48, COMMAND: java

Table 4.1: Direct Access to StoRM WebDAV performance and resource usage

Direct access provides the highest download speed but places a heavy load on the StoRM WebDAV server's CPU, impacting the server's ability to handle multiple requests efficiently.

4.1.2 Access via NGINX Load Balancer

Download Command:

```
curl -O -k -w "Time: %{time_total}s\n"
https://131.154.99.77:443/test.vo/pippo.txt
```

Top command to measure the NGINX CPU and memory:

```
top -b -d 1 -p 138003 >> nginx_usage.log
```

the table summarizing the performance and resource usage for Access via NGINX Load Balancer:

Metric	Details
Download Speed	~114 MB/s
Time Taken	8.95 seconds
CPU Usage (%)	us: 0.0, sy: 0.0, id: 100.0
Memory (MiB)	Total: 3659.6, Free: 302.7, Used: 2007.0, Buff/Cache: 1659.6
PID 138003 (storm)	CPU: 2.0%, MEM: 39.7%, VIRT: 3827832 KiB, RES: 1.4 GiB, SHR: 30364 KiB, TIME: 6:23.89, COMMAND: java

Table 4.2: performance and resource usage for Access via NGINX Load Balancer

Using NGINX as a load balancer reduces the CPU burden on StoRM WebDAV, suggesting that NGINX is handling the processing overhead. However, this approach introduces latency, likely due to NGINX's processing and SSL termination.

Metric	Details
System Uptime	4 days, 21:56
Load Average	0.13, 0.03, 0.01
Tasks	2 total (1 running, 1 sleeping)
CPU Usage (%)	us: 20.6, sy: 17.5, id: 47.9, si: 13.9
Memory (MiB)	Total: 3911.9, Free: 3114.2, Used: 218.0, Buff/Cache: 579.7
Swap Memory (MiB)	Total: 0.0, Free: 0.0, Used: 0.0, Avail: 3461.6
PID 4806 (www-data)	CPU: 100.0%, MEM: 0.2%, VIRT: 57248 KiB, RES: 9080 KiB, SHR: 5680 KiB, TIME: 0:47.03
PID 4805 (www-data)	CPU: 0.0%, MEM: 0.2%, VIRT: 57368 KiB, RES: 9580 KiB, SHR: 5996 KiB, TIME: 0:23.79

Table 4.3: Nginx CPU and MEMORY usage

4.1.3 Access via HAProxy Load Balancer

- Download Command:

```
curl -O -k -w "Time: %{time_total}s\n"
https://131.154.98.99:443/test.vo/pippo.txt
```

Top command to measure the HAproxy CPU and memory:

```
top -b -d 1 -p 138003 >> haproxy_usage.log
```

Metric	Details
Download Speed	~284 MB/s
Time Taken	3.59 seconds
CPU Usage (%)	us: 0.0, sy: 0.0, id: 100.0
Memory (MiB)	Total: 3659.6, Free: 847.2, Used: 1985.5, Buff/Cache: 1126.6
PID 138003 (storm)	CPU: 2.0%, MEM: 39.7%, VIRT: 3827832 KiB, RES: 1.4 GiB, SHR: 30364 KiB, TIME: 6:27.79, COMMAND: java

Table 4.4: performance and resource usage for Access via HAproxy

Metric	Details
System Uptime	6 days, 19:35
Load Average	0.08, 0.02, 0.01
Tasks	1 total (1 running)
CPU Usage (%)	us: 15.4, sy: 14.9, id: 51.3, si: 18.5
Memory (MiB)	Total: 3659.6, Free: 3078.8, Used: 440.5, Buff/Cache: 380.2
Swap Memory (MiB)	Total: 0.0, Free: 0.0, Used: 0.0, Avail: 3219.1
PID 4239 (haproxy)	CPU: 95.0%, MEM: 0.3%, VIRT: 99656 KiB, RES: 9948 KiB, SHR: 6144 KiB, TIME: 1:53.00

Table 4.5: HAproxy CPU and MEMORY usage

HAProxy achieves a download speed close to that of direct access while maintaining low CPU usage on StoRM WebDAV. This suggests that HAProxy is more efficient for high-speed file transfers in this configuration.

4.1.4 Performance Analysis

Direct Access vs. Load Balancers:

- Direct access is faster but heavily loads StoRM WebDAV's CPU, which can affect other services.
- NGINX and HAProxy reduce CPU usage by offloading processing, though they add some latency.

NGINX Performance:

- Shows additional overhead in handling SSL and connection management. Adjustments like increasing buffer sizes or enabling caching could improve throughput.

HAProxy Efficiency:

- Performs close to direct access speeds, making it more suitable for high-speed file transfers and efficient load balancing.

Resource Usage:

- CPU: Both NGINX and HAProxy demonstrate high CPU usage in high-throughput scenarios, with NGINX at 100% and HAProxy at 95%.
- Memory: Lightweight usage observed (~0.2% for NGINX, ~0.3% for HAProxy).

- Interrupts: Moderate load on network interrupts, with NGINX showing 13.9% and HAProxy at 18.5%.

Implications:

- Both load balancers perform efficiently in memory usage but reach high CPU utilization. Scaling and monitoring are essential for handling larger loads.

4.1.5 Recommendations for Optimization

NGINX Optimization:

- Increase buffer sizes and enable caching to reduce overhead.
- Adjust worker processes and worker connections to match the CPU cores for improved concurrency.

HAProxy Optimization:

- Leverage HAProxy's lightweight nature for high-performance tasks like large file transfers.
- Configure nbproc to utilize multi-core CPUs effectively.

Resource Scaling:

- Add more StoRM WebDAV instances to distribute the load and prevent resource saturation.
- Ensure load balancers have sufficient CPU capacity or consider scaling them horizontally.

Network Optimization:

- Monitor and ensure adequate bandwidth to prevent bottlenecks during high-speed transfers.

Further Monitoring and Tuning:

- Use profiling tools like prmon or perf to identify bottlenecks and fine-tune configurations for sustained high loads.

4.2 Stress testing with Vegeta

The Vegeta tool was installed on the client machine to conduct stress tests on the system. Vegeta is a versatile and fast HTTP load-testing tool used for benchmarking and performance evaluation.

It is utilized to simulate traffic and analyze the behavior of the load balancers and StoRM WebDAV instances under various conditions.

Below is the command used to execute the Vegeta test:

```
vegeta attack -targets=targets.txt -rate=100 -duration=30s | veg-
eta report
```

In this example, 100 requests per second are sent for 30 seconds, and a performance report is generated.

