FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Lehrstuhl für Informatik 4 · Verteilte Systeme und Betriebssysteme

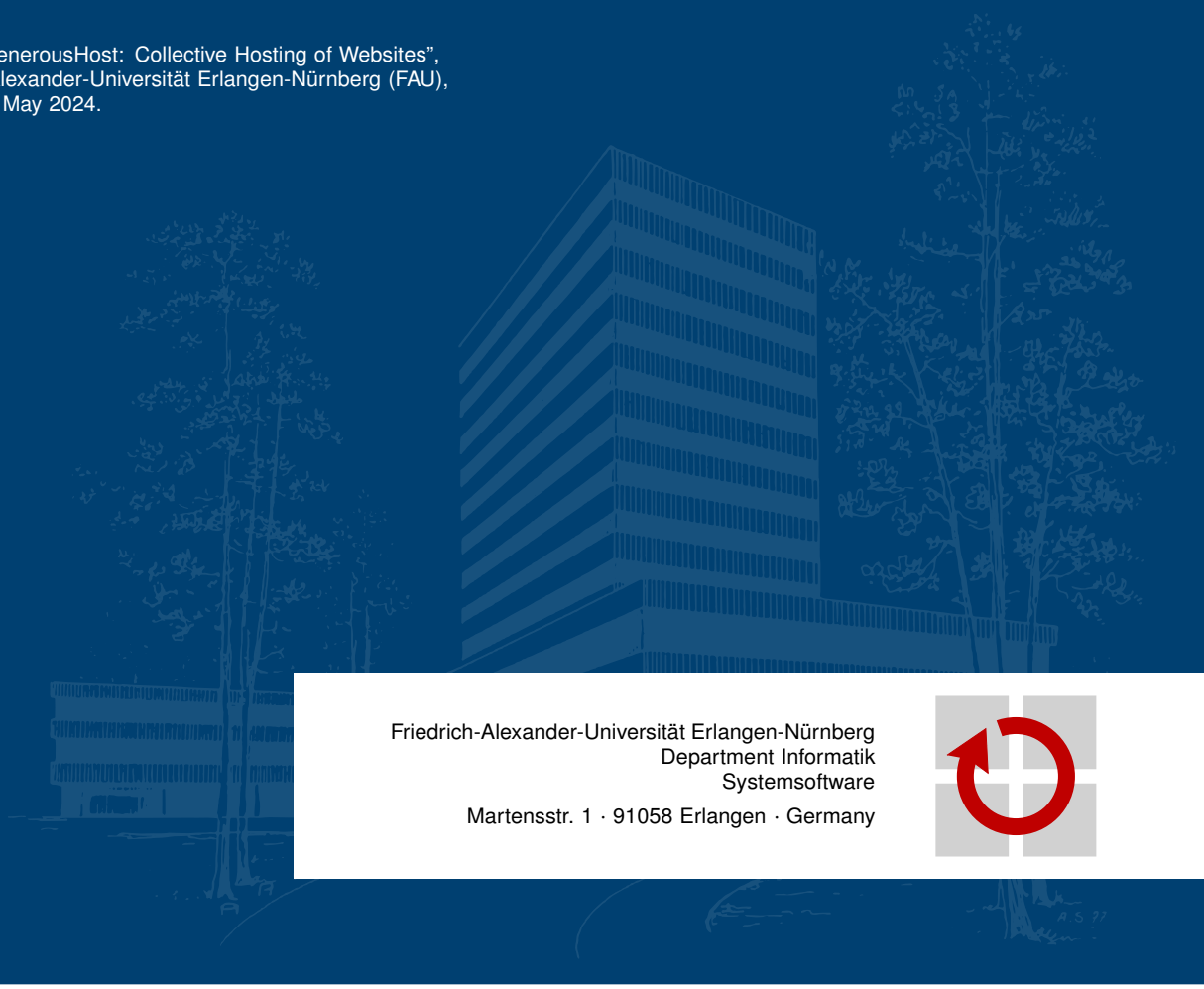
Steve Richard Pegouen

GenerousHost: Collective Hosting of Websites

Masterarbeit im Fach Informatik

31. Mai 2024

Please cite as:
Steve Richard Pegouen, “GenerousHost: Collective Hosting of Websites”,
Master’s Thesis, Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU),
Dept. of Computer Science, May 2024.



Friedrich-Alexander-Universität Erlangen-Nürnberg
Department Informatik
Systemsoftware
Martensstr. 1 · 91058 Erlangen · Germany

GenerousHost: Collective Hosting of Websites

Masterarbeit im Fach Informatik

vorgelegt von

Steve Richard Pegouen

geb. am 23. April 1996
in Bafoussam

angefertigt am

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Department Informatik
Friedrich-Alexander-Universität Erlangen-Nürnberg

Betreuer:	**Arne Vogel, M. Sc.** **Harald Böhm, M. Sc.**
Betreuender Hochschullehrer:	**Prof. Dr.-Ing. Rüdiger Kapitza**
Beginn der Arbeit:	**1. December 2023**
Abgabe der Arbeit:	**31. Mai 2024**

Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Declaration

I declare that the work is entirely my own and was produced with no assistance from third parties. I certify that the work has not been submitted in the same or any similar form for assessment to any other examining body and all references, direct and indirect, are indicated as such and have been cited accordingly.

(Steve Richard Pegouen)
Erlangen,
31. Mai 2024

ABSTRACT

In our interconnected world, web sites are indispensable, forming the foundation of our digital society by providing essential access to numerous services. However, ensuring their ongoing availability and performance is expensive. The substantial cost of server maintenance, development, and infrastructure often pose significant challenges, especially for innovative projects. This financial strain can stifle the growth and accessibility of valuable funding solutions that keep the digital world open and thriving for everyone. Finding a secure way to sustainable funding web services remain a challenging and elusive goal. The method of financing, such as the subscription model which requires payment to use service limits access and excludes many people who cannot afford it.

In this thesis, we present `GenerousHost`, a novel approach that leverages underutilized resources on volunteer's computers to support the hosting of web services. It enables individuals or organizations to contribute resources, thereby distributing the hosting task across a diverse network of contributors. This decentralises the hosting infrastructure, thereby reducing the financial burden on developers. `GenerousHost` offers contributors the flexibility to determine where the service should be hosted, while ensuring a superior quality of service and low latency for contributors, which could be a motivating factor for contributors to donate their resources. `GenerousHost` integrates Trusted Execution Environment (TEE), such as AMD SEV-SNP, to ensure secure computing within the infrastructure provided by contributors. Furthermore, It utilises the Raft algorithm for data synchronisation and the Hypertext Transfer Protocol (HTTP) for exchanging messages between nodes, with the aim of getting them to communicate with each other. We evaluate `GenerousHost` by implementing a web application for hosting and a load generator in order to provide the system with a load. Our evaluation shows that `GenerousHost` can process a huge number of requests simultaneously in less time than a system with a single server. In addition to this, it has been found that this approach is more fault tolerant and more secure than systems that use a single server, as it allows data to be secured by replicating it on several servers.

KURZFASSUNG

In unserer vernetzten Welt sind Webdienste unverzichtbar. Sie bilden die Grundlage unserer digitalen Gesellschaft, da sie den Zugang zu zahlreichen Möglichkeiten ermöglichen. Die Sicherstellung ihrer ständigen Verfügbarkeit und Leistung ist jedoch teuer. Die beträchtlichen Kosten für Serverwartung, Entwicklung und Infrastruktur stellen oft eine große Herausforderung dar, insbesondere für innovative Projekte. Diese finanzielle Belastung kann das Wachstum und die Zugänglichkeit wertvoller Finanzierungslösungen behindern, die die digitale Welt für alle offen und lebendig halten. Einen sicheren Weg zur nachhaltigen Finanzierung von Webdiensten zu finden, bleibt ein schwieriges und schwer zu erreichendes Ziel. Die Finanzierungsmethode, wie das Abonnementmodell, das eine Zahlung für die Nutzung des Dienstes erfordert, schränkt den Zugang ein und schließt viele Menschen aus, die es sich nicht leisten können.

In dieser Arbeit stellen wir `GenerousHost` vor, einen neuartigen Ansatz, der nicht ausgelastete Ressourcen auf Computern von Freiwilligen nutzt, um das Hosting von Webdiensten zu unterstützen. Es ermöglicht es Einzelpersonen oder Organisationen, Ressourcen beizusteuern und so die Hosting-Aufgabe auf ein vielfältiges Netzwerk von Mitwirkenden zu verteilen. Dadurch wird die Hosting-Infrastruktur dezentralisiert und die finanzielle Belastung der Entwickler verringert. `GenerousHost` biete den Mitwirkenden die Flexibilität, den Ort zu bestimmen, an dem der Dienst gehostet werden soll, und gewährleistet gleichzeitig eine überragende Dienstqualität und geringe Latenzzeiten für die Mitwirkenden, was ein Motivationsfaktor für die Mitwirkenden sein könnte, ihre Ressourcen zu spenden. `GenerousHost` integriert eine vertrauenswürdige Ausführungsumgebung (TEE), wie AMD SEV-SNP, um eine sichere Datenverarbeitung innerhalb der von den Teilnehmern bereitgestellten Infrastruktur zu gewährleisten. Außerdem nutzt es den Raft-Algorithmus für die Datensynchronisierung und das HTTP-Protokoll für den Austauch von Nachrichten zwischen Knoten, um sie miteinander kommunizieren zu lassen. Wir evaluieren `GenerousHost`, indem wir eine Webanwendung für das Hostig und einen Lastgenerator implementieren, um das System mit Last zu versorgen. Unsere Auswertung zeigt, dass `GenerousHost` eine große Anzahl von Anfragen gleichzeitig in kürzerer Zeit verarbeiten kann als ein System mit einem einzelnen Server. Darüber hinaus hat sich gezeigt, dass dieser Anzatz fehlertoleranter und sicherer ist als Systeme, die einen einzelnen Server verwenden, da es die Sicherung der Daten durch Replikation auf mehrere Server ermöglicht.

CONTENTS

1 INTRODUCTION

Websites play a fundamental role in today's digital society, serving as vehicles for information, communication, and social interaction. They provide access to a wide range of services, making a significant contribution to optimizing our daily activities. However, the permanent availability of websites on the internet is dependent on the existence of an efficient web hosting infrastructure. This can be achieved through specialized service providers or by website providers setting up an infrastructure of their own, which enables websites to be accessed by the general public.

Web hosting plays a fundamental role in the provision of online services, it involves renting space and connectivity on servers to make a web site accessible on the internet [52]. Nowadays, several web hosting solutions are offered by companies such as dedicated servers, cloud hosting services, shared hosting and Content Management System (CMS) that offer integrated hosting solutions [29, 57]. These services are generally available on a subscription basis, the cost of which varies according to user requirements. However, the autonomous management of servers, although an alternative to traditional subscriptions, requires expertise not only in maintenance but also in server security and investment to increase hosting capacity. These hosting solutions are becoming increasingly expensive due to the growth in demand and the complexity of hosted services, which represents a major challenge for initiatives such as Open Source projects, which often operate on limited budgets and rely on donations to provide free services. These projects are particularly vulnerable, as escalating costs threaten their viability. This raises the crucial question of how to fund the maintenance of this hosted service without having to resort to funding methods that could compromise the confidentiality and privacy of users such as paid advertising.

In our exploration of hosting solutions that minimize expenses and provide a secure and easily scalable framework for hosting services, we are evaluating the potential of `GenerousHost`, which leverages advancements in distributed systems like cloud computing, volunteer computing and trusted execution environment. `GenerousHost` is a new hosting paradigm that operates on a decentralized model where the user community, referred to as `donors`, shares their unused computing resources to support the hosting of the web services they may themselves use. By equipping individual donor's computers with virtual machines that have security mechanisms to protect data and software needed to host web services, the conventional single-provider hosting model is transformed into a more resilient and distributed model providing multiple service provider. This strategy aims to improve scalability and security while ensuring user privacy. This approach offers an economically viable alternative to other hosting methods, addressing both operational efficiency and data protection.

However, `GenerousHost` also presents critical challenges that require attention. One such challenge is the balanced distribution of resources among donor users' machines. To ensure stable and reliable performance of the hosted service, it is essential to guarantee optimal allocation that considers

the variability of processing capacities and storage space among donor machines. Maintaining data consistency and synchronization across all donor machines is a significant technical challenge. Therefore, it is crucial to have a mechanism that maintains data integrity while enabling real-time updating without conflict or loss of information to ensure the system's reliability. Additionally, data security is another challenge in collective hosting, which requires ensuring the confidentiality of sensitive data and maintaining high service availability. The governance and regulatory challenges of the decentralized model accentuate the complexity of collective hosting. To ensure the uninterrupted operation of the system, clear policies and an effective regulatory framework are required for the governance of the collective hosting network. This includes regulating donor entry and exit, as well as ensuring equitable distribution of charges.

Despite these challenges the development and adoption of `GenerousHost` as a web hosting solution is motivated by several factors that address the challenges of cost, performance, and security. `GenerousHost` is based on the principle of distributed collaboration, offering a potential solution to the challenges of scalability, cost, and performance. The service is distributed across several hosts, reducing the risk of breakdowns and downtime in the event of a server failure. Therefore, it promotes greater reliability and continuity of operations, increasing user confidence in the availability and quality of hosted services.

The objective of this study is to create and assess a practical solution for hosting websites collectively. The system must effectively manage distributed resources, guarantee the security of hosted data, and maintain service availability even in the presence of potential failures or attacks.

The introduction of `GenerousHost` into the web services catalog represents a transformative shift in how hosting is approached, offering significant benefits over other hosting methods.

The project make two significant contributions:

1. We are presenting the design and implementation of `GenerousHost` a novel approach, which exploits the unused resources of volunteers who wish to support the hosting of web service.

2. We assess the performance of `GenerousHost`.

The remainder of this work is structured as follows: first, we explore the Background (Chapter 2) where essential context and foundational theories are established. Following this, in the Design (Chapter 3), we present our architectural strategies and the design principles. This leads into the Implementation (Chapter 4), where we describe the practical steps and methodologies used to create our solution. The Evaluation (Chapter 5) follows, assessing our approach against specific criteria to demonstrate its efficacy and utility. Lastly, the Related Work (Chapter 6) places our findings within the broader academic landscape, comparing and contrasting with similar efforts in the field.

2 BACKGROUND

In this chapter, we delve into the fundamental aspects that are essential to understand our topic : the collective hosting of websites. The first sub-chapter will examine distributed systems, highlighting their crucial importance to the resilience and efficiency of collective hosting. The second will focus on virtualization, exploring how virtual instances of computing resources facilitate the efficient use of hardware and underpin cloud infrastructures. The third will delve into cloud computing, providing the necessary foundations for understanding the key infrastructures of our collective hosting system. The fourth will delve into Trusted Execution Environments (TEEs), discussing how secure areas within processors can safeguard data and computation in cloud environment. The fifth sub-chapter will focus on the Hypertext Transfer Protocol (HTTP), analysing its impact and role in web communication between distributed servers. Additionally, we will discuss the proxy concept and its importance in a collective hosting architecture.

2.1 Distributed Systems

In today's interconnected world, distributed systems have become an integral part of many aspects of computing, enabling complex and large-scale operations across multiple domains and industries. A common definition of a distributed system is a set of interacting, independent computers that operate as a single, coherent entity in the eyes of users [15, 60]. This definition emphasises two distinctive aspects of distributed systems, namely the autonomy of the computing elements of the system and the transparency of their operation, which allows users to perceive the system as a single entity [60]. Distributed systems link computers that are geographically dispersed but share a common purpose and are connected by a network. They are essential for applications that require the processing of large amounts of data or significant computing power, distributed across multiple locations to optimise performance, reduce costs or improve resilience.

In the context of collective hosting, the importance of distributed systems is particularly crucial. They enable the load of data processing and requests to be spread over several servers, also known as nodes, thereby reducing dependency on a single server or location. This increases the resilience and availability of the service. By exploiting the capacity of multiple computers, provided by volunteer users, collective hosting becomes more sustainable and less susceptible to single points of failure, making the service more robust and reliable for all users of the hosted service.

2.1.1 Characteristics of Distributed Systems

Distributed systems have distinctive characteristics that make them indispensable for managing complex operations across diverse technological sectors. Among these features, decentralisation

ensures that there is no single point of failure, distributing tasks and resources across multiple independent nodes to minimise the risk of system-wide failures. This structure enhances resilience and robustness, making services less vulnerable to component-specific failures.

At the same time, fault tolerance is another essential quality, enabling the system to maintain continuous operation even in the presence of node failures or network disturbances. Thanks to redundancy techniques such as replication and check-pointing [32], distributed systems are able to maintain continuous operation even in the presence of component failures. Replication involves creating multiple copies of jobs for each running job, which are then executed on different hosts. This strategy ensures that, in the event of a host failure or network disconnection, at least one of the replicated tasks will be successfully completed. Check-pointing also enables the distributed system to regularly back up its status to reliable, stable storage. In case of a failure, the system can restart from the last backup point, significantly reducing downtime and minimizing data loss.

Finally, distributed systems are scalable, allowing for dynamic addition or removal of nodes to adapt processing capacity to workload demands. This flexibility is crucial in environments where computing resource requirements can vary considerably, enabling systems to provide additional resources during peaks in demand and to reduce capacity when they are less needed, thereby optimizing efficiency and operating costs. These characteristics make distributed systems a reliable and scalable solution to modern technological challenges.

2.1.2 Types of Distributed Systems

Distributed systems can be classified into three main types based on their use and the nature of the tasks they perform: distributed computing systems, distributed information systems, and distributed pervasive systems [59]. Each category is designed to meet specific requirements for large-scale data processing and management. In the field of distributed computing systems, there are three primary sub-types that are essential for data processing: cluster computing, grid computing, and cloud computing [60]. In the context of our work, we will focus specifically on cloud computing, which offers a flexible and scalable infrastructure via the internet. This will be discussed in greater detail in Section 2.3, where the implications of cloud computing in our study will be elaborated.

