Extending the Functionality of the Realm Gateway

Maria Riaz

School of Electrical Engineering

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

Espoo 25.09.2019

Supervisor

Prof. Raimo Kantola

Advisors

Juha-Matti Tilli

Hammad Kabir

Aalto University
School of Electrical
Engineering

Aalto University
School of Electrical Engineering

Aalto University, P.O. BOX 11000, 00076 AALTO
www.aalto.fi
Abstract of the master's thesis

Author Maria Riaz

Title Extending the Functionality of the Realm Gateway

Degree programme Masters in Computer, Communication and Information Sciences

Major Communications Engineering **Code of major** ELEC3029

Supervisor Prof. Raimo Kantola

Advisors Juha-Matti Tilli, Hammad Kabir

Date 25.09.2019 **Number of pages** 86 **Language** English

Abstract

The promise of 5G and Internet of Things (IoT) expects the coming years to witness substantial growth of connected devices. This increase in the number of connected devices further aggravates the IPv4 address exhaustion problem. Network Address Translation (NAT) is a widely known solution to cater to the issue of IPv4 address depletion but it poses an issue of reachability. Since Hypertext Transfer Protocol (HTTP) and Hypertext Transfer Protocol Secure (HTTPS) application layer protocols play a vital role in the communication of the mobile devices and IoT devices, the NAT reachability problem needs to be addressed particularly for these protocols.

Realm Gateway (RGW) is a solution proposed to overcome the NAT traversal issue. It acts as a Destination NAT (DNAT) for inbound connections initiated towards the private hosts while acting as a Source NAT (SNAT) for the connections in the outbound direction. The DNAT functionality of RGW is based on a circular pool algorithm that relies on the Domain Name System (DNS) queries sent by the client to maintain the correct connection state. However, an additional reverse proxy is needed with RGW for dealing with HTTP and HTTPS connections.

In this thesis, a custom Application Layer Gateway (ALG) is designed to enable end-to-end communication between the public clients and private web servers over HTTP and HTTPS. The ALG replaces the reverse proxy used in the original RGW software. Our solution uses a custom parser-lexer for the hostname detection and routing of the traffic to the correct back-end web server. Furthermore, we integrated the RGW with a policy management system called Security Policy Management (SPM) for storing and retrieving the policies of RGW. We analyzed the impact of the new extensions on the performance of RGW in terms of scalability and computational overhead. Our analysis shows that ALG's performance is directly dependent on the hardware specification of the system. ALG has an advantage over the reverse proxy as it does not require the private keys of the back-end servers for forwarding the encrypted HTTPS traffic. Therefore, using a system with powerful processing capabilities improves the performance of RGW as ALG outperforms the NGINX reverse proxy used in the original RGW solution.

Keywords NAT, Realm Gateway, HTTP, HTTPS, Application Layer Gateway, Security Policy Management

Preface

This master thesis has been carried out in the department of Communications and Networking at Aalto University. It is completed under the 5G-FORCE research project of Aalto University.

First of all, I would like to thank Allah for the countless blessings in my life. I am deeply indebted to my professor Raimo Kantola for providing me with the opportunity to work on this research topic and sharing valuable insights throughout the time I was working under his supervision. I would like to thank my advisors Juha-Matti Tilli and Hammad Kabir for their intellectual guidance, particularly Juha who was always patient with me and addressed any ambiguities I encountered during the process. His knowledge on the research area proved crucial in the realization of this thesis.

A big thank you to all my friends for their constant support especially Hassaan for his technical insights during the course of my studies and Umar and Kinza for encouraging me and being there for me whenever I needed help. Finally, I would like to express my gratitude to my family, particularly my parents for their unconditional love and prayers.

Otaniemi, 25.9.2019

Maria Riaz

Contents

List of Figures

List of Tables

Abbreviations

AI	Artificial Intelligence
ALG	Application Layer Gateway
AL-NAT	Application Layer NAT
API	Application Program Interface
APNIC	Asia-Pacific Network Information Centre
ARP	Address Resolution Protocol
ASP	Answer Set Programming
AVP	Attribute Value Pair
CES	Customer Edge Switching
CG-HCPCLI	Carrier Grade HTTP Connect proxy client
CGN	Carrier Grade NAT
CIDR	Classless Inter-Domain Routing
CR-LF	Carriage Return-Line Feed
DDoS	Distributed Denial of Service
DFSM	Deterministic Finite State Machine
DMTF	Distributed Management Task Force
DNAT	Destination Network Address Translation
DNS	Domain Name System
DPI	Depp Packet Inspection
FQDN	Fully Qualified Domain Name
FTP	File Transfer Protocol
GB	Gigabytes
GSMA	Global System for Mobile Association
GUI	Graphical User Interface
HTML	Hyper Text Markup Language
HTTP	Hyper Text Transfer Protocol
HTTPS	Hyper Text Transfer Protocol Secure
IANA	Internet Assigned Numbers Authority
ICE	Interactive Connectivity Establishment
IETF	Internet Engineering Task Force
IGDP	Internet Gateway Device Protocol
ILASP	Inductive Learning of Answer Set Programs
IoT	Internet of Things
IP	Internet Protocol
IPSec	Internet Protocol Security
IPv4	Internet Protocol version 4
IPv6	Internet Protocol version 6
ISP	Internet Service Provider
ISO	International Organization for Standardization
M2M	Machine to Machine
NAPT	Network Address Port Translator
NAT	Network Address Translation
NAT-PMP	Network Address Translation Port Mapping Protocol
NIC	Network Interface Controller

OS	Operating System
OSI	Open Systems Interconnection
PBMS	Policy Based Management Systems
PCP	Port Control Protocol
PDP	Policy Decision Point
PEP	Policy Enforcement Point
PKI	Public Key Infrastructure
PMS	Policy Management System
PMT	Policy Management Tool
PR	Policy Repository
QUIC	Quick UDP Internet Connection
REST	Representational State Transfer
RGW	Realm Gateway
RIR	Regional Internet Registry
SDN	Software Defined Networking
SFQDN	Service Fully Qualified Domain Name
SIP	Session Initiation Protocol
SLA	Service Level Agreement
SMTP	Simple Mail Transfer Protocol
SNAT	Source Network Address Translation
SNI	Server Name Indication
SNMP	Simple Network Management Protocol
SPM	Security Policy Management
SSL	Secure Sockets Layer
STUN	Session Traversal Utilities for NAT
TCP	Tranmission Control Protocol
TLS	Transport Layer Security
TTL	Time To Live
TURN	Traversal Using Relays around NAT
UDP	User Datagram Protocol
UPnP	Universal Plug and Play
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
5G	Fifth Generation (of mobile network)

1 Introduction

With the advent of 5G and Internet of Things (IoT) there has been a substantial increase in the number of connected devices. According to a report released by GSM Association (GSMA) Intelligence in 2019, there are expected to be 25 billion connected devices by the end of 2025 out of which 5.8 billion users will be mobile network subscribers, accounting for 71% of world's population [1]. All these devices need IP addresses to communicate with each other and connect to the services provided by the realm of Internet Service Providers (ISPs). Internet Protocol version 4 (IPv4) suffers from various technical challenges in this regard, ranging from issues in routing, security issues and the most prominent issue of scalability due to its limited address space [2].

The problem of IPv4 address exhaustion was predicted in the late 1980s when Internet started experiencing tremendous, de-centralized growth. Concurrent evolution of social web and mobile technology helped in increasing the global reach of Internet, occupying the usable IP address space. By 2011, the last of the two unreserved /8 address blocks were allocated by Internet Assigned Numbers Authority (IANA) to Asia-Pacific Network Information Centre (APNIC), which is the Regional Internet Registry (RIR) in the Asia Pacific region. Various solutions were proposed to overcome this problem of IPv4 address depletion. Development and deployment of a successor protocol to IPv4, namely, Internet Protocol version 6 (IPv6) [3], Network Address Translation (NAT) [4] and Classless Inter-Domain Routing (CIDR) [5] were some of the proposed solutions adopted to mitigate the problem.

Development of NAT managed to reduce the address exhaustion problem but it had limitations. In the traditional NAT, devices in private network initiate a connection towards devices in the public Internet using a publicly reachable IP address and NAT stores the IP and port mapping in a translation table. However, this poses a question of reachability for users in the public domain when there is no existing mapping in the NAT for a particular public host and the destined private host. This reachability problem has been addressed by various NAT traversal solutions like User Datagram Protocol (UDP) Hole Punching, STUN [6], TURN [7] and ICE [8] but they are not optimal for mobile networks.

To cater to the problem of reachability of hosts behind NAT, a solution was proposed called Realm Gateway (RGW) [9][10]. Realm Gateway acts as a Source NAT (SNAT) for outbound connections from the private network hosts and a Destination NAT (DNAT) for traversing inbound connections from the clients in public realm towards the private hosts situated behind the RGW. RGW uses a circular pool containing a fixed number of NAT outbound IP addresses that are used for creating dynamic binding between the public and private domain users. Whenever a user in the public domain initiates a connection towards a private host sitting behind RGW, an IP address is temporarily allocated from RGW's circular pool in response to the Domain Name System (DNS) query sent by public client which is released upon establishment

of a successful connection.

RGW works well for most of the protocols that can be identified correctly by the 5-tuple based on a single flow. However HTTP and HTTPS connections operate differently, where a browser might initiate multiple connections using different source ports to the web server for fully retrieving the contents of a page. This could be problematic for RGW which relies on Circular Pool for temporarily allocating an IP address for a particular hostname in response to a DNS query and maintaining the correct NAT binding based on the client's port and IP address. It could result in stalling of the connections when the same IP address being accessed by the browser becomes available for next allocation. Thus a reverse proxy is needed in conjunction with RGW to ensure smooth connectivity using HTTP and HTTPS protocols and help in efficient utilization of available address space to meet the demands of 5G and IoT.

1.1 Research Problem

The purpose of this thesis is to extend the functionality of Realm Gateway by adding better support for widely used application layer protocols particularly HTTP and HTTPS to improve the scalability of the existing software. Additionally, the thesis involves studying a Security Policy Management (SPM) system and integrating it to RGW to improve its usability and offer more fine-grained control over the host policies.

Currently RGW uses an open-source web server for reverse proxying. The open-source web proxy works well with HTTP connections but connections using HTTPS run over Transport Layer Security (TLS) and the web proxy requires server certificates and private keys to decrypt the traffic for forwarding it to the correct host. This creates a potential security vulnerability as RGW could be operated by an ISP and this decryption of traffic would give them access to sensitive user data. The decryption followed by re-encryption also degrades the performance of HTTPS connection by increasing the time taken for connection establishment in comparison to HTTP connection.

1.2 Objective and Scope

The thesis aims to solve the connectivity and scalability problem for application layer protocols, specifically HTTP and HTTPS by developing an application layer gateway and integrating it with RGW to study the changes in the performance of the system. RGW is a policy dependent solution and in the original design the host policies are loaded from a local repository. This raises a usability concern making it harder to customize the host policies. Therefore, one aspect of the thesis involves integrating a policy management system with RGW to update the policies remotely and in a user-friendly manner.

Using a custom parser-lexer [21], ALG detects the protocol and hostname based on information within the payload and routes the connection to correct internal private IPv4 address. This proves beneficial specifically for HTTPS that relies on encryption mechanisms to ensure secure communication. For this purpose, ALG requires changes only at the NAT middlebox and no changes are required at the endpoints.

Although there are some known application layer protocols such as File Transfer Protocol (FTP), Simple Mail Transfer Protocol (SMTP) and other custom protocols that can operate over TLS, those are considered out of the scope of this thesis and we will only discuss HTTPS. ALG can be customized to work on non-standard ports for accessing web services but that would require modifying the policies of RGW before running an ALG. Hence ALG cannot handle HTTP and HTTPS connections on non-standard ports unless specified explicitly using the RGW policies. SPM is discussed but the specifics of improving security using policy control are out of the scope of this thesis.

1.3 Structure

The thesis is structured into six chapters. The following Chapter 2 provides the necessary background information in order to understand the working of Application Layer Gateway designed for this thesis. It discusses application layer protocols, functionality and operation of a traditional NAT and also NAT that operates on layer 7 of the OSI (Open Systems Interconnection) model known as application layer NAT. This chapter also gives an overview of the existing application layer gateways and their implementation mechanisms. Finally, the policy management system is explained.

The third chapter presents a detailed analysis of Realm Gateway, the design principles of the software and how it offers reliable communication by maintaining a reputation system for different entities. Chapter 4 explains the design of custom Application Layer Gateway, its architecture and in what aspects it is better than the existing solutions. The integration to RGW is also explained in Chapter 4. The interface to the policy database based on Representational State Transfer (REST) architecture is used for configuring the policies of RGW and ALG and it is discussed at the end of 4^{th} chapter.

The results obtained from testing the developed solution are analyzed in Chapter 5. The last chapter discusses the outcome of the implemented solution and also gives a brief overview of the future work.

2 Background

This chapter provides a thorough background of the topics relevant in understanding the key notions of this thesis. In the beginning application layer protocols that enable end-to-end communication between different entities are described specifically addressing HTTP and HTTPS. This is followed by an explanation of NAT and examining the working of an application layer NAT. Existing solutions of application layer gateways are then analyzed and towards the end of the chapter the role of security policy management systems in enhancing network security is discussed.

2.1 Application layer protocols

A set of rules that ensure smooth data transmission between end devices form a protocol. Protocols in a network are divided into various layers to aid the process of message exchange between different users having varied network requirements. In the stack of communication protocols, all the protocols interact with one another where each higher layer is served by a layer below it.

International Organization for Standardization (ISO) defined the system interactions of different layers along with their functionality in the Internet Protocol suite. The model comprises of 7 abstraction layers and is known as Open Systems Interconnection (OSI) model [11]. Internet Engineering Task Force (IETF) also designed a protocol stack model known as TCP/IP model that consists of four layers namely; network access layer, internet layer, transport layer and application layer, with application layer being the top layer. Figure 1 illustrates the mapping of layers in these two different models.

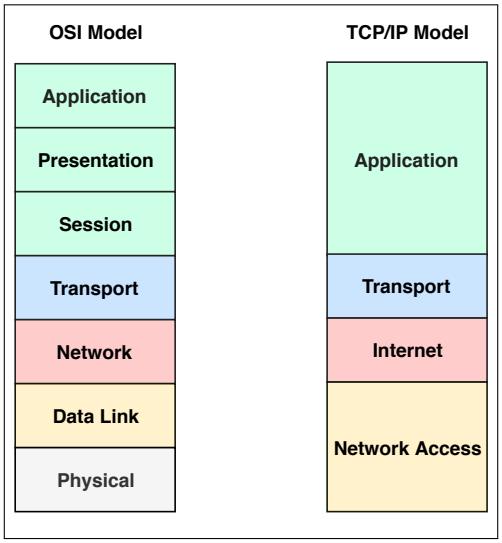

Figure 1: OSI model vs TCP/IP model

In the OSI model, application layer provides an interface between underlying network and the user applications. Application layer protocols are designed to perform the

task of framing, encoding, error reporting of data packets and use different mechanisms for authentication and ensuring privacy as described by RFC 3117 [12]. To exchange data, application layer uses various protocols like FTP, SMTP, HTTP, HTTPS, DNS to name a few. Today most of the well-known application layer protocols have TCP or UDP as the underlying transport protocol. Different server applications use distinct port numbers standardized by IANA and these ports in conjunction with IP addresses identity the target destination and the service being requested.

HTTP and HTTPS form the basis of world wide web. They are two of the most widely used application layer protocols that are expected to play an important role in smooth adoption of IoT and machine to machine (M2M) communication [13]. Their working and functionalities are explained in the following subsections.

2.1.1 HTTP

HTTP is a stateless [1] application protocol that follows the client-server model for communication in which the tasks are divided between the provider of service, namely servers and the clients requesting those resources [14]. The request is by default sent to port 80 of the web server commonly using TCP as the underlying protocol to ensure reliability. Being a stateless protocol, each of the request and response message pairs exchanged between the client and server can be processed independently by the receiving end without retaining information about the previous messages exchanged in the session.

During a typical HTTP session, the client first establishes a TCP connection with a server. When the TCP 3-way handshake has been completed, a request message is sent by the client which is composed of a request method, a resource, headers and some additional content as shown in Figure 2. After processing the request, the server sends a response message which contains the protocol version, a three-digit status code, a human-readable status and response headers in addition to some optional content [13].

In the previous versions of HTTP (version 0.9 and 1), a new connection was created for every request/response pair. To overcome the shortcomings of HTTP/1.0, HTTP/1.1 emerged as the first HTTP standard version in 1997 only four months after the documented release of HTTP/1.0. HTTP/1.1 has been widely adopted owing to the performance optimizations it offered including the keep-alive connections, caching mechanism and chunked encoding transfers to name a few. Using the keep-alive headers, the same connection was used for sending multiple requests which significantly improved performance by reducing latency as the 3-way TCP handshake was not initiated for every new request.