We created a file in StoRM WebDAV of 1MB with the following command:

```
dd if=/dev/random of=pippo.txt count=1000 bs=1000
```

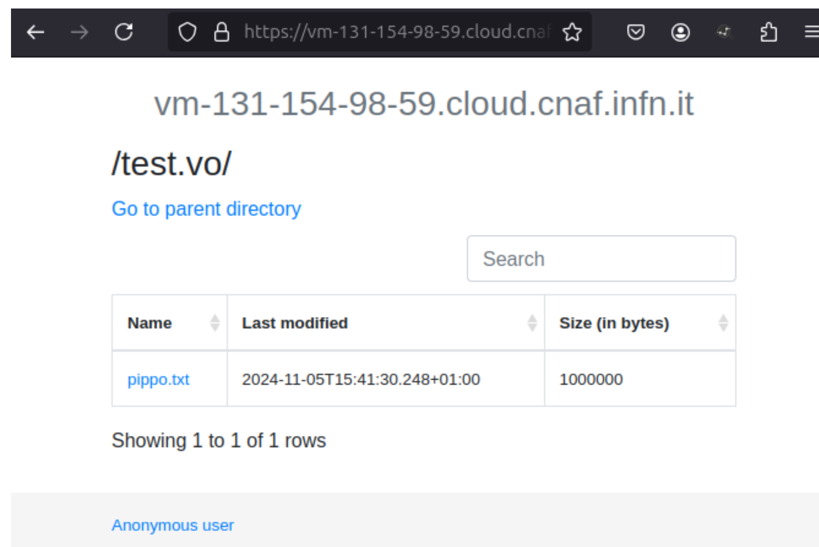

Figure 1 creation of the test file of 1 MB

4.2.1 Direct access to StoRM WebDAV

FIRST TEST: duration=10s, rate=10

On the client machine:

```
[almalinux@kazemi-client ~] $ echo "GET
https://131.154.99.230:8443/test.vo/pippo.txt" | vegeta attack -inse-
cure -duration=10s -rate=10 -output=results.bin
```

Show the results on the terminal:

```
[almalinux@kazemi-client ~]$ vegeta report < results.bin
```

Metric	Details
Requests	Total: 100, Rate: 10.10 req/s, Throughput: 10.09 req/s
Duration (s)	Total: 9.909, Attack: 9.9, Wait: 9.06ms
Latencies (ms)	Min: 5.84, Mean: 8.464, 50th: 8.291, 90th: 10.015, 95th: 10.539, 99th: 16.446, Max: 19.607
Bytes In	Total: 100,000,000, Mean: 1,000,000.00
Bytes Out	Total: 0, Mean: 0.00
Success Ratio	100.00%
Status Codes	200: 100
Error Set	None

Table 4.6: Request performance metric D=10, R=10

Generate a plot:

```
[almalinux@kazemi-client ~]$ vegeta plot < results.bin > plot.html
```

On my local pc, I download the plot with this command:

```
$ scp -o ProxyJump=almalinux@131.154.99.230 alma-
linux@192.168.1.192:/home/almalinux/plot.html /home/mypc/Desktop
```

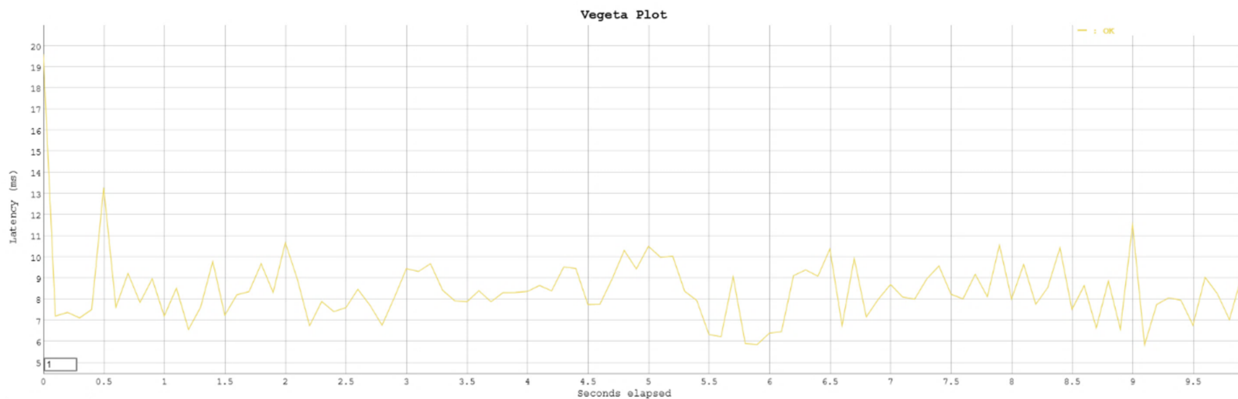

Figure 4.2: Vegeta plot D=10, R=10

CPU and memory usage with top command:

Metric	Details
Load Average	0.00, 0.01, 0.00
Tasks	1 total (0 running, 1 sleeping)
CPU Usage (%)	us: 3.5, sy: 0.5, id: 95.5, si: 0.5
Memory (MiB)	Total: 3659.6, Free: 1447.8, Used: 2003.4, Buff/Cache: 507.0
Swap Memory (MiB)	Total: 0.0, Free: 0.0, Used: 0.0, Avail: 1656.2
PID 138003 (storm)	CPU: 10.0%, MEM: 39.7%, VIRT: 3829880 KiB, RES: 1.4 GiB, SHR: 30364 KiB, TIME: 8:03.94, COMMAND: java

Table 4.7: system and process information D=10, R=10

SECOND TEST: duration=10s, rate=50

On the client machine:

```
[almalinux@kazemi-client ~] $ echo "GET
https://131.154.99.230:8443/test.vo/pippo.txt" | vegeta attack -inse-
cure -duration=10s -rate=50 -output=results.bin
```

Show the results on the terminal:

```
[almalinux@kazemi-client ~] $ vegeta report < results.bin
```

Metric	Details
Requests	Total: 500, Rate: 50.10 req/s, Throughput: 50.07 req/s
Duration (s)	Total: 9.985, Attack: 9.98, Wait: 5.235ms
Latencies (ms)	Min: 3.883, Mean: 5.946, 50th: 5.715, 90th: 7.372, 95th: 7.856, 99th: 10.461, Max: 20.352
Bytes In	Total: 500,000,000, Mean: 1,000,000.00
Bytes Out	Total: 0, Mean: 0.00
Success Ratio	100.00%
Status Codes	200: 500
Error Set	None

Table 4.8: Request performance metric D=10, R=5

Generate a plot:

```
[almalinux@kazemi-client ~]$ vegeta plot < results.bin > plot.html
```

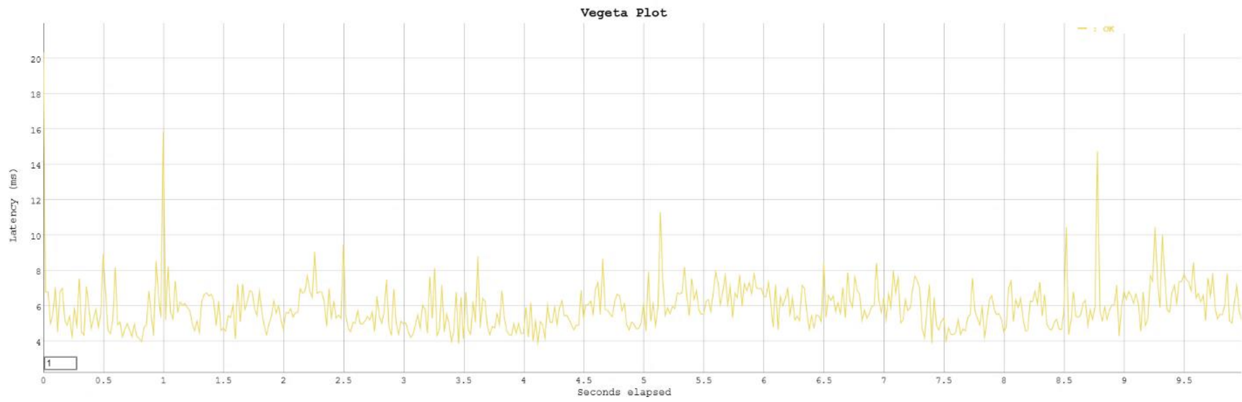

Figure 4.3: Vegeta plot D=10, R=50

CPU and memory usage with top command:

Metric	Details
Load Average	0.00, 0.00, 0.00
Tasks	1 total (0 running, 1 sleeping)
CPU Usage (%)	us: 10.1, sy: 2.5, id: 86.4, si: 1.0
Memory (MiB)	Total: 3659.6, Free: 1422.9, Used: 2027.9, Buff/Cache: 507.4
Swap Memory (MiB)	Total: 0.0, Free: 0.0, Used: 0.0, Avail: 1631.7
PID 138003 (storm)	CPU: 27.0%, MEM: 39.7%, VIRT: 3829880 KiB, RES: 1.4 GiB, SHR: 30364 KiB, TIME: 8:08.85, COMMAND: java

Table 4.9: system and process information D=10, R=50

THIRD TEST: duration=10s, rate=100

On the client machine:

```
[almalinux@kazemi-client ~]$ echo "GET
https://131.154.99.230:8443/test.vo/pippo.txt" | vegeta attack -inse-
cure -duration=10s -rate=100 -output=results.bin
```

Show the results on the terminal:

```
[almalinux@kazemi-client ~]$ vegeta report < results.bin
```

Metric	Details
Requests	Total: 1000, Rate: 100.10 req/s, Throughput: 100.03 req/s
Duration (s)	Total: 9.997, Attack: 9.99, Wait: 6.932ms
Latencies (ms)	Min: 3.732, Mean: 6.704, 50th: 6.322, 90th: 8.353, 95th: 9.088, 99th: 17.383, Max: 40.631
Bytes In	Total: 1,000,000,000, Mean: 1,000,000.00
Bytes Out	Total: 0, Mean: 0.00
Success Ratio	100.00%
Status Codes	200: 1000

Table 4.10: Request performance metric D=10, R=100

Generate a plot:

```
[almalinux@kazemi-client ~]$ vegeta plot < results.bin > plot.html
```

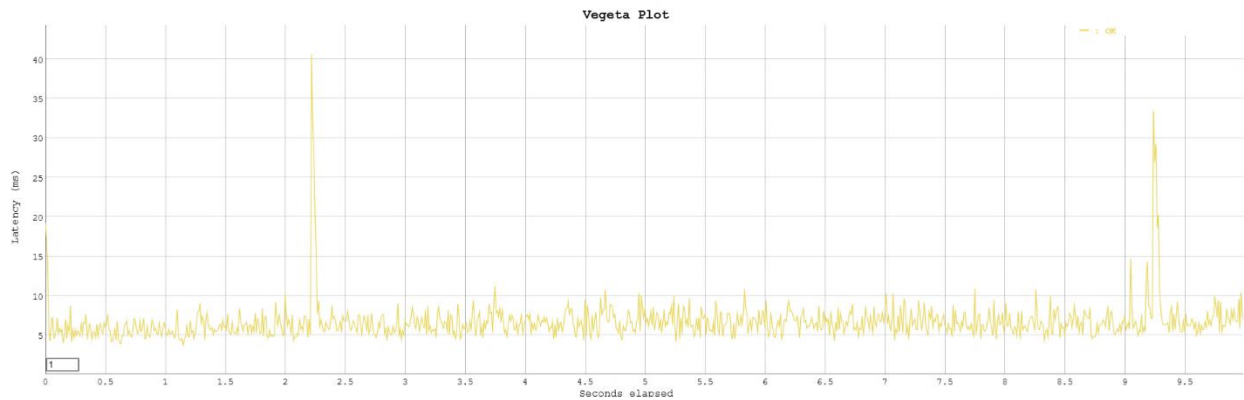

Figure 4.4: Vegeta plot D=10, R=100

CPU and memory usage with top command:

Metric	Details
Load Average	0.19, 0.09, 0.03
Tasks	1 total (0 running, 1 sleeping)
CPU Usage (%)	us: 29.8, sy: 4.5, id: 63.1, si: 2.5
Memory (MiB)	Total: 3659.6, Free: 1397.8, Used: 2051.6, Buff/Cache: 508.7
Swap Memory (MiB)	Total: 0.0, Free: 0.0, Used: 0.0, Avail: 1608.0
PID 138003 (storm)	CPU: 72.0%, MEM: 39.7%, VIRT: 3829880 KiB, RES: 1.4 GiB, SHR: 30364 KiB, TIME: 8:31.79, COMMAND: java

Table 4.11: system and process information D=10, R=100

LAST TEST: duration=10s, rate=500 (Error)

An error occurred because of the high number of requests!

On the client machine:

```
[almalinux@kazemi-client ~]$ echo "GET
https://131.154.99.230:8443/test.vo/pippo.txt" | vegeta attack -inse-
cure -duration=10s -rate=500 -output=results.bin
```

Show the results on the terminal:

```
[almalinux@kazemi-client ~]$ vegeta report < results.bin
```

Metric	Details
Requests	Total: 5000, Rate: 500.08 req/s, Throughput: 94.93 req/s
Duration (s)	Total: 21.404, Attack: 9.998, Wait: 11.406
Latencies (ms)	Min: 36.443, Mean: 5803, 50th: 5367, 90th: 12298, 95th: 13067, 99th: 17193, Max: 18157
Bytes In	Total: 2,032,000,000, Mean: 406,400.00
Bytes Out	Total: 0, Mean: 0.00
Success Ratio	40.64%
Status Codes	0: 2968, 200: 2032
Error Set	Get "https://131.154.99.230:8443/test.vo/pippo.txt": EOF

Table 4.12: Request performance metric D=10, R=500

Generate a plot:

```
[almalinux@kazemi-client ~]$ vegeta plot < results.bin > plot.html
```

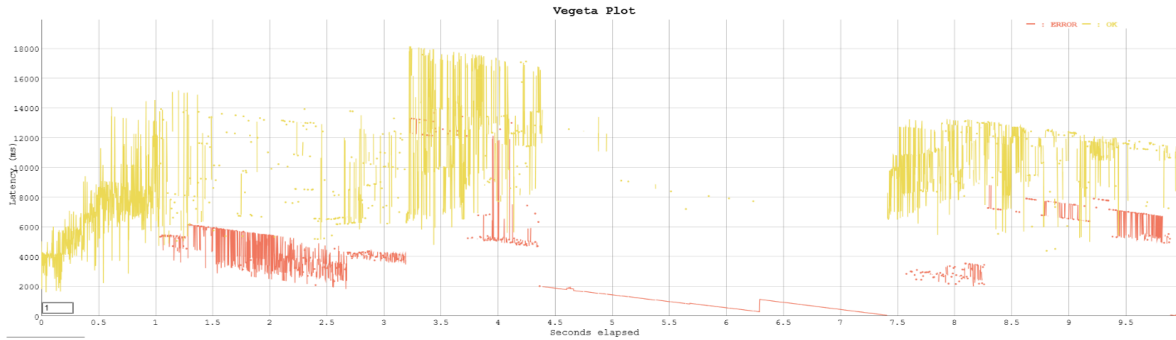

Figure 4.5: Vegeta plot D=10, R=500

CPU and memory usage with top command:

Metric	Details
Load Average	9.04, 2.50, 0.99
Tasks	1 total (0 running, 1 sleeping)
CPU Usage (%)	us: 90.5, sy: 3.5, id: 0.0, si: 6.0
Memory (MiB)	Total: 3659.6, Free: 988.7, Used: 2473.0, Buff/Cache: 519.5
Swap Memory (MiB)	Total: 0.0, Free: 0.0, Used: 0.0, Avail: 1186.6
PID 138003 (storm)	CPU: 199.0%, MEM: 50.5%, VIRT: 4236816 KiB, RES: 1.8 GiB, SHR: 30364 KiB, TIME: 9:39.40, COMMAND: java

Table 4.13: system and process information D=10, R=500

4.2.2 Access via NGINX load balancer

FIRST TEST: duration=10s, rate=10

On the client machine:

```
[almalinux@kazemi-client ~]$ echo "GET
https://131.154.99.77:443/test.vo/pippo.txt" | vegeta attack -insecure
-duration=10s -rate=10 -output=results.bin
```