2.1.3 Consensus Algorithms

In a distributed system, achieving perfect synchronisation and coherence is challenging, particularly in the face of issues such as node failures, communication latency and security threats. As Ongaro [43] has highlighted, consensus enables a set of machines to function as a coherent group capable of surviving the failures of some of its members. This process, by which several entities reach a general agreement on a certain state of the system, enables uniform and synchronised progress to be made. To ensure the continued seamless functioning of the system, consensus algorithms represent a crucial tool, facilitating the enhancement of the system's overall reliability and efficiency. Among consensus mechanisms, Byzantine Fault Tolerance (BFT) and Crash Fault Tolerance (CFT) are prominent strategies. BFT is designed to handle not only crashes but also malicious behaviors within the system [9]. It ensures that the system can reach consensus even when some nodes act in unpredictable or adversarial ways, which ma include providing false or misleading information due to malicious attacks, software bugs, or hardware malfunctions. On the other hand, CFT addresses situations where nodes fail by crashing [10] and ceasing to respond, without engaging in deceitful actions. This model is inherently less complex and less resource-intensive than BFT, as it does not need to mitigate the potential malicious intentions of nodes. Prominent CFT algorithms include

Paxos, developed by Lesly Lamport [26], and the Raft algorithm, developed by Diego Ongaro and John Ousterhout [44], which we will primarily focus on in our work.

Introduced in 2014 [44], the Raft algorithm was designed to offer a more comprehensible and robust consensus method than other consensus algorithms in distributed systems. It stands out for its cleaner structure and ease of application, in contrast to other algorithms such as Paxos. Raft organises servers, also known as nodes in a distributed system, into three states: leader, follower and candidate. As illustrated in Figure 2.1, the transitions between states are clearly depicted. Raft provides efficient management of log replication and leader election, which are important elements in that algorithm for maintaining data coherence in distributed systems.

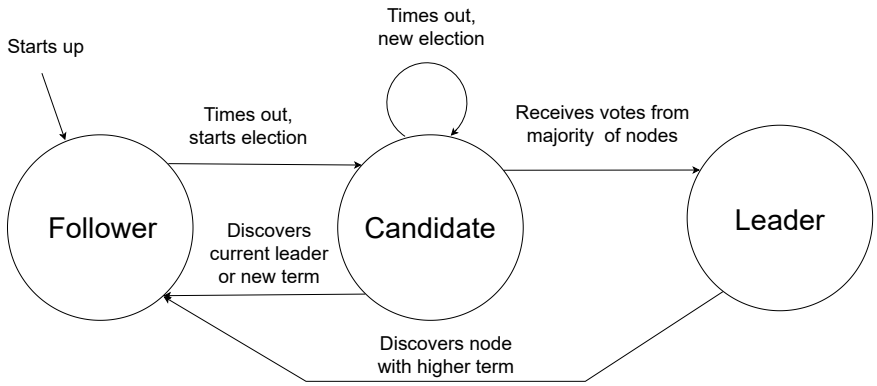


Figure 2.1 – State transition model for the Raft algorithm [20]

The leader election process is a central mechanism that ensures coherence and coordination between nodes. Initially, all the nodes in Raft begin in the follower state, which is a passive state where they only respond to requests from the leader. The election of the leader is initiated by a mechanism based on a waiting period. If a follower receives no signal, known as a `heartbeat`, from the current leader for a predetermined period of time, it can assume that the leader is failing or that there is a network problem. In response to this lack of communication, the follower changes state to become a candidate, which is the transition that initiates the election process.

As a candidate, the node first allocates a vote to itself and then sends voting requests in a message called `VoteRequest` to all the other nodes on the network. For a node to become leader, it must receive a majority of votes from all the active nodes in the group of servers taking part in the election, also known as a cluster. If a candidate receives this majority, he becomes leader.

Once elected, the leader takes over the management of the cluster by sending regular heartbeats to all followers to maintain its authority and to prevent a new election being triggered. These heartbeats are also used to confirm that the leader is active and functional. The leader remains

functional as long as it is able to communicate these signals effectively to the other nodes. If the leader stops responding, the election process can be reinitialised, and a new leader can be elected using the same process.

The leader receives all the client commands that need to be executed by the distributed system. These commands are first added to the leader's log. Each log entry comprises the command itself, an index which indicates the position of the command in the log and a term, which indicates a period of time during which the leader has been elected. Once a command is in its log, the leader replicates this entry on the followers' logs in a message called `AppendEntries`. It sends replication messages containing the log entries to all the followers. The leader then waits for the majority of followers to confirm that they have received and saved the log entry before moving on to the `commit stage`. After receiving confirmation from the majority of followers, the log entry is considered committed. The leader then applies the entry to its local state and sends the result of the command execution back to the client. The leader also informs the followers in his next heartbeat that they can also apply the entry to their local state. This log replication process ensures that all modifications to the system state are durable and consistent across the cluster, even in the event of a node failure.

2.2 Virtualization

Virtualization is a fundamental technology that has transformed the management and utilization of computing resources. It involves creating virtual instances of computing resources, such as servers, storage devices, and networks, allowing multiple virtual machines (VMs) to run on a single physical machine [22]. This technology enhances the efficient use of hardware resources and provides a flexible and scalable framework for managing computing environments.

Virtualization can be categorized into several types, each serving distinct purposes. A key type relevant to our work is server virtualization, which partitions a physical server into multiple virtual servers, each capable of running its own operating system and applications, as illustrated on Figure 2.2. This approach is widely implemented using virtualization technologies such as Quick EMUlator (QEMU) [12].

The benefits of virtualization are numerous and crucial for hosting solutions. One of the primary advantages is resource optimization; virtualization maximises the utilisation of physical resources by allowing multiple VMs to run on a single hardware platform [21]. This reduces the necessity for additional physical servers and lowers operational costs.

2.3 Cloud Computing

The advent of distributed computing services has rapidly transformed the way companies and individual users access and manage information technology (IT) resources. Cloud computing refers to an architecture where several servers interconnected via a digital network are used as a single computer, offering virtualisation of resources such as networks, servers, applications, data storage and services, accessible on demand by the end user [63]. Virtualization is a key enabler of cloud computing. By abstracting physical hardware and creating resources, virtualization allows cloud service providers to efficiently manage and allocate their infrastructure. Cloud computing can be categorized into three main service models Infrastructure as a Service (IaaS), Platform as a Service (PaaS), Software as a Service (SaaS) [38], that meet different needs, from access to virtual servers to complete development platforms and ready-to-use software applications.

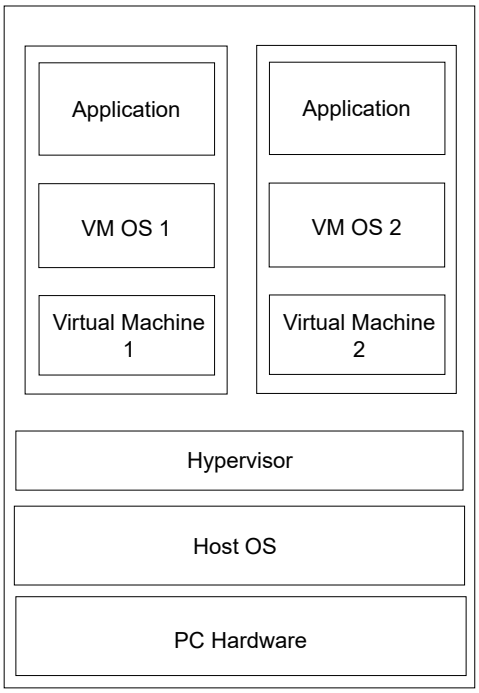


Figure 2.2 – Server Virtualization

Among the different models of cloud services, IaaS is particularly well-suited for the demands of collective hosting. It offers computing resources such as virtual machines and other infrastructure components critical such as storage and network capabilities for hosting web services in a decentralized manner. This setup allows users of hosted services to contribute resources to a shared hosting environment, enhancing scalability and flexibility while maintaining control over their individual contributions. Another significant advantage of IaaS in the context of collective hosting is its independence from physical infrastructure. By utilizing virtual machines and other virtualized components, IaaS decouples the physical hardware from the software layer. This separation allows each participant to host services without impacting the operation of their own system's hardware, ensuring seamless and efficient hosting operations.

2.4 Trusted Execution Environment

In the context of a rapidly evolving technological landscape, characterised by the increasing complexity and interconnectedness of systems, the necessity for robust data security measures is becoming increasingly apparent. In response to these challenges, trusted execution environments (TEEs) have been developed to address a critical need for protection, particularly in cloud architectures where the use of virtualisation and the outsourcing of resources amplifies security risks as data leakage or hypervisor vulnerability [11]. TEEs represent a hardware extension, enabling the execution of sensitive code in a protected environment separate from the main operating system and hyper-

visor [5]. By isolating operations within the TEE, data authenticity, confidentiality and integrity are guaranteed, which is important for maintaining system security in the face of today's threats like replay attacks. Within this secure framework, sensitive data is stored in a protected manner, using advanced encryption mechanisms [53]. This secure storage ensures that only authorised applications within the TEE can access this important information, thereby preserving its integrity and confidentiality, even if the device in which the TEE is integrated is compromised.

A number of significant processor manufacturers, including AMD and Intel, have developed TEE solutions tailored to their respective architectural frameworks. Among these technologies, AMD has introduced Secure Encrypted Virtualization (SEV), while Intel has proposed its own system, Intel SGX [24]. In the context of this study, we will focus on the solution proposed by AMD, examining how SEV contributes to the security of data and applications in virtualized environments, particularly in the cloud.

2.4.1 AMD Secure Encrypted Virtualization

The AMD SEV technology, which was introduced in 2017 [37], has been designed with the objective of enhancing the security of cloud environments through the implementation of significant innovations in virtual memory management. AMD has developed extensions, including SEV-Encrypted State (SEV-ES) and SEV-Secure Nested Paging (SEV-SNP).

The AMD SEV-SNP technology safeguards virtual machines in terms of confidentiality and integrity by implementing rigorous access controls that prevent any unauthorised manipulation or access to data and code, thereby effectively protecting VMs against both internal and external threats.

From an external perspective, SEV-SNP serves to segregate the VM, thereby safeguarding it from potential interference from hypervisor and other malevolent VMs on the same host. This isolation is achieved by encrypting the VM's memory with a unique key, preventing any unauthorised access. This guarantees strict separation between VMs and defends against side channel attacks. Furthermore SEV-SNP ensures that VMs can only access and modify the data that they have written, including during the transfer of memory pages or the migration of VMs. This feature employs a reverse mapping table (RMP) to confirm that only the rightful owner of a memory page can modify it, thereby protecting against replay attacks and data corruption [28].

Internally, this technology enables the establishment of different levels of privilege within the VM itself, through the use of Virtual Machine Privilege Levels (VMPL). These levels of privilege determine the access permissions for various operations within the VM. Consequently, even in the event of an internal compromise, the impact can be contained by limiting the access capabilities of the affected components.

2.4.2 VM Security Verification

The security verification process of VMs using AMD SEV-SNP technology is based on an attestation system. At the time of VM initialisation, SEV-SNP generates encryption keys used to secure the VM's memory to ensure data security to confirm its authenticity and integrity.

Attestation with SEV-SNP is carried out by the AMD Secure Processor (ASP), an independent component integrated within the main processor, which oversees the security of the VMs. When the VM is configured on the host AMD SEV-SNP, the ASP receives the initial code and data in form of an image send by the hypervisor.

To access encrypted resources, which are encrypted at the initialisation of the VM, the VM must prove that its configuration has not been altered and that it is running in a secure environment to

obtain the decryption key from a key server. To do this, the initial image and data of the VM are compared with the version held by the ASP, as provided by the hypervisor. If there is a match, the VM will start up successfully and will obtain the necessary key to decrypt its memory.

To check the integrity of a VM by a third party, an attestation report can be used. To obtain it, the VM makes a request including additional random data, such as a public key, to the ASP as in Figure 2.3, which produces an attestation report that compiles essential information such as the hashes of the initial data, the identity of the VM, firmware versions, and other information [25]. This report is signed with the versioned chip endorsement key (VCEK) which is unique for each AMD Central Processing Unit (CPU) and confirms the report's authenticity.

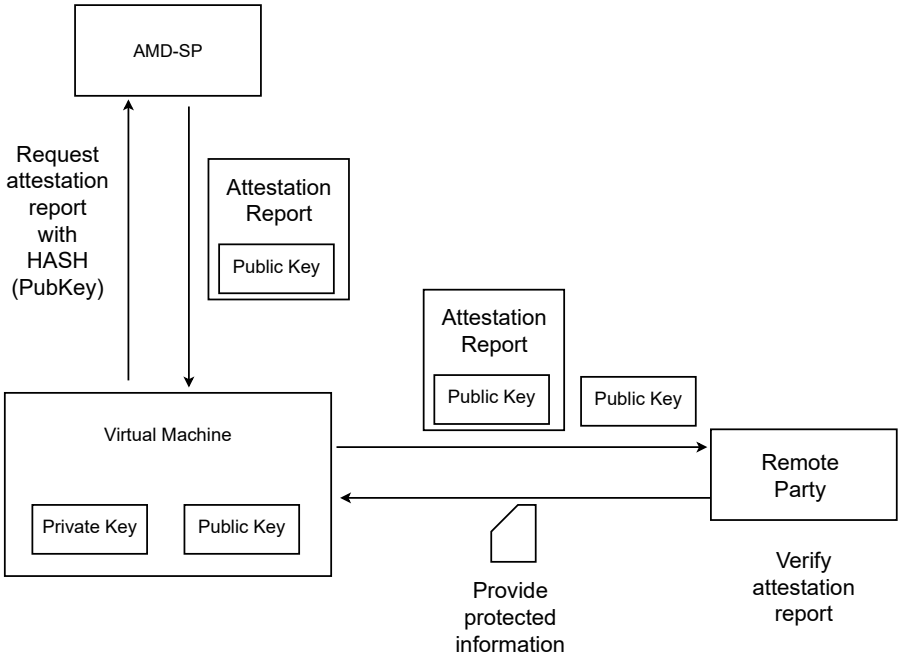


Figure 2.3 – Attestation Report Cycle [25]

This report includes specific data provided by the VM, which are used by the verifying entities to confirm that the VM operates in a secure environment and that its configurations have not been altered. The verifying entities must also check the signature of the report.

To reinforce trust in these attestation reports, AMD has developed a Key Distribution Service (KDS), which certifies the authenticity and the validity of the VCEKs used to sign the reports, ensuring that these reports are issued by authentic and updated components.

In short, TEEs, in particular AMD-SEV, are a critical component for reinforcing security in the strategy of collective web site hosting within cloud environments. AMD-SEV and its subsequent development, such as SEV-SNP, offer sophisticated security mechanisms that ensure the integrity, authenticity and confidentiality of data handled in VMs in this distributed hosting architecture.

2.5 Hypertext Transfer Protocol

At the heart of Internet communication is the Hypertext Transfer Protocol (HTTP), an application-level protocol that enables the transmission of information between user web browsers and servers that host web content. Originally developed in 1990, HTTP was designed to facilitate basic interactions between a client, typically a web browser, and a server [41]. Characteristically, HTTP operates as a stateless protocol, meaning it does not maintain any state between individual transactions but to manage state during successive interactions, techniques such as cookies are often employed. HTTP has been progressively modified to support the increasing complexities of contemporary network architectures, particularly those that incorporate distributed web hosting. In our research, we are using HTTP/1.1 due to it widespread compatibility within distributed environments.

The HTTP/1.1 version marked a turning point in the evolution of the HTTP protocol, introducing a number of significant new features. One of these was the introduction of persistent connections, which allow multiple requests and responses to transit over a single Transmission Control Protocol (TCP) connection [66]. This optimisation reduces latency and improves the use of network resources. In addition, HTTP/1.1 innovated with pipelining, allowing several successive requests to be sent without waiting for responses to previous requests [56], which improves traffic management and server efficiency. Furthermore, the HTTP/1.1 protocol introduced the technique of chunked transfer encoding [30], which facilitates the management of HTTP messages by breaking them in to several part when the total size is not known in advance. This method not only allows for more efficient handling transmission of data over the network but also helps avoid transferring large volumes of data in a single operation, enhancing overall network performance.

2.5.1 Client-Server Model

The client.server architecture is a central framework in network interactions, typically involving clients such as web browsers. In this model, the client initiates communication by sending a request to the server via TCP, which is the primary protocol used to ensure reliable and ordered transmission of data across the network. TCP segments the data into packets, manages their transmission, and ensures their correct reassembly at the destination.

Once the server receives these packets, it processes the request and uses TCP again to send the required data back to the client. This structured exchange of request and response, facilitated by TCP, allows for efficient information exchange. It supports a variety of user activities on the web, including accessing web pages, submitting forms, and using online applications. This reliability and efficiency in handling data are what make TCP indispensable in the client-server communication model. Figure 2.4 illustrates a single client-server model.

2.5.2 HTTP Messages

Communication between clients, which are usually browsers, and servers takes place by exchanging HTTP messages. These messages are structured into requests and responses to facilitate exchanges in accordance with the rules of the HTTP protocol. Each message is made up of a start line, headers and often a message body. The start line of a request includes an HTTP method such as GET or POST which are widely used in our study, the target URL and the protocol version, while in the response, it contains the protocol version, a status code, and a descriptive sentence about the status. To visualize the structure of HTTP messages more clearly, please refer to the Figure 2.5, which presents both the HTTP messages and the dynamic between the client and the server.