[1] HTTP by design is a stateless protocol but modifications at the client end can help in maintaining the information about the session state and make it behave in a stateful manner for e.g, by using cookies

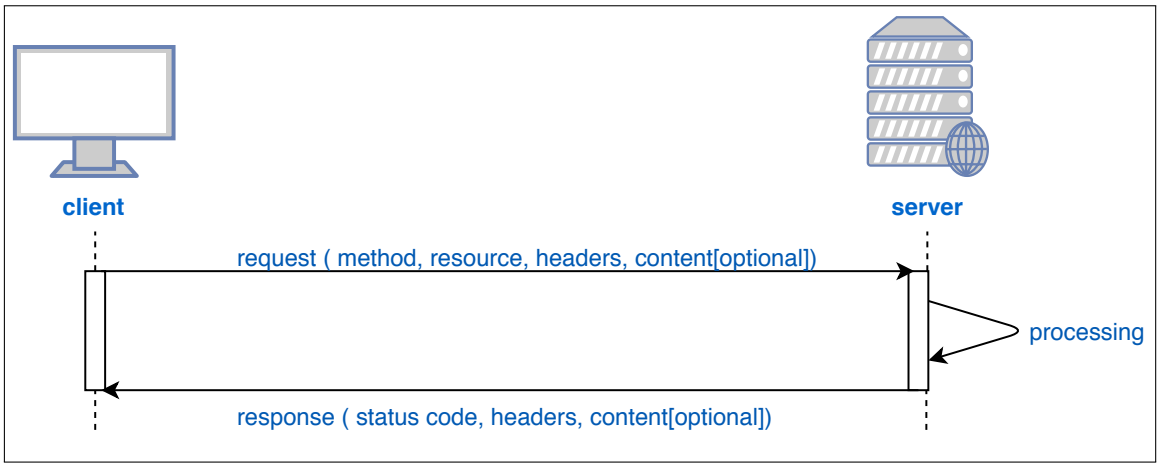

Figure 2: Client-Server model in HTTP

HTTP protocol uses uniform resource identifiers (URIs) for identifying the resources requested by the client. The requested resource could be a file including data, text, images, some other hypermedia content or even an IoT application interacting with an IoT object and is referenced using a uniform resource locator (URL). HTTP URL is constructed using a scheme that involves specifying the hostname, the port on which the server is listening, path to the resource followed by query string and a fragment identifier. Specifying the port, query and fragment identifier is optional. Also the fragment identifier is appended by the client and not sent to server in the HTTP request. HTTP URL for requesting an HTML file located in 'img' repository from the domain 'www.example.com' is shown in Figure 3 as an example.

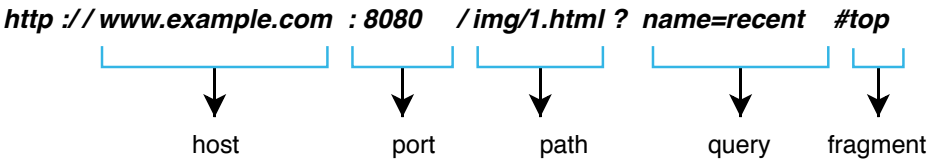

Figure 3: Syntax for URL in http scheme

HTTP Request Message

Request message comprises of a Request-Line which identifies the Request-Method, resource identifier, version of HTTP protocol being used, headers followed by the CR-LF (carriage return-line feed) sequence that indicates the end of a line. Request-Methods determine how the server should handle the requested resource identified by the resource identifier. Eight different Request-Methods are mentioned in RFC 2616 [14] and they are case-sensitive. Table 1 provides a comprehensive summary of the methods and what they expect the server to do. After the request method, the resource identifier is specified followed by the protocol version supported by the

Methods	Functionality
GET	requests the resource specified by the URI
PUT	store the requested resource to the specified URI
HEAD	requests only the header of the resource specified by the URI
POST	requests the server to pass the data to the resource specified by URI
DELETE	requests the specified resource at the given URI to be deleted
TRACE	requests the server to send a duplicate of the request message initially sent by the client
CONNECT	method used by a proxy for switching the TCP/IP connection to a secure tunnel
OPTIONS	requests information regarding the methods supported by the server

Table 1: HTTP Request methods and their description

client. Request headers are defined in the next line and they contain an attribute value pair (AVP) separated by a colon. A detailed list of request headers is specified in the RFC 2616 [14] from which the following three are considered most important.

User-Agent: Specifies the information about the requesting user or client. User-Agent field mostly contains the information about the web browser sending the request. For example, if the requesting web browser is Mozilla then the User-Agent field might look like this:
`User-Agent: Mozilla/6.0 (Windows NT 6.1; Win64; x64; rv:48.0) Gecko/20100101 Firefox/48.0`

Accept-Language: Specifies the language preferred for communication by the user-agent. If no language is specified than the server should assume that all languages are equally preferred.

Host: Specifies the name of the host and the port from which the resource is being requested. This field is compulsory in a request message. If no port is defined, then the request is sent to the default port 80. An example Host-header can be:
`Host: www.google.com:8080`

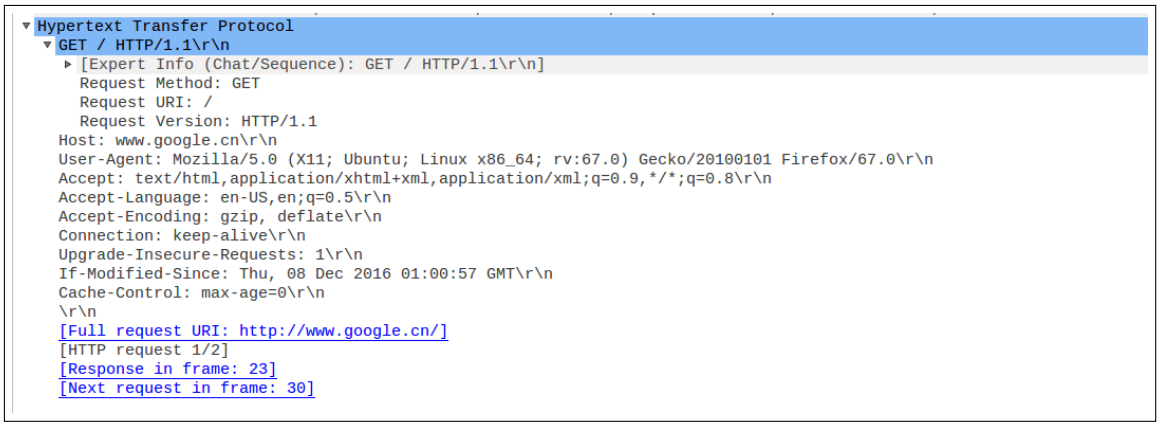

Figure 4: HTTP Request to fetch the page 'www.google.cn'

HTTP request message sent to the domain 'www.google.cn' contains a number of different headers as indicated by the wireshark capture in Figure 4.

HTTP Response Message

The response message contains the version of HTTP supported by server, the status code in response to the requested message and the textual phrase for that particular status code. The status line is followed by a CR-LF sequence to indicate the end-of-line. Then there are some response headers followed by an optional response body.

Status code in the status-line are broadly classified into five categories. The first digit of the code is representative of the code class for any particular code [15].

1xx : They are called informational codes that signify the request has been received and is being processed.
2xx : They are called success codes which indicate successful receiving and processing of request.
3xx : They are called redirection codes which indicate that the user agent needs to take some additional action for the request to be processed.
4xx : They are called client error codes and are an indication of syntax error in the request or unavailability of the requested resource.
5xx : They are called server error codes which indicate that the server was unable to process the request.

The details of some of the popular status codes are given in Table 2.

Code	Meaning
200 OK	request was received and successfully processed
300 Multiple Choices	requested data has been moved
301 Moved Permanently	data was found on a temporary URI and has been moved to a new location
304 Not Modified	request was successful and the data was not modified since last accessed
400 Bad Request	server is unable to process the request
401 Unauthorized	requires user authentication for processing the request
403 Forbidden	server processed the request but cannot fulfill it due to some reason
404 File Not Found	server didn't find anything specified by the requested URI
500 Internal Server Error	server encountered some internal problem due to which it cannot process the request
550 Permission Denied	current user does not have the permission to perform the requested action.

Table 2: Common HTTP Response Codes

Following the status line, there are response headers in the message which also contain a field value pair separated by a colon. These field provide help in understanding the response sent by the server and include Accept-Ranges, Age, Retry-After, Location, Server, ETag, www-Authenticate, Proxy-Authenticate, Vary, Warning.

A wireshark capture showing a response message by the server of host 'www.google.cn' is shown in Figure 5.

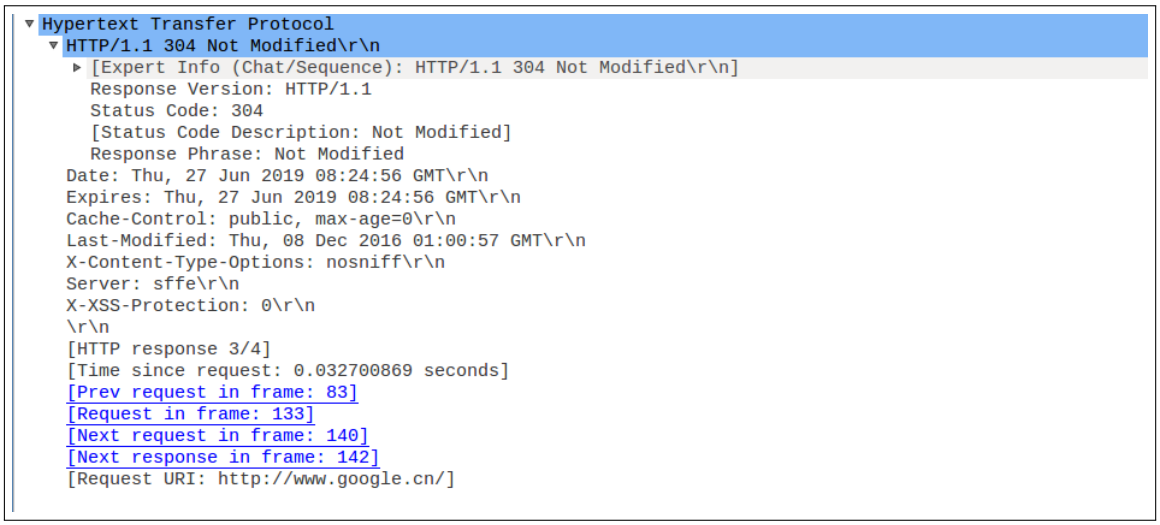

Figure 5: HTTP response sent by 'www.google.cn'

HTTP Headers

In addition to the request and response headers, HTTP messages also contain some general headers and entity headers that help the client and server in understanding messages of each other.

General headers: These headers are general for both request and response messages but do not apply to the message body being sent. These can be used in request messages or response messages depending on the context in which they are being used. They include Date, Pragma, Cache-Control, Connection, Trailer, Transfer-Encoding, Upgrade, Via, Warning. Date, Cache-Control and Connection are the most commonly used general headers.

Date: This header includes the time at which the message is created and it should follow the format specified in RFC 1123 [16].

Cache-Control: This general-header field indicates how the caching mechanism works for the specified resource. Cache-control field is further broken down into directives that control different aspects of caching. The following example indicates that the resource can be cached by any browser since it is marked public and the fetched response can be reused for the next 90 seconds.

```
Cache-Control: public, max-age = 90
```

Connection: This field helps in specifying the type of connection between the client and server. In HTTP/1.1 the default is set to keep the connection alive after processing of the single request. It must be explicitly set to close if the client wants the connection to terminate after completion.

Entity Headers: These headers apply to the entity body being transferred. Allow, Content-Encoding, Content-Length, Content-Location, Content-MD5, Last-Modified, Content-Type, Content-Language, Expires are all different types of entity headers.

HTTP is arguably the most widely used protocol which is being extended day by day by improving the limitations in the previous versions. It should be noted that the message sent from the client is relayed through a number of different networks before reaching the client. There might be some intermediary proxy servers or gateways that modify the request before it can be received by the destined server. These proxy servers aim at reducing the processing time by serving cached responses whenever possible. Hence, proper security mechanisms are needed to ensure that integrity and privacy of data is maintained when it is being transferred. For transferring sensitive data, another protocol is used in conjunction with HTTP called transport layer Security (TLS) protocol resulting in the development of secure version of HTTP called HTTPS.

HTTP/2 was published in 2015 which offered improvements to the commonly used version of HTTP, HTTP/1.1 [17]. Another version of HTTP is in the development phase known as HTTP/3 which works over QUIC (Quick UDP Internet Connection) transport protocol that operates over UDP to ensure secure communication. The implemented solution in the thesis is not designed to work with plain-text HTTP/2 but it can support the encrypted version of HTTP/2, called HTTP/2 over TLS.

2.1.2 HTTPS

When Internet started gaining traction among the network users, a need for secure communication channel surfaced. HTTPS was invented which allowed the receiver and sender of data applications to encrypt the traffic and provide authentication mechanisms to verify the end hosts using TLS as the underlying protocol instead of encapsulating the HTTP messages directly using TCP. Previously HTTPS was primarily used for exchanging sensitive information like banking credentials but today almost 50% of the world's websites have switched to HTTPS [18]. The security mechanisms enforced by TLS maintain data confidentiality, data integrity and also provide server authentication mechanisms so that the client knows if the recipient server is who it claims to be. The main goal of HTTPS is to securely exchange data. It makes the system resilient against network attacks by eliminating spoofing and modification of the data at network level thereby creating a secure channel for communication.

The syntax of HTTPS URI scheme is similar to that of HTTP except the keyword 'http' is replaced by 'https' in the http scheme described in Figure 3. HTTPS connection is directed to port 443 of the destined server by default unless otherwise specified. The HTTPS message includes a request or status line followed by headers and a message body. Unlike HTTP, in HTTPS different cryptographic operations are performed on the message to encrypt it. It is also possible to upgrade an existing HTTP connection to an HTTPS connection which is specified using the upgrade

header in HTTP message the details of which can be found in RFC 2817[19].

Learning TLS protocol is crucial to understand the working of HTTPS. TLS provides security by encrypting the http request message before it is sent by the client and the same message on arrival at the server is decrypted. This is achieved using various methods, such as exchanging keys using the asymmetric public key infrastructure (PKI) to authenticate the web server, using key exchange mechanisms for securing keys for symmetric ciphers and using hashing algorithms to compare if the data has been tampered with during transit. In the beginning of an HTTPS session, the client and server undergo a TLS handshake phase followed by data compression, encryption and validating the origin of the message which is handled by TLS record protocol. After the end points are successfully authenticated only then the application data is exchanged in an encrypted format. Steps involved in TLS session establishment are shown in Figure 6.

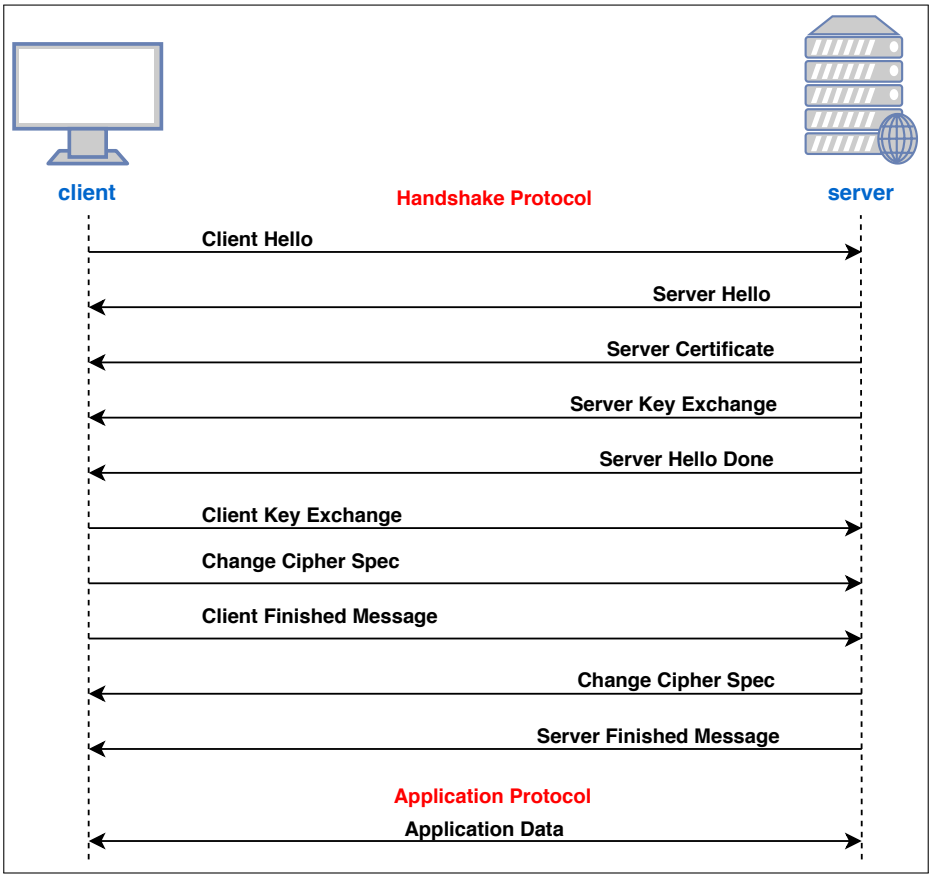

Figure 6: TLS session establishment

On establishment of a successful TCP connection, HTTPS client sends a TLS ClientHello message which includes the information about the version, compression methods and cipher suites supported by the client and some other information necessary for carrying a secure communication. The server then responds with a Server Hello message indicating the encryption settings it supports and additionally sends

its certificate to the client, which is most often a web browser, to verify server's authenticity. It is also possible to ensure two-way authentication which involves the client authentication using a client certificate if the client has specified it in the initial Hello message. In case of two-way TLS, the server requests the client's certificate and verifies it thereby adding an extra layer of security. This authentication is done using X.509 certificates based on PKI [20]. After verifying the server certificate, the client then sends a symmetric secret key known as the pre-master secret created using server's public key and also the information about the algorithm which is used to generate a master key. The server then decrypts the pre-master secret key with the help of it's private key. The pre-master key is used to compute a master key by the server which is similar to the master key generated by the client. These identical master keys are used to generate the symmetric session keys for encrypting all the subsequent data exchanged in the session.