Show the results on the terminal:

```
[almalinux@kazemi-client ~]$ vegeta report < results.bin
```

Metric	Details
Requests	Total: 100, Rate: 10.10 req/s, Throughput: 10.09 req/s
Duration (s)	Total: 9.911, Attack: 9.9, Wait: 11.161ms
Latencies (ms)	Min: 8.757, Mean: 18.136, 50th: 10.805, 90th: 12.701, 95th: 26.304, 99th: 281.767, Max: 339.085
Bytes In	Total: 100,000,000, Mean: 1,000,000.00
Bytes Out	Total: 0, Mean: 0.00
Success Ratio	100.00%
Status Codes	200: 100
Error Set	None

Table 4.14: Request performance metric D=10, R=10

Generate a plot:

```
[almalinux@kazemi-client ~]$ vegeta plot < results.bin > plot.html
```

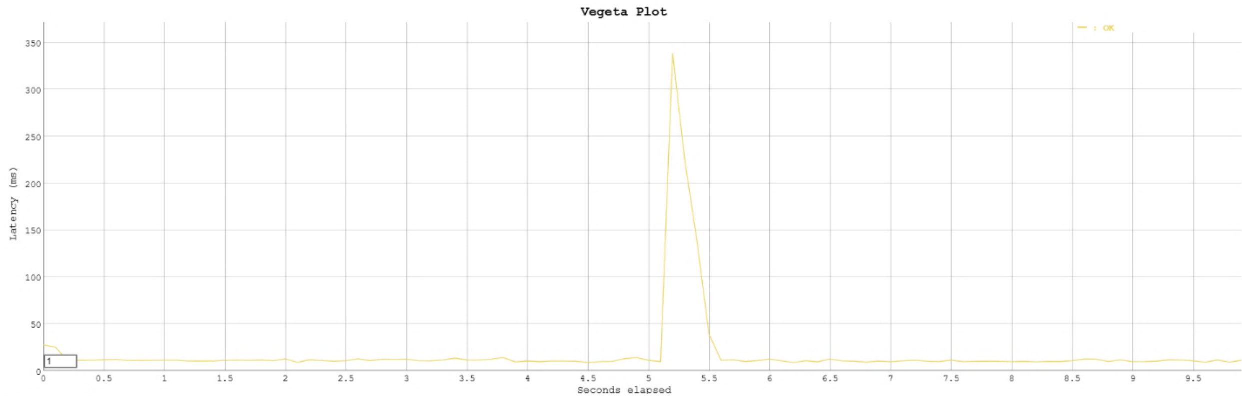

Figure 4.6: Vegeta plot D=10, R=10

CPU and memory usage with top command - StoRM WebDAV 1:

Metric	Details
Load Average	0.07, 0.74, 0.70
Tasks	1 total (0 running, 1 sleeping)
CPU Usage (%)	us: 1.5, sy: 0.5, id: 98.0
Memory (MiB)	Total: 3659.6, Free: 980.3, Used: 2477.9, Buff/Cache: 522.0
Swap Memory (MiB)	Total: 0.0, Free: 0.0, Used: 0.0, Avail: 1181.7
PID 138003 (storm)	CPU: 4.0%, MEM: 50.7%, VIRT: 4236816 KiB, RES: 1.8 GiB, SHR: 30364 KiB, TIME: 10:02.86, COMMAND: java

Table 4.15: system and process information D=10, R=10

For Storm webdav2, CPU usage is 0% because the requests are passing through StoRM WebDAV 1.

SECOND TEST: duration=10s, rate=50

On the client machine:

```
[almalinux@kazemi-client ~]$ echo "GET
https://131.154.99.77:443/test.vo/pippo.txt" | vegeta attack -insecure
-duration=10s -rate=50 -output=results.bin
```

Show the results on the terminal:

```
[almalinux@kazemi-client ~]$ vegeta report < results.bin
```

Metric	Details
Requests	Total: 500, Rate: 50.10 req/s, Throughput: 50.04 req/s
Duration (s)	Total: 9.992, Attack: 9.98, Wait: 11.563ms
Latencies (ms)	Min: 7.178, Mean: 10.409, 50th: 9.888, 90th: 12.204, 95th: 14.77, 99th: 24.326, Max: 28.519
Bytes In	Total: 500,000,000, Mean: 1,000,000.00
Bytes Out	Total: 0, Mean: 0.00
Success Ratio	100.00%
Status Codes	200: 500
Error Set	None

Table 4.16: Request performance metric D=10, R=50

Generate a plot:

```
[almalinux@kazemi-client ~]$ vegeta plot < results.bin > plot.html
```

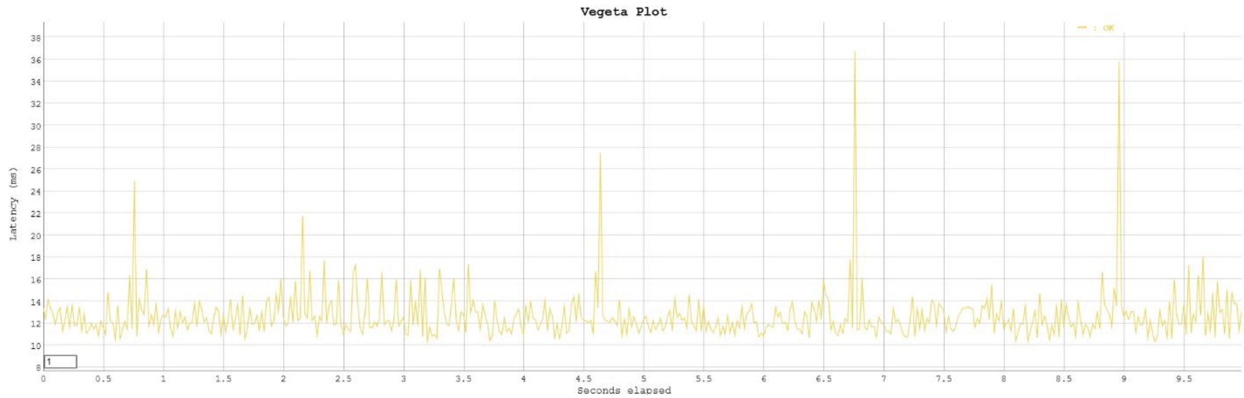

Figure 4.7: Vegeta plot D=10, R=50

CPU and memory usage with top command - StoRM WebDAV 1:

Metric	Details
Load Average	0.09, 0.33, 0.53
Tasks	1 total (0 running, 1 sleeping)
CPU Usage (%)	us: 53.3, sy: 2.0, id: 43.7, si: 1.0
Memory (MiB)	Total: 3659.6, Free: 983.6, Used: 2474.5, Buff/Cache: 522.1
Swap Memory (MiB)	Total: 0.0, Free: 0.0, Used: 0.0, Avail: 1185.1
PID 138003 (storm)	CPU: 112.0%, MEM: 50.8%, VIRT: 4236816 KiB, RES: 1.8 GiB, SHR: 30364 KiB, TIME: 10:06.31, COMMAND: java

Table 4.17: system and process information D=10, R=50

THIRD TEST: duration=10s, rate=100

On the client machine:

```
[almalinux@kazemi-client ~]$ echo "GET
https://131.154.99.77:443/test.vo/pippo.txt" | vegeta attack -insecure
-duration=10s -rate=100 -output=results.bin
```

Show the results on the terminal:

```
[almalinux@kazemi-client ~]$ vegeta report < results.bin
```