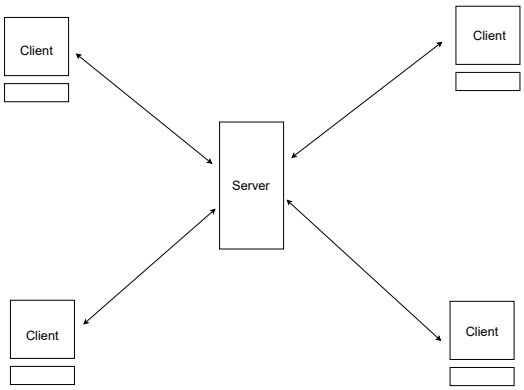


Figure 2.4 – Client-Server Model

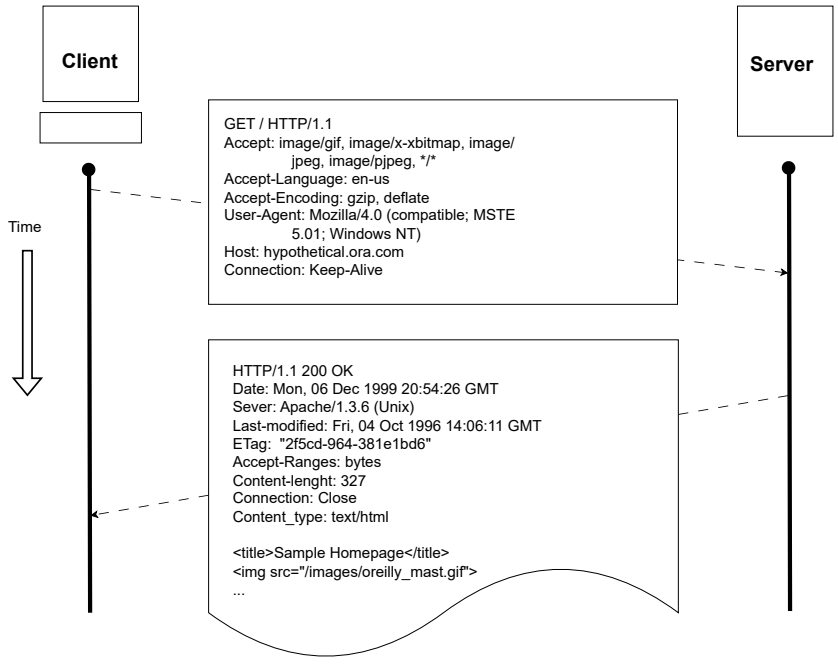


Figure 2.5 – Client-Server Interaction exchanging HTTP messages [64]

Request methods determine the actions that the client wishes to perform on resources, in particular on a server. The GET method is used to retrieve information from a server. When a user clicks on a link or types a URL in the browser address bar, a GET request is sent to the server to request the content of the specified page. GET requests are designed to have no side-effects, which means that they must not modify the state of the server. they are ideal for carrying out requests that do not change the information stored on the server, such as searches or data filtering. The parameters for the GET request are often included in the URL itself. Unlike GET, the POST method is used to send data to the server to create or modify a resource. It is typically used to submit web forms. When a user fills in a form and submits it, the information from the form is encapsulated in the body of the POST request and sent to the server, where it can be used to update a database or perform other actions that change the state of the server. The data sent by POST is included in the body of the request, which means that larger quantities of data and more complex types of data can be sent compared with GET. Unlike GET parameters, POST data is not exposed in the URL, which makes the POST method more secure for the transmission of sensitive data.

The headers, forming key-value pairs, transmit vital information such as cache directives, content details such as type and size, authentication information and many other metadata, and also facilitate structured data transmission between the client and the server. The message body, which may be absent in certain requests such as GET, is used to send data in requests or to return data requested in responses, such as the HTML of a page or JavaScript Object Notation (JSON) data. There also exist response status codes, which indicate the result of the request, which are important because they provide crucial feedback on each interaction. These codes indicate the success or failure of a request and are categorized to reflect various types of responses. Table 2.1 provides a detailed overview of various status codes that we will frequently encounter in our work.

2.6 Web Proxy

Web server proxies function as intermediaries between clients and servers, acting as mediators who transfer HTTP messages between the two parties [17]. There are various types of proxy servers, each serving distinct purposes within network architectures. Common types include forward proxies, transparent proxy, anonymous proxies, public proxies and reverse proxies.

Particularly relevant to our research is the reverse proxy. Unlike a forward proxy that serves clients within a local network protecting and anonymizing internal users, a reverse proxy sits in front

Code	Signification	Category
200	OK	Success
301	Moved Permanently	Redirection
400	Bad Request	Client Error
401	Unauthorized	Client Error
403	Forbidden	Client Error
404	Not Found	Client Error
500	Internal Server Error	Server Error
502	Bad Gateway	Server Error
503	Service Unavailable	Server Error

Table 2.1 – HTTP Code Status in our Study

of one or more web servers and acts as an intermediary for servers rather than clients. It intercepts requests from clients before they reach the servers.

Proxies play a versatile dual role in network architectures as depicted in Figure 2.6, acting both as servers and as clients, which significantly enhances their utility in managing and optimizing network interactions. As servers, proxies directly handle requests from clients by providing the needed responses, effectively fulfilling client requests on their own. Conversely, when proxies receive requests that must be passed on to a web server, they switch roles and functions as clients. Figure illustrates the operational dynamics of proxies within a network infrastructure.

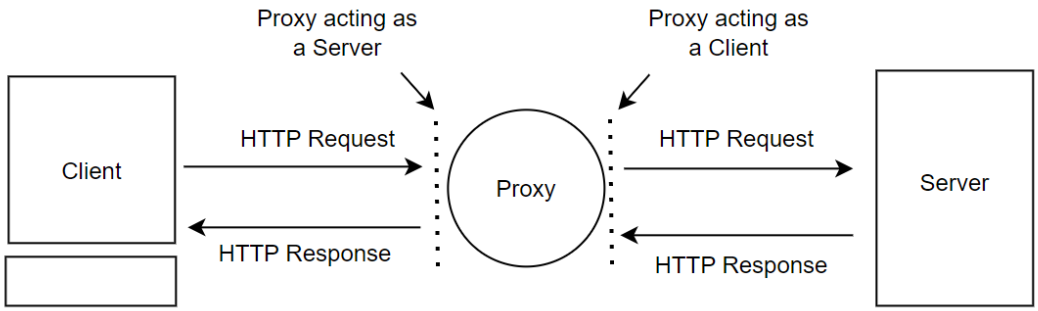


Figure 2.6 – Proxy acting as Client and Server

3 DESIGN

This chapter introduces the design of GenerousHost, which represents a notable advancement in the field of web hosting designed to address the shortcomings of current web hosting models. It begins with a problem statement that identifies and clarifies the specific challenges our system seeks to overcome. Following this, we present a comprehensive system model that articulates both the operational environment and the security landscape of GenerousHost. Finally, the general design of the system is presented, detailing components and actors interactions within the system.

3.1 Problem Statement

The contemporary web hosting landscape is largely dominated by models that struggle with scalability, cost-efficiency, and security [8]. Existing hosting solutions such as dedicated servers and shared hosting involve significant costs and complexity in scaling [4, 23]. These models require significant investments in physical infrastructure and ongoing maintenance, leading to high operational costs and complex scaling processes involving lengthy delays due to the need for acquiring, installing, and rigorously testing new hardware, along with managing resource allocation. Additionally, centralized hosting models are prone to service outages as they rely on single points of failure for data processing and storage [39, 48]. Even minor issues like hardware malfunctions, software issues, or power outages can quickly lead to widespread disruptions, affecting all users and exposing the inherent vulnerabilities in maintaining service continuity in the hosting environment. These challenges significantly constrain the potential of small to medium-sized enterprises and individual developers in particular, because they often lack the resources necessary for substantial investments in hosting infrastructure.

In response to these multifaceted issues, our proposed GenerousHost aims to revolutionize the hosting paradigm by decentralizing it. By utilizing the unused computing resources of the computers belonging to users of the hosted services or organizations, who will be referred to as donors in our work, our model aims to create a distributed and easily scalable web hosting environment, reinforced with security measures to protect both data and system operations. To achieve this, we incorporate Trusted Execution Environments (TEE) as security measure to ensure that critical tasks are processed within a secure, isolated area, safeguarding from external threats and system vulnerabilities.

3.2 System Model

The system model for GenerousHost serves as a foundational framework that defines the operational environment and the security landscape. This model is constructed to include a set of assumptions

that shape the expected behavior of the system under normal and adversarial conditions. It aims to ensure clarity in the system's operational boundaries and to prepare for potential security threats outlined in our attacker models.

3.2.1 Assumptions

Several key assumptions guide the design and operational aspects of the collective hosting system. These assumptions serve as foundational principles upon which the system's architecture and functionality are built.

Among these assumptions, they are the willingness of donors to contribute their computing resources to the system. Despite the potential for untrustworthy intentions among donors, their participation is driven by the control they gain over the service location and the continuous access to the service even if the central server experiences downtime. It is also essential that donors possess adequately equipped hardware and the necessary minimum resources to support the hosting of the service.

In our system, all donor machines function as nodes that contribute to hosting services. These nodes are geographically dispersed and connected by network that enables communication through message exchanges. We assume the network operates under partial synchrony and is unreliable, which may lead to delayed or lost messages, with connection possibly failing and nodes potentially disconnecting and continuing to run independently. Nodes are expected to crash or fail at arbitrary times and can restart and rejoin the system. On each nodes runs a VM with restricted access to the host computer, which prevents to unauthorized access, safeguards the node and the application of the host computer. Donors may freely join or exit the collective hosting system. We also assume that external attackers cannot intercept messages sent between nodes nor access the VMs installed on each node. Lastly, service providers do not trust the resources, so there is a mechanism to ensure that their application is running in a safe environment. By choosing to host their services utilizing `GenerousHost`, service providers can focus more on developing and enhancing their services, while spending less time on the technical complexities of server capacity expansion and scaling.

3.2.2 Attacker Models

When designing `GenerousHost`, several attacker models must be considered to ensure the system's availability, integrity and confidentiality. These models help identify a broad range of potential threats that could disrupt the seamless functionality of the system. By thoroughly analyzing and mitigating these threats, we can safeguard the system against disruptions and protect sensitive data, thereby maintaining the service's availability, integrity, and confidentiality.

In our system, a particularly significant threat scenario arises from the potential unauthorized access to data stored and processed within the VMs on donors' computers, either by malicious donors themselves or by external attackers who have managed to gain control of the donor's machine. To counteract this, `GenerousHost` employs TEEs, which are important for data security by protecting VM memories against data corruption and unauthorized access [55]. TEEs utilized encryption mechanisms specifically designed to thwart memory access attacks, ensuring that even if attacker accesses encrypted data, decrypting it without proper authorization remains highly unlikely. TEEs significantly strengthens the security of our hosting environment, ensuring that sensitive information is protected against both direct and sophisticated breaches such as replay attacks [55], Ciphertext side channel attacks [33].

Additionally, the distributed nature of our hosting system, which spreads the hosting duties across multiple donor machines, inherently reduces the risk of denial-of-service (DoS) attacks. While

this approach does not completely eliminate the possibility of DoS attacks, it significantly mitigates their impact by dispersing the load and making it difficult for attackers to disrupt the entire service.

3.3 Theoretical Framework

GenerousHost operates as a decentralized system, enabling individuals to enhance its network by donating server resources. This collective hosting approach leverages servers spread across various locations to improve web service resilience and performance. It reduces the impact of local disruptions and optimizes service delivery by bringing resources closer to end-users, thereby minimising latency and improving the quality of the service. The theoretical underpinnings of GenerousHost incorporates the use of TEE for secure data execution and concepts from distributed computing systems, particularly cloud computing, redundancy and fault tolerance, which are integral for preventing total system failure, ensuring continuous availability, and maintaining operations despite component failures.

Operational mechanisms employed by GenerousHost include data synchronisation algorithms in particular consensus algorithms, which are employed to maintain consistency across servers. Additionally, load balancing techniques are employed to evenly distribute workloads, thus preventing any server from becoming overwhelmed. This framework inherently supports scalability, as more donors contribute computing resources, the system can expand to handle increased loads efficiently. Moreover, the use of redundancy and decentralization ensures that GenerousHost can reroute traffic to operational nodes in the event of nodes failures, which significantly enhances its reliability and ensures service continuity.

3.4 System Architecture

The system architecture has been designed with scalability and flexibility in mind, ensuring that the various components and actors interact seamlessly with one another. Each component plays a specific role within the system, collectively contributing to the system's overall functionality.

3.4.1 System Entities

The dynamics of our system are driven by a number of actors, including the service provider and a diverse range of service users. The architectural framework is founded upon a number of fundamental components, including the core hosting infrastructure, and donor machines. Figure 3.1 provides a high-level overview of the system, which illustrates both its components and actors.

3.4.1.1 System Users

GenerousHost is mainly composed of three main users. The first is the service provider, who is the developer and requires hosting resources to support their project. He is responsible for the hosting services and delivering enhancements and new features. The second is the donor, who is a volunteer who may or may not use the service provided by the developer but want to support projects by donating their underused resources. Donors act as cloud providers for the service provider, as they provide the hosting infrastructure to developers. Finally, there are end-users who simply utilise the service offered by developers through GenerousHost.

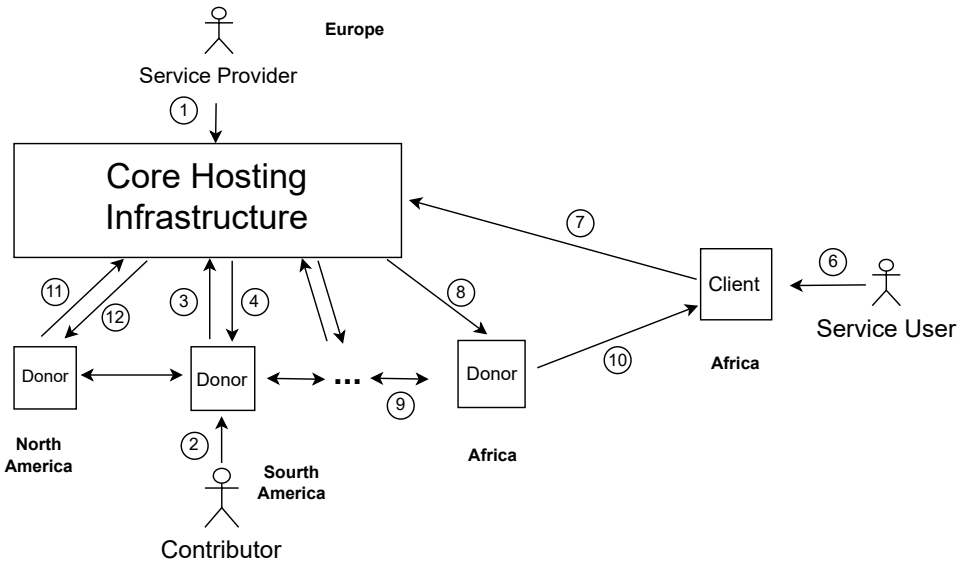


Figure 3.1 – GenerousHost Sytem Overview

3.4.1.2 Technical Components

With regard to the technical components, the core hosting infrastructure is comprised of a group of servers, as illustrated in Figure 3.2. This includes the management server, the central hosting server, and the load balancer. Each server group fulfils a distinct role within the infrastructure.

The management server is responsible for the registration and deregistration of donor computers within the system. The primary function of this server is to maintain an up to date list of active machines and manage their participation status. Additionally, it plays a crucial role in facilitating the integration of new donors into the system. During the registration process, the management server collects data on the donor machines that register. In contrast, the central hosting server on their side provides the web services that donors assist in hosting, They serve as the backbone of the service delivery, ensuring that the hosted applications are running and accessible to end-users. The function of load balancing is to distribute incoming requests to the servers within the system. The load balancer employs metrics to identify the most appropriate server to handle each request. This decision is based on proximity information, derived from several metrics including round-trip time, number of hops, geographic distance, bandwidth availability, and more [54]. By optimizing request routing based on these metrics, the load balancer is able to significantly improve response times.

3.5 Donors Integration into the System

The process for a donor to join the system begins after the service provider has made the service operational by using `GenerousHost`, which corresponds to step 1 in Figure 3.1. A donor may become a member of a hosting network by first expressing an interest in joining. This involves selecting the project they wish to support, registering on the corresponding website (step 3), and downloading the virtual machine with an identifier along with the necessary software to host the service.

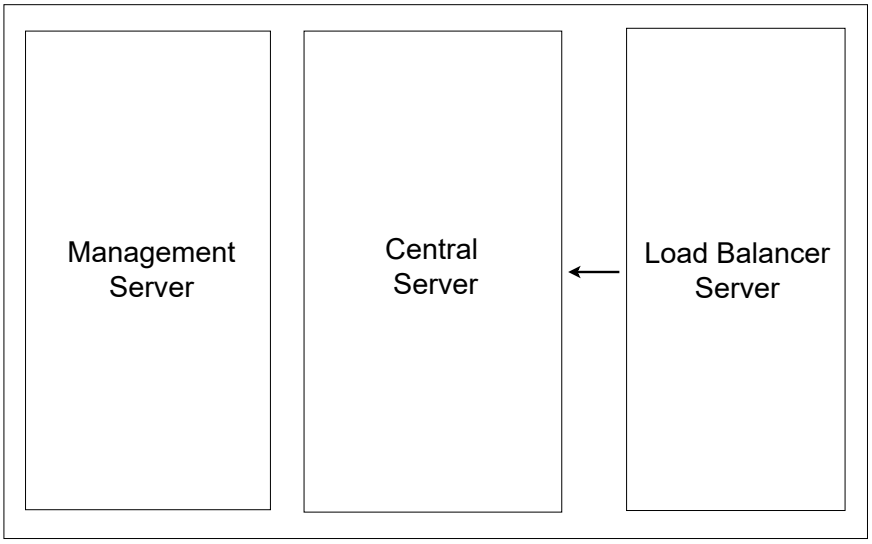


Figure 3.2 – Core Hosting Infrastructure

Once the registration is complete, the management server has collected all necessary information including IP address and location data from the new donor. To mark the successful integration of the new donor machine, the management server sends (step 4), after the deployment and start up of virtual machine, the list of coordinates and the IP addresses of already active donors so that the donors can communicate.

This deployed virtual machine, secured by the use of TEE, is equipped with all the necessary tools for hosting the service. Within this machine, a web proxy plays a crucial role in the synchronization of data between the donor machines. Additionally, a web server is installed, which houses the service to be hosted. Figure 3.3 shows the configuration of the donor machine after the virtual machine installation.

Beyond the virtual machine, there are also applications and processes belonging to the contributor that are running. Once the tools are installed and all components are operational, the donor machine proceeds to synchronize with other machines already hosting the service corresponding to step 9 in Figure 3.1. This synchronization updates its data so that it can start responding to incoming requests. After this step, the machine is fully integrated into the network and ready to contribute effectively to the distributed hosting environment.

3.6 Donors Exiting the System

In light of the fact that donors are able to freely join and exit the system, the process for a donor to leave the system begins when the donor decides to disengage. The donor communicates this intention by sending a message to the management service, informing it to their decision to leave

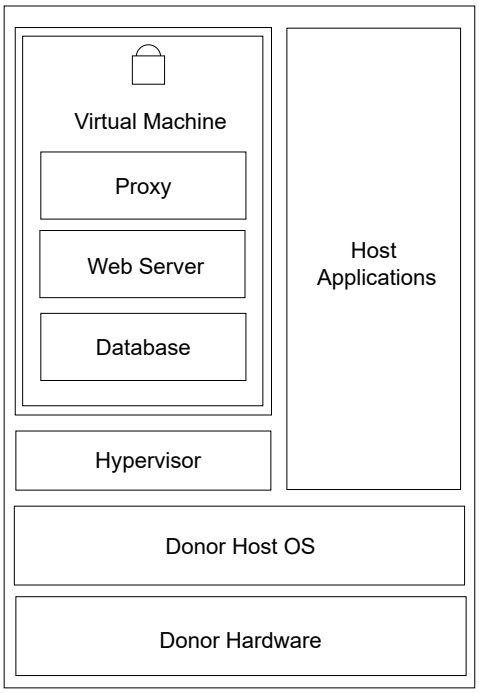


Figure 3.3 – Donor Computer Configuration

(step 11 in Figure 3.1). Upon receipt of this message, the management service proceeds to deregister the donor from the system.