As an example, HTTPS request is sent to 'www.aalto.fi'. The first client Hello message is unencrypted and contains information about the encryption settings as seen in the wireshark capture of Figure 7.

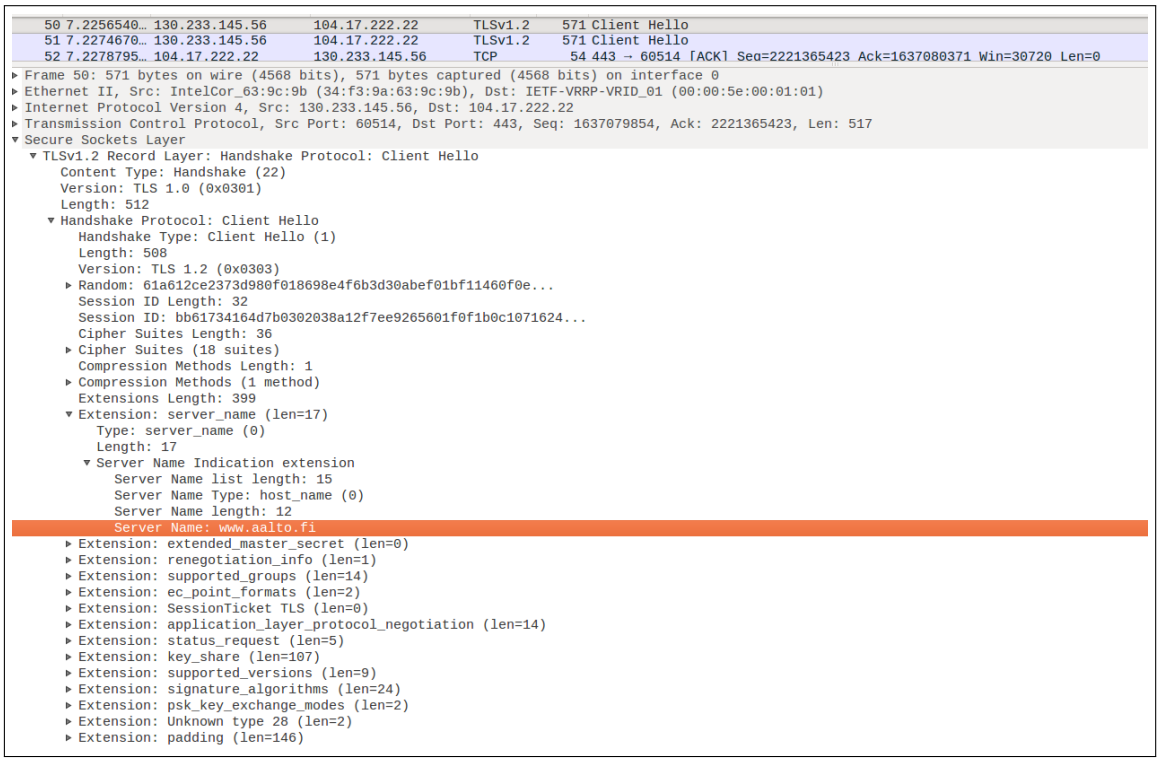

Figure 7: TLS ClientHello message sent to 'www.aalto.fi'

After the TLS Handshake is completed, server with the hostname www.aalto.fi residing at the IP address 104.7.222.22 sends the encrypted application data which cannot be deciphered without the session keys and is visible in plaintext only to the client. Figure 8 indicates that the data would appear encrypted to any other application other than the intended client.

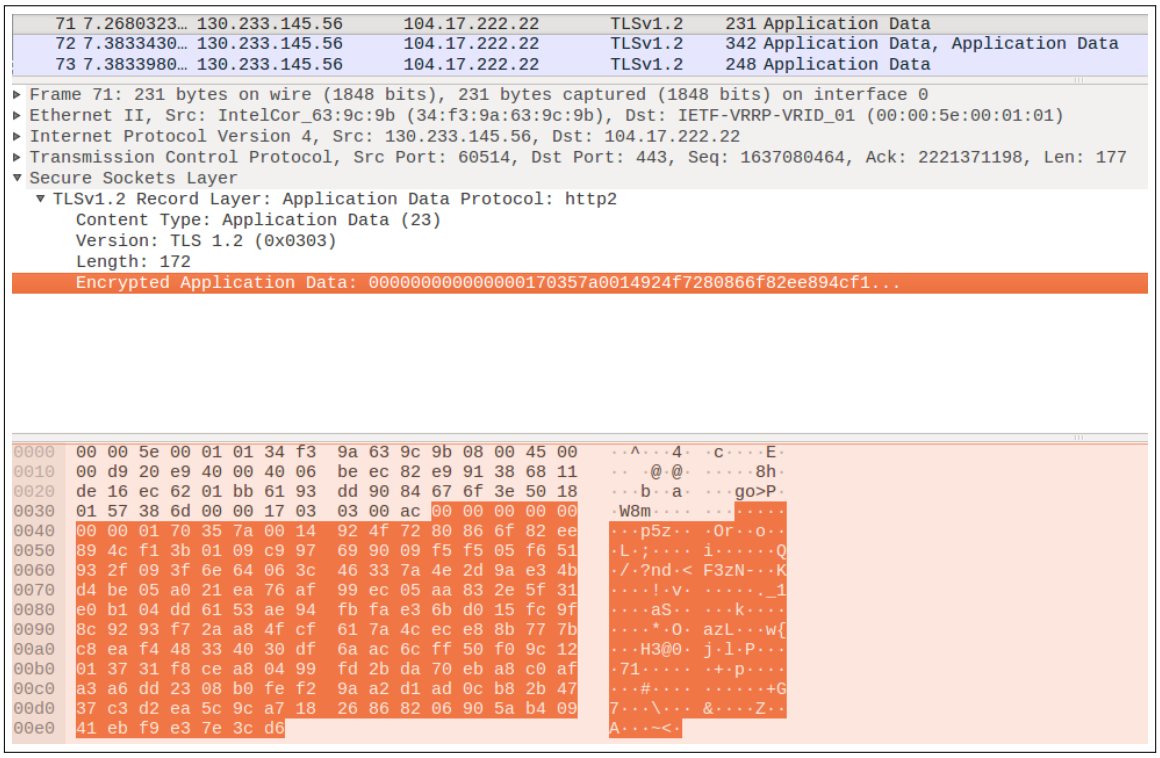

Figure 8: Encrypted Application Data sent by www.aalto.fi

Secure connections are being standardized for all websites and HTTPS is the protocol being used for public Internet because of its easy adoption. In the beginning, HTTPS experienced some trouble with virtual hosting for servers having different hostnames but same IP addresses as there was no mechanism to specify the server's hostname the client wanted to connect to during the TLS handshake. This problem is solved in the subsequent versions of TLS, where the client notifies the web server about the hostname with which it wants to initiate a connection using an additional field known as Server Name Indication (SNI). The popular desktop browsers of today namely, Internet Explorer, Mozilla Firefox, Google Chrome, Opera and Safari have a support for SNI extension in their latest versions. ALG designed in this thesis establishes an HTTPS connection with the correct server without using server certificates by using a custom parser-lexer [21] which relies on SNI field value for identifying the hostname. Even though HTTPS adds latency in comparison to HTTP and some additional load is experienced by the server due to the TLS Handshake messages exchanged in the beginning, it is expected that HTTPS would play an important role in the adoption and expansion of IoT devices.

2.2 Network Address Translation

NAT was designed for alleviating the problem of IPv4 address exhaustion caused by the expansion of Internet. It aims at connecting private realm of Internet users with users in the public domain. The basic functionality of NAT involves reusing

of existing IP address space through IP address translation. This can be done by modifying the source IP address of a packet sent by a private host to a globally accessible IP address available to a NAT gateway. This way a single NAT IP address can be used to represent an entire network of private hosts. The technique of hiding multiple IP addresses behind a single or a small pool of IP addresses is known as IP masquerading which gives an impression to a user in the public domain as if the packet is originating from the public host whereas in reality NAT is responsible for forwarding the modified packet and storing the IP mapping of each private host to the publically accessible IP address.

Organizations having hosts that do not require external connectivity are assigned addresses that are not routable on the internet. These addresses in the private domain are divided into three different blocks according to RFC 1918 [22] and include addresses in the range:

- 10.0.0.0 - 10.255.255.255 (10/8 prefix)
- 172.16.0.0 - 172.31.255.255 (172.16/12 prefix)
- 192.168.255.255 - 192.168.255.255 (192.168/16 prefix)

Additionally, a separate address block was reserved for Carrier-Grade NAT (CGN) used by Internet Service Providers to offer services to a wider range of end customers. These IP addresses belong to the range 100.64.0.0/10. The address block allowed mapping of private addresses of customer networks first to a pool of private IP addresses available to ISP before mapping it to a public IP address. The deployment of a specific address range for CGN allowed ISPs to ensure that the private addresses being used for routing to their core network are not similar to the IP addresses of their customer networks and the CGN address range can also be shared amongst different ISPs for internal routing purposes.

NAT that creates a private-to-public IP mapping is known as basic NAT but there is another variant of NAT that uses port number in conjunction with IP address during the translation and it is called Network Address Port Translation (NAPT). Three different types of NAT are discussed as follows:

Static Address Binding: In static address binding, a one-to-one address mapping is created between the private host and globally routable IP address of the NAT gateway for achieving connectivity to the outside world and the mapping is stored in a NAT table. This mapping is bidirectional and hence allows both private hosts as well as public hosts to initiate the communication as the mapping is fixed. For example, a web server can have a local IP address for the private domain and additionally a public IP address accessible to other users in the internet as indicated by Figure 9. For an outgoing connection, the source IP of the web server is changed to match NAT's public IP address and for an incoming connection the destination IP is modified. However, this type of binding does not contribute to mitigating IP

address depletion and is rarely used due to its limited scalability.

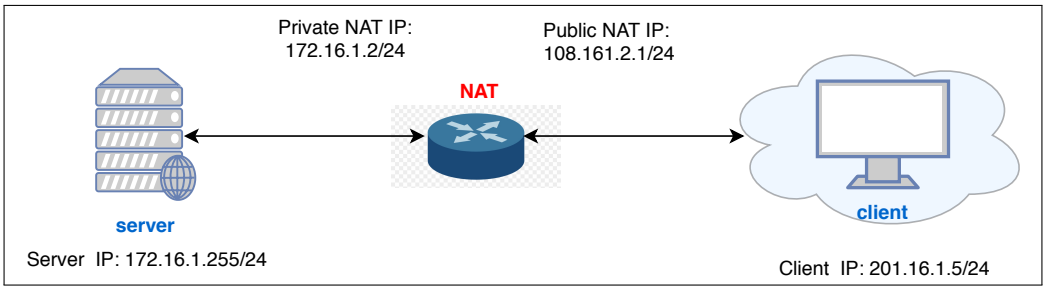

Figure 9: One-to-One NAT translation

Dynamic Address Binding: This kind of mapping is normally known as many-to-one mapping as it allows several private hosts to be represented using only a single IP address or a pool of few IP addresses. In the setup shown in Figure 10, three hosts achieve connectivity with the outside world through NAT's public IP address 108.161.2.1/24. A pool of IP addresses can also be used when dealing with large networks where one IP address is used from the pool for establishing a session and on successful establishment of a unique connection, the IP address is released and becomes available for next binding. A limitation of this variant of NAT is that it allows the private hosts to initiate a connection to a public domain host but the inverse is not possible since there is no previous record in the NAT table for new connection due to dynamic binding.

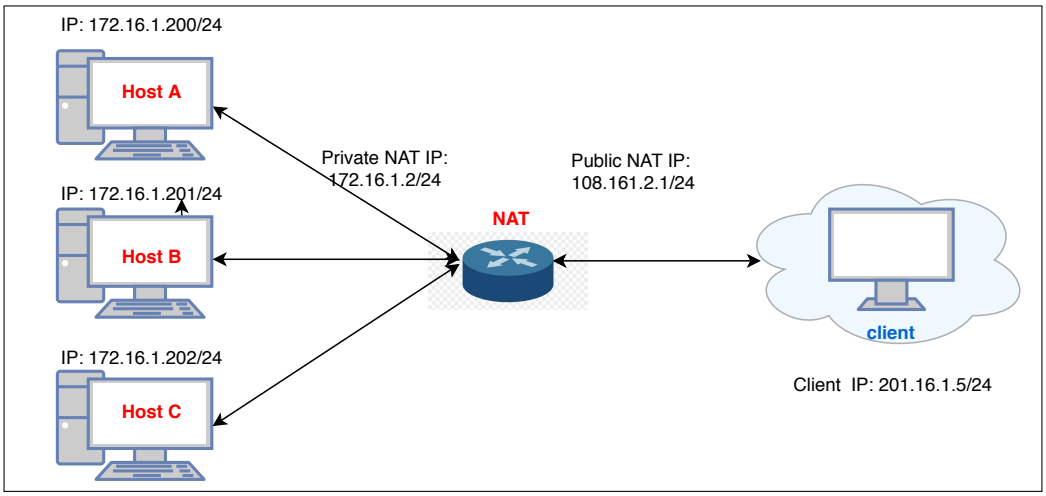

Figure 10: Many-to-One NAT translation

Another problem occurs when the clients select their own ports for establishing an outbound connection using NAT's public IP address. If both clients communicate using the same source port and NAT outbound IP address with the public hosts, the communication is bound to fail as NAT would not be able to distinguish the correct destined host for the traffic sent by the public host. Static port forwarding can allow

one of the hosts to run a particular service, for example by allowing a web service only on Host A in Figure 10, NAT knows that all the web traffic is intended for Host A but this is not a scalable solution.

Port Address Translation: Another variant of NAT uses IP address in conjunction with port numbers for creating NAT binding and it is called NAPT or NAT overload. Similar to the dynamic address binding, it creates a many-to-one mapping allowing several private hosts to be represented using one IP address or a small pool of IP addresses. The binding it creates includes the source port of the NATed host as well as the port number used by NAT for forwarding the traffic of a particular host. To avoid the routing problem experienced when using Dynamic NAT, no two hosts are assigned the same NAT port. NAPT is the most commonly used NAT variant as it requires only a single IP address for translation and is easily scalable thus it works best for small organizations having limited public IP addresses.

One reason for the wide adoption of NAT is the flexibility it offers in establishing an end-to-end connectivity without requiring any modifications by the end hosts. NAT middlebox itself is responsible for creating and storing the IP and port mapping without the higher protocols having to know about it. Hence, NAT is often used with a firewall that hides the identity of clients and protects them from various attacks by only allowing legitimate traffic to pass through and blocking any traffic that does not match the destination specified in the NAT look-up table. Application layer gateways also work with NAT for packet filtering to check for malicious payload.

NAT has limitations, especially when it comes to establishing bi-directional connectivity. Additional support is required in order to allow hosts in public realm to initiate a connection towards hosts behind a NAT network. Some techniques were proposed to solve the NAT reachability problem called NAT traversal techniques. A few of these techniques are explained briefly.

Session Traversal Utilities for NAT (STUN): It is a protocol that follows the client-server model for its working and is specified in RFC 5389 [6]. STUN itself doesn't solve the NAT traversal problem but can be considered as a tool used by other protocols to alleviate the connectivity problem. It helps the end hosts in identifying if they are protected using NAT as well as the NAT outbound IP address and port using which they can communicate with the other hosts. For STUN to work properly, at least one STUN server having two IP addresses is required with which the NAT private hosts communicate to exchange information about the NAT policies implemented at both ends. On learning the information from the STUN server, the two ends can pass this information using an application layer protocol to the other end.

Traversal Using Relays around NAT (TURN): TURN is a relay protocol that was created as an extension to STUN. It uses a TURN server in the public domain for forwarding the TCP or UDP packets between hosts. The clients behind the NAT need to create a session with the TURN server first, specifying the intended

destination in order to obtain a public IP address and port that is reserved for that particular client. The destination would then forward all the communication to the TURN server which relays it to the correct client behind a NAT server. However it is a resource intensive process and adds delay in forwarding the packets to the end-host. Also any problem encountered by TURN server could break the entire communication.