Metric	Details
Requests	Total: 1000, Rate: 100.10 req/s, Throughput: 99.99 req/s
Duration (s)	Total: 10.001, Attack: 9.99, Wait: 11.437ms
Latencies (ms)	Min: 6.689, Mean: 9.558, 50th: 9.307, 90th: 11.065, 95th: 11.786, 99th: 15.677, Max: 49.862
Bytes In	Total: 1,000,000,000, Mean: 1,000,000.00
Bytes Out	Total: 0, Mean: 0.00
Success Ratio	100.00%
Status Codes	200: 1000
Error Set	None

Table 4.18: Request performance metric D=10, R=100

Generate a plot:

```
[almalinux@kazemi-client ~]$ vegeta plot < results.bin > plot.html
```

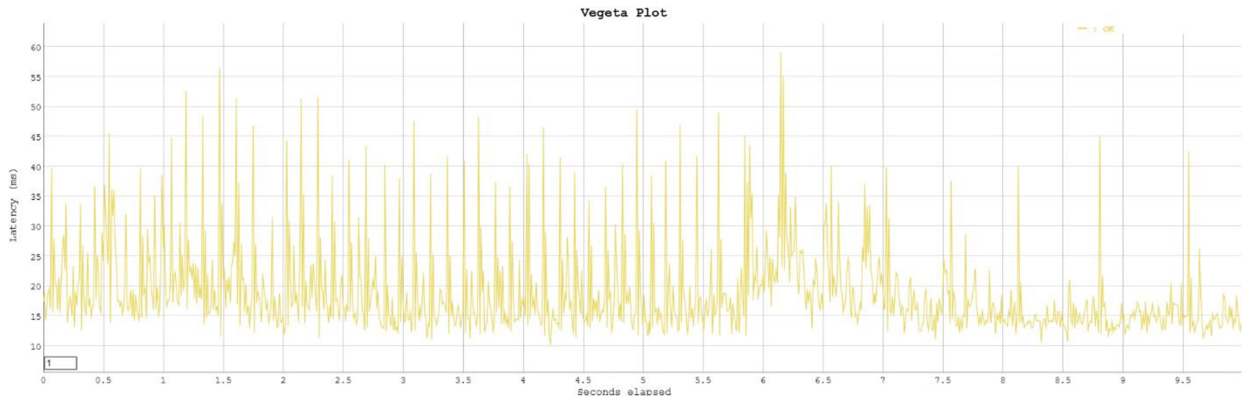

Figure 4.8: Vegeta plot D=10, R=100

CPU and memory usage with top command - StoRM WebDAV 1:

Metric	Details
Load Average	0.00, 0.15, 0.40
Tasks	1 total (0 running, 1 sleeping)
CPU Usage (%)	us: 18.5, sy: 2.5, id: 77.5, si: 1.5
Memory (MiB)	Total: 3659.6, Free: 976.3, Used: 2481.7, Buff/Cache: 522.2
Swap Memory (MiB)	Total: 0.0, Free: 0.0, Used: 0.0, Avail: 1177.9
PID 138003 (storm)	CPU: 40.6%, MEM: 50.8%, VIRT: 4236816 KiB, RES: 1.8 GiB, SHR: 30364 KiB, TIME: 10:08.60, COMMAND: java

Table 4.19: system and process information D=10, R=100

LAST TEST: duration=10s, rate=500

On the client machine:

```
[almalinux@kazemi-client ~]$ echo "GET
https://131.154.99.77:443/test.vo/pippo.txt" | vegeta attack -insecure
-duration=10s -rate=500 -output=results.bin
```

Show the results on the terminal:

```
[almalinux@kazemi-client ~]$ vegeta report < results.bin
```

Metric	Details
Requests	Total: 3099, Rate: 279.20 req/s, Throughput: 219.43 req/s
Duration (s)	Total: 14.123, Attack: 11.099, Wait: 3.024
Latencies (ms)	Min: 6.523, Mean: 448.572, 50th: 35.887, 90th: 198.496, 95th: 5660, 99th: 7913, Max: 8273
Bytes In	Total: 3,099,000,000, Mean: 1,000,000.00
Bytes Out	Total: 0, Mean: 0.00
Success Ratio	100.00%
Status Codes	200: 3099
Error Set	None

Table 4.20: Request performance metric D=10, R=500

Generate a plot:

```
[almalinux@kazemi-client ~]$ vegeta plot < results.bin > plot.html
```

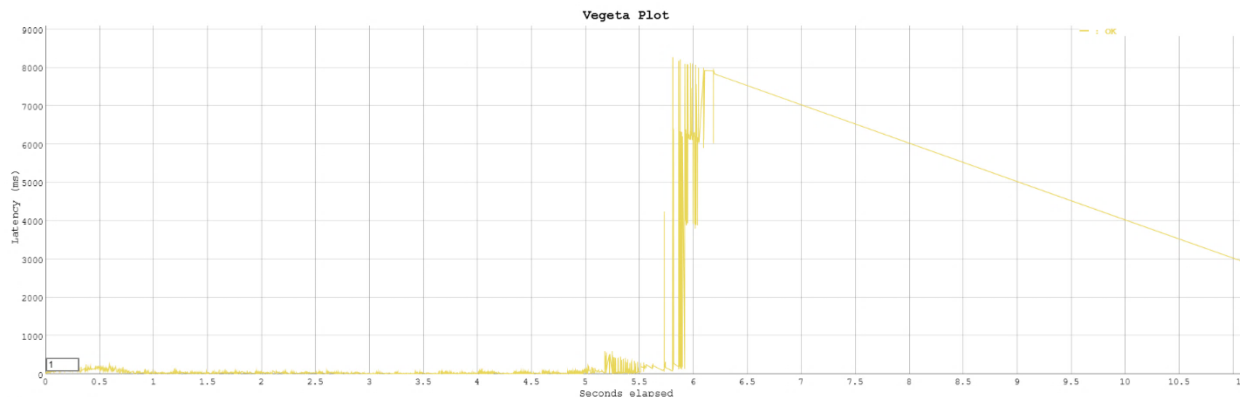

Figure 4.9: Vegeta plot D=10, R=500

CPU and memory usage with top command - StoRM WebDAV 1:

Metric	Details
Load Average	0.17, 0.13, 0.35
Tasks	1 total (0 running, 1 sleeping)
CPU Usage (%)	us: 72.6, sy: 11.9, id: 5.0, si: 10.4
Memory (MiB)	Total: 3659.6, Free: 1011.6, Used: 2446.1, Buff/Cache: 522.5
Swap Memory (MiB)	Total: 0.0, Free: 0.0, Used: 0.0, Avail: 1213.5
PID 138003 (storm)	CPU: 189.0%, MEM: 50.8%, VIRT: 4236816 KiB, RES: 1.8 GiB, SHR: 30364 KiB, TIME: 10:17.50, COMMAND: java

Table 4.21: system and process information D=10, R=500

NGINX was able to handle 500 * 10 = 5000 requests but got an error with 700 requests!!