Once deregistration has been completed, the list of available donors is updated to ensure that the load balancer no longer routes new requests to this node. It is of the utmost importance during this process that the donor does not have any ongoing operations in progress, as this may result in data loss. This precaution helps to maintain consistency, integrity and continuity of the hosted services.

In response to the deregistration (step 12 in Figure 3.1), the management service transmits commands to uninstall the virtual machine that was previously installed on the donor's machine. The objectives of this uninstallation is to free up the resources that were previously allocated. Once the virtual machine and any associated configuration have been removed, the donor's contribution to the network is deemed complete.

3.7 End-User Access to Hosted Services

The interaction between end users and hosted services within our system is a multi-step process. The interaction begins when the service is operational and the user decides to access it. This represents step 6 of Figure 3.1, where the user's intentions to utilize the service.

The user's journey starts with a request to access the service (step 7 in Figure 3.1) to the core Hosting Infrastructure. Once the request reaches the load balancer, it is then routed to the server

best positioned to provide the optimal response time (step 8 in Figure 3.1). This decision is made based on number of metrics, including the number of network hops required to reach the server [54]. The aforementioned metrics facilitate the delivery of a user experience that is characterised by minimal latency.

The server selected for processing the user's request is the one that receives the request. In the event that the request results in a change to the server's state, step 9 in Figure 3.1 is initiated before a response is sent. This step involves synchronizing all changes across every server in the hosting network to maintain consistent state information. Finally, the server responds to the user (step 10 in Figure 3.1), thus completing the interaction cycle.

3.8 Service Update

Updating services is a critical process for improving functionality and resolving issues with the service hosting in the system. In a distributed hosting environment, this process is particularly delicate because it requires updates to be rolled out without interrupting the service, ensuring that the updates are not only seamless to end users, who should notice no downtime, but also uniformly applied across all hosting servers. In order to achieve this, we adopt a strategy of phased updates based on the blue-green deployment approach [14], where a group of servers is selected and temporarily decouple from the operational network for updates while other servers remain operational to handle user requests. This approach minimizes downtime and maintains service availability throughout the update process. As the selected servers are updated, any new data handled by the operational servers is synchronised back to the updated servers to maintain data consistency across the network. This ensures that when these servers are reintegrated into the service, they are up to date and can seamlessly continue service delivery. After updates are applied, a thorough testing phase ensures that the updated servers operate correctly without introducing errors. If the tests fail, the update process is interrupted. The system then initiates a rollback procedure to restore the servers to their previous stable state. Once a group of servers passes these tests, the update process is replicated on the next group of servers in a rolling fashion until all servers in the network have been updated.

The phased update approach may encounter significant challenges due to compatibility issues, particularly those related to database schema changes. When updating an application, developers often need to alter the database schema, which defines the structure and organization of the data. These changes can include adding new tables, modifying existing ones, or changing the relationships between tables. If not handled properly, these can lead to data loss, corruption, or system errors. To mitigate these risks, developers must implement code changes with backward compatibility in mind [19]. This means ensuring that the new version of the application can still work with the old version of the database schema, allowing for a seamless transition. One effective strategy for achieving this is the use of transitional database schemas.

Transitional database schemas represent intermediary steps that facilitate the coexistence of both old and new data models during the update process. They play a crucial role in ensuring an uninterrupted transition without disrupting the functioning of the service. In the event that new data fields or tables are required, they are incorporated into the existing schema without removing or altering the existing ones. This approach allows the old version of the application to continue functioning without any disruption while the new version starts using the new structures. By maintaining the old schema alongside the new one, developers ensure compatibility and continuity. In some cases, the application is updated to write data to the old and new formats. It ensures that all data in both formats, which is especially useful if a rollback to the previous version becomes

necessary. Having data in both formats provides a safety net, allowing developers to reverts the old schema without losing any data. Service providers must also write migration scripts to migrate data to the new schema format. These scripts run in the background and ensure that all data conforms to the new structure. During this phase, the application may be designed to handle both old and new formats, reading from the appropriate structure as needed. This gradual migration helps in identifying and resolving any issues without affecting the overall functionality. Once the transition is complete and the schema is fully adopted, the old structures can be safely removed. This cleanup phase eliminates any legacy components that are no longer needed.

4 IMPLEMENTATION

This chapter explores the transformation of the `GenerousHost` system from conceptual design to a fully functional distributed hosting solution. It delves into the systematic deployment of the system's infrastructure, the strategic integration of software components, and the adoption of security measures to safeguard data and ensure the integrity of the entire system. By detailing each phase of the implementation process this chapter highlights how abstract theoretical concepts are realized into practical, operational architecture.

4.1 Environment Setup

In the development of our work, we have selected hardware to meet the demands of our collective hosting system. Employing the right equipment is important for ensuring optimal performance, security, and reliability of the service.

Our infrastructure is based on a Personal Computer (PC) running Ubuntu 22.04.4 LTS as operating system. This machine is powered by AMD EPYC processors. A pivotal aspect of our setup is the integration of AMD's advanced security extension, AMD-SEV-SNP. This feature is implemented in all host machines, which then support the installation of secure guest virtual machines enabled with AMD-SEV-SNP.

AMD-SEV-SNP is utilized in our system to enhance the security of data processed and stored on virtual machines [45]. This technology facilitates the creation of isolated execution environments that protect the VMs against potential attacks, even if the hypervisor is compromised [46]. For managing these VMs, we employ QEMU, a machine emulator for its compatibility with various system architectures [7]. One of the advantages of using QEMU in our system is its support for QEMU Copy On Write version 2 (QCOW2) images. These images are a type of disk image format used by QEMU [40], which features a better disk compression ratio, making it an efficient choice for virtual disk storage. QEMU images primarily serve to store operating systems and applications in a virtual format. A disk image complete representation of a physical hard drive, including all data, partitions, and file systems necessary to run an OS. This allows VM's users to create, save and clone complete working environments with ease. QEMU images thus facilitate the management of operation systems, enabling the testing of different configurations without affecting the host operating system.

Each virtual machine in our system is preconfigured with a suite of essential software stacks tailored to facilitate seamless hosting service operations. The base of this configuration is an Ubuntu operating system, which provides a stable platform for the subsequent applications. A key component of the setup within the VM is a reverse proxy, developed using Node.js, particularly the framework Express.js. We have named this reverse proxy `raft-node`. It intercepts all incoming request before

they reach the final applications. It includes a Raft algorithm module, implemented in JavaScript. This module guarantees that all incoming POST requests are executed by all servers in precisely the same order, thereby maintaining uniform data consistency across the entire server network. In addition to the reverse proxy, virtual machine houses a web server using Nginx [58], which serves our web application that we aim to provide services to users. Complementing Nginx, we use MariaDB as our database management system for data persistence [50]. The Table 4.1 summarizes the software stack used in our VMs.

Type	Software
Operating System	Ubuntu 22.04
Web Server	Nginx
Database Server	MariaDB
Proxy Server	raft-node
Web Application	HotCRP and Test-app

Table 4.1 – Summary of software stack in the VMs.

The configuration described will be uniformly implemented across our system for all donors. This ensures that every donor experiences consistent and reliable interactions, reflecting our commitment to maintaining standardized processes. By applying the same setup universally, we can guarantee that the system's performance and security are consistent for all users.

In our core hosting infrastructure, a group of servers interacts to ensure continuous availability of the hosted service, providing users with permanent access to the service. Among these servers, we have a management server that handles the management of the various donor machines, a server dedicated to managing the main service so that if no donor is available, the core hosting infrastructure continues to serve users. Additionally, we have a load balancer that distributes client requests among the servers where the final application is hosted.

4.2 Virtual Machine Startup

During the implementation of our system, we leveraged the AMDSEV project available on GitHub [2]. This project enables us to build the guest Linux kernel, QEMU and the open virtual machine firmware (OVMF) Basic Input Output System (BIOS) required to launch a SEV-SNP guest on our Linux machine with an AMD processor that supports SEV-SNP security feature. Upon completing the build of the SEV-SNP guest, we executed scripts that we implemented to prepare the QEMU image and the user data image. This user data image includes user account configurations, packages to be installed, and necessary files to set up the hosting environment inside the VM, such as files related to the reverse proxy, other scripts files and the application we intend to host on the VM.

The AMDSEV project provides a script called `launch-qemu` that facilitates the configuration and startup of a VM. This script accepts options that define the VM's configuration upon startup. These options include specifying the QEMU image to be used, the user data image, the type and the number of virtual processors the VM will use, the amount of memory, the network configuration options, the option to enable SEV-SNP security and several other options. In our implementation for starting a VM, we used the options and values summarized in Table 4.2, in addition to the default options provided by the `launch-qemu` script in the AMDSEV project.

Once the options and values are provided, the launch-qemu script is executed using the root privileges, allowing the VM to start up properly.

Option	Meaning	Value
`-hda`	Hard disk file	sev-guest.qcow2
`-cdrom`	CDROM	user-data.img
`-smp`	Number of virtual cpus	16
`-mem`	Guest memory size in MB	8192
`-default-network`	Enable default usermode networking	
`-sev-snp`	Launch SNP guest	

Table 4.2 – Virtual Machine Startup Options

Before the VM is launched, an unencrypted image containing no secrets is prepared. This image contains only the initial code and data necessary for the VM to start. A cryptographic hash called launch-digest of this image is created by the ASP, representing the exact content of the image and securely stored it. At the end of the launch process, a signed identity block, containing data that uniquely identifies the VM and the expected launch-digest, can be submitted [55]. The ASP compares the launch-digest it holds with the one in the identity block. If the two match, the VM starts successfully; otherwise, the launch fails to prevent the execution of potentially malicious or unverified code.

In our implementation environment, we needed to run multiple VMs with the same configuration and software stack simultaneously. To achieve this, we create a script called `run`. This script takes an ID parameter, which serves as a unique identifier for each VM instance within our environment. This script creates a VM instance by duplicating the base QEMU images, which were pre-configured with all necessary settings. Additionally, the script sets up TAP interfaces on the host to enable network communication with the instantiated VMs.

4.3 Virtual Machine Network Configuration

In a SEV-SNP guest VM, the default networking backend is the User Networking. This type of networking has significant limitations, including the poor performance, lack of ICMP traffic support, and the inability for the guest to be directly accessible from the host or external network [49]. It does not allow bidirectional communication or message exchange between two VMs or between a VM and the host machine. To enable essential communication between system entities, we modified the default configuration of the VM within the `launch-qemu` file.

We replaced the `user` network type with the `tap` type, which allows for a more robust and performant networking integration. The TAP backend enables the VMs network interfaces to connect directly to the host's network, facilitating bidirectional communication. To configure the network interface of the VMs, we created a TAP interface for each instance, associating a unique identifier and name. Then, we configured a virtual network card for each VM, linking this card to the created TAP interface and assigning a unique MAC address to each VM using the instance identifier, ensuring the uniqueness of each MAC address. This uniqueness is crucial to avoid network conflicts and to clearly identify each machine on the network. Table 4.3 shows the configuration we uses for our implementation.

Furthermore, we set up the VMs to obtain an Internet Protocol (IP) address to ensure they are correctly identified and accessible in the host's network. This setup not only allows for better management of network resources but also enhances the efficiency of communication between VMs and also between VMs and the host.

Command/Option	Meaning
`-netdev tap`	Use a TAP network device.
`id=vmnic${INSTANCE_ID}`	Assigns a unique identifier to the network interface.
`ifname=tap${INSTANCE_ID}`	Specify the name of the TAP interface.
`script=no`	Disables automatic configuration scripts for the TAP interface.
`downscript=no`	Disables scripts when the TAP interface is deactivated.
`-device e1000`	Uses a virtual e1000 network card.
`netdev=vmnic${INSTANCE_ID}`	Associates the virtual network card with the created TAP interface.
`mac=52:54:00:12:34:5${INSTANCE_ID}`	Assigns a unique MAC address to the virtual network card.
`romfile=""`	Specifies that no ROM file is used for the virtual network card.

Table 4.3 – Virtual Machine Network Configuration

4.4 SEV-SNP Attestation Reports

Ensuring the integrity and security of VMs in our system is of paramount importance, particularly as we operate in an environment where the donor machines are not trusted, and VMs can be manipulated to extract information or for other malicious purposes. To verify the confidentiality and the integrity of VMs, AMD's SEV-SNP technology offers a mechanism for attesting to the state of a VM, ensuring its trustworthiness and safeguarding the data it encompasses.

4.4.1 Attestation Report Creation

An SEV-SNP attestation report is a binary file that outlines the state and security measures of a VM. This report is crucial for validating that a VM is running in a secure and unaltered state. In our project, we used a tool called `snp-guest` to generate these attestation reports for the VMs we used. This utility communicates with the AMD SEV-SNP guest firmware device, enabling several operations, including creating attestation reports, reading them to access their content and more other operations[62].

After the VM has started, `snp-guest` is used to generate the attestation report by providing the tool specific parameter including a data file that could containing information, such as public key and that we would like to add in the attestation report.

4.4.2 Attestation Report Contain

An attestation report contains several key pieces of information about the VM and the platform the VM is running on [47]. Among this information is the version of the report, the guest security version number indicating the security version of the guest software. Following this, the guest policy defines the security policies and capabilities for the VM. This includes other several information. Additionally , the report includes Family ID and Image ID, which are unique identifiers for the processor family and the VM image, respectively. The virtual machine privilege level (VMPL) indicates the privilege level of the VM, where higher levels have more privileges and control over resources.

The signature algorithm specifies the algorithm used for signing the attestation report. The current trusted computing base (TCB) provides details about the current version of critical security components like the microcode, SNP, TEE, and Boot Loader. Moreover, the platform info field in the report indicates security feature enabled on the platform, such as transparent secure memory encryption (TSME) and simultaneous multithreading (SMT).

The report data field contains user-specific or session-specific data, often used to link the report to specific data or sessions. Another critical field is the measurement, which is represented as a hexadecimal string, resulting from the application of a cryptographic hash function such as SHA-256 on the initial state of the VM. It is used to verify the integrity of the loaded software. The report ID is a unique identifier for the attestation report, used for tracking and referencing. Additionally, the report ID migration agent is an identifier for the migration agent, used for migrated VMs. The value of these IDs are also in hexadecimal format in the attestation report.

The chip ID is a unique identifier for the processor, used to verify the origin of the report. The committed TCB filed represents the committed version of the TCB. Finally, the signature (R and S) fields in the report are components of the digital signature, ensuring the report's authenticity and integrity.

Table 4.4 provides an overview of the fields in the attestation report, distinguishing between information related to the guest SEV-SNP VM and the platform.

Guest SEV-SNP VM	**Platform**
FAMILY_ID	CHIP_ID
IMAGE_ID	PLATFORM_INFO
GUEST_SVN	CURRENT_TCB
MEASUREMENT	COMMITED_TCB
ID_KEY_DIGEST	REPORTED_TCB
AUTHOR_KEY_DIGEST	LAUNCH_TCB
POLICY	
REPORT_ID	
MA_REPORT_ID	

Table 4.4 – Attestation Report Content [47]

4.4.3 Measurement Calculation and Validation

In our project to verify the integrity of the VMs, we computed the hash of the initial state of the VM to compared it with the measurement provided in the attestation report generated within the SEV-SNP guest VM. To accomplish this we used a Python-based tool called `sev-snp-measure`, available on GitHub [61]. This command-line utility calculates the expected measurement of an AMD SEV-SNP guest VM. Using the options of the tools listed in Table 4.5, we specified the configuration values of our VMs and compared the generated hashes with the measurements in the attestation reports to ensure the VM's status is trusted.

Option	**Description**
`-mode`	Guest mode.
`-vcpus`	Number of guest virtual cpus.
`-ovmf`	OVMF file to calculate the hash from.

Table 4.5 – Options for Measurement Calculation

4.5 Management Server

The management server is specifically designed to streamline the management of donors within our system. It is implemented using `Express.js`, a minimalist framework for Node.js and `Axios` module in order to provide donors with API endpoints, which are used to handle the registration of donor's details, including their unique identifier and IP address. This server listen on port 3006 in our system and uses Express's JSON middleware to process incoming JSON data, which is essential for receiving and handling node information.

Donor registration is managed through a POST route that we named `register`. When a donor wants to register it send a POST request to this route with its information, including its identifier and IP address. The server checks if the node is already registered by iterating through the list of registered nodes. If node is not registered, it is added to the list, and a confirmation message is sent to acknowledge that the registration was successful otherwise a response is send to notify that the node is already registered.

Once a donor is registered, the complete list of registered donors is sent to all donors in the system. This task is accomplished using the HTTP communication module Axios to send POST requests to specific endpoint on each donor node. This facilitate seamless communication and coordination across the system.

This server acts as a central registry, maintaining an updated list of all active donors and their respective connection points. Additionally, it delivers important information regarding active servers to the load balancer, enabling it to update its server list in real time. This updating ensures that the load balancer can always effectively distribute traffic just to active server.

4.6 Application

The web server in our system hosts a PHP-based web application called HotCRP. This application interacts with the MariaDB database server for reading and saving user data. HotCRP is a conference management web application that provides users with various features and roles [27].

The application allows user to create accounts, log in, and access their personal profiles to manage their submissions, reviews, and communications with conference organizers. It enables authors to submit their work through an interface, update their submissions, and add supplementary files. Reviewers are assigned to submitted papers and can access the submissions, add comments, and provide detailed evaluations.

4.7 Reverse Proxy Server

In each VM in our system, a reverse proxy server written in Node.js with the Express framework operates. This server acts as an intermediary between the client and the final application hosted on the web server, through which all requests pass. It integrates several JavaScript modules, including the Raft module, which implements the consensus algorithm we use in our system to ensure data synchronisation between nodes. This module handles the leader election process and log replication, which are essential mechanisms for maintaining data consistency in our system.

Each proxy server in our system has an instance of the Raft module with a unique identifier. Communication between each instance is done via HTTP messages, mostly sent using the JavaScript module Axios. Our proxy servers have endpoints, notably the `request-vote` and `append-entries` endpoints, used respectively in the leader election process and log replication.