Interactive Connectivity Establishment (ICE): ICE is another NAT traversal technique developed by incorporating the functionality of STUN and TURN protocols. ICE ensures peer-to-peer communication by obtaining information from different point-to-point links about the addresses reachable by the remote host and then establishing a connection between the ICE client and remote host using the best route after performing connectivity checks using STUN and TURN protocol. However ICE adds considerable delay in setting up a session.

Universal Plug and Play (UPnP): It is a set of methods that allow devices on different networks to discover each other and then using various services exchange data with each other. As a solution to NAT traversal, Internet Gateway Device Protocol (IGDP) was developed using UPnP so devices can communicate with routers or other gateways which can dynamically open ports for creating sessions between public host and NAT devices. It doesn't use any authentication mechanisms and is thus not secure.

Port Control Protocol (PCP): It is a protocol which enables the end-hosts behind a NAT or firewall, to determine how the translation and forwarding of incoming IP packets takes place[23]. It operates by creating static mapping between the external IP address, port number and protocol and the internal IP address, port number and protocol being used and informing the remote client about the mapping. It is similar to NAT hole punching though it does not require applications to send keep-alive messages to check if the connection is still alive. It has a fixed life span associated with each mapping. It acts as a successor to NAT Port Mapping Protocol (NAT-PMP).

NAT hole punching: It is a technique that allow hosts behind the NAT device to communicate with each other using a third-party server. The server stores information about the session as well as the NAT mapping including the private and public IP address used by the client. This information is then relayed to the other client as well for establishing a successful connection. Since a valid port is used for communication, firewall policies allow the traffic to pass through. It works for TCP and ICMP protocols but it is most commonly used for UDP.

IETF recommends STUN, TURN and ICE protocols to be used for solving NAT traversal problem. All of these solutions have limitations and are not ideal for mobile network devices. These solutions require changes in the Mobile communication applications for the support of these protocols thus adding complexity to the application's design. To keep the binding active, these protocols require the mobile devices to

send keep-alive messages thereby reducing their battery life. Moreover, additional overhead is added in the session establishment thereby making the communication undesirable. Using ALG with NAT is another solution proposed to solve the NAT traversal problem but it also has its shortcomings as discussed in more detail in Section 2.3.

2.2.1 Application Layer NAT

In a traditional NAT, address translation takes place on the IP network layer. In case of NAPT, UDP/TCP layers are also modified by the NAT gateway to ensure the packets have the correct transport headers. Assembling of the fragmented data packets might also be needed along with IP translation. This translation is transparent for the end devices. However, not all applications are designed to work with NAT transparently and might break especially those in which the the IP and port information is encrypted in the payload. Examples of such protocols include File Transfer Protocol (FTP), Session Initiation Protocol (SIP) and audio-visual protocols that follow H.323 to name a few.

Application layer protocols, especially that use authentication and encryption mechanisms, need modification in traditional NAT to provide end-to-end connectivity. One solution is to modify the communication protocols to make them NAT-friendly as exemplified by [24]. This approach is not feasible as it requires changing the standards defined for all the application protocols that are incompatible with NAT. Another solution is to use NAT with some ALG that monitors the incoming traffic and creates dynamic port mappings. This approach of using NAT with ALG is discussed in the next section. Another method involves modifying NAT middlebox to detect the IP and port information from the payload of the incoming packet without the support of ALG and forwarding it to the correct end host, maintaining transparency. Such an implementation of NAT that monitors the application layer traffic for detecting IP and port information is called Application Layer NAT.

An application layer NAT works essentially in a manner similar to a traditional NAT though it additionally requires a SYN proxy for functioning properly, which is a TCP proxy that filters the incoming connections for any potential SYN packet flooding attacks. Like a proxy server, the SYN proxy acts as a gateway and is responsible for relaying all the incoming and outgoing traffic to and from the endpoint. AL-NAT requires a SYN proxy because it needs to establish a connection between the client before it can establish a connection with the server. It requires a TCP or a higher level proxy for forwarding of the packets in a synchronized manner.

A NAT equipped with SYN proxy is shown in Figure 11. When a public client attempts to connect to a private host behind a NAT, a TCP 3-way handshake is first completed with SYN proxy. The SYN proxy is embedded in an application layer NAT to assist in its operation however it is a separate software component. NAT uses its public IP address for sending a response to the client and thus acts

as an endpoint in the initial stages of the connection. Application data sent by the client after successful establishment of a connection is buffered by the NAT and is used for detecting the information about the IP address and port of the destined private host for correctly forwarding it. On detecting the information from the payload necessary for creating a successful connection towards the private server, the application layer sends a modified SYN to the private host having a source IP address of the public client and stores this mapping in the NAT table. If the private host does not have any active service running it can response with a Reset flag in which case the same packet should be forwarded to the public client after TCP header modification. Application layer NAT then forwards the buffered application data to the requested server and any subsequent communication between the server and the client takes place transparently.

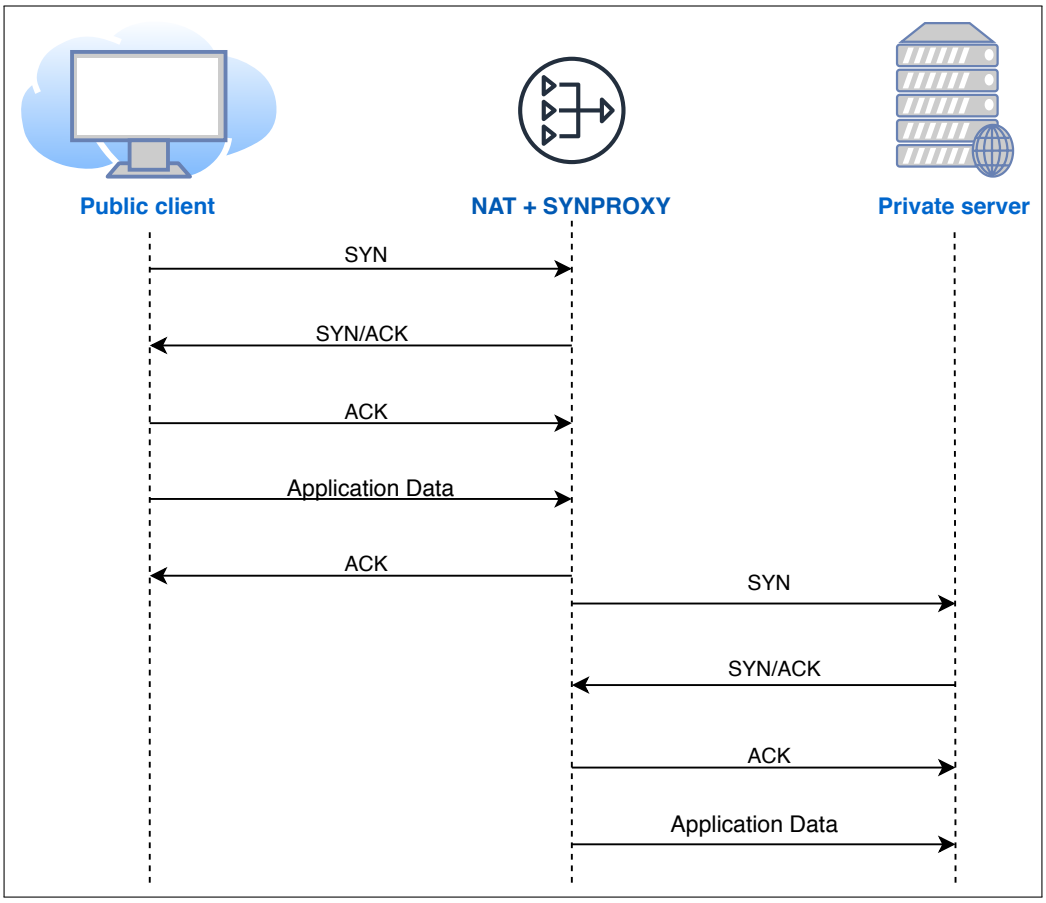

Figure 11: Connection establishment in SYN proxy

References to Application Layer Gateway can be found in the literature and some developed ALG solutions exist but Application Layer NAT is a novel concept that has not been explored. An implementation of a Linux user-space application layer NAT that works on top of TCP is available on github and it is called ldpairwall [25]. This application layer NAT does not enforce a security policy and must be used

in conjunction with a firewall for added security. It is currently implemented in C language.

2.3 Application Layer Gateway

Application Layer Gateway (ALG) can be defined as a software component used for monitoring application payload for making various application protocols compliant with NAT in addition to performing various other functions. It is also used for allowing an IPv4 network to communicate with an IPv6 network using IP translation as mentioned in RFC 2766 [26]. Based on the application payload, it can either create a new dynamic binding rule for allowing the incoming packets to traverse through the NAT to the destined private host or modify the application data stream so that it can be passed without any restrictions. In a NAT ALG, the translation of addresses is done on application layer unlike a traditional NAT that works on network layer. In order to maintain end-to-end connectivity, an ALG appears as the end-point for the hosts in the public domain and after inspection and modification of the incoming requests, it forwards the packets to the destined host thus acting as a proxy server and hence ALGs are also alternatively known as Application Level Proxy. There are some application layer protocols that require the support of an ALG in the presence of a NAT device, such as FTP, SIP, H.323 and Simple Network Management Protocol (SNMP), with FTP being the protocol for which ALG support is most commonly added as addressed in RFC 2428 [27].

FTP uses control connections and data connections for successfully transferring the file from the FTP server to the FTP clients. The information about the address, port and parameters for the data session are conveyed using the control session. After the FTP server receives the correct information, file transfer is initiated. FTP ALG has two modes of operation, active mode and passive mode. In the active mode of operation, the ALG monitors the application data stream for the PORT command to identify the IP address and port number used for establishing a connection by the FTP server. Whereas in the passive mode of operation, the response sent by the server is examined to look for the PASV command containing the IP address and port information. NAT then translates this private IP address to a publically routable address to which the client connects as indicated by Figure 12. The operating system TCP stack would be responsible for adjusting the sequence numbers and acknowledgment numbers for smooth communication. Once the session has been completed, the firewall policy rule created by the NAT firewall is removed.

The technique of deep packet inspection (DPI) is employed by an ALG for seamless transfer of data between different networks. This technique on the one hand is a very effective way of detecting the protocol and analyzing the application stream but simultaneously adds a lot of complexity to the software design. ALGs also tend to be protocol-specific, which in essence means that a custom ALG has to be designed for each protocol. This allows ALG to handle any application-specific traffic but it can be problematic in some cases where for example, ALG encounters an application header

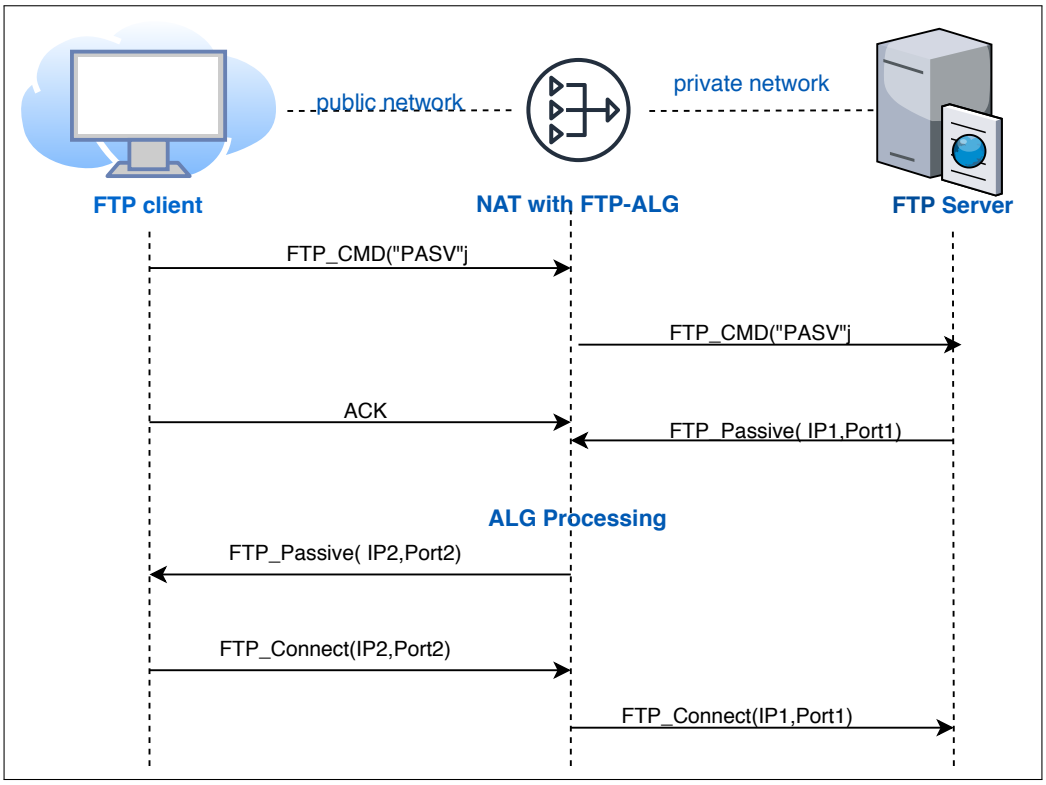

Figure 12: Passive mode of communication in FTP ALG

it was not designed to handle. Hence, changes in a specific application protocol might require similar changes to be reflected in the ALG designed for that protocol inducing additional monetary and computational costs.

ALG imposes a restriction that the application payload must not be encrypted in order for it to be analyzed by the ALG. For example, if the data is secured using IPSec (Internet Protocol Security) then allowing the traffic to pass through a NAT ALG is not possible. This is because IPSec uses some inherent cryptographic algorithms using which the receiving end of the IPSec tunnel can detect if the application data has been tampered. IPSec can only be used with a NAT if the gateway acts as the end-point of the tunnel [28]. In case the application data is encrypted, then there should be some mechanism using which an ALG can decrypt the traffic. For example, HTTPS works on top of TLS thus NAT gateway would require the certificate and keys of the private hosts it serves in order for correctly routing the traffic to the host in the internal network. This imposes a security concern for the private networks, especially if the NAT is operated by third parties like ISPs.

Commercial-use ALGs either have a built-in firewall or are always used with a firewall along with a NAT. This combination of software components makes the system resilient against many Distributed Denial-of-Service (DDoS) attacks. Windows operating system has an ALG service available and also supports third-party plugins. Linux uses the Netfilter framework to implement various ALG. A number of

vendors have developed Application level firewalls; Palo Alto Networks, F5 Networks, Imperva and Juniper to name a few. However most of these are proprietary softwares designed for custom application protocols or use cases.

A very famous web proxy that is open-source and used commercially is called Squid [29]. It offers many filtering capabilities for HTTP messages but for HTTPS it either requires the use of server certificates for establishing the connection to the correct server or uses the HTTP CONNECT method to first establish a secure HTTP tunnel between the client and server to relay the encrypted traffic. The use of CONNECT Method requires first sending a request using the CONNECT header in the HTTP message and then switching to a secure connection. This can be done for outbound connections from the private hosts but requires client-side modifications for establishing inbound HTTPS connections. One example of such client-side library for solving the NAT reachability problem is called Carrier grade HTTP CONNECT proxy client (CG-HCPCLI) [30]. Prototype of an ALG that works with TLS protocol is available on Gitlab [31] written using Rust programming language. It resembles in some aspects to the custom ALG designed during this thesis.

2.4 Policy Management System

Internet technologies have seen a great deal of advancement in the last few decades. This technological sophistication comes at a price of security threats and attacks becoming more advanced with each passing day. Administrators of networks need a set of defined rules for protecting their systems and ensuring the correct access control mechanisms are in place. For this purpose, a framework is needed to define all the rules in a comprehensive manner. Before explaining the policy management system and its functionalities it is essential to understand the terms associated with a policy and the requirements of a policy management system as specified by IETF.

2.4.1 Overview of Policy

Defining policy

The meaning of policy varies slightly according to the field or area it is being used in. According to Merriam-Webster, policy can be defined as a set of actions taken to determine the future decision in view of one's interests [32]. In the domain of networking, policy is described as a set of rules that govern and control how the network resources are allocated in the most efficient way possible.

A network security policy charts out the principles that determine the type of traffic allowed through the system, users who can access the system resources and the conditions under which the policies are to be enforced. The network policies add value to the system by maintaining privacy, confidentiality and integrity of the network and directing a system's behaviour under various conditions, for example, threat detection. In essence, policies allow to tailor a generic security engine to the

environment, the traffic as well as the concrete expectations of the users and admin of the network or host.

Characteristics of a policy

A good network security policy has the following characteristics:

- Policy is reflective of the network's goals. Since a network policy aims at protecting the system, it should include information about the users having access to the system and the rights given to each user.
- The policy should be modifiable only by a selected group of people for e.g., administrator.
- Policies should be written in a manner that is easy to understand even for a simple user of the policy management system.
- A policy should be easy to configure and enforceable using tools where applicable.
- Updating one set of policies should not conflict the operation of the rest of the policies. For example, changing policies for one type of network traffic should not affect the rest of the policies.