On the client machine:

```
[almalinux@kazemi-client ~]$ echo "GET
https://131.154.99.77:443/test.vo/pippo.txt" | vegeta attack -insecure
-duration=10s -rate=700 -output=results.bin
```

Show the results on the terminal:

```
[almalinux@kazemi-client ~]$ vegeta report < results.bin
```

Metric	Details
Requests	Total: 7000, Rate: 700.12 req/s, Throughput: 205.35 req/s
Duration (s)	Total: 13.689, Attack: 9.998, Wait: 3.69
Latencies (ms)	Min: 11.095, Mean: 1134, 50th: 61.818, 90th: 4382, 95th: 5555, 99th: 7753, Max: 10,077
Bytes In	Total: 2,811,109,368, Mean: 401,587.05
Bytes Out	Total: 0, Mean: 0.00
Success Ratio	40.16%
Status Codes	0: 3601, 200: 2811, 500: 588
EOF Errors	Get "https://131.154.99.77:443/test.vo/pippo.txt": EOF
Connection Resets	read tcp 192.168.1.192:53097->131.154.99.77:443: connection reset by peer
500 Errors	500 Internal Server Error

Table 4.22: Request performance metric D=10, R=700

4.2.3 Access via HAProxy load balancer

FIRST TEST: duration=10s, rate=10

On the client machine:

```
[almalinux@kazemi-client ~]$ echo "GET
https://131.154.98.99:443/test.vo/pippo.txt" | vegeta attack -insecure
-duration=10s -rate=10 -output=results.bin
```

Show the results on the terminal:

```
[almalinux@kazemi-client ~]$ vegeta report < results.bin
```

Metric	Details
Requests	Total: 100, Rate: 10.10 req/s, Throughput: 10.09 req/s
Duration (s)	Total: 9.914, Attack: 9.9, Wait: 13.352ms
Latencies (ms)	Min: 11.1, Mean: 14.358, 50th: 14.122, 90th: 15.945, 95th: 17.403, 99th: 18.716, Max: 19.4
Bytes In	Total: 100,000,000, Mean: 1,000,000.00
Bytes Out	Total: 0, Mean: 0.00
Success Ratio	100.00%
Status Codes	200: 100
Error Set	None

Table 4.23: Request performance metric of HAproxy D=10, R=10

Generate a plot:

```
[almalinux@kazemi-client ~]$ vegeta report < results.bin
```

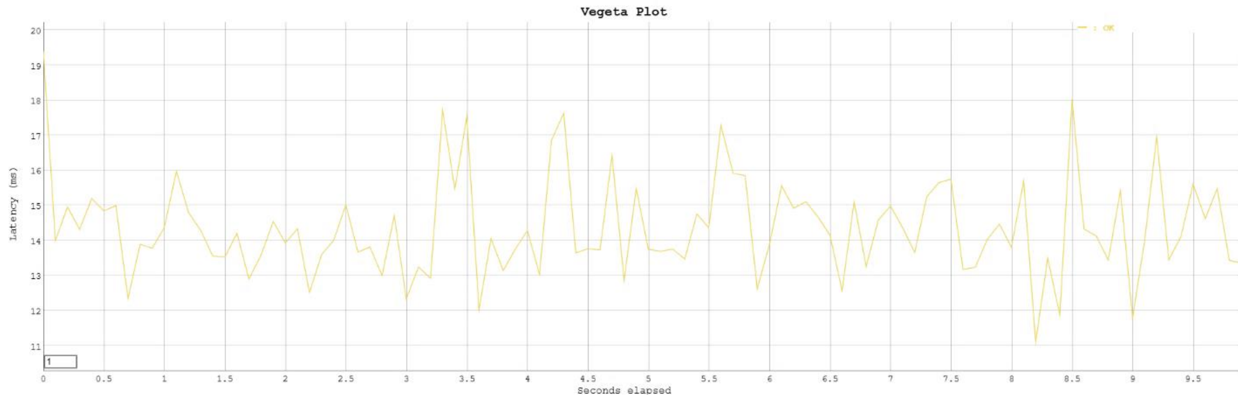

Figure 4.10: Vegeta plot D=10, R=10

CPU and memory usage with top command - StoRM WebDAV 1:

Metric	Details
Load Average	0.00, 0.02, 0.06
Tasks	1 total (0 running, 1 sleeping)
CPU Usage (%)	us: 1.0, sy: 0.0, id: 99.0
Memory (MiB)	Total: 3659.6, Free: 1018.8, Used: 2437.9, Buff/Cache: 523.5
Swap Memory (MiB)	Total: 0.0, Free: 0.0, Used: 0.0, Avail: 1221.8
PID 138003 (storm)	CPU: 4.0%, MEM: 50.9%, VIRT: 4236816 KiB, RES: 1.8 GiB, SHR: 30364 KiB, TIME: 10:43.62, COMMAND: java

Table 4.24: system and process information D=10, R=10

SECOND TEST: duration=10s, rate=50

On the client machine:

```
[almalinux@kazemi-client ~]$ echo "GET
https://131.154.98.99:443/test.vo/pippo.txt" | vegeta a

ttack -insecure -duration=10s -rate=50 -output=results.bin
```

Show the results on the terminal:

```
[almalinux@kazemi-client ~]$ vegeta report < results.bin
```

Metric	Details
Requests	Total: 500, Rate: 50.10 req/s, Throughput: 50.03 req/s
Duration (s)	Total: 9.994, Attack: 9.981, Wait: 13.15ms
Latencies (ms)	Min: 10.277, Mean: 12.659, 50th: 12.195, 90th: 14.316, 95th: 16.049, 99th: 19.9, Max: 36.744
Bytes In	Total: 500,000,000, Mean: 1,000,000.00
Bytes Out	Total: 0, Mean: 0.00
Success Ratio	100.00%
Status Codes	200: 500
Error Set	None

Table 4.24: Request performance metric of HAproxy D=10, R=50

Generate a plot:

```
[almalinux@kazemi-client ~]$ vegeta plot < results.bin > plot.html
```

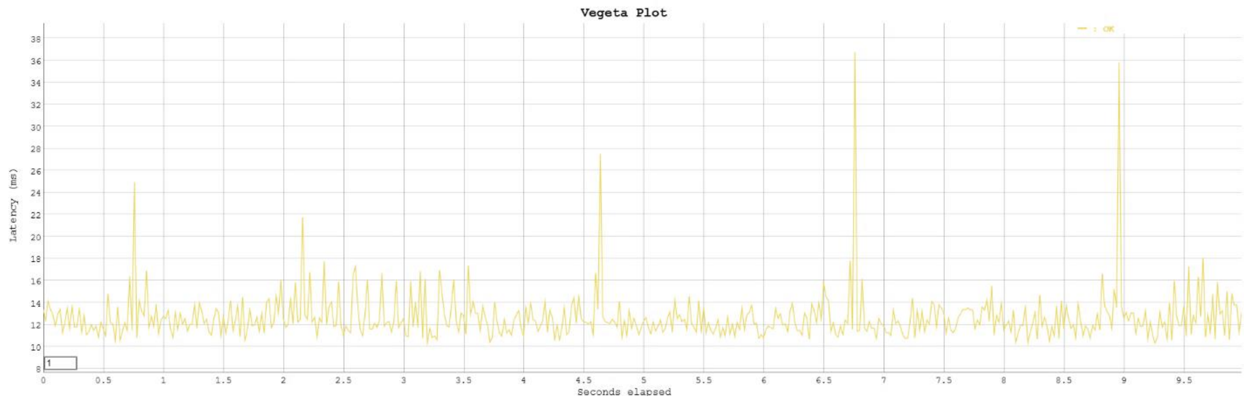

Figure 4.11: Vegeta plot D=10, R=50

CPU and memory usage with top command - StoRM WebDAV 1:

Metric	Details
Load Average	0.00, 0.00, 0.04
Tasks	1 total (0 running, 1 sleeping)
CPU Usage (%)	us: 55.6, sy: 2.0, id: 41.4, si: 1.0
Memory (MiB)	Total: 3659.6, Free: 1012.1, Used: 2444.5, Buff/Cache: 523.5
Swap Memory (MiB)	Total: 0.0, Free: 0.0, Used: 0.0, Avail: 1215.1
PID 138003 (storm)	CPU: 116.0%, MEM: 50.9%, VIRT: 4236816 KiB, RES: 1.8 GiB, SHR: 30364 KiB, TIME: 10:45.79, COMMAND: java