When each proxy server starts, the node contacts the management server to register, providing its identifier and IP address. in return it receives a list of other nodes already active in the system. Immediately after registration, the Raft protocol is started on the node. If, after a random time, the node does not receive any message from a leader or other nodes, it initiates the leader election process. It transitions from follower state to candidate state and then submits its candidacy to other servers by sending an HTTP request message via the Axios module to the `request-vote` endpoint of each node in its node list. In response, the candidate node receives vote responses via HTTP response messages and becomes the leader if the majority of nodes in the system respond positively to vote request. Once elected, the leader node periodically send heartbeat requests to the `append-entries` endpoint of all other nodes with follower state to let other nodes know that it is always present in the system as leader.

As soon as the proxy server receives a request, if it is a read request such as a GET request then this request is sent directly to the web server using the Http-proxy JavaScript module to be applied to the final application. However, if it is a write request, such as a POST request, which will change the state of the data in the web application, then this request is submitted to the Raft module so that it can be replicated on all nodes in the system in order to synchronise the data. Figure 4.1 illustrates this process, showing how the proxy server handles different types of requests and how Raft module replicates requests.

If the node receiving the request is not the leader, the request is redirected to the leader node using a JavaScript module Http-proxy. Once the leader has received the request passed to it by the follower, it extracts the key part of that request, such as the URL, headers and the body. The

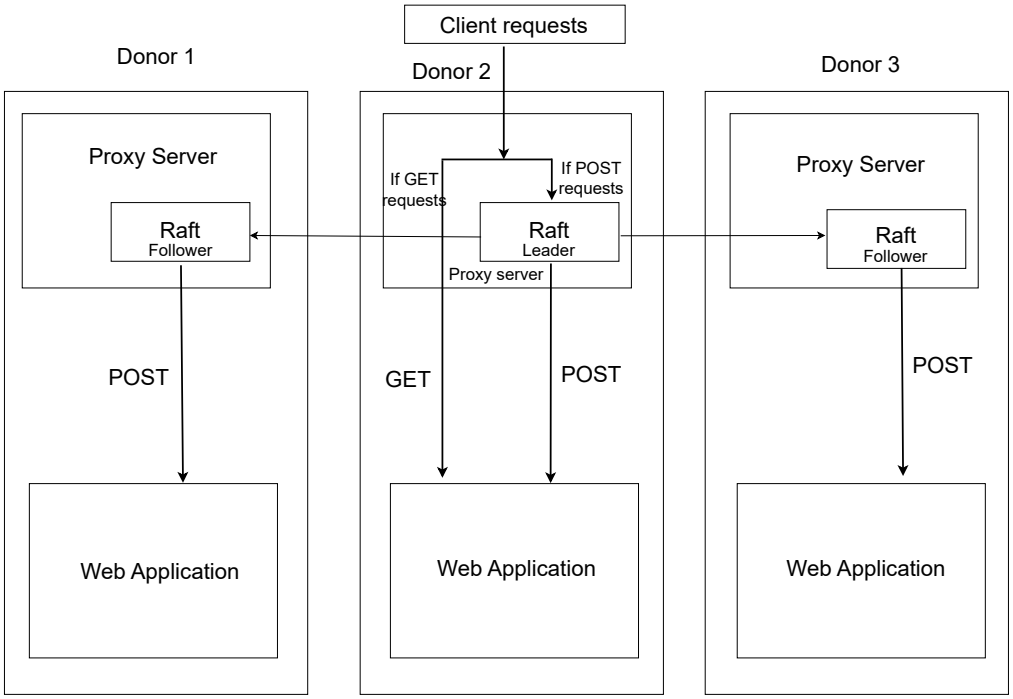


Figure 4.1 – Proxy Server Handling Requests

leader then creates a log entry for each write request to track and manage the request. This log entry includes all the extracted information. After creating the log entry, the leader replicates these entries to the follower nodes. This is done via POST requests to the `append-entries` endpoint of each follower node using the Axios module. This ensures that all follower nodes are updated with the new entries, maintaining consistency across all nodes in the system.

As soon as each follower node receives the request and adds it to its logs, it sends a positive response to the leader to confirm that it has successfully received the request it must to the final application. The leader then applies the request in its log to the hosted web application using Axios. Next, the leader send a heartbeat to other nodes to indicate that they can also apply the request to the application on their own web server, also using Axios. Once the request has been applied to the final application of each node, the server that initially received the requests send the response back to the client as show in Figure 4.2.

In our system, our implementation of Raft ensures that all nodes maintain a consistent state of data. If a follower node disconnects and misses some entries it can request the leader to resend the missing entries when it reconnects, as the follower node receives the system state from the leader who sends the index of the entries already applied. The leader maintains an index of the entries that have been applied in the web application to ensure all followers are up-to-date. When a follower reconnects and detects it is missing some entries, it will respond negatively to the heartbeat message sent by the leader. This informs the leader that the follower's entries are not up-to-date. The leader then sends the missing entries to the follower through heartbeats, using HTTP messages sent via the Axios module.

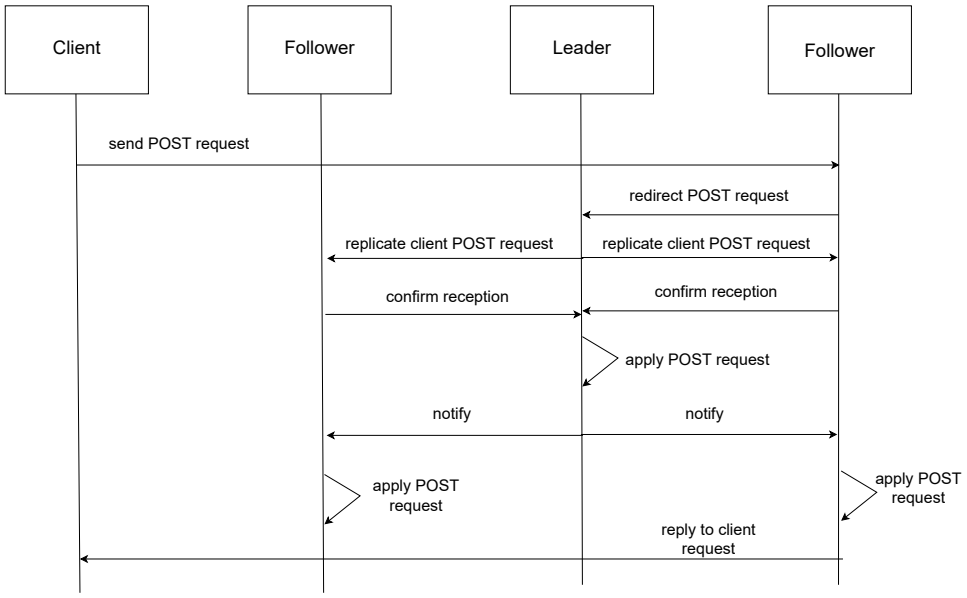


Figure 4.2 – Summary of the POST Request Replication Process

Our system is also fault-tolerant. This means it can continue to response to client request even when some components fail. If the leader node fails or is considered faulty, the system initiates a new leader election process. During this election process, the remaining nodes vote to select a new leader. This new leader then takes over the responsibilities of managing and coordinating the write requests in the system. This fault-tolerant mechanism ensures high availability and reliability of the system, allowing it to handle node failures gracefully without disrupting the overall functionality and consistency of the data.

5 EVALUATION

In this chapter, we will conduct an evaluation of `GenerousHost`. Our primary focus will be on analysing the server response times, the throughput and the processing duration for read and write request, which are critical metrics for assessing the performance of our hosting infrastructure. By scrutinizing these metrics, we aim to gain a detailed understanding of how quickly our servers handle requests and deliver content to users, as well as how many request the servers can handle per second.

5.1 Experiment Tools

In order to carry out an in-depth evaluation of our system, we have developed two separate applications. The first is a web application design to be hosted on the servers in the test environment, which enables performance to be analysed under real conditions. The second application is a load generator, designed to simulate end-user behaviour by sending requests to each server. This simulation enables us to test the performance of our infrastructure by reproducing various usage situations.

5.1.1 Test Application

The application we used to perform the evaluation is a simple web application that performs arithmetic operations and stores the results in a database. It consists of two main forms. The first form allows users to enter two number and choose either addition or multiplication. Once the form is submitted, the values are retrieved and the result of the operation is calculated using PHP: Hypertext Preprocessor (PHP). This result is then inserted into a MySQL database. If the insertion is successful, a confirmation message with the result is displayed on the page. The second form allows the user to retrieve a specific result by entering its identifier. Upon submission, an SQL query is executed to search the database for the corresponding result. If a result is found, it is displayed on the page; otherwise, a message indicates that the result has not been found. The application uses an extension for MariaDB called `mysqli` to manage connections and interactions with the database.

5.1.2 Load Generator

The second application among our experimental tools is a load generator, which is a Bash script designed to test the performance of web servers by simulating end-user behaviour. The script takes several parameters as input including the percentage of write requests, the percentage of read requests, the total number of requests to be sent, and the percentage of requests destined for each

specified server. The target servers are defined in a list of URLs and the requests are distributed according to the percentages specified.

The script initiates the process by calculating the number of requests to send to each server based on the percentages provided. For each server,a set of results files is generated, comprising one for write requests and one for read requests.

The script defines a function which send requests to the servers. For each server, it calculates the number of write and read requests to send, and sends these requests using `curl` tool. Write requests send random data for addition, while read requests retrieve results by identifier. The response time and status code are recorded in the corresponding files. As soon as the client sends a request and receives a response, it immediately sends the next request without rate limiting. Once all the requests have been sent, the script generates files summarising the test results for each server. The files are formatted into tables displaying the information collected, including the type of request, the response time, the status code, the throughput and the duration of the operation according to the request type.

5.2 Test Environment

The evaluation of our proposed solution was conducted on a Linux machine configured with the specifications presented in Table 5.1. It is power by an AMD processor, providing sufficient computational power for handling multiple virtualized environments simultaneously.

Component	**Specification**
Operating System	Ubuntu 22.04.4 LTS
Architecture	x86-64
CPU(s)	32
CPU Model	AMD EPYC 9124 16-core
Memory	124GB

Table 5.1 – Resource Specification of the Test Infrastructure

Our test environment comprises of several elements as illustrated in Figure 5.1, all interacting within the same network. These include a management service, a load generator, and four VMs with software stacks for hosting the test application and incorporating the reverse proxy needed to perform data synchronisation between the servers. The VMs representing the donors in our system are configured identically, with their allocated resources detailed in Table 5.2. The management service allows the various VMs to register within the system and obtain information about other VMs. The load generator emulates end user behaviour by issuing requests to the servers.

Component	**Specification**
Operating System	Ubuntu 22.04.4 LTS
Architecture	x86-64
CPU Model	AMD EPYC-v4 Processor
Number of vCPU	16
Memory	7.2GB

Table 5.2 – Specifications of VMs for the Test Environment.

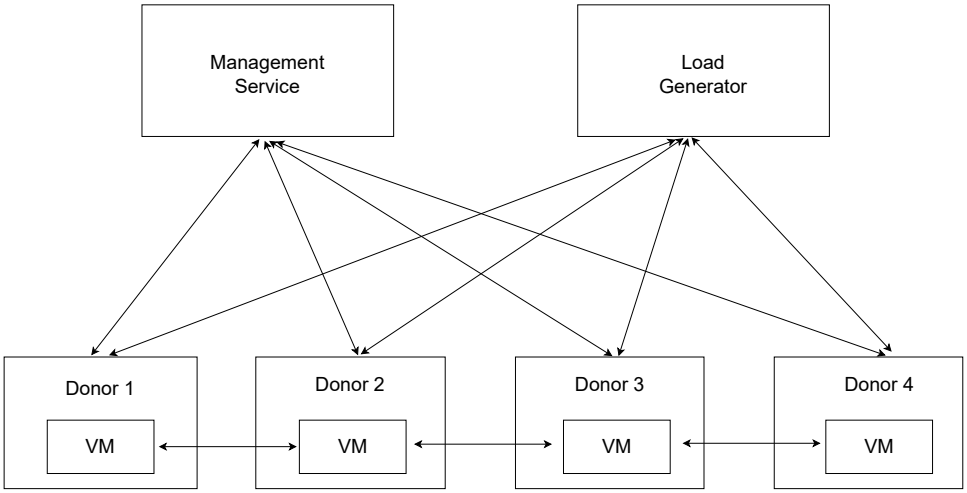


Figure 5.1 – Test Environment

5.3 Evaluation Questions

In evaluating our collective hosting system, we aim to answer a number of questions that will enable us to analyse the performance and reliability of our hosting infrastructure. By identifying strengths and weaknesses, we can optimize the system. The questions we aimed to answer included:

1. What are the response time for read and write requests?
2. Is there a significant difference between the response times of different servers?
3. How do response times vary according to the number of requests sent to each server?
4. How many requests the servers processes per second?
5. How long does it take each server to process each type of request?

The responses to these questions permit the assessment of the overall performance of read and write operations, such as GET and POST requests. They also enable the identification of the worst cases of performance and examine the possible causes. Furthermore, they also facilitate the determination of whether certain servers are performing less well than others, to assess the system's ability to handle high loads, and to identify system reliability problems. In addition, they enable us to determine the proportion of requests that respond quickly and to set performance targets. Specifically, the throughput measures the system's capability to handle a given workload. High

throughput indicates that the system can process many requests quickly, which is essential for a good web hosting service. Additionally, knowing the time required to process each type of request for each server helps identify performance differences between request types. This can reveal inefficiencies specific to read or write requests, thereby aiding in optimization. Consequently, this information is essential for improving the overall efficiency and reliability of the system.

5.4 Test Approach

After setting up our test environment, we planned the test approach to ensure a comprehensive evaluation of our system's performance. This approach includes the execution of specific evaluation cases, the measurement of performance metrics, and the analysis of collected data to derive meaningful insights.

5.4.1 Evaluation Cases

We have identified two primary evaluation cases. These cases are outlined as follows:

1. Case utilizing one VM without our implemented solution: This case served as a baseline to compare the system's performance prior to the application of our proposed solution. It facilitates the understanding of the initial performance metrics and underscores the improvements introduced by `GenerousHost`. In this case, the entire request load is handled by a single hosting server.
2. Case utilizing four VMs with our implementation: This case evaluated the system following the deployment of our proposed solution utilizing four VMs. This configuration enables the assessment of the performance improvements provided by `GenerousHost`. In this case, the total load is distributed equally across the four hosting servers within the system.

In each case, we utilized a specific quantity of requests classified as low, medium and high as presented in Table 5.3. The total quantity of requests is divided equally into 50% read and 50% write requests. This classification facilitates an understanding of the system's performance, provides insights into uptime and overall reliability, and enables the determination of whether the system behaves consistently across different scenarios.

Load Level	Number of Requests	Read/Write Split
Low	1000	500 Read / 500 Write
Medium	4000	2000 Read / 2000 Write
High	8000	4000 Read / 4000 Write

Table 5.3 – Classification of Load Levels

5.4.2 Metrics for Measurement

During the evaluation we measured the following metrics:

- **Response Time**: measures the time taken to respond to read and write requests. It provides an indication of the system's performance under various load conditions. Lower response times indicate better performance and a more efficient system.

- **Throughput**: indicates the number of requests processed per second. It reflects the system's capacity to handle a given workload. Higher throughput values suggest that the system can efficiently manage a larger volume of requests.

- **Processing Time**: measures the time taken by each server to process specific types of requests, such as read or write operations. It helps identify potential inefficiencies and performance differences between request types, aiding in the optimization of server performance.

5.5 Performance Analysis

The data collected during the evaluation of each case will be analyzed in this section based on the quantity of requests. This analysis aims to elucidate the system's performance under varying request numbers, which are categorized as low, medium, and high request volumes. Additionally, we aim to understand how different types of requests (read and write) are managed by the servers in each evaluation case.

5.5.1 Evaluation with Low Request Volume

During our evaluation, we gathered data in the two evaluation cases with a load of 1000 requests, divided into 500 read requests and 500 write requests. We then measured the variation in the response time for each type of request, as well as the variation in response time as a function of the number of request, the throughput of each type of request and the processing time for each type of request.

In the first case, concerning the response time for the read request, the box plot in Figure 5.2 provides a visual representation of the response time distribution for read requests handled by the single hosting server in the VM. The minimum response time, represented by the lower whisker is approximately 0.8 millisecond (ms). This value indicates the fastest response recorded during the evaluation. The maximum response time, as indicated by the upper whisker, is around 1.2 ms, suggesting that some requests experienced higher than expected response times. The interquartile range shows that the middle of the data lies between 0.9 ms and 1 ms, with a median response time of about 0.95 ms. This analysis highlights that while the majority of read requests were processed efficiently, there are instances where response times spiked.

With regard to the write requests, as illustrated in Figure 5.3, the minimum response time, is approximately 1 ms, indicating the fastest response time, while the maximum response time is around 2 ms. Nevertheless, several outliers extend up to approximately 8 ms, indicating that some write requests experienced significantly higher response times. The interquartile range, which represents the middle of the data, spans from about 1.1 ms to about 1.13 ms, with a median response time of approximately 1.2 ms.

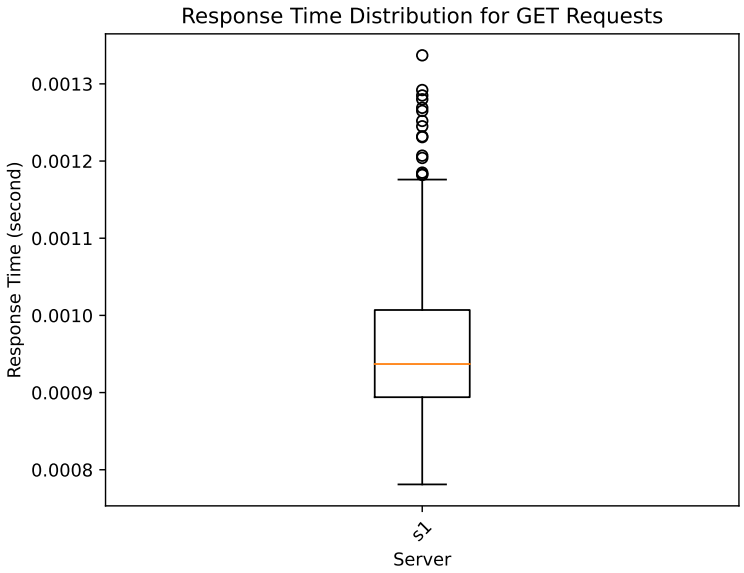


Figure 5.2 – Response Time Distribution for Read Requests on Single Server (Low Volume Request)

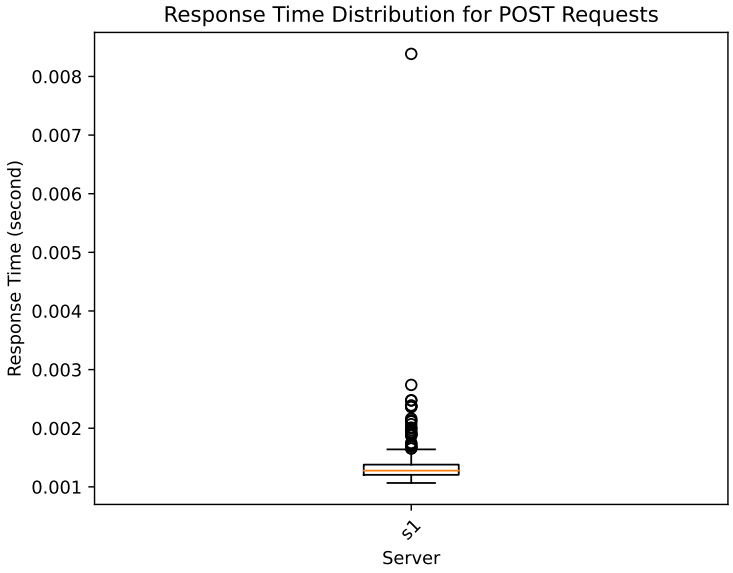


Figure 5.3 – Response Time Distribution for Write Requests on Single Server (Low Volume Request)

By observing the response time variation for the read requests by the request number in Figure 5.4, we can notice that the response times fluctuate significantly between approximately 0.8 ms and 1.3 ms throughout the 500 requests. Initially, the response times are relatively stable around 1

ms, with occasional peaks and troughs. Several spikes occur around request numbers 200, and 400, where the response times briefly exceed 1.29 ms.