Types of policy

IETF has broadly categorized policies into seven types depending on their usage and functionality in RFC3060 [33].

Configuration Policies: As the name implies, these policies are used to configure the system. For example, setting up forwarding rules for a specific subnet on a router.

Installation Policies: Policies that define what can be installed on a network component can be classified as installation policies. For example, installation of dependency softwares on a server can be specified using installation policies.

Motivational Policies: Policies that validate if a policy's purpose has been achieved or not can be termed as motivational policies. These policies can be further categorized into configuration and installation policies.

Usage Policies: These policies define the guidelines about how different network components should behave based on some usage data. For example, upgrading the forwarding policies of a router for a privileged user.

Error and Event Policies: These policies are triggered in case of a certain event or an error. For example, restoring a system from the last backup point after system failure.

Service Policies: Service policies are used for identifying different services available in the network. Usage policies are then used to implement the action that is needed in the presence of a certain service policy. For example all backup routers would forward the updated routing table in a round-robbin manner.

Security Policies: These policies are used for implementing security mechanisms in the system. For example policies that define access control to various resources are part of the security policy group.

2.4.2 IETF Requirements for a Policy Management System

IETF in collaboration with Distributed Management Task Force (DMTF), a group that creates open standards for IT infrastructures, came up with a model for a policy-management system that can be applied to different types of policies [33]. The model is formed using four basic blocks as illustrated in Figure 13.

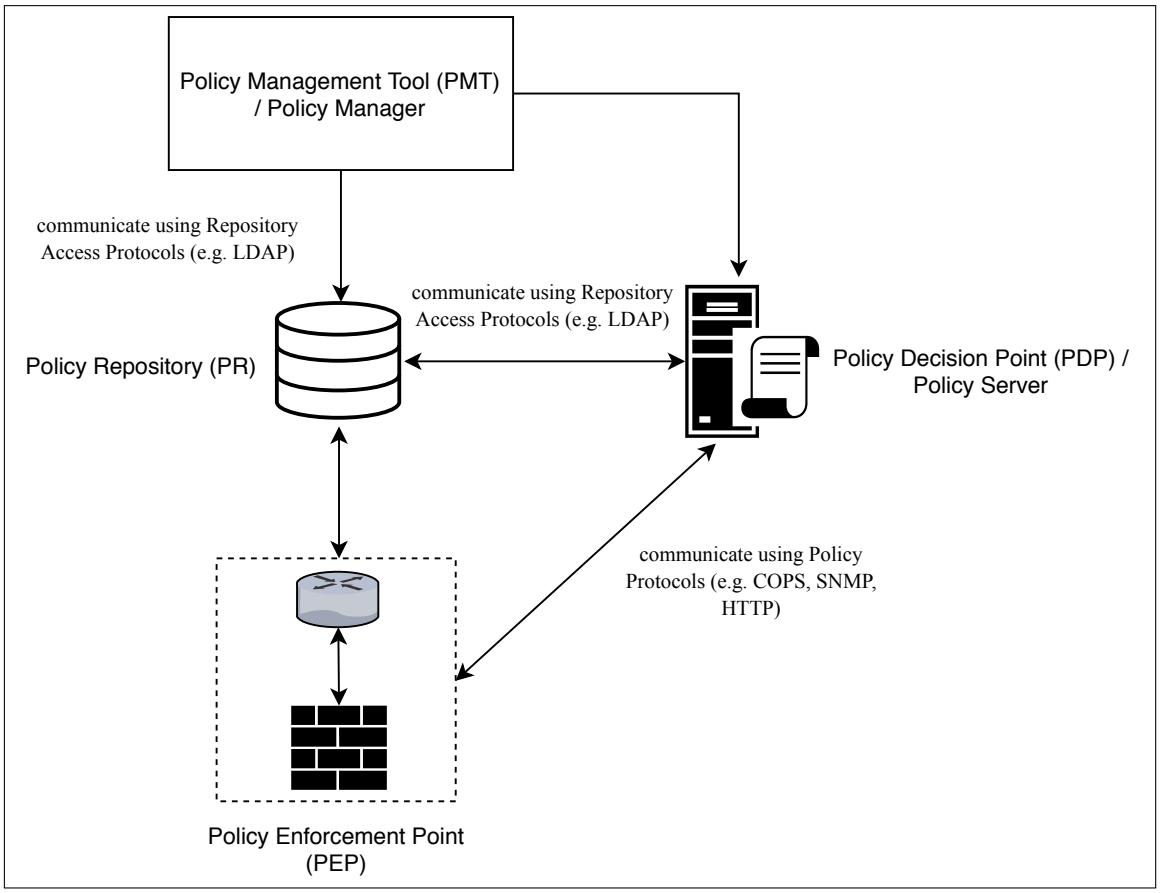

Figure 13: Components of a Policy Management System

Policy Management Tool: A policy management tool (PMT) is a software component used by the administrator for accessing and modifying the policies that need

to be implemented in the network.

Policy Repository: Policy repository (PR) is the repository or database for storing all the policies. These policies are to be stored in a standard model, inter-operable between different network components. MySQL database is often used for storing the policies.

Policy Decision Point: Policy decision point (PDP) is a network node responsible for retrieving the policies from PR. Additionally, PDP also notifies the Policy Enforcement Point (PEP) about any modifications in the PR and conveys them to PEP. It is responsible for taking decision against policies.

Policy Enforcement Point: Policy Enforcement Point is the network device which is responsible for enforcing the policy specified by PDP. Firewalls are example of network components that can act as PEPs.

These components need some standard protocols to communicate with one another. A general flow of policy enforcement in a system using a reactive approach would first involve the PEP such a router, to enforce some policy when it encounters certain type of traffic and then it would send a request to the policy server or PDP. The policy server would then retrieve the particular policy from the repository based on the characteristics specified by the router and take the final decision of allowing or disallowing the incoming traffic. Most firewalls that acts as PEP however use the proactive approach where the policies to be enforced are conveyed to the PEP before the start of its operation. Using a proactive approach is beneficial as it does not have a single central point of failure. The ALG designed in the thesis uses a proactive approach for enforcing the policies.

2.4.3 Existing Policy Management Systems

Various Policy Based Management Systems (PBMS) have been developed over time and each serves its own purpose, tailored to the network for which it is designed. Since management of large scale distributed systems is a tedious task, these PBMS assist by automating the management procedure. Flexible policies are used which can be changed with minimal effort and adapt to the changing environments. One such system is developed by Cisco called Cisco Security Manager [34]. This software offers policy management, object management, management of various events as well as troubleshooting tools. However it's biggest drawback is that it is inter-operable with only Cisco products which is a big setback especially in the context of IoT and cyber-physical systems that would involve products from different vendors.

A Policy based Management System that works using Software Defined Networking (SDN) based on the principles of Open-Flow has been discussed in [35]. However most of these PBMS are designed keeping in mind the network operator, so end-devices don't have much control over the user policies that are defined in the policy framework.

Artificial Intelligence (AI) combined with 5G will be the enabler for the world of intelligent connectivity and the network of these intelligent devices would require policies for internal as well as external interactions. Conventional Policy management tools need to be updated to handle these smart devices, proper security policies need to be imposed to protect them against the myriad of attacks. Policy-models adopted after learning the behavior of the system over time that can self-generate optimal policies would be needed in the future. One such approach based on a generative policy-based model has been recently proposed in [36]. This model is based on Inductive Learning of Answer Set Programs (ILASP) which is an inductive approach to an already defined programming language, Answer Set Programming (ASP) used in reasoning applications. This is still a research based approach and has not been tested for different networking conditions. Also because of the complexity involved in its implementation, it may not be ideal to be managed by end hosts.

There is a need to design a policy management system that can be controlled by the customer networks, where they can define the type of flows that can be admitted to their networks. This task of policy creation should be automated, having a user-friendly interface and can be modified or managed with as little effort as possible. One such open-source PMS has been designed, called Security Policy Management (SPM)[37]. It aims at configuring both network level policies as well as user-specific policies, including the task of policy creation and validation and can be accessed remotely using a REST (Representational State Transfer) API. This framework is used in this thesis for storing and accessing policies of the RGW as well as the policies for configuring the application layer gateway developed in the thesis.

3 Realm Gateway

This chapter presents a detailed analysis on Realm Gateway, its working and design principles. A thorough knowledge of Realm Gateway is essential in understanding the extensions proposed in this thesis to Realm Gateway's architecture and how they improve the design in terms of connectivity and scalability. The chapter is divided into three main sections. Section 3.1 addresses the question why we need RGW. Section 3.2 describes the architecture of RGW, what are the various software components that collectively form RGW. The design choices are discussed in section 3.3 along with the reputation system based on which the resources are allocated to different end-hosts.

3.1 Motivation

To mitigate the problem of IPv4 address exhaustion several solutions have been proposed over the course of years, one of which is the development of a Network Address Translation (NAT). However, the adoption of NAT presents some challenges of its own, the most crucial of which is the issue of reachability in peer-to-peer communication. The end-hosts in the public realm may not be able to reach the end-hosts in the private realm if no previous address translation mapping is available in the translation table of NAT serving the private network users. Some NAT traversal techniques have emerged in this regard but each with its own limitation as discussed in Section 2.2.

A network prototype was developed by Jesus Llorente Santos as his Master thesis in 2012 for ensuring end-to-end connectivity between users [10]. The prototype was later on modified and is available as an open source software called Realm Gateway [38]. It incorporates the functionality of a firewall for filtering unwanted traffic flows and also acts as a NAT to provide security and accessibility to the private end users. For outgoing connections from the private network, it acts as a Source NAT (SNAT) while for incoming connections initiated towards the private network it serves as a Destination NAT (DNAT) thus solving the reachability problem encountered in a traditional NAT. RGW uses a pool of IP addresses which is referred as 'Circular Pool' for creating a dynamic binding in response to a DNS query sent by the client to enable bi-directional communication. This pool contains a limited number of IP addresses that are globally accessible from the Internet. After successful creation of a TCP connection, the IP address is again released to the circular pool for subsequent connections.

To make the communication more reliable, RGW maintains a reputation system for all the entities involved in communication, advocating that the end-devices should be given more control over the accepted traffic flows. The design of RGW is based on David Clark's trust-to-trust principle which implies that the end-devices should have control over the services they accept based on who they trust. RGW is designed to make the network more secure and reliable which is also a requirement for the future

5G networks.

3.2 Architecture

Realm Gateway Software is written using Python programming language and is built for Linux OS (Operating System) environments. Broadly, it is composed of four main modules a DNS server, a basic NAT, circular pool of public IP addresses and a Firewall, acting as a gateway between the public and private network space indicated by Figure 14.

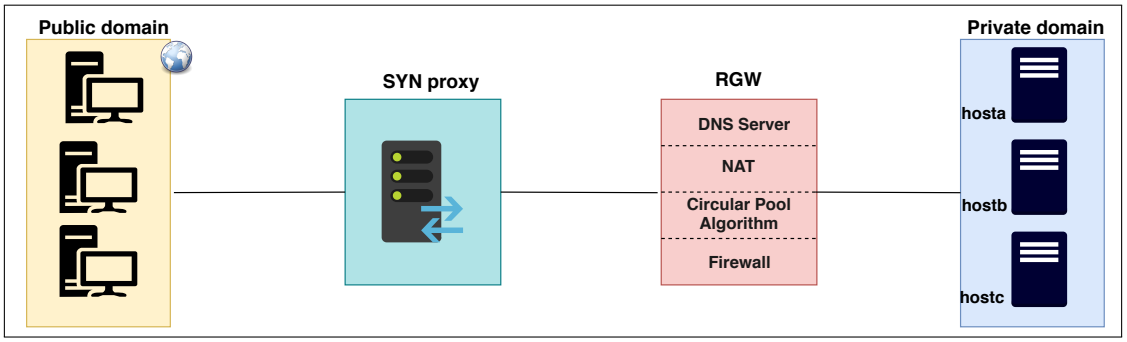

Figure 14: Components involved in the operation of RGW

Netfilter, a Linux kernel framework, is used for implementing the functionality of packet filtering and network address translation in RGW. A custom DNS server is developed using Python's DNS module and the DNS server acts as an authoritative server for the domains served by the RGW, resolving DNS queries based on its resource records. RGW also acts as a DNS forwarder for relaying DNS queries of hosts residing outside the private network to other DNS servers for resolution. Circular Pool helps in creating dynamic binding in response to the DNS query sent by the client for enabling peer-to-peer communication. It has a limited number of globally accessible IP addresses for creating TCP connections that are reserved for a duration of 2 seconds and upon establishment of a connection, the address is released back into the Circular Pool and is reused by following connections. All of these components communicate with one another and contribute to RGW functioning reliably. A SYN proxy is used with RGW which allows filtering of spoofed TCP-SYN traffic as well as rate limiting the number of new TCP connections. Two different variants of SYN proxy have been tested with RGW, one of them is a custom SYN proxy designed by Juha Matti-Tilli in Linux User Space based on netmap and L Data Plane(LDP) [39], the other SYN packet proxy is a module in Linux's netfilter framework.

To deal with HTTP connections, an additional HTTP Proxy Server is required. This is because RGW relies on the state of Circular Pool IP addresses when establishing connections. Using HTTP, a browser can initiate multiple requests for retrieving the contents of a web page. If the client does not send a new DNS query

and uses the previously allocated circular pool IP address for sending multiple HTTP requests then it could result in stalling of connections if the previously reserved IP address for the client is allocated to some new connection. The original Realm Gateway implementation has Nginx server configured to act as a reverse proxy for forwarding the HTTP messages to the correct RGW host. These components are briefly discussed for a deeper insight.

3.2.1 Netfilter

To improve network security Linux kernel makes use of the Netfilter system and implements various functions related to filtering and mangling of packets received on the IP network layer in the Linux protocol stack. Netfilter contains a number of different hooks which allow different callback functions to be executed when packets traverse through different layers of the networking stack. Table 3 indicates the five different hooks that allow modules to interact with the traffic at different points of the Linux protocol stack. These hooks then determine what needs to be done based on the properties of the packet.

Hook	Functionality
NF_IP_PRE_ROUTING	Triggered by incoming packets before any routing decision has been made for them
NF_IP_POST_ROUTING	Triggered after routing decision has been executed for outgoing traffic or traffic forwarded within the network
NF_IP_LOCAL_IN	Triggered after incoming traffic has been routed which was intended for local network
NF_IP_LOCAL_OUT	Triggered by outgoing traffic intended for the local network
NF_IP_FORWARD	Triggered when incoming traffic has been routed and it needs to be forwarded to another host

Table 3: Netfilter Hooks

The most important kernel modules that interact with netfilter hooks include ip_tables, ip6_tables, eb_tables and arp_tables. These modules interact with the network traffic using administrative tools having similar name in the userspace. Iptables and ip6tables are responsible for managing the filtering rules for IPv4 and IPv6 packets while arptables and ebtables deal with rules for ARP frames and Ethernet frames respectively. These userspace tools contain tables to enforce different operations on the traffic by using rule sets called chains. Rules define what is to be done when a packet matching a certain criteria such as source IP address, destination IP address or protocol to which the packet belongs etc. is encountered. Rules are arranged in tables on the basis of a priority value which implies that a rule having higher priority would be given precedence over other rules matching the same criteria.

Rules related to network traffic are categorized into five tables namely, raw, mangle, filter, nat and security. Filter table is the default table used when no other table has

been defined. It is used for deciding which packets can be admitted or rejected for further processing. Raw table helps in configuring packets that should be exempt by the connection tracking module in Linux. The connection tracking module if used can inspect packets that belong to a certain state of a connection, for example listening for packets belonging to a certain flow and performing some action on them. Mangle table helps in changing the packet headers to meet some requirements that affect the routing or forwarding of packets. Modifying Time To Live (TTL) value in the IP header is one example of packet mangling. NAT table modifies the source or destination IP address of the packets to allow them to traverse through a NAT network whereas security table is used for enforcing access control networking rules.

There are five default rule chains, prerouting, postrouting, input, forward and output that are triggered by the netfilter hooks when they interact with traffic at various points in the network stack. Each table has some default rule chains. Additionally, users can also define their own rule sets using custom chains. When a certain rule in a rule chain is matched, the action that needs to be executed for the packet is defined using Targets. Accepting, dropping and rejecting packets are all potential targets.

Using the iptables utility, RGW accepts and rejects traffic flows to and from the private hosts. It has a number of custom chains defined for executing different operations for example rate limiting the traffic originating from a particular private host. These iptables rules are defined in policy templates that are loaded during the initialization of RGW and can be customized.

3.2.2 DNS Server

Realm Gateway acts as a DNS resolver for domains registered in the public realm requested by private hosts in the outbound direction. The DNS query is resolved recursively by publicly accessible DNS servers and the DNS response is used by RGW for establishing a NATed connection between the private service and the user in the public domain.