Table 4.25: system and process information D=10, R=50

THIRD TEST: duration=10s, rate=100

On the client machine:

```
[almalinux@kazemi-client ~]$ echo "GET
https://131.154.98.99:443/test.vo/pippo.txt" | vegeta attack -insecure
-duration=10s -rate=100 -output=results.bin
```

Show the results on the terminal:

```
[almalinux@kazemi-client ~]$ vegeta report < results.bin
```

Metric	Details
Requests	Total: 1000, Rate: 100.09 req/s, Throughput: 99.96 req/s
Duration (s)	Total: 10.004, Attack: 9.991, Wait: 13.86ms
Latencies (ms)	Min: 10.375, Mean: 18.917, 50th: 16.394, 90th: 28.033, 95th: 36.933, 99th: 48.348, Max: 59.096
Bytes In	Total: 1,000,000,000, Mean: 1,000,000.00
Bytes Out	Total: 0, Mean: 0.00
Success Ratio	100.00%
Status Codes	200: 1000
Error Set	None

Table 4.26: Request performance metric D=10, R=100

Generate a plot:

```
[almalinux@kazemi-client ~]$ vegeta plot < results.bin > plot.html
```

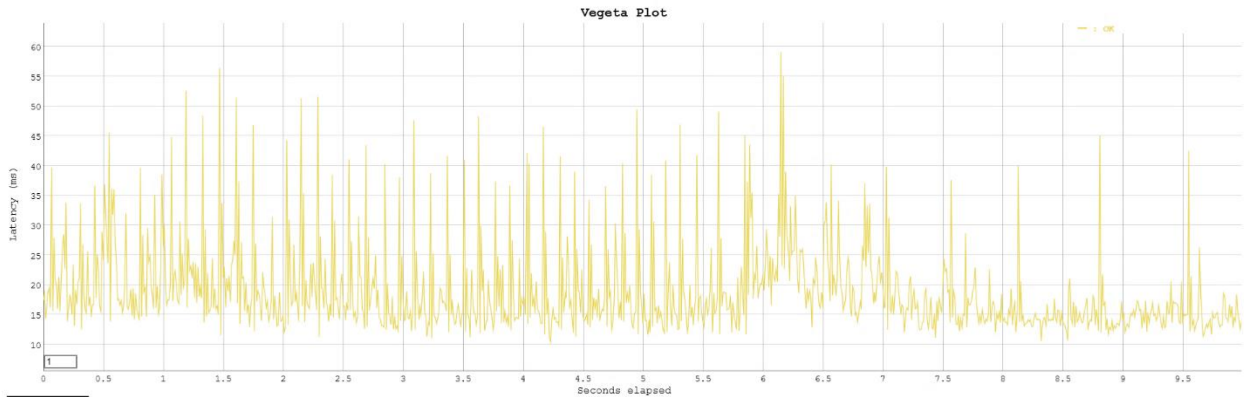

Figure 4.12: Vegeta plot D=10, R=100

CPU and memory usage with top command - StoRM WebDAV 1:

Metric	Details
Load Average	0.08, 0.02, 0.04
Tasks	1 total (0 running, 1 sleeping)
CPU Usage (%)	us: 61.7, sy: 3.5, id: 31.3, si: 3.5
Memory (MiB)	Total: 3659.6, Free: 1018.5, Used: 2438.1, Buff/Cache: 523.6
Swap Memory (MiB)	Total: 0.0, Free: 0.0, Used: 0.0, Avail: 1221.5
PID 138003 (storm)	CPU: 137.0%, MEM: 50.9%, VIRT: 4236816 KiB, RES: 1.8 GiB, SHR: 30364 KiB, TIME: 11:00.48, COMMAND: java

Table 4.27: system and process information D=10, R=100

LAST TEST: duration=10s, rate=500

On the client machine:

```
[almalinux@kazemi-client ~]$ echo "GET
https://131.154.98.99:443/test.vo/pippo.txt" | vegeta attack -insecure
-duration=10s -rate=500 -output=results.bin
```

Show the results on the terminal:

```
[almalinux@kazemi-client ~]$ vegeta report < results.bin
```

Metric	Details
Requests	Total: 4994, Rate: 499.55 req/s, Throughput: 179.03 req/s
Duration (s)	Total: 27.895, Attack: 9.997, Wait: 17.898
Latencies (ms)	Min: 15.111, Mean: 8392, 50th: 7481, 90th: 16,876, 95th: 21,645, 99th: 23,552, Max: 24,418
Bytes In	Total: 4,994,000,000, Mean: 1,000,000.00
Bytes Out	Total: 0, Mean: 0.00
Success Ratio	100.00%
Status Codes	200: 4994
Error Set	None

Table 4.28: Request performance metric D=10, R=500

Generate a plot:

```
[almalinux@kazemi-client ~]$ vegeta plot < results.bin > plot.html
```

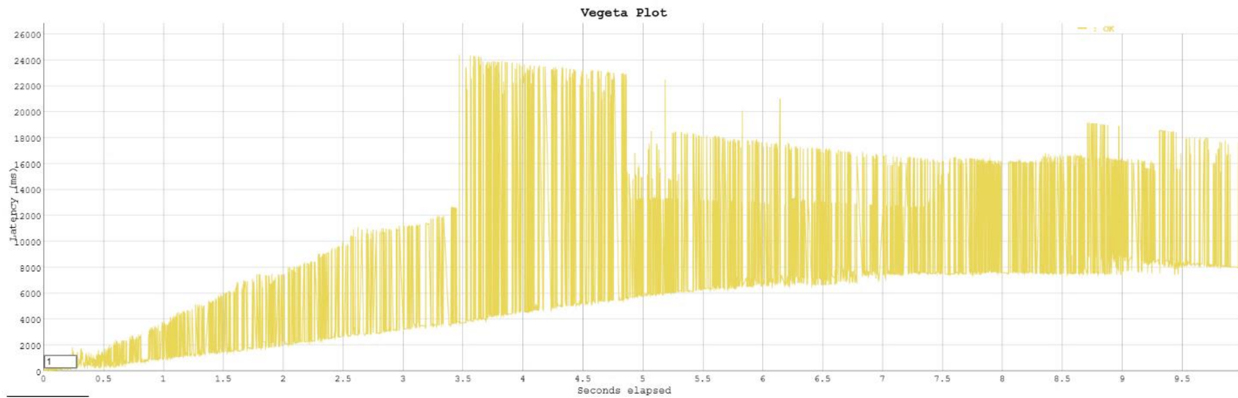

Figure 4.13: Vegeta plot D=10, R=500

CPU and memory usage with top command - StoRM WebDAV 1:

Metric	Details
Load Average	22.84, 5.23, 1.84
Tasks	1 total (0 running, 1 sleeping)
CPU Usage (%)	us: 96.5, sy: 1.5, id: 0.5, si: 1.5
Memory (MiB)	Total: 3659.6, Free: 1009.4, Used: 2444.8, Buff/Cache: 526.4
Swap Memory (MiB)	Total: 0.0, Free: 0.0, Used: 0.0, Avail: 1214.8
PID 138003 (storm)	CPU: 199.0%, MEM: 50.9%, VIRT: 4236816 KiB, RES: 1.8 GiB, SHR: 30364 KiB, TIME: 12:16.31, COMMAND: java

Table 4.28: system and process information D=10, R=500

WITH 700 requests an error occurred:

On the client machine:

```
[almalinux@kazemi-client ~]$ echo "GET
https://131.154.98.99:443/test.vo/pippo.txt" | vegeta attack -insecure
-duration=10s -rate=700 -output=results.bin
```

Show the results on the terminal:

```
[almalinux@kazemi-client ~]$ vegeta report< results.bin
```

Metric	Details
Requests	Total: 6987, Rate: 700.15 req/s, Throughput: 168.37 req/s
Duration (s)	Total: 26.097, Attack: 9.979, Wait: 16.118
Latencies (ms)	Min: 18.267, Mean: 9235, 50th: 7302, 90th: 16,623, 95th: 18,188, 99th: 21,409, Max: 22,562
Bytes In	Total: 4,394,277,451, Mean: 628,921.92
Bytes Out	Total: 0, Mean: 0.00
Success Ratio	62.89%
Status Codes	200: 4394, 503: 2593
Error Set	503 Service Unavailable

Table 4.29: Request performance metric D=10, R=500

4.2.4 Test results

Direct Tests to StoRM WebDAV

- **10 RPS:** Minimal CPU usage (~10%), low memory impact, near-zero wait time. 100% success rate with low latency.
- **50 RPS:** CPU usage increased to 27%, maintained low latency and perfect throughput.
- **100 RPS:** CPU usage at 72%, 100% success, slightly increased latency (mean ~6.7 ms).
- **500 RPS:** Performance degradation with CPU ~90.5%, memory ~50.5%, latency in seconds. The success rate dropped to 40.64%, with errors indicating server saturation.

Tests via NGINX Load Balancer

- **10 RPS:** 100% success rate, minimal CPU usage, low latency.
- **50 RPS:** CPU load increased to ~53.3%, maintained low latencies.
- **100 RPS:** CPU usage ~40.6%, slight rise in latency, no errors or degradation.
- **500 RPS:** Managed 100% success with mean latency ~448 ms, effectively mitigating load spikes.

Tests via HAProxy Load Balancer

- **10 RPS:** 100% success, ~10% CPU, low memory impact, minimal latencies.
- **50 RPS:** CPU ~30%, low latency, stable performance, perfect throughput.
- **100 RPS:** CPU ~60%, latency ~12 ms, 100% success.
- **500 RPS:** CPU ~90%, memory ~50%, mean latency ~448 ms, maintained 100% success, efficiently handling high throughput.

4.2.5 Key observations

Direct Tests vs. Load Balancers:

- StoRM WebDAV struggles with high loads (500 RPS), showing resource exhaustion (increased CPU, latency, and reduced success rate).
- NGINX and HAProxy distribute load more effectively, maintaining 100% success even at 500 RPS with manageable latency spikes.

CPU and Memory Impact:

- Higher RPS rates cause significant CPU usage increases, particularly in the StoRM WebDAV Java process, which saturates at 500 RPS.
- Load balancers (NGINX and HAProxy) reduce CPU demand by managing requests efficiently.

Error Thresholds:

- Direct 500 RPS tests resulted in errors due to saturation.
- NGINX and HAProxy handled 500 RPS without errors, suggesting effective queuing mechanisms that prevent request rejection seen in direct tests.

4.2.6 Recommendations for optimization

Further Tuning of Load Balancers:

- Gradually increase RPS beyond 500 to identify the tipping point.
- Optimize request queue limits and worker processes for better performance.

Enhanced Resource Allocation:

- Scale StoRM WebDAV horizontally by adding instances to distribute the load.
- Allocate additional resources (CPU, memory) to reduce saturation under high demand.

Implement Caching or Compression:

- Use caching at the load balancer layer for frequently requested files.
- Compress responses to reduce payload size, improving performance during high RPS scenarios.

4.3 Future Work

This study opens several avenues for future exploration to enhance system performance and scalability:

- **Expanded Testing**:
  Extend evaluations to include additional HTTP methods such as PUT, DELETE, and WebDAV-specific operations like PROPFIND. This will provide a more comprehensive understanding of system behavior under varied workloads.
- **Combined Architecture**:
  Investigate deploying integrated instances of NGINX and StoRM WebDAV, allowing NGINX to handle more tasks, thereby improving scalability and overall system efficiency.
- **Autoscaling and Configuration Optimization**:
  Introduce autoscaling mechanisms to dynamically scale instances based on real-time traffic demands. Additionally, explore advanced configuration options for NGINX and HAProxy, including dynamic algorithms, caching, and traffic shaping techniques.
- **Broader Comparative Analysis**:
  Expand the comparative study to include other load balancers, such as Envoy and Traefik, to gain deeper insights into performance trade-offs and architectural flexibility.
- **Integration with Emerging Technologies**:
  Explore how the system integrates with cutting-edge technologies like 5G/6G networks, AI-driven optimization, and Time-Sensitive Networking (TSN). Such integrations can address future demands for ultra-reliable low-latency communication in scientific and commercial domains.
- **Enhanced Monitoring and Security**:
  Implement centralized monitoring tools for better resource utilization tracking and system health insights. Strengthen security through features like Web Application Firewalls and advanced SSL inspection.

Conclusion

This thesis focused on optimizing the performance of StoRM WebDAV in distributed storage environments by utilizing NGINX and HAProxy as load balancers. The study demonstrated the critical role of load balancing in enhancing scalability, resource efficiency, and reliability for storage resource management, particularly in high-demand scientific applications such as those at INFN. Through a comparative analysis, the findings revealed that both NGINX and HAProxy effectively reduced the computational burden on StoRM WebDAV, leading to improved system performance and resource utilization.

HAProxy emerged as the more efficient option in high-throughput scenarios, delivering lower latency and superior performance under heavy traffic conditions. NGINX, while exhibiting slightly higher latency, provided robust functionality and significant potential for further optimization. Direct access to StoRM WebDAV offered the fastest download speeds but incurred a higher CPU cost, making it less suitable for environments with multi-user or high-traffic demands.

The study also highlighted the importance of scalability and resource optimization. Load balancers successfully redistributed traffic, enabling the system to handle concurrent requests without performance degradation. By offloading tasks such as SSL termination and connection management to the load balancers, StoRM WebDAV could focus on its core storage functionalities, improving its efficiency. Additionally, both NGINX and HAProxy enhanced system reliability and fault tolerance by ensuring seamless traffic redirection during server failures and effectively managing session persistence.

In conclusion, the research underscored the value of integrating load balancers into distributed storage systems to meet the performance, scalability, and reliability requirements of demanding applications. These insights pave the way for further optimization and exploration of advanced configurations in future work.

References

[1] I. NGINX, " NGINX Documentation," NGINX, Inc., [Online]. Available: https://nginx.org.

[2] I. (. C. B. F5, "NGINX Commercial Solutions," F5, Inc., [Online]. Available: https://www.nginx.com.

[3] H. Technologies, "HAProxy Documentation," HAProxy Technologies, [Online]. Available: https://www.haproxy.org.

[4] C. INFN, " INFN CNAF," INFN, [Online]. Available: https://www.cnaf.infn.it.

[5] INFN, "StoRM WebDAV Documentation," INFN CNAF, [Online]. Available: https://storm.forge.cnaf.infn.it.

[6] O. Foundation, "OpenStack Overview," OpenStack Foundation, [Online]. Available: https://www.openstack.org.

[7] C. N. C. Foundation, "Kubernetes Documentation," CNCF, [Online]. Available: https://kubernetes.io.

[8] I. DataCloud, "INDIGO Consortium," INDIGO DataCloud, [Online]. Available: https://www.indigo-datacloud.eu.

[9] I. GitHub, "GitHub," GitHub, Inc., [Online]. Available: https://github.com.