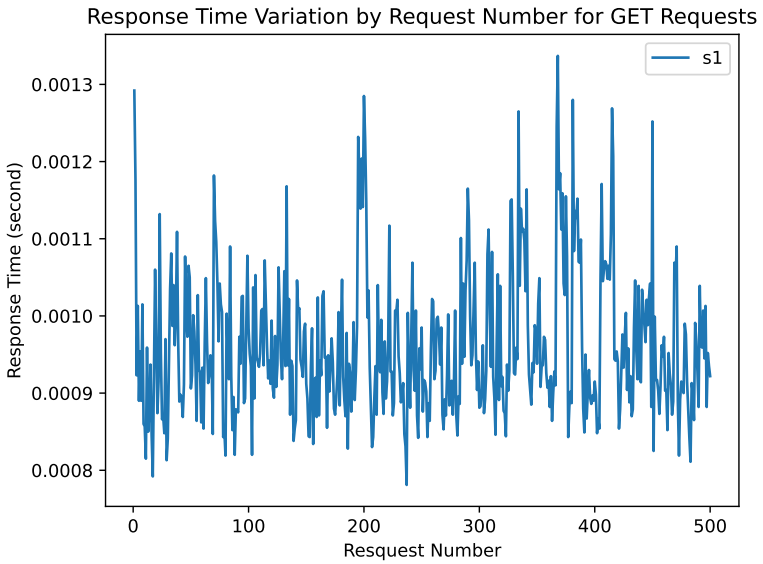


Figure 5.4 – Response Time Variation by Read Request Number (Low Volume Request)

The analysis of the write request response time graph in Figure 5.5 shows an initial spike to approximately 8 ms. This may be attributed to several factors. Initially, the system may experience an increase in overhead due to the presence of cold caches, where data is not yet cached. This can result in longer access times. Furthermore, this initial delay may also result from the establishment of new connections.

Following this initial phase, the response time stabilizes at approximately 1 ms, indicating that the caches have been warmed up, connections have been established, and the system has reached a steady state. The minor fluctuations, which do not exceed 3 ms could be attributed to routine variations in network latency, transient processing loads, or occasional resource contention within the server.

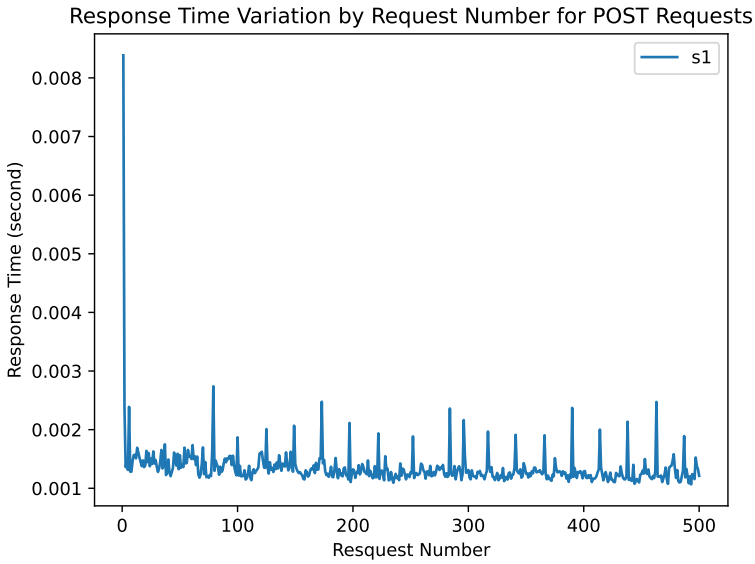


Figure 5.5 – Response Time Variation by Write Request Number(Low Volume Request)

Regarding the data gather about the processing time and the throughput of each request type on the hosting server, as Figure 5.6 illustrates for both read and write requests. The blue bar indicates the write duration for 500 requests at around 4 seconds (s), whereas the cyan bar shows the read duration at approximately 5 s. The red dot represents the write throughput, which is 125 requests/s, while the orange dot indicates the read throughput at 100 requests/s. These observations indicate that the duration of write requests is slightly shorter than that of read requests for the same number of requests.

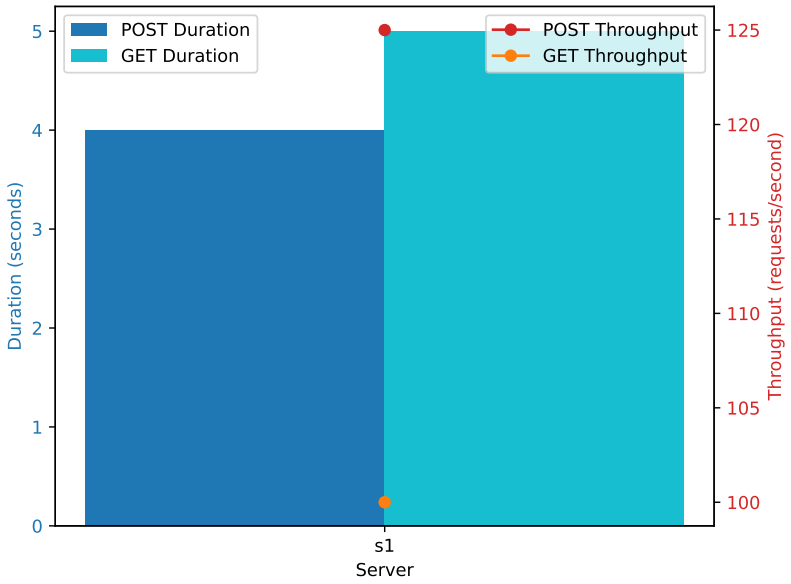


Figure 5.6 – Servers Throughput and Processing Time (Low Volume Request)

Focusing on the second evaluation case, where we deployed our implementation on four VMs. In this scenario, we distributed 1000 requests across the VMs, with each VM handling 25% of the total requests. Specifically, each VM processed 125 write requests and 125 read requests. Regarding the response time for read requests, Figure 5.7 indicates several insights. The median response time for the majority server is approximately 2 ms, while the minimum response times are almost similar, just above 1 ms. However, there are notable differences in the variability of response times. The maximum response times show more significant discrepancies, with server S2 having the highest maximum response time approximately 4 ms.

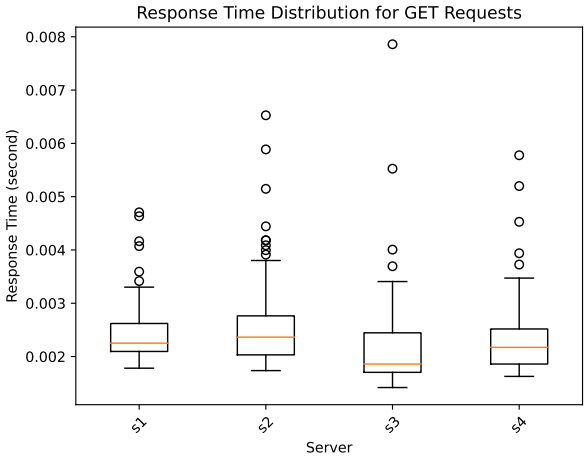


Figure 5.7 – Response Time Distribution for Read Requests on four Servers (Low Volume Request)

Observing the Figure 5.8 about the response time distribution for write requests across the four servers, the median response times are higher compared to read requests, around 15-25 ms, with the minimum response time slightly above 15 ms for server S2 and the maximum response time between the four servers approximately 32 ms for server S4. These results indicate that server S2 is faster than other servers, while server s4 is slowest in processing write requests.

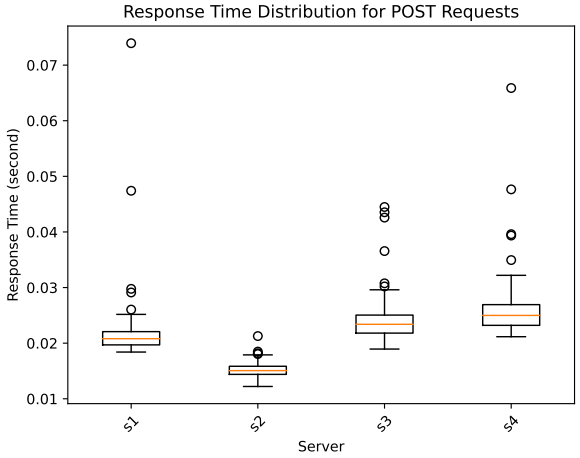


Figure 5.8 – Response Time Distribution for Write Requests on four Servers (Low Volume Request)

Concerning the response time variation on each server for read request by request number, Figure 5.9 shows that all servers experience some variability in response time, with occasional spikes. Server S2 and S3 show higher spikes compared to S1 and S4, indicating periods of increased response times. Overall, the majority of response times for all servers remain below 3 ms, with most fluctuations occurring around 2 ms to 3 ms.

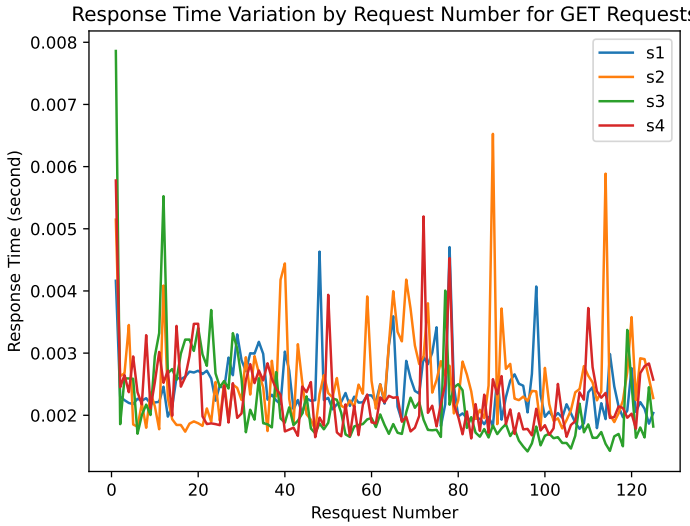


Figure 5.9 – Response Time Variation by Read Request Number on Four Server (Low Volume Request)

For write requests Figure 5.10 reveal that the server S2 demonstrates the most consistent performance with lowest response time, averaging around 15 ms to around 25 ms. In contrast other servers show greater variability in response times. This consistent and rapid response time can be justified by the fact that we are using the Raft consensus algorithm. Given this, server S2 might be the current leader, which explains its superior performance. As leader, server S2 is always the first to commit a write request. This ensures that it does not have to wait acknowledgements from other servers before responding, resulting in a quicker response time. In contrast, the other servers, which are followers, must wait for the leader's confirmation before they can respond, leading to higher variability and generally slow response times.

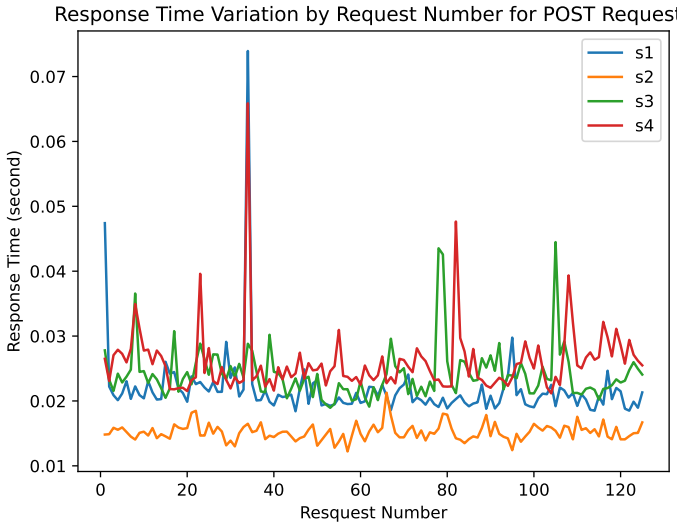


Figure 5.10 – Response Time Variation by Write Request Number on Four Server (Low Volume Request)

With regard to the throughput and the processing time for both read and write requests, Figure 5.11 indicates that server S4 has the longest processing times for write requests at 5 s. In contrast, the server S1 has the longest processing time for read requests at 2 s, while other servers have a better processing time for read requests at 1 s. On the throughput side, the line chart shows that server S2, S3, and S4 achieve the highest read throughput at 120 requests per second, whereas the throughput for write requests does not exceed 40 request per second.

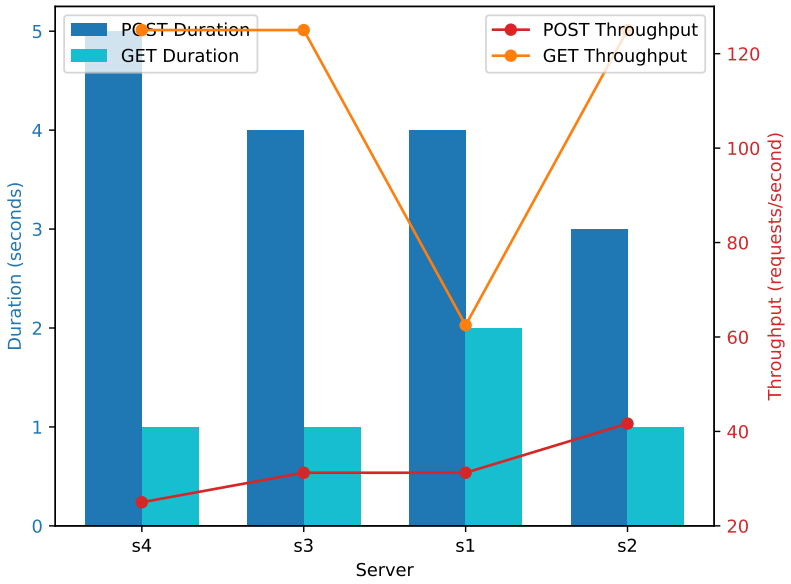


Figure 5.11 – Servers Throughput and Processing Time for 1000 Requests by Four VMs (Low Volume Request)

Comparing the two evaluation cases, we observed distinct differences in performance metrics. In the single server case, the median response time for read requests was lower at 0.95 ms, and write requests showed a median of 1.2 ms. Processing times for read and write requests were 5 s and 4 s, respectively, with throughput of 100 requests per second for reads and 125 requests per second for writes. In contrast, the case with four VMs had higher median response times for read requests at 2 ms and for write requests at 15-25 ms. The case with four VM achieved higher read throughput at 120 requests per second but showed greater variability in response times (between approximately 1-4 ms for read requests and approximately 10-30 ms for write requests) and lower throughput for write requests (between almost 25-40 request per second) compared to the evaluation with a single hosting server.

5.5.2 Evaluation with Medium Request Volume

To further evaluate the performance of our distributed web hosting system, we extended our analysis to medium request volume. We processed a total of 4000 requests, divided into 2000 read requests and 2000 write requests, in both the case with a single VM without our hosting approach and the case with four VM with our solution.

In both cases, there was not too much variation in the maximum and minimum values for read and write requests in comparison to the evaluation with a low volume of requests. In the single server case, the maximum response time for read requests remained around 1.2 ms and the minimum response time around 0.8 ms. For write requests, the maximum and minimum response time were approximately 2 ms and 1 ms, respectively. The same observation was also made for the variation of the response time variation by request number for read and write requests. This can be observed in the Figures 5.12, 5.13, 5.14, 5.15, for the first case of evaluation with a single server and in Figures 5.16, 5.17, 5.18, 5.19 for the second case with four servers.