Realm Gateway acts as an authoritative DNS server for the protected servers behind RGW. This essentially means that DNS query pertaining to a web domain for a server behind RGW is resolved at the end by RGW itself. For ensuring reliability and determining the authenticity of the unknown sender, RGW sends a truncated DNS response for a DNS Query sent over UDP. The client is then expected to send the DNS query using TCP. RGW uses the CNAME challenge for resolving the requested domain name to an IP address. In the CNAME challenge, the initial DNS response sent by the RGW contains a CNAME or canonical name for the domain rather than the actual IP address using which the domain can be accessed. The client is sent the actual IP address for the domain when it sends a DNS query using the specified CNAME domain. The cname challenge sent by RGW is unique for every DNS query. It should be noted that all these phases related to the DNS operation of RGW are

specified using policies.

DNS response is sent containing one of the available IP addresses in the circular pool. This method of choosing one of the available public IP address from the circular pool makes it difficult to guess which IP address will be selected by RGW next for sending the DNS response and carry out a DoS attack. TTL value is set to zero in the DNS response to prevent the DNS servers from caching the response and thus sending a new DNS query for every new connection. However, a vulnerability in the current implementation is that some DNS servers use the cached DNS response for generating new DNS queries despite RGW setting a TTL value of zero in DNS response. This results in the same CNAME challenge being generated for subsequent connections from the same public client.

On receiving the TCP SYN packet from the public client, the public IP address is released back to the circular pool for establishing further connections. For added security, RGW is used in conjunction with a SYN proxy that limits the incoming SYN packets, eliminating SYN spoofing. SYN proxy is important for RGW's operation as it prevents the attackers sending spoofed SYN packets from stealing the circular pool state essential for establishing new connections.

3.2.3 Circular Pool

Circular Pool Algorithm plays an essential role in the design of RGW. Public IP addresses in the circular pool are utilized in a round robin manner. When one IP address in the circular pool is reserved for an incoming connection, the IP address goes in the waiting state. After successful establishment of a TCP connection the IP address is released back to the circular pool or if no packet is received for the reserved IP address for a duration of 2 seconds, the entry in the forwarding table expires and the IP address is released back to the circular pool for assigning it to new connections.

The scalability of RGW is largely dependent on the size of IP address pool as exhaustion of IP addresses in the circular pool results in blocking of incoming connections. If no circular pool IP address becomes available for establishing a connection after the client has sent the DNS queries N times, the subsequent connections are blocked. This is true only when a service is requested using a FQDN rather than a Service FQDN (SFQDN)[2] for a service that uses the custom ALG. Having a larger pool of IP addresses can reduce the chance of blocking but simultaneously increases the address space available for carrying out attacks [9]. Thus RGW makes use of a reputation system for admitting new connections explained in the next section.

[2]The concept of SFQDN stems from the SRV DNS records and is used to identify the service requested on a particular domain using the DNS query. For example a web service on a particular host can be specified as www.host.gwa. An alternative approach combines the port number and transport protocol for defining the service such as tcp80.host.gwa.

3.3 Design Principles

RGW with its circular pool algorithm was designed to overcome the NAT reachability problem. The size of circular pool is an important factor that determines the scalability of software and it is measured by the number of private hosts being served by RGW. To limit the number of connections admitted to RGW, a reputation system is introduced. Using the reputation system, flows from the DNS servers related to new connections are admitted depending on the current load on the system and previous reputation of the associated DNS server sending DNS query. This ensures that only legitimate traffic is admitted through and priority clients have a higher chances of getting the requested services under attack conditions.

3.3.1 Reputation System

To offer better protection against attacks, RGW maintains a reputation system for all the DNS Servers sending DNS Queries for requested domains. Based on the IP address information for each DNS server, RGW classifies the DNS servers into three categories, whitelist, greylist and blacklist. DNS Servers having prior SLAs (service level agreements) belong to the whitelist category. By default if no prior reputation information is available for a DNS Server it is admitted into the greylist while if some DNS server misbehaves it is dropped into the blacklist category. Candidates of each list are treated differently under different load conditions. Under extremely high load conditions, traffic from only whitelisted DNS servers is accepted. Under normal to high load conditions, flows from greylisted DNS servers are also accepted. This reputation is associated with an aging mechanism to ensure that if some DNS Server is given a bad reputation due to a client involved in some malicious activity for instance due to hijacking, then the DNS Server should be able to regain the reputation with time. Hijacking refers to an attack where the attacker gets access to the communication between two hosts and impersonates one of them to carry out a malicious activity like a denial-of-service (DoS) attack. The dynamic method of reputation collection can also encourage the ISPs to participate in improving the security of entities involved.

4 Custom Application Layer Gateway

This chapter discusses in detail the development of a custom Application Layer Gateway. The motivation for designing custom ALG for RGW is explained in the beginning of the chapter. Next, architecture of custom ALG is described followed by the design principles that aim to improve security and scalability of the software. The chapter further entails an analysis on how the current software implementation is integrated with RGW. Towards the end of the chapter role of policy database in storing the policies of RGW and custom ALG is explained along with the modifications made to RGW's code for seamless communication with security policy database.

4.1 Motivation

Realm Gateway is developed as a solution to overcome the NAT reachability problem [9][38]. It was revealed during the initial testing of RGW that it faced compatibility issues when dealing with HTTP and HTTPS connections rendering clients in the public domain unable to properly connect to web servers located behind RGW. When a client accesses a web page using HTTP/1.0 it initiates multiple HTTP connection requests to retrieve all the different embedded content such as videos and images. Some web browsers might retrieve the web page using multiple requests even in HTTP/1.1. If the client requests to access the web page using HTTPS during the same session a new connection request is sent. The compatibility issue originates if a new DNS query is not sent for each subsequent HTTP request with a different source port as the associated circular pool IP address allocated previously during the DNS response is released to the circular pool and can be reallocated for some new connection. This problem stems from the fact that RGW relies on DNS queries for maintaining the connection state and forwarding the data between public client and the web server. Hence the compatibility issue is attributed to the circular pool algorithm of RGW which relies on domain names for maintaining the correct connection states.

To overcome the challenges involved in establishing HTTP and HTTPS connections RGW is used with a reverse HTTP Proxy Server. A reverse proxy server is a type of proxy server responsible for forwarding the client requests to the correct server behind the NAT or firewall. If the initial DNS query sent by the client requests the IP address of the web server using SFQDN, a more specific address binding is created by RGW in which in addition to the source IP address, the service requested can be identified based on the standard service port. Utilizing the iptables a static mapping is created whereby all web traffic destined for the standard HTTP/HTTPS ports is forwarded to a proxy server internal to the RGW which is responsible for forwarding the connection to the correct web server and also relaying the response. In case the service is not specified in the initial DNS query, a temporary allocation of IP address for a particular domain is carried out using the circular pool. On receiving a TCP packet for a destination port reserved for HTTP/HTTPS web servers, the IP address is released to the circular pool and all subsequent traffic for the connection

is handled directly by the proxy server.
Realm Gateway uses NGINX as a reverse proxy for handling web traffic. For establishing an HTTPS connection, NGINX requires the server certificates and private server keys during the initial TLS handshake phase to determine the IP address of the web server for which the traffic is intended. NGINX encounters a problem when different domains are hosted on the same IP address. Some solutions have been proposed but each has its drawbacks [40]. Also, NGINX decrypts the HTTPS traffic and re-encrypts it again to make routing decisions before sending to the back-end server. This raises a security concern as the proxy server can be operated by third parties that can gain access to sensitive data when it is decrypted. Additionally, the decryption and re-encryption increase the connection establishment time in comparison to a simple HTTP connection. To overcome the shortcomings experienced by NGINX reverse proxy, a custom ALG is designed that is inter-operable with RGW software.

4.2 Proposed Architecture

ALG separates the network into two distinct regions, a public domain consisting of clients requesting various services and a private domain composed of the servers sitting behind a firewall/NAT. The custom ALG is designed to work with HTTP and HTTPS protocol and it functions as an intermediary for the public client accessing web services on the web server sitting behind a firewall/NAT. It ensures transparency by assuming the role of a server for the public clients, forwarding HTTP responses received by the back-end servers, while appearing as a client to the web servers by forwarding the HTTP requests sent by the original clients in the public domain as shown in Figure 15. Consequently, the end-clients are not aware of the presence of ALG. It has a multi-process, event-driven architecture with one master process and multiple child processes responsible for establishing the connections and the specification of multi-process approach is discussed in subsection 4.2.1.
In addition to the logical unit that executes the code for ALG, it has three main components that ensure its smooth operation, a storage unit, a custom parser-lexer, and a log management module. The storage unit contains information about the domains hosted on the web servers sitting behind ALG and also includes the details related to some configuration parameters of ALG. Custom parser-lexer helps in identifying the hostname after the protocol detection and is thus responsible for forwarding the data to and from the correct web server. Log management module handles the recording of important information about the end hosts besides reporting errors encountered during the operation of ALG.

Storage unit contains information about the domains hosted by web servers and the corresponding IP address and port number on which each domain operates. It also includes the list of users allowed to run ALG software as well as the list of TCP ports on which ALG listens for incoming connections. Furthermore, the storage unit entails information about the duration after which the client has to re-initiate a TCP connection with the ALG if the parser-lexer cannot detect a valid hostname. All this

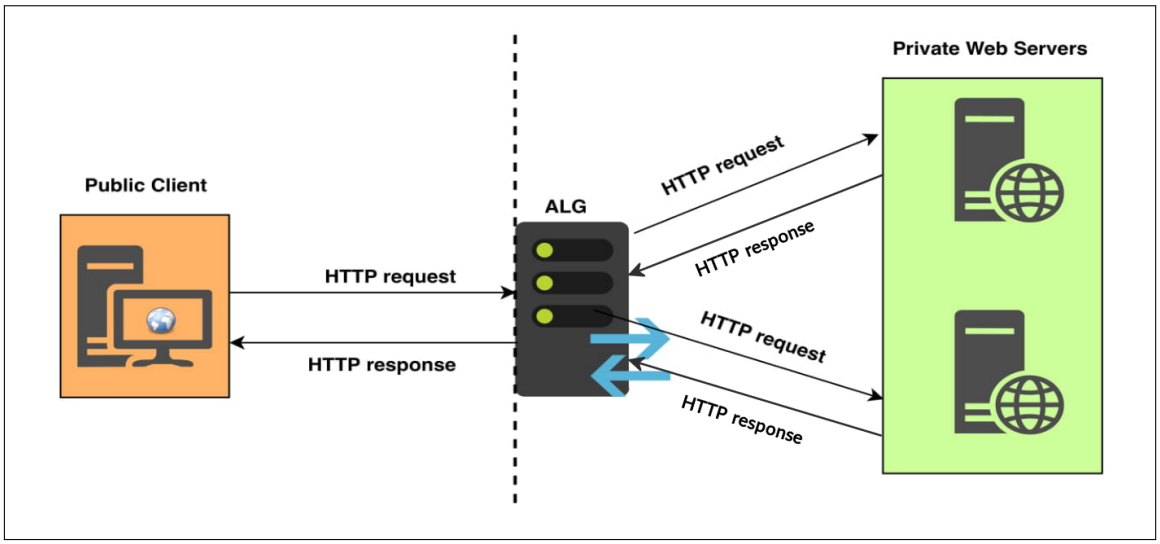

Figure 15: ALG architecture

information can be stored in a remote database or locally in a file on the hard disk of the system running ALG. In the current design, all this information is retrieved from a remote database and stored in a configuration file on the local system. If the lookup table, containing the mapping of the hostname to their respective IP addresses and ports, is updated or any new hostname is registered in the remote database the modifications are also reflected in the local file as described Section 4.5.

ALG uses a custom parser-lexer for detecting the hostname from the first HTTP or HTTPS message sent by the client. On successfully identifying the hostname, ALG validates if it knows the requested domain and looks for the corresponding IP address and port number by checking against the configuration file. ALG then forwards the connection to the correct web server. Parser-lexer is used only in the HTTP/HTTPS connection establishment phase and ALG's logical unit directly handles all subsequent application data.

Another main component of ALG is the log management module used for recording information related to HTTP/HTTPS session such as Public client's IP address, source port, requested web domain and application layer protocol for creating the connection. Any errors encountered during the operation of ALG are also logged that can help the operator in troubleshooting or fixing software bugs. Log management module allows the system user to specify the verbosity level using which they can log different information. ALG supports four different logging levels namely debug, info, warning, and error. Increasing the logging level gives the user a more detailed description of the working of ALG at the expanse of increasing the time taken for writing to the logging file.

All the components that form ALG work in a synchronized manner for its successful operation. ALG has been designed to work with RGW and thus it relies

on RGW's custom DNS Server for responding to the DNS Queries for the domains served by RGW. If using ALG as a stand-alone component an authoritative DNS server would also be required.

4.2.1 Connection Establishment

ALG is written in Python and is designed for Linux OS environments. It relies on the operating system's TCP/IP stack for handling incoming connections. The TCP/IP stack uses socket objects for creating new connections and exchanging data between client and server. ALG supports a master-slave architecture which achieves concurrency by using multiple processes. A process is defined as an instance of an application running in the memory of an operating system. The current solution splits end-to-end connection into two separate connections which are referred to as connection-halves. Figure 16 illustrates a high-level abstraction of ALG's process model where each child process handles communication in one direction of the HTTP/HTTPS connection. Linux kernel manages the two processes belonging to a connection independently.

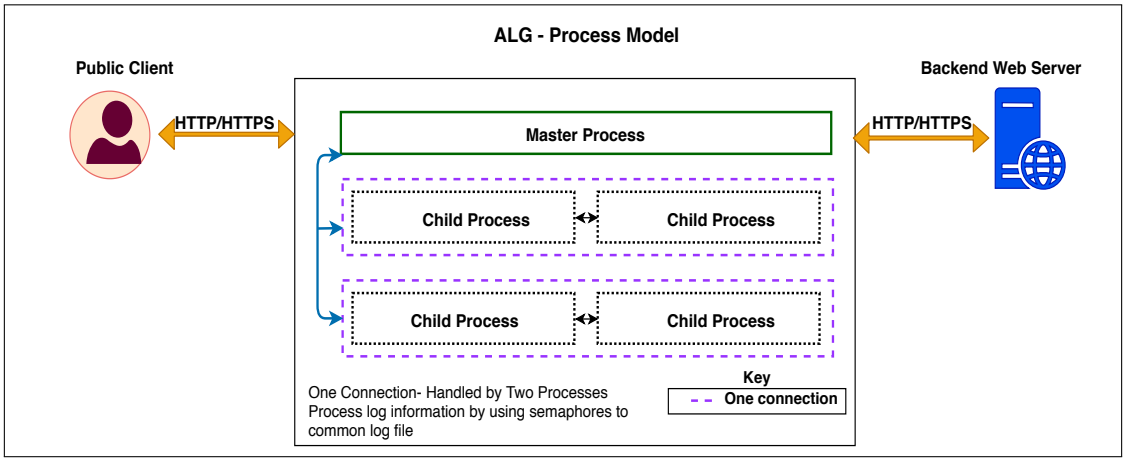

Figure 16: ALG Process Model

The master process performs privileged operations requiring root permissions, such as the opening of listening sockets on standard ports for HTTP/HTTPS, reading the values of configuration parameters from the configuration file and limiting the maximum number of simultaneous connections. The event-driven functionality of ALG's architecture is associated with the select module of the OS. The master process accepts new incoming connections from the clients using select module which is based on the non-blocking I/O model. In a non-blocking I/O model, a program can continue performing other tasks while one function is waiting for an input/output operation to complete. This allows the master process to execute the rest of the program code until a new client connection becomes available. Figure 17 presents the steps involved in the connection establishment using ALG. After accepting the new connection, the master process executes a system call to fork a child process. The child process is then responsible for the hostname detection and establishing

the other half of the connection with the back-end server. The child process forks another child process for handling the other direction of the connection.

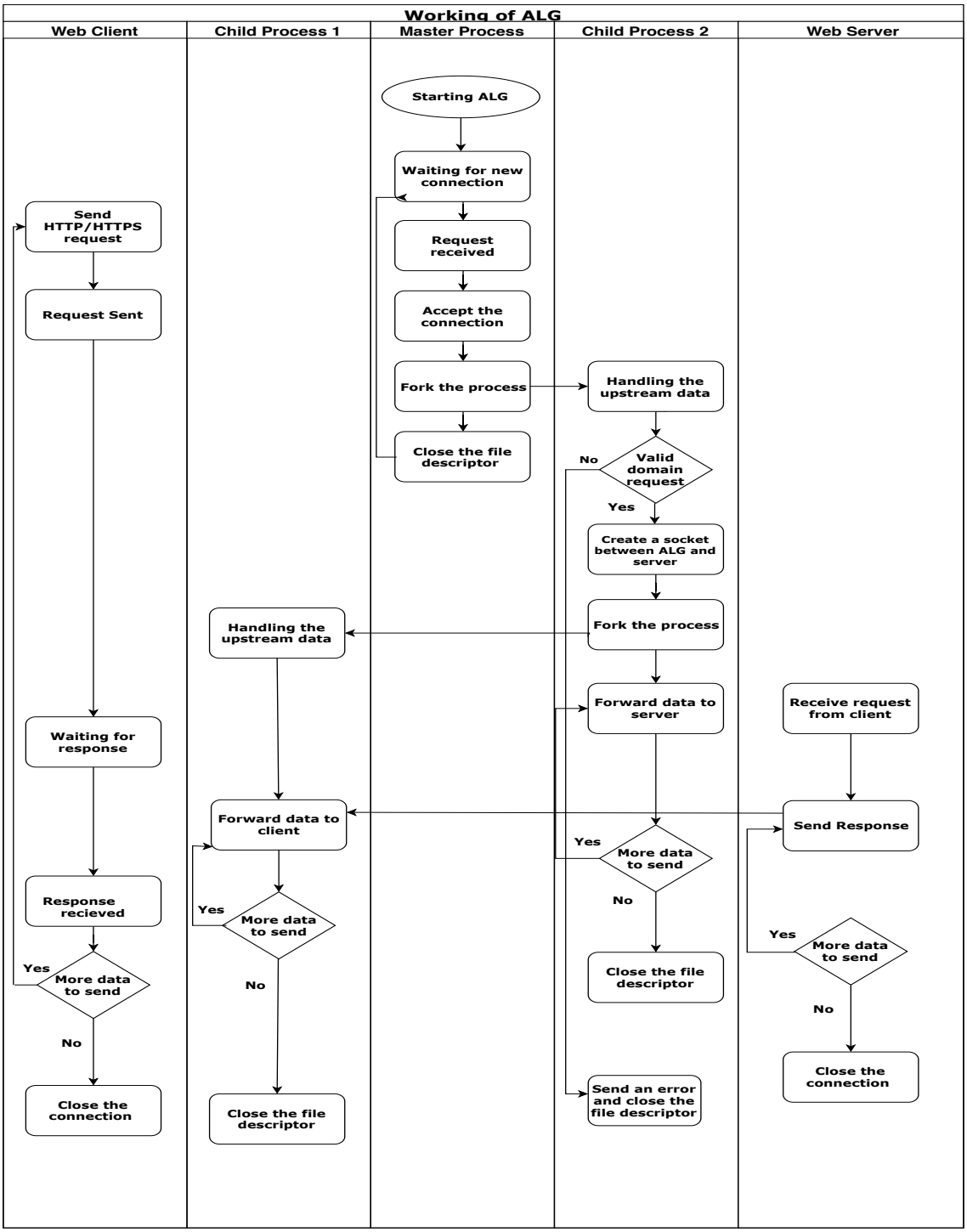

Figure 17: Flow Diagram of connection establishment in ALG

The forked child process constitutes the accepted HTTP/HTTPS connection socket from the client. The master process then delegates the forked connection to the child process, closing its copy of client connection and waits for accepting new client connections. The child process is then responsible for establishing the other half of the connection between ALG and the server. Child process having information about both ends of the connection forks another process. Hence, one process is responsible for forwarding data sent by the server while the other process transmits the data sent by the client. ALG is inter-operable with IPv4 and IPv6 networks.

Each connection is handled by two different processes and consequently, the total process instances initialized by ALG are equivalent to twice the number of incoming client connections. Logging in a multi-process environment is attained using operation system semaphores. Semaphores are used for synchronizing the operation of various processes when they are competing for the same system resources. When executing the logging functionality, any process that requires to log information acquires the semaphore and checks the size of the log file. If it matches the global log file size limit it is an indication that the log needs to be rotated. Once logging is complete, the file is available for other processes to access.

4.2.2 Lexers and Parsers

A compiler is a software that translates human-readable code written in a high-level programming language to a machine-readable code. This requires analyzing the source code. Analysis of source code can be broadly divided into two parts namely lexical analysis and syntax analysis.

Lexical analysis, also known as lexing or scanning, is carried by the lexer, a program designed to read characters from the input and convert them into meaningful tokens known as lexemes. Each lexer has it's own pre-defined lexical grammar using which it identifies the sequence of characters resembling a token. Lexing can be thought of as the first task carried out by a compiler. Syntax analysis alternatively known as parsing is the technique that determines the relationship between the input strings and evaluates if the input is following the rules of a formal grammar. The relationship can be indicated using data structures such as parse trees. Parsing is most often preceded by lexing whereby the input to the parser generator is the stream of tokens created by the lexer.

Application Layer Gateways often employ the DPI technique for analyzing the contents of application payload before routing it to the correct host. Deep packet inspection is resource intensive and complex. ALG designed in this thesis has to parse the HTTP/HTTPS packet payload only to detect the protocol and resolve the hostname of the web server requested by the remote client which is essential in maintaining the correct connection state and forwarding the application data. Packet parsing is considered the first step in the HTTP/HTTPS connection establishment phase after a TCP connection has been completed between ALG and the remote

client. For detecting the hostname, ALG relies on a tool known as YaLe developed by Juha-Matti Tilli [21].

YaLe combines the functionality of a lexer and parser into one tool. It is designed for parsing network protocols in a non-blocking, event-driven manner making it more efficient than the existing network protocol parsers and lexers. It is based on a state-machine architecture where the lexer is dependent on parser state for its operation. Parser and lexer are integrated into one tool which uses a pre-defined context grammar. The source code of YaLe is written using C language and the source code is compiled and linked to a shared object loadable as a separate Python module in the ALG.

The lexer in YaLe implements the longest-match lexing using a deterministic finite state machine (DFSM) which depends on the parser's state. In the longest-match lexing, alternatively called maximal munch lexing, the lexer generator constructs tokens using the maximum possible characters it can interpret matching the grammar rules based on the input stream. It uses bounded statically allocated backtrack buffers to execute lexing. The parser is a table-driven LL(1) parser that uses a statically allocated parser stack. Whenever new data arrives, the parser calls functions that match the input into associated terminal/non-terminal symbols to which the grammar rules can be applied, generating the output.

We have designed the ALG to work with HTTP protocol and HTTPS protocol but it can work with any application layer protocol that operates on top of TLS supporting SNI extension. In an HTTP connection, YaLe parses the HTTP payload to look for the host header. It determines the hostname from the host header based on the HTTP [3] grammar that was specified during the implementation of the tool. YaLe breaks down the HTTP request into tokens like request line, request headers followed by an end-of-line sequence. There are separate rules defined for each of these tokens and an in-built function of YaLe is called to return the hostname to a system call of ALG.

In a similar manner, when an HTTPS message is encountered by the ALG, after the correct protocol detection it is fed to YaLe for identifying the hostname using an ALG system call. YaLe looks for the SNI extension in the TLS message sent by the client during the TLS handshake phase to determine the hostname requested by the client. YaLe detects the hostname from the TLS message based on TLS [4] grammar. The job of YaLe is complete after the hostname detection and all subsequent web traffic for that connection is handled by the ALG.

[3]HTTP grammar can be found in the same repository as the YaLe software. Link to the grammar file is https://github.com/Aalto5G/yale/blob/master/test/httppy.txt

[4]TLS grammar constitutes of six nested parsers. They can be found in files named as ssl1.txt to ssl6.txt in YaLe repository. Link to the first file is https://github.com/Aalto5G/yale/blob/master/test/ssl1.txt

4.3 Design Principles

A software needs to adhere to some design principles to improve its performance in terms of scalability, reliability, security or efficiency. The main objective in designing ALG is to improve the performance of RGW when dealing with HTTP and HTTPS connections and provide better support for these application layer protocols. Some key considerations were kept in mind while designing the architecture of ALG.

Integration Requirements

ALG's design cannot interfere with the working of existing network components. It must seamlessly integrate with RGW's software with minimum modification in RGW's code. The connection between the remote client and the web server behind ALG is to be established transparently. Also ALG cannot not modify the application layer protocols rather support the existing protocols.

Scalability Requirements

RGW's software is developed to overcome the address exhaustion problem of IPv4 networks. ALG can not aggravate the address depletion problem by using a large pool of IP addressees to proxy the connection from the client to server. The solution cannot limit the number of remote connections and the concurrent clients handled at a particular instance except for enforcing a security policy.

Complexity Requirements

ALG cannot increase the complexity of RGW after integration. The solution must not be slow and thus should not have a complicated design. Commercial ALG's use DPI technique that is resource intensive and modifications in the protocol require extensive changes in the commercial ALG. To overcome this problem, ALG must have a simple architecture that can support updated versions of application layer protocols without making a lot of modifications to the source code. The architecture should be easy to understand for other developers as well.

Security Requirements

The design should improve security of the end-to-end HTTP/HTTPS connections. It has to maintain trust of the end-users in the network by maintaining encryption and not decrypting network traffic in case of HTTPS connections. It should protect the system against an HTTP/HTTPS DoS attack by limiting the established connections to a reasonable threshold. Also if a connection is idle for a specified duration of time it should close the connection.

Explanation of Design Choices

We implemented a solution which is in accordance with the specified design requirements. One aspect taken into consideration while designing ALG was to ensure only configured users are able to run the software. Starting ALG script requires root permissions as it involves binding to ports below 1024. Once the listening sockets are created, ALG has a feature to drop privileges and then the ALG's program code is executed using the privileges of the configured users. This list of permitted users is loaded from a configuration file with a pre-defined set of users who can access ALG. The reason for making this design choice can be attributed to the security risk of running network daemons with administrator rights. In case the process is compromised, the user can end up having access to the full system.

ALG has a multi-process architecture which uses Python's socket objects. As RGW is written in Python, the choice of programming language is the same for ALG, allowing people having only Python skills to maintain both software components. The design involves a master process and two child processes for establishing one HTTP/HTTPS connection. The master process handles all the global operations. Updating the policies associated with registering and mapping hostname of web servers to IP addresses is also done by the master process which warrants that there would be no conflict in policies accessed by different child processes.

For implementing the rate-limiting functionality, token bucket algorithm is used. The algorithm handles new connections at a fixed rate until a connection limit is reached. When a new connection is encountered, it is checked against the available connection tokens in the bucket. If the connection limit has not been reached, the new connection is accepted and the available token count is decreased by one. However if ALG hits the maximum connection limit set by the user, all subsequent connections are dropped until new tokens become available. This prevents overloading of the back-end web servers and ALG. This is another technique to ensure that a DoS attack cannot exhaust system resources.

HTTP/HTTPS connection establishment phase between the client and ALG is crucial to ALG's operation. Only after the web server sends the first HTTP/HTTPS response to the ALG for forwarding to the respective remote client the connection is considered successful. ALG has a connection timeout that can be configured by the user before running ALG. This timeout specifies the duration for which ALG waits to receive the application data containing the requested hostname. If the client doesn't send a valid host in the application data during the connection timeout ALG closes the connection. The connection timeout in the implementation is set to 2 seconds.

TCP sockets in Python by default operate in blocking I/O mode. This essentially means that any socket call would block the execution of other functions until the operation associated with the socket call has been completed. For example if

receive call is used on a socket, then no other operation can be performed by the ALG until an error occurs or the socket receives at least one byte of data. The event-driven functionality of ALG is associated with the non-blocking mode of operation of TCP sockets. An in-built Python library is used where a socket-timeout is specified. If no operation is performed during the specified socket-timeout, the system returns from the function call and can perform other operations.

Since HTTP/HTTPS uses TCP as the underlying protocol, the keep-alive mechanism of TCP sockets is used to ensure that idle connections do not consume system resources. If a connection is not properly closed by the back-end server or the client connection is broken, ALG uses the keep-alive timeout to close the associated connections. In the current design, if ALG detects that a connection has been left idle for the specified duration, it sends keep-alive probes to the socket to wait for a response. If no response is received from the associated socket, ALG assumes that the connection is dead and closes the connection.

Another important aspect of ALG's design is the implementation of a custom log management system. A log rotation mechanism is used to protect the system against a disk space attack. Log rotation is the process by which a current log file is renamed after it exceeds the size limit and a new log file is set up for subsequent log entries. A maximum log size is defined in the configuration policy of the ALG. This ensures that logging does not consume a lot of space on the system's hard disk. Log rotation also makes it easier to analyze log data. A log level can be specified as a configuration parameter to indicate what information should be added into the log file. Increasing the log level makes debugging the problems easier at the expanse of increasing the time taken by a connection when writing to a log file.

4.3.1 Benefits

The model of ALG offers some advantages in comparison to other existing commercial ALG solutions. Most of the ALGs available for commercial use are not available as open-source software. The idea of designing ALG for RGW software is to have a software package that is available for the research community and can be improved by using iterative design methodology.

ALG's design does not involve any extensive DPI processing. It parses the application data only to detect the protocol and hostname and after the connection is established at both ends successfully, it is only responsible for forwarding the HTTP/HTTPS traffic resulting in a simple design. In comparison, commercial ALG solutions involve DPI technique that involves modifying the packet headers and other resource intensive processing.

ALG has been tested for HTTP/HTTPS connections but it can support any application protocol that uses TLS as the underlying protocol and supports SNI extension. This implies that ALG's source code would not need to be modified to support other

application layer protocols using TLS. Most commercial ALG's are designed for specific protocols and the connectivity breaks down when they encounter application data they were not designed to handle.

The most significant advantage of using the custom ALG is evident when dealing with encrypted traffic. Unlike other ALG solutions, custom ALG does not decrypt the HTTPS traffic for making routing decision to the upstream server as can be seen in NGINX reverse-proxy server. As a result, the latency of establishing an HTTPS connection is the same as an HTTP connection in the custom ALG. To forward the encrypted traffic without any modifications improves security of the system in comparison to other solutions.

As the ALG establishes the connection transparently, multiple web servers can be hosted using the same IP address and thus it supports RGW's objective of alleviating the IPv4 address exhaustion problem. ALG supports both IPv4 and IPv6 as the underlying network protocols. However, currently RGW's software does not support dual-stack functionality and can only support client-side IPv4 networks. Hence, when ALG is used in conjunction with RGW it can only handle IPv4 networks though the web-servers can operate on IPv6 networks.

4.3.2 Drawbacks

ALG is designed using Python as the programming language and Python has an inherent drawback of having a slower execution time. Though it is easier to understand and code using Python, it does not exploit the multi-core functionality of an OS. These shortcomings of Python are responsible for ALG having a slower processing time in comparison to other ALGs written using different programming languages such as C, C++ or Java.

The multi-process architecture is used for achieving concurrency in ALG as explained previously in section 4.2.1. This means that the custom ALG consumes more system resources than other application layer gateways which use threading or other techniques for handling multiple parallel connections. This design choice hinders the scalability of ALG as the number of connections it can handle is limited by the architecture of the system it runs on. Multiple processes consume memory and memory exhaustion can hinder the operation of ALG. However ALG is not primarily designed to be run on client machines and thus using better hardware can reduce the impact of this problem.

Another functionality of custom ALG is rate-limiting the number of simultaneous connections that it can manage. It uses the token-bucket algorithm for implementing the rate-limiting mechanism. However, in the current design if the number of simultaneous connections exceed the defined connection limit, all the subsequent connections are dropped until new tokens become available. No queuing mechanism has been implemented to ensure packets, belonging to new connections after the maximum

connection count has been reached, are stored in a buffer for re-transmission on the availability of new tokens. This saves the system from overloading but the client has to re-initiate the connection with ALG for connection establishment with the back-end web server.

The current design of ALG is a proof of concept and improvements can be made to the software to enhance its performance.

4.4 Integration to Realm Gateway

RGW relies on the Netfilter framework of the Linux kernel for establishing connections between the end hosts. The policies associated with packet forwarding and implementing NAT functionality are defined in the RGW and enforced using iptables. These policies allow bi-directional communication based on various network protocols. ALG is designed for HTTP and HTTPS protocol thus to integrate it with RGW, policies related to the forwarding of web traffic are added in iptables. These policies ensure that all web traffic is forwarded to ALG when it is received by Linux kernel. Since ALG is tailored to work with HTTP and HTTPS application layer protocols to make the web servers behind RGW accessible to the public internet, we only consider inbound connections that are initiated by the public clients towards the private web servers.

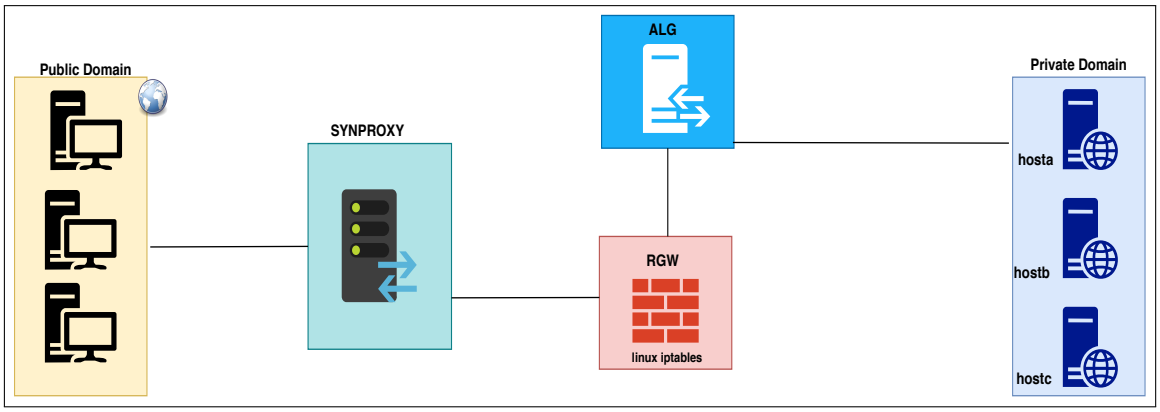

Figure 18: Network components involved in establishing HTTP/HTTPS connection using ALG

ALG is used in conjunction with various network components when used with RGW as shown in Figure 18. All the TCP traffic is first received by SYN proxy which limits the new TCP connections and also eliminates SYN spoofing. The TCP packets are then forwarded to Linux iptables having different policies enforced by RGW. The operation of ALG is concerned predominantly with the iptables though RGW has several other components indicated in Figure 14. ALG runs as a separate service on the system running RGW software. RGW relies on DNS Queries for domain name resolution of the web servers and maintaining the correct connection state. If the

web service is also specified in the initial DNS query sent by the client using SFQDN, the Linux kernel directly forwards the web traffic to ALG bypassing the circular pool. However, if the initial DNS query only contains FQDN of the private host, an IP address is reserved from the circular pool of RGW for establishing TCP connection. If the client sends HTTP/HTTPS request on standard HTTP/HTTPS service ports after the TCP 3-way handshake is complete, RGW forwards the connection to ALG and releases the reserved IP address to the circular pool for subsequent connections.