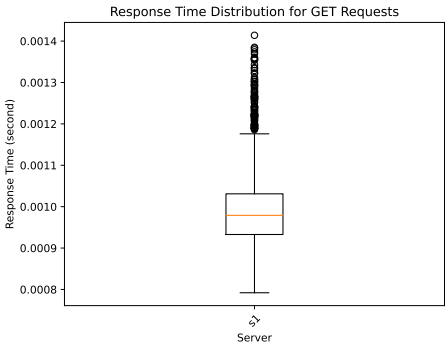


Figure 5.12 – Response Time Distribution for Read Requests on Single Server (Medium Volume Request)

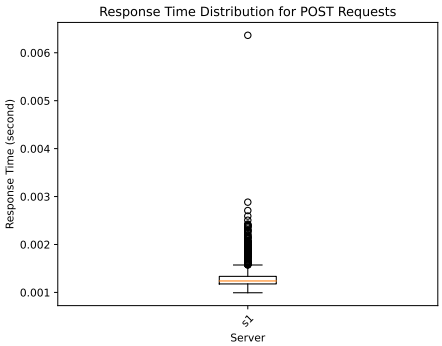


Figure 5.13 – Response Time Distribution for Write Requests on Single Server (Medium Volume Request)

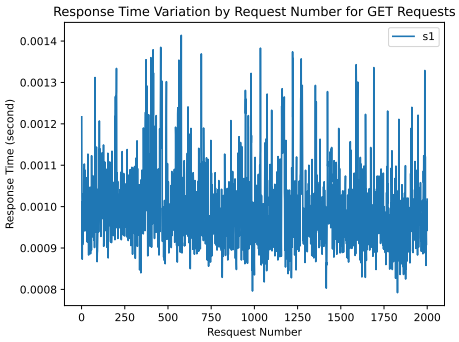


Figure 5.14 – Response Time Variation by Read Request Number (Medium Volume Request)

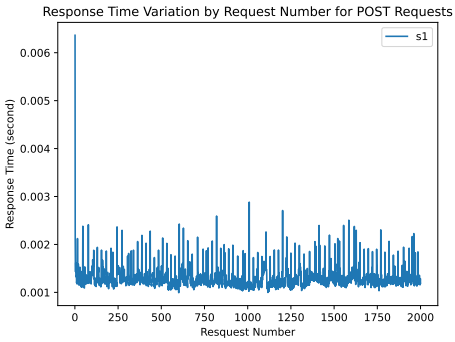


Figure 5.15 – Response Time Variation by Write Request Number (Medium Volume Request)

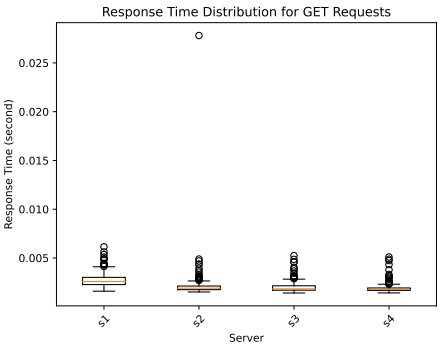


Figure 5.16 – Response Time Distribution for Read Requests on Four VMs (Medium Volume Request)

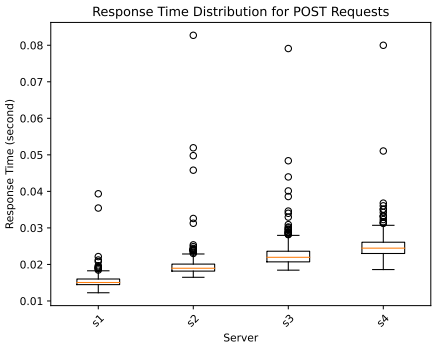


Figure 5.17 – Response Time Distribution for Write Requests on Four VMs (Medium Volume Request)

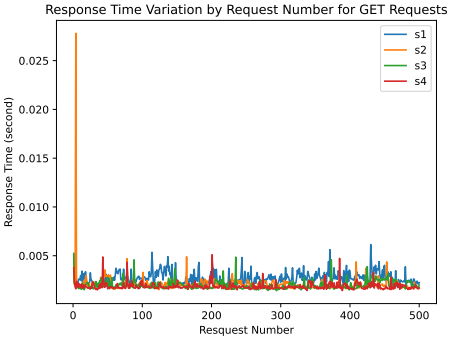


Figure 5.18 – Response Time Variation by Read Request Number (Medium Volume Request)

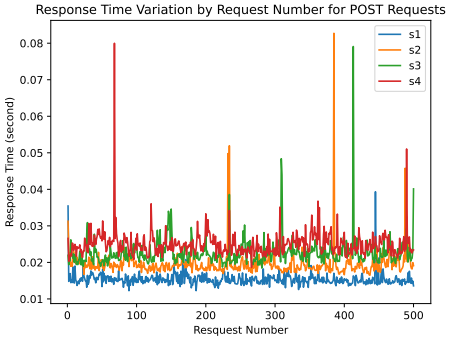


Figure 5.19 – Response Time Variation by Write Request Number (Medium Volume Request)

Nevertheless, the processing time and throughput values exhibited discrepancies in this instance, where the request load was of a medium volume. On the single hosting server, as Figure 5.20 presents, both read and write requests have the same processing time of approximately 18 s. Additionally, the throughput is identical for both read and write requests around 111 requests per second. This indicates that the single hosting server demonstrates balanced performance for both operations, handling them with equal efficiency in terms of processing time and throughput. One possible reason for the identical processing times and throughputs could be that the server did not experience any significant interruptions or context switching during the request handling. This uninterrupted processing ensures that both read and write requests are managed without delayed caused by other system activities.

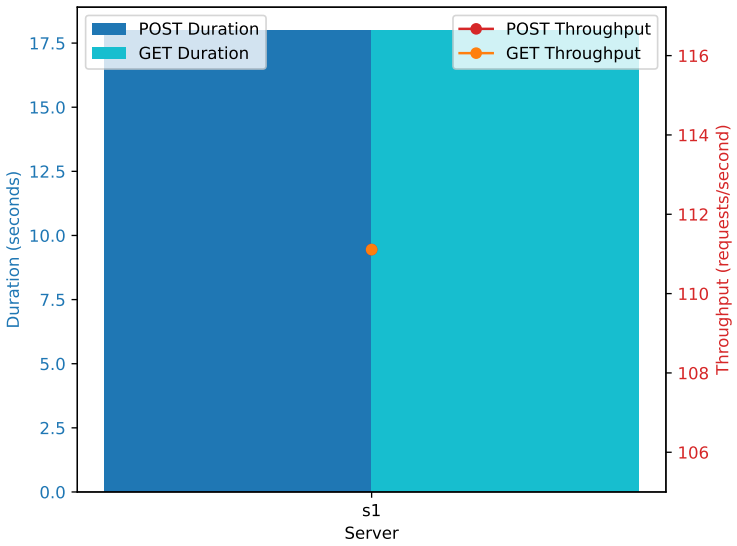


Figure 5.20 – Servers Throughput and Processing Time for 4000 Requests by a single VM

On the case with four VMs, Figure 5.21 indicates the processing duration and the throughput for read and write requests across the four servers. For write requests, the processing duration is significantly higher across all servers, with S4 having the longest duration at 17 s and S1 the shortest at 12 s. In contrast, read requests have a much shorter processing duration at 5 s for all servers. Throughput for read requests remains consistent at 100 requests per second for all servers, while write requests throughput is almost the same as in the situation with lower load volume, it remains between 25-40 requests per second.

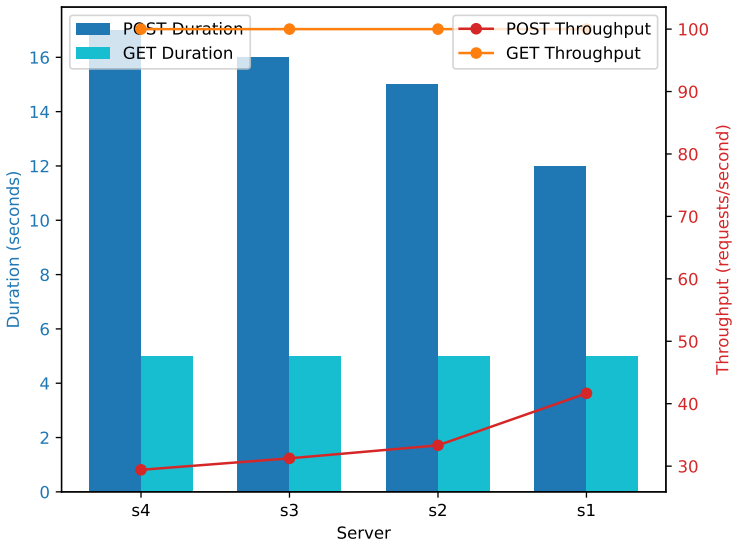


Figure 5.21 – Servers Throughput and Processing Time for 4000 Requests by four VMs

5.5.3 Evaluation with High Request Volume

In order to gain further insight, an additional evaluation was carried out under high request volume conditions. In this scenario, a total of 8000 requests were processed, maintaining the previous split of 4000 read requests and 4000 write requests. In term of response times, the data collected in the case of a single server and in the case of four servers did not show much difference from the data already presented in the previous scenarios as the following Figures 5.22, 5.27, 5.24, 5.25, 5.26, 5.27, 5.28, 5.29 are presented for both cases.

On the other hand, there was a slight difference in the time taken to process read and write requests in the case of a single server. As Figure 5.30 presents, it took approximately one second longer for the single server to process all write requests than read requests, and the throughput of read requests in this scenario was higher than that of write requests. In contrast, the throughput remained unchanged in the case of four servers, as observed in the scenario with medium loads of requests (100 request per second for read requests and between 25-40 requests per second for write requests) as showed in Figure 5.31.

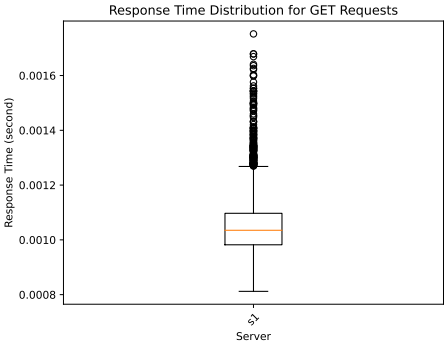


Figure 5.22 – Response Time Distribution for Read Requests on Single Server (High Volume Request)

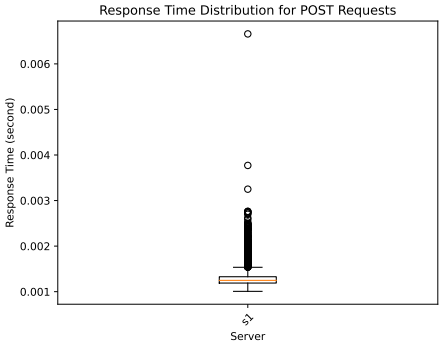


Figure 5.23 – Response Time Distribution for Write Requests on Single Server (High Volume Request)

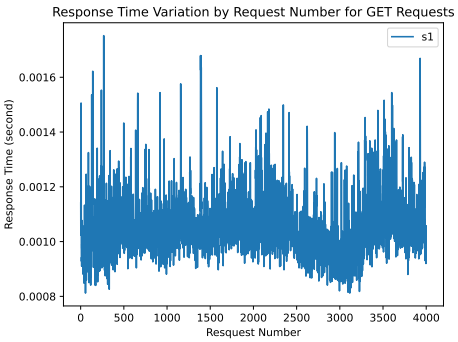


Figure 5.24 – Response Time Variation by Read Request Number (High Volume Request)

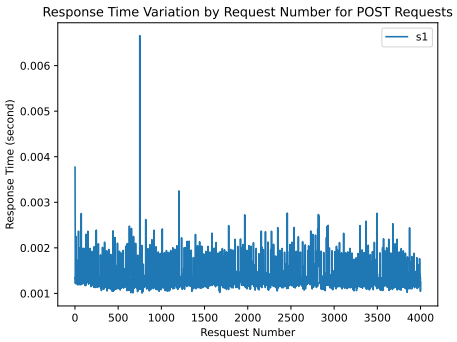


Figure 5.25 – Response Time Variation by Write Request Number (Medium Volume Request)

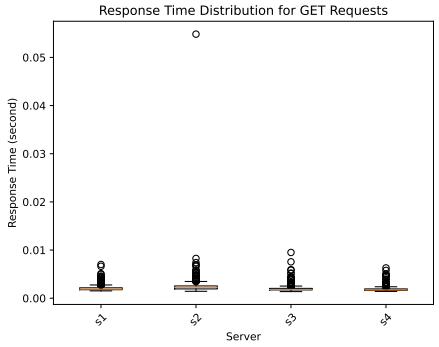


Figure 5.26 – Response Time Distribution for Read Requests on Four VMs (High Volume Request)

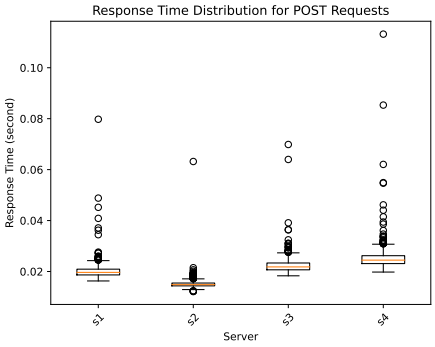


Figure 5.27 – Response Time Distribution for Write Requests on Four VMs (High Volume Request)

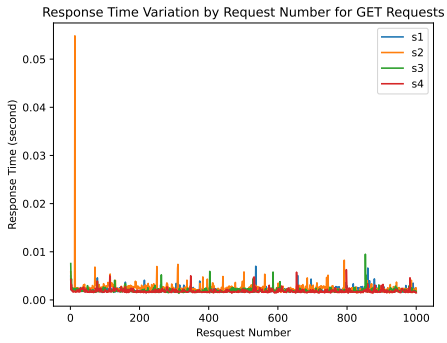


Figure 5.28 – Response Time Variation by Read Request Number (High Volume Request)

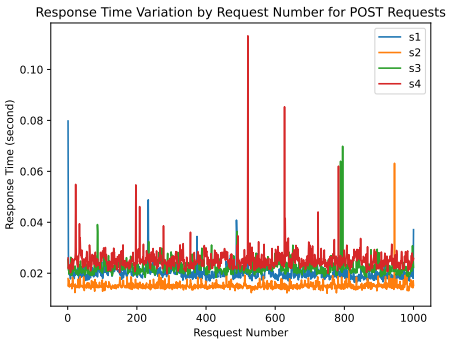


Figure 5.29 – Response Time Variation by Write Request Number (High Volume Request)

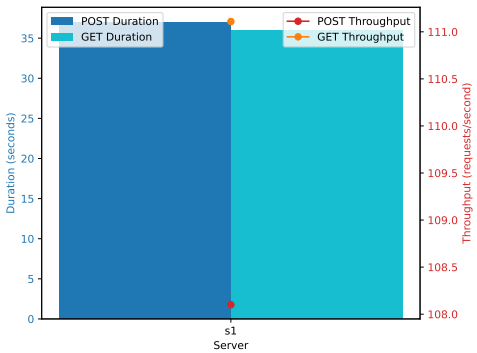


Figure 5.30 – Servers Throughput and Processing Time for 8000 Requests by a single VM

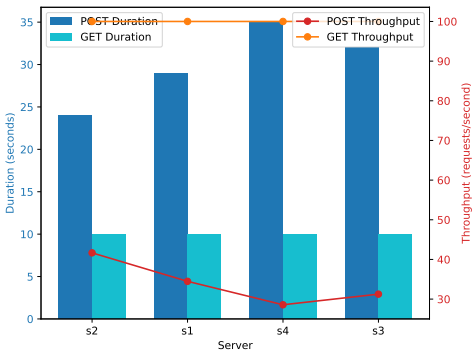


Figure 5.31 – Servers Throughput and Processing Time for 8000 Requests by four VMs

5.6 Discussion

The primary objective of our evaluation was to compare how read and write requests are handled in two different system configurations: our proposed distributed web hosting solution, which leverages the unused resources of volunteers, and a configuration with a single server handling all incoming requests. The objective was to identify the enhancements offered by `GenerousHost` and to determine potential areas for optimization within this approach.

The results of this evaluation demonstrate that, under varying quantities of requests, the single server configuration processes requests with a higher throughput than the one in the case of our deployment with four VMs and also observes better response times in request processing. In fact, for 4000 write requests and 4000 read requests, the system with a single server achieves an average response time of approximately 1.1 ms for read requests and about 1.5 ms for write requests, with respective throughputs of 111 requests per second and slightly above 108 requests per second. However, the configuration with four VMs, for the same quantity of requests, shows a response time for read requests slightly above the single server configuration (around 2 ms) with a constant throughput of 100 requests per second, and a response time for write requests much higher than the single server configuration (on average approximately 23 ms) with a throughput not exceeding 40 requests per second. Nevertheless, the single server takes a longer time to complete all requests in comparison to `GenerousHost` with four VMs, which distributes incoming requests across multiple servers, enabling faster processing of a larger number of requests. For the same number of requests as before, the single server takes a total of almost 73 seconds to process 8000 requests. In contrast, `GenerousHost`, by distributing these requests, completes processing in approximately 45 seconds when the requests are executed simultaneously.

In our deployment with four server, analyses indicates that servers are more efficient at handling read requests than write requests. Read request processing maintains a constant throughput of 100 requests per second across all servers, even as the number of requests increases, as observed in our evaluation with 4000 and 8000 requests. In contrast, the throughput for processing write requests is lower on all servers, varying between 25 and 40 requests per second. This performance difference can be attributed to the use of a data synchronisation algorithm between servers (Raft). Although Raft introduces some latency in processing write requests, it offers significant benefits in terms of security and fault tolerance. The Raft algorithm ensures the consistent replication of data across multiple servers by synchronising data across all servers. This process protects against data loss in the event of a server failure. Furthermore, by ensuring that decisions are made through consensus among servers, `GenerousHost` becomes more resilient and capable of continuing to function correctly even in the presence of faults, thus enhancing the overall reliability of the system. Consequently, although the processing of write requests is slower, this trade-off is justified by the improved data security and system resilience to faults.

6 RELATED WORK

GenerousHost, as well as many other projects, is based on leveraging unused resources from volunteers or willing organizations worldwide via the internet to support various projects. Although GenerousHost and other projects differ in their application domains, they share several common similarity. Among these projects is Berkleley Open Infrastructure for Network Computing (BOINC) [3], a middleware that allows volunteers to contribute computing resources to scientific projects, often to increase the computational power of these projects. Several initiatives benefit from this platform, including Albert@Home [1], GPUGRID [18], and many others [35].

Like GenerousHost, BOINC uses virtual machines to allocates space on volunteers' computers, ensuring that the code executed does not harm or interfere with the host machine's code. However, BOINC does not permit the execution of sensitive data such as passwords or personal information and does not offer a mechanism for secure computing. In contrast, GenerousHost, based on AMD's SEV-SNP technology that provides the necessary security to ensure the handling of sensitive data.

BOINC features an architecture where each VM is autonomous and independent, limiting the platform's use to specific, independent computational tasks. Once the results are obtained, they are returned to the server, and all data related to the task is deleted. Consequently, this type of platform cannot be used for web hosting due to data consistency issues and the lack of synchronization between the nodes of the system, unlike GenerousHost.

Another project similar to GenerousHost is XtremWeb [13], which differs from BOINC in that volunteer machines can communicate with others. However, this communication is not secure despite the use of secure execution relies on sandboxing, meaning an attacker could intercept messages transiting between two machines (workers). Conversely, in our system, nodes can securely exchange information. XtremWeb relies on the assumption that the code executed on volunteer machines is trustworthy. In contrast, GenerousHost allows of unstrusted code without affecting the host machine due to the isolation of VMs and the host machine's data. Another disavantage of XtremWeb is that resources can only be used when the volunteer is not using their computer, conditioning the usage of resources. However, in GenerousHost, VMs can run alongside the host's applications, even when the host is using their computer.

Other works exploiting resources have tried to differentiate themselves by not using VMs but by leveraging resources through web browsers, as seen in projects like Javelin [6], Alchemi [36], and POPCORN [42]. These projects aim to reduce the installation effort of necessary tools to participate in the resource contribution network by using web browser. However, as Anderson explained [3], they are rarely used because clients may close the browser, and scientific applications written in C and Fortran do not supports this. In contrast, our system not only allows the exploitation of diverse resources but also supports various applications beyond web hosting, such as distributed computing. Moreoever, there are also web-based projects that do not offered trusted computation

at the participant side [16], including Alkamai NetSession [67], Pando [31], and Legion [34]. In contrast, other projects like EDGAR can support trusted computation while considering untrusted participants. However, EDGAR provides its service with advertisements, which can be removed if the end-user donates their resources [16]. This may dissuade end-users who do not wish to contribute and do not want advertisements from utilising the service. In `GenerousHost`, end-users utilize the service without constraints, but are encourages to donate their resources by the ability to decide the location of the service and enjoy better quality of service, even if the central service experiences downtime.

Other systems like BitTorrent [65] and Gnutella [51], primarily focus on file sharing and rely on volunteers who provide their resources to support this file transfer. While these systems share a similarity with our system in terms of utilizing volunteer resources for distributing content, they do not use virtual machines to isolate the file of the host machine to the file to be transfer and they cannot handle sensitive data securely. Their singular objective is to facilitate the distribution of files among users. However, this objective does not allow for the optimal exploitation of the overall resources, as the resources remain unused when there is no file transfer. In contrast, `GenerousHost`, in addition to distributing content, can support many other services.

7 CONCLUSION

In our work, we presented GenerousHost an approach, which leverages unused computing resources from volunteers' computers to support the hosting of web services. This approach confers benefits not only on service providers and donors but also on end-users, who enjoy better service quality and reduced response time due to the proximity of donors. We presented how GenerousHost takes advantages of trusted execution technology to ensure secure computing, enabling the handling of sensitive data in cloud environments. Additionally, we highlighted the importance of data synchronization using the Raft protocol to achieve data consistency and maintain a fault-tolerant system. In our evaluation we showed that GenerousHost can handle a significantly larger number of simultaneous requests compared to a system with a single server. In this approach, read requests are handled more efficiently by the system's servers compared to write requests, due to the data synchronisation required during write operations. However, it is crucial to ensure data consistency within the system and the service availability.

7.1 Future Work

In term of future work, there are several areas for improvement to explore, notably the implementation of a service update mechanism. Developing efficient procedures for deploying updates without service interruptions and minimizing downtime will be crucial. Additionally, implementing a load balancing mechanism will be essential to better distribute requests among servers to enhance overall system performance. Moreover, further evaluation of GenerousHost, especially when donors are not on the same network or on the same host, should be conducted to better understand how the system behaves under these conditions.

7.1.1 Service Update

In order to implement updates to the hosted services within our system, we could adopt a phased update strategy. This method entails selecting a group of servers for updates while the remaining servers continue to operate normally, thus ensuring uninterrupted service delivery to clients, as illustrated in Figure 7.1. Once the updates for the selected group have been completed and the data migration has been successfully executed on the updated server, the process will be repeated with the next group until all servers have been updated. In the event that the update has failed, the process must be interrupted so that the error can be rectified before the updates are restarted. To orchestrate this process, a software tool for configuration management, application deployment and IT orchestration could be used. The tool should allow secure connection to the servers to minimise attack surges and simplify security management.

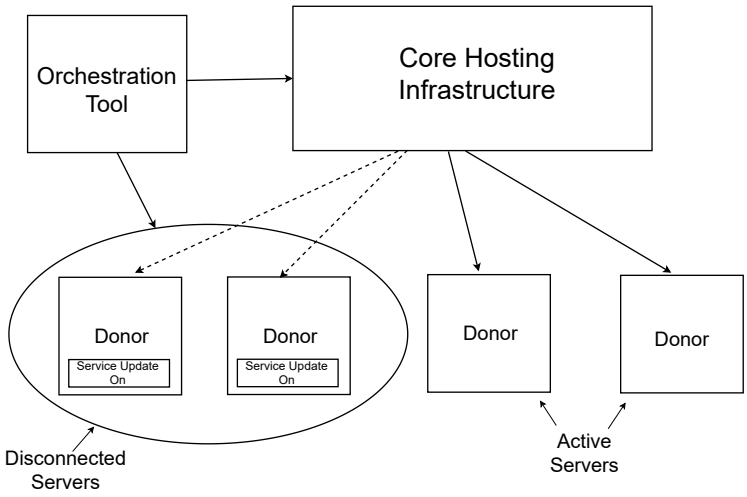


Figure 7.1 – Service Update

7.1.2 Load Balancing

The load balancer in our system plays an important role. Upon receiving the load balancer a client's request, it must evaluate the current state of the network to route the request efficiently. A remote process on the web server could assist the load balancer in gathering information, including the average load, CPU and memory usage, the number of active connections and more over [54]. Additionally, The load balancer must also have information about the client like the round-trip time to the servers. This information is important for making informed decisions about directing the client's request to the server that is best positioned to handle it efficiently and promptly.

7.1.3 Further Evaluation

In the course of our work, we evaluated `GenerousHost` on a single host machine with several guest virtual machines residing on the host. To gain further insight, it would also be beneficial to conduct further evaluation and observe the system's behaviour in different networks and with guest machines residing on different hosts. Conducting evaluations in different network environments, we can assess the system's reliability and performance. This could assist in the identification of potential improvements or issues that may not be apparent in a single host setup, which has been used in our work. Moreover, testing with guest virtual machines on different hosts will allow to evaluate the efficiency of communication and data transfer between hosts in the distributed system.

LIST OF ACRONYMS

TEE	Trusted Execution Environment
HTTP	Hypertext Transfer Protocol
CMS	Content Management System
BFT	Byzantine Fault Tolerance
CFT	Crash Fault Tolerance
VMs	virtual machines
QEMU	Quick EMUlator
IT	information technology
IaaS	Infrastructure as a Service
PaaS	Platform as a Service
SaaS	Software as a Service
SEV	Secure Encrypted Virtualization
SEV-ES	SEV-Encrypted State
SEV-SNP	SEV-Secure Nested Paging
RMP	reverse mapping table
VMPL	Virtual Machine Privilege Levels
ASP	AMD Secure Processor
KDS	Key Distribution Service
CPU	Central Processing Unit
TCP	Transmission Control Protocol
JSON	JavaScript Object Notation
PC	Personal Computer
QCOW2	QEMU Copy On Write version 2
OVMF	open virtual machine firmware
BIOS	Basic Input Output System
IP	Internet Protocol
TCB	trusted computing base
TSME	transparent secure memory encryption

SMT	simultaneous multithreading
PHP	PHP: Hypertext Preprocessor
ms	millisecond
BOINC	Berkleley Open Infrastructure for Network Computing
DoS	denial-of-service

LIST OF FIGURES

LIST OF TABLES

REFERENCES

[1] *Albert@Home*. Accessed on: 24-05-2024. URL: https://albertathome.org/fr/home.

[2] AMDESE. *AMDSEV*. Accessed on: 19-05-2024. URL: https://github.com/AMDESE/AMDSEV/tree/snp-latest.

[3] David P Anderson. "BOINC: a platform for volunteer computing." In: *Journal of Grid Computing* 18.1 (2020), pp. 99–122.

[4] Desire Athow. *Shared hosting vs dedicated hosting: Which plan should I choose?* Accessed on: 08-05-2024. 2020. URL: https://www.techradar.com/news/shared-hosting-vs-dedicated-hosting-which-plan-should-i-choose.

[5] Manuel Barbosa et al. "Foundations of hardware-based attested computation and application to SGX." In: *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE. 2016, pp. 245–260.

[6] Roger M Bartlett and Russell J Best. "The biomechanics of javelin throwing: a review." In: *Journal of sports sciences* 6.1 (1988), pp. 1–38.

[7] Fabrice Bellard. "QEMU, a fast and portable dynamic translator." In: *USENIX annual technical conference, FREENIX Track*. Vol. 41. 46. California, USA. 2005, pp. 10–5555.

[8] Harvey Brennan. *Overcoming Common Challenges in Server Hosting and Cloud Solutions for Digital Agencies*. Accessed on: 08-05-2024. 2023. URL: https://www.datacentreplus.co.uk/overcoming-common-challenges-in-server-hosting-and-cloud-solutions-for-digital-agencies/.

[9] Hua Chai and Wenbing Zhao. "Byzantine fault tolerance for services with commutative operations." In: *2014 IEEE International Conference on Services Computing*. IEEE. 2014, pp. 219–226.

[10] Rachid CHAMI. *Your Journey To Consensus (Part 1) — Crash Fault Tolerance and Paxos*. Accessed on: 28-04-2024. 2019. URL: https://medium.com/@chamirachid/your-journey-to-consensus-part-1-6a88a6f818f6.

[11] Harshal Charde. *Increasing Virtualization Trend, Associated Risks, and Ways to Mitigate them*. Accessed on: 27-04-2024. 2022. URL: https://cloudlabs.ai/blog/virtualization-risks-and-ways-to-mitigate-them/.

[12] Susanta Nanda Tzi-cker Chiueh and Stony Brook. "A survey on virtualization technologies." In: *Rpe Report* 142 (2005).

[13] Gilles Fedak et al. "Xtremweb: A generic global computing system." In: *Proceedings first IEEE/ACM international symposium on cluster computing and the grid*. IEEE. 2001, pp. 582–587.

[14] Martin Fowler. *Blue Green Deployment*. Accessed on: 09-05-2024. 2010. URL: https://martinfowler.com/bliki/BlueGreenDeployment.html.

[15] Vijay K Garg. *Principles of distributed systems*. Vol. 3144. Springer Science & Business Media, 2012.

[16] David Goltzsche et al. *Edgar: Offloading Function Execution to The Ultimate Edge*. Tech. rep. Tech. rep. 2021. doi: 10.24355/dbbs. 084-202111031112-0.

[17] David Gourley and Brian Totty. *HTTP: the definitive guide*. " O'Reilly Media, Inc.", 2002.

[18] *gpugrid.net*. Accessed on: 24-05-2024. URL: https://www.gpugrid.net/.

[19] Marcin Grzejszczak. *Zero-Downtime Deployment With a Database*. Accessed on: 09-05-2024. 2016. URL: https://dzone.com/articles/zero-downtime-deployment-with-a-database-1.

[20] Dongyan Huang, Xiaoli Ma, and Shengli Zhang. "Performance analysis of the raft consensus algorithm for private blockchains." In: *IEEE Transactions on Systems, Man, and Cybernetics: Systems* 50.1 (2019), pp. 172–181.

[21] IBM. *What is virtualization?* Accessed on: 18-05-2024. 2016. URL: https://www.ibm.com/topics/virtualization.

[22] Nancy Jain and Sakshi Choudhary. "Overview of virtualization in cloud computing." In: *2016 Symposium on Colossal Data Analysis and Networking (CDAN)*. IEEE. 2016, pp. 1–4.

[23] Paul James. *Major Web Hosting Challenges and Solutions*. Accessed on: 08-05-2024. URL: https://www.infince.com/blog/major-web-hosting-challenges-and-solutions.

[24] Patrick Jauernig, Ahmad-Reza Sadeghi, and Emmanuel Stapf. "Trusted execution environments: properties, applications, and challenges." In: *IEEE Security & Privacy* 18.2 (2020), pp. 56–60.

[25] David Kaplan. "Hardware VM Isolation in the Cloud: Enabling confidential computing with AMD SEV-SNP technology." In: *Queue* 21.4 (2023), pp. 49–67.

[26] Jonathan Kirsch and Yair Amir. "Paxos for system builders: An overview." In: *Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware*. 2008, pp. 1–6.

[27] Eddie Kohler. *HotCRP*. Accessed on: 20-05-2024. URL: https://hotcrp.com/.

[28] Kevin Kollenda. "General overview of AMD SEV-SNP and Intel TDX." In: (2023). URL: https://sys.cs.fau.de/extern/lehre/ws22/akss/material/amd-sev-intel-tdx.pdf.

[29] Jeff Lantz. "Understanding Website Hosting." In: *GPSolo* 37.1 (2020), pp. 40–43.

[30] Eric D. Larson. *HTTP Chunking*. Accessed on: 26-04-2024. URL: https://www.oracle.com/technical-resources/articles/javame/chunking.html.

[31] Erick Lavoie et al. "Pando: personal volunteer computing in browsers." In: *Proceedings of the 20th International Middleware Conference*. 2019, pp. 96–109.

[32] Abdeldjalil Ledmi, Hakim Bendjenna, and Sofiane Mounine Hemam. "Fault tolerance in distributed systems: A survey." In: *2018 3rd International Conference on Pattern Analysis and Intelligent Systems (PAIS)*. IEEE. 2018, pp. 1–5.

[33] Mengyuan Li et al. "A systematic look at ciphertext side channels on AMD SEV-SNP." In: *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2022, pp. 337–351.

[34] Albert van der Linde et al. "Legion: Enriching internet services with peer-to-peer interactions." In: *Proceedings of the 26th International Conference on World Wide Web*. 2017, pp. 283–292.

[35] *List of volunteer computing projects*. Accessed on: 24-05-2024. URL: https://en.wikipedia.org/wiki/List_of_volunteer_computing_projects.

[36] Akshay Luther et al. "Alchemi: A. NET-based Enterprise Grid Computing System." In: *International conference on Internet computing*. Citeseer. 2005, pp. 269–278.

[37] Saeid Mofrad et al. "A comparison study of Intel SGX and AMD memory encryption technology." In: *Proceedings of the 7th International Workshop on Hardware and Architectural Support for Security and Privacy*. 2018, pp. 1–8.

[38] San Murugesan and Irena Bojanova. "Cloud computing: an overview." In: *Encyclopedia of cloud computing* (2016), pp. 1–14.

[39] N-able. *Centralized Networks vs Decentralized Networks*. Accessed on: 08-05-2024. 2018. URL: https://www.n-able.com/blog/centralized-vs-decentralized-network.

[40] Kevin Nguetchouang et al. "SVD: A Scalable Virtual Machine Disk Format." In: *IEEE Transactions on Cloud Computing* (2024).

[41] Henrik Nielsen et al. *Hypertext Transfer Protocol – HTTP/1.1*. RFC 2616. June 1999. DOI: 10.17487/RFC2616. URL: https://www.rfc-editor.org/info/rfc2616.

[42] Noam Nisan et al. "Globally distributed computation over the internet-the popcorn project." In: *Proceedings. 18th International Conference on Distributed Computing Systems (Cat. No. 98CB36183)*. IEEE. 1998, pp. 592–601.

[43] Diego Ongaro. "Consensus: Bridging theory and practice." PhD thesis. Stanford University, 2014.

[44] Diego Ongaro and John Ousterhout. "In search of an understandable consensus algorithm." In: *2014 USENIX annual technical conference (USENIX ATC 14)*. 2014, pp. 305–319.

[45] Petar Paradžik, Ante Derek, and Marko Horvat. "Formal Security Analysis of the AMD SEV-SNP Software Interface." In: *arXiv preprint arXiv:2403.10296* (2024).

[46] Joana Pecholt and Sascha Wessel. "CoCoTPM: Trusted platform modules for virtual machines in confidential computing environments." In: *Proceedings of the 38th Annual Computer Security Applications Conference*. 2022, pp. 989–998.

[47] Jeremy Powell. *AMD SEV-SNP Attestation: Establishing Trust in Guests*. Accessed on: 20-05-2024. 2022. URL: https://www.amd.com/content/dam/amd/en/documents/developer/lss-snp-attestation.pdf.

[48] James Purton. *Why the centralization vs decentralization debate is not clear cut*. Accessed on: 08-05-2024. 2023. URL: https://www.servers.com/news/blog/why-the-centralization-vs-decentralization-debate-is-not-clear-cut.

[49] QEMU. *Documentation/Networking*. Accessed on: 19-05-2024. URL: https://wiki.qemu.org/Documentation/Networking.

[50] Federico Razzoli. *Mastering MariaDB*. Packt Publishing Ltd, 2014.

[51] Matei Ripeanu. "Peer-to-peer architecture case study: Gnutella network." In: *Proceedings first international conference on peer-to-peer computing*. IEEE. 2001, pp. 99–100.

[52] JD Rockefeller. *Web Hosting Guide for Beginners*. JD Rockefeller, 2016.

[53] Mohamed Sabt, Mohammed Achemlal, and Abdelmadjid Bouabdallah. "Trusted execution environment: What it is, and what it is not." In: *2015 IEEE Trustcom/BigDataSE/Ispa*. Vol. 1. IEEE. 2015, pp. 57–64.

[54] Dheeraj Sanghi, Pankaj Jalote, and Puneet Agarwal. “Using proximity information for load balancing in geographically distributed Web server systems.” In: *Eurasian Conference on Information and Communication Technology*. Springer. 2002, pp. 659–666.

[55] AMD Sev-Snp. “Strengthening VM isolation with integrity protection and more.” In: *White Paper, January* 53 (2020), pp. 1450–1465.

[56] Mishal Shah. *Browser behaviour in HTTP/1.1 vs HTTP/2*. Accessed on: 26-04-2024. 2021. URL: https://mishal.dev/browser-http-behaviour/.

[57] Aayush Shukla. *Web Hosting*. Accessed on: 07-03-2024. 2020. URL: https://aayushshukla13.wordpress.com/2020/08/11/example-post-3/.

[58] Rahul Soni. *Nginx*. Springer, 2016.

[59] Andrew S Tanenbaum and Maarten van Steen. “Distributed Systems.” In: ().

[60] Maarten Van Steen and Andrew S Tanenbaum. “A brief introduction to distributed systems.” In: *Computing* 98 (2016), pp. 967–1009.

[61] virtee. Accessed on: 31-05-2024. URL: https://github.com/virtee/sev-snp-measure.

[62] virtee. *snpguest*. Accessed on: 20-05-2024. URL: https://github.com/virtee/snpguest.

[63] Mythry Vuyyuru et al. “An overview of cloud computing technology.” In: *International Journal of Soft Computing and Engineering* 2.3 (2012), pp. 244–247.

[64] Clinton Wong. *Http pocket reference: Hypertext transfer protocol*. " O’Reilly Media, Inc.", 2000.

[65] Raymond Lei Xia and Jogesh K Muppala. “A survey of bittorrent performance.” In: *IEEE Communications surveys & tutorials* 12.2 (2010), pp. 140–158.

[66] John Yannakopoulos. “Hypertext transfer protocol: A short course.” In: *University of Crete* (2003).

[67] Mingchen Zhao et al. “Peer-assisted content distribution in akamai netsession.” In: *Proceedings of the 2013 conference on Internet measurement conference*. 2013, pp. 31–42.