RGW operates by using three different types of policies, namely policies related to private hosts, policies related to the reputation system of remote clients and policies to enforce iptables rules. The iptables policies can be customized to handle web traffic on other ports apart from the standard port 80 and 443 used for HTTP/HTTPS service. These policies are reflected in the NAT table and filter table of the kernel. For connections having the SFQDN specified, the decision for forwarding the connection to ALG is done before it can be forwarded to the control plane of RGW. Whereas TCP connections established using RGW's circular pool and destined for specified web service ports, first the circular pool IP address is released back to the pool and then the connection is forwarded to ALG.

ALG listens on the wildcard IP address of 0.0.0.0 which signifies that it is listening for connections destined for all IP addresses available on the local machine running ALG. ALG binds to a list of ports specified using configuration policies. When used with RGW software in a containerized environment, ALG binds to all the IP addresses available to RGW. Thus when the client specifies the SFQDN in the DNS query, the TCP connection between the client and the server is established using RGW's public IP address. Using the wildcard address, ALG also listens to the IP addresses belonging to the circular pool of RGW. This allows the ALG to handle connections established using circular pool IP addresses. Policies for forwarding the connection to ALG after releasing the circular pool IP address are specified by the RGW in the nat table and filter table of Linux kernel. Application data sent by the web server to the client is forwarded by ALG using circular pool's IP address as the underlying TCP source IP address.

Preference is given for binding the sockets to the wildcard address in ALG because it has some advantages over binding to a specific IP address. If the ALG was bound only to RGW's public IP address, some additional policies would be needed for the HTTP/HTTPS connections which are established using circular pool IP address. The advantage is also evident when RGW's circular pool IP addresses are changed. In the current design any modifications in the RGW's circular pool IP addresses would not require any changes in ALG's code. Even if IP addresses are added to RGW's circular pool dynamically, ALG would be inter-operable with the modified RGW's software. Another benefit of using the wildcard address is related to the operation of RGW. If the RGW's component is restarted, ALG would still enable the client and the web server to continue exchanging application data until the HTTP/HTTPS connection is closed by either party. This ensures that the

application data for established connections is not lost even if RGW shuts down and starts again.

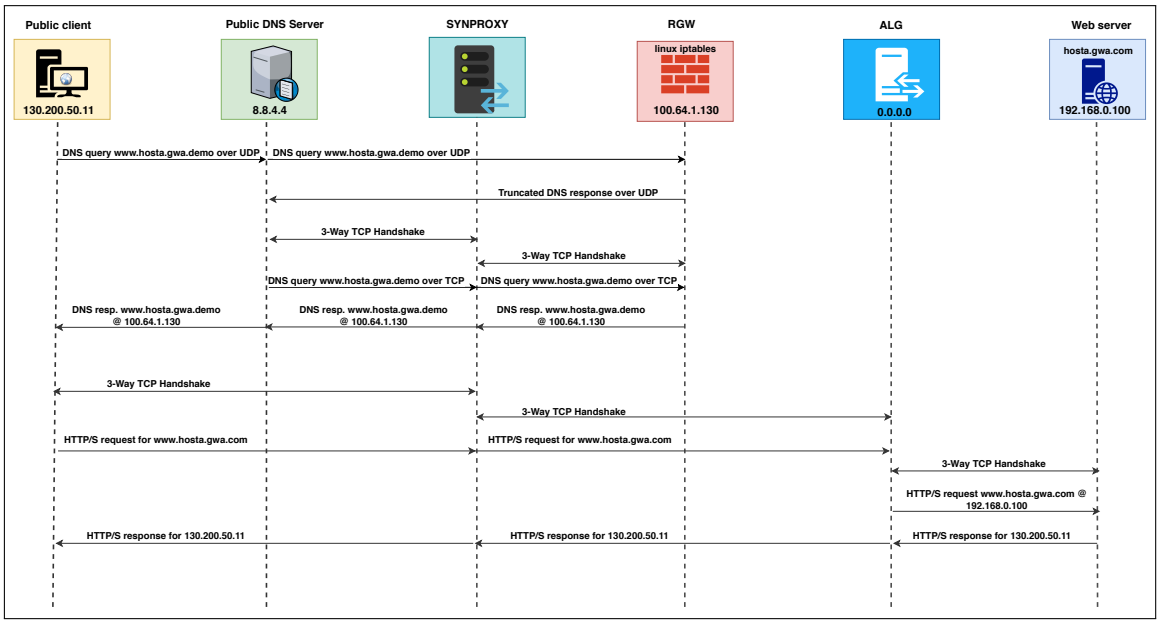

Figure 19: HTTP/HTTPS connection establishment using SFQDN in ALG integrated with RGW

Figure 19 illustrates the flow of establishing HTTP/HTTPS connection between a remote client and a web server behind RGW. When a remote client sends a DNS Query for the domain hosta.gwa.com over UDP, RGW's authoritative DNS Server sends a truncated DNS response. After the 3-way TCP handshake between the public DNS resolver and SYN proxy is successful, the SYN proxy initiates the 3-way handshake with RGW and and then forwards the DNS query sent by the public DNS resolver for the domain wwww.hosta.gwa.com. RGW responds to the DNS query indicating that the requested domain can be accessed using RGW's public IP address. When the remote client sends the application data, it is handled directly by the ALG and it is then responsible for forwarding all the web traffic between the client and the web server.

If the service is not specified in the requested FQDN by the remote client, RGW establishes the TCP connection using one of the publically accessible IP addresses of the circular pool of RGW. The circular pool IP address is chosen in a round-robin manner based on the state of the IP address. The IP address that is not marked in the waiting state can be used for connection establishment. Once the application data is received on standard HTTP/HTTPS ports, the RGW forwards the connection to ALG using the pre-defined iptables policies and the IP address is released to the circular pool for new connections. ALG then establishes the connection with the web server after detecting the hostname and forwarding the application data on the IP address belonging to the web server as represented by Figure 20. It should

be noted that iptables forward the application data to ALG in response to a DNS query containing FQDN only if the client initiates a connection using SYN packet. If some other TCP traffic is sent by the client as a result of carrying a flooding attack, iptables drop the TCP connection.

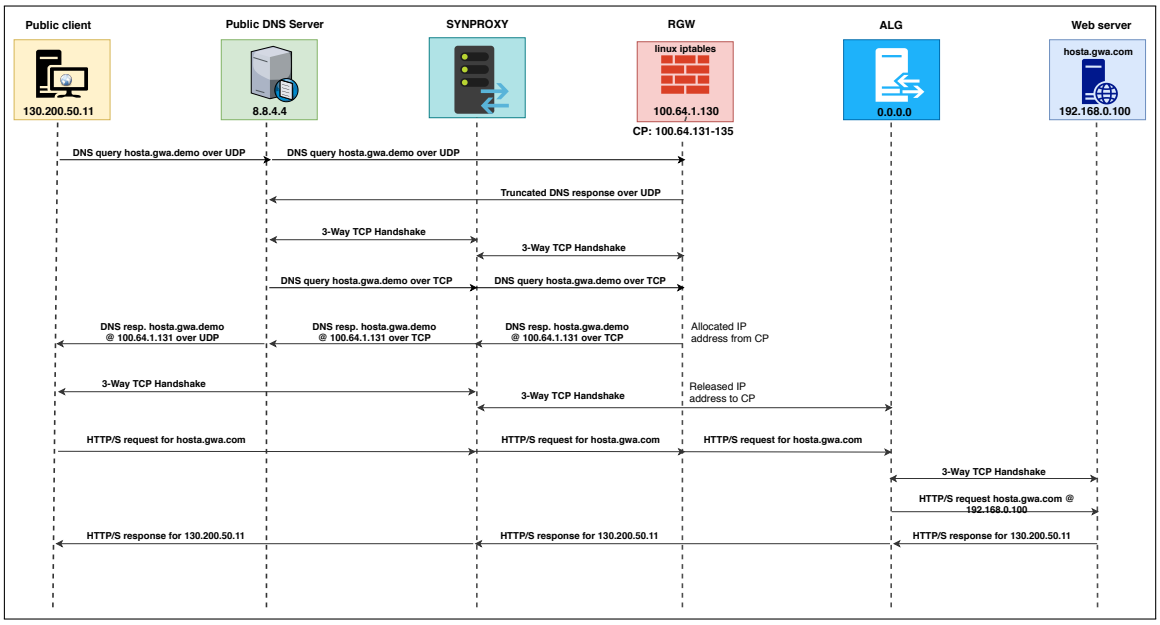

Figure 20: HTTP/HTTPS connection establishment using FQDN in ALG integrated with RGW

When the application data sent by the client is received by the ALG, it first detects the application layer protocol based on the first byte in the application data. Once the application layer protocol has been detected, ALG then uses YaLe [21] for detecting the hostname of the web server. ALG then establishes the connection to the IP address and port on which the domain is operating by matching against the domain information stored in its configuration file.

ALG is integrated with RGW using a wildcard IP address and iptable policies of RGW. No major modifications are made to RGW's code for ALG to inter-operate with RGW for handling HTTP/HTTPS connections.

4.5 Policy Database

A policy managaement system was developed by Hassaan Mohsin as part of his Master thesis called Security Policy Management (SPM) for a customer edge switching (CES) node [37][41]. The purpose of SPM is to allow end hosts behind the CES node to have more fine-grained control over the traffic they can accept. Managing policies using a database is more efficient as it makes it easier to make modifications. Policy sharing becomes easier and more accessible using a policy management system which helps in improving the scalability of the system.

4.5.1 Overview of SPM

SPM consists of a Policy-API using which policies are stored in a policy database. The policy management system provides a RESTful API for creating, modifying or removing policies in the database. It allows the remote users to include the user-specific policies through a Django-based web interface. The user input made using the Django web server, generates an HTTP request to the Policy-API of SPM for executing the requested action. The Policy-API can handle GET, PUT, DELETE and POST queries sent using the Django framework. All the policies are stored in JSON format.

The policy database in SPM constitutes of two main databases; Bootstrap_Policies and Session_Policies. Session_Policies contain policies related to the connection establishment and enforcing firewall rules for the private hosts while the policies related to the operation and configuration of the software are stored in the Bootstrap_Policies database. SQL client is used for initiating the connections to the two databases establishing one TCP connection with each database.

SPM is implemented using Python programming language and relies on two different servers for its operation namely a Django web server for providing the GUI (graphical user interface) to the users for editing the policies and the Policy-API server that is based on the REST architecture for managing policies and interacting with MySQL policy database. All the validity checks regarding the policies are performed by the Policy-API server in SPM. In the current implementation of SPM, the SQL client used to retrieve the policies using Policy-API server resides on the local system but it can be relocated to a remote setup. A TCP connection is established between the SQL client and Policy-API server. The SQL Client requests for a policy using a URI, which is parsed by the Policy-API server to extract the parameter values for forming the HTTP request. An SQL query is then generated for interacting with the MySQL database and the response is returned to the SQL client by the API-Server.

In SPM, the request for a policy can be sent using the Django framework or using the CES node directly which is served by the Policy-API server. However, in the design of this thesis since the objective was integrating RGW and ALG to the policy database, the policy management is done from the ALG and RGW directly without using the Django web interface.

4.5.2 Integration of RGW to SPM

The policies in RGW can be broadly classified into three different types namely, the policies concerning the traffic flows of hosts served by RGW, policies regarding the circular pool allocation based on the reputation system and the policies regarding the connection establishment by implementing NAT and firewall functionality. In RGW's original design, the policies for admitting flows were stored and fetched from a local repository. One objective of the thesis is to integrate RGW with the existing policy management system used for CES node known as SPM.

The three different types of policies in RGW were stored separately into different files using YAML format. YAML is a data-serialization language primarily used for storing and transmitting data in the configuration files. In the original RGW software, all the policies were loaded when RGW was initialized and any changes in the host policies after running RGW, were not reflected in RGW's operation without restarting the software. This is a shortcoming as any modifications in the host policies were not reflected in the RGW. To overcome this problem, RGW is integrated with SPM.

In SPM, a user is identified based on the FQDN. This user ID acts as a key for managing or retrieving any information regarding the specific user. RGW's policies are stored into the two different databases of SPM. All the policies regarding the circular pool allocation, iptables or reputation system are deemed as configuration policies necessary for RGW'S operation and are stored in the Bootstrap_Policies database. Whereas the host policies containing the information about the admitted flows and services supported by each host are added into the Session_Policies database.

In the current model of RGW integrated with SPM, RGW's policies are fetched using the Policy_API server from the SPM's database. An asynchronous HTTP client from Python's AIOHTTP library is used in RGW for fetching the policies from the Policy_API server by specifying the URL. The policies are retrieved using an asynchronous co-routine which does not hinder the execution of the rest of RGW's operation. The policy database is polled after every 10 seconds in asynchronous manner without disrupting the execution of other functions in RGW. The policies are then reinitialized in RGW, replacing the previous policies so that any modifications regarding the user policies become effective in subsequent communication[5].

4.5.3 Integration of ALG to SPM

The current implementation of ALG does not separate the policies regarding the configuration parameters of ALG and the policies for mapping the hostnames of web servers to corresponding IP address, port number combination into distinct categories. All the policies of ALG are stored in the Bootstrap_Policies database of SPM.

The policies related to the configuration parameters of ALG if modified, are not reflected during ALG's operation until ALG is restarted. Only the policies concerning the domain names of web server can be updated in the ALG. The updates could include modification of IP address or port on which a host is operating or the type of application layer protocol a web server supports. Registering new web servers with ALG is also possible after its initialization. The structure of the ALG's policies is depicted in Figure 21. ALG's policies are categorized based on the function they

[5]The modifications in RGW's software with the integration to SPM are available in the forked repository at https://github.com/Maria-Riaz/RealmGateway/tree/ldpsynproxy

perform and each policy has several attributes.

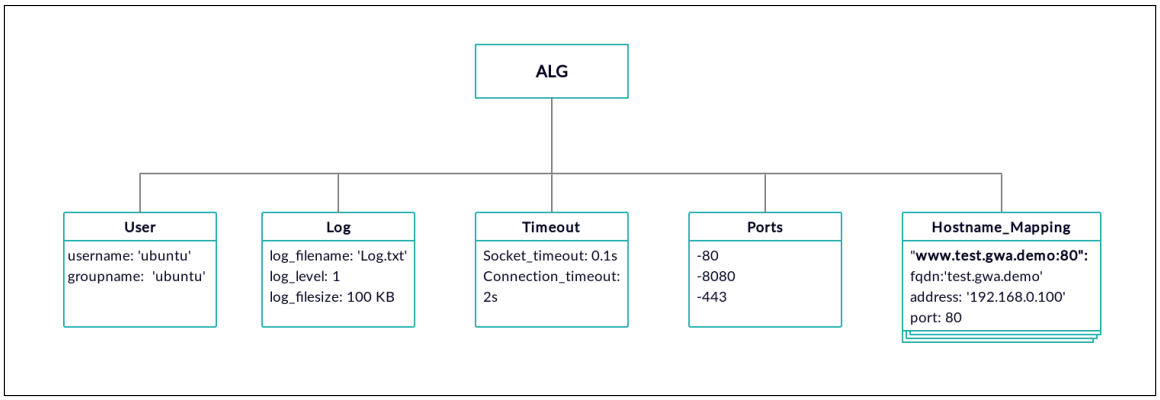

Figure 21: Structure of ALG policies

The policies are loaded at the start of ALG and are necessary for its operation as they include information regarding the ports on which ALG listens for incoming connections, HTTP connection timeout and the list of allowed users in addition to other data which is crucial for ALG's operation. The policies are not retrieved by the ALG directly from the policy database. A separate HTTP REST client is running on the same system running the ALG. The HTTP REST client executes as a separate Python program and is responsible for retrieving the ALG policies from SPM and storing them locally in a file.

The HTTP client polls SPM after every 10 seconds. Using the Python program, a copy of the newly fetched policies is retrieved and temporarily stored in a list for comparing against the local file containing previously stored policies. The newly fetched policies replace the policies stored in the local file only if the policies have been modified. If the ALG policies have been modified since the SPM was last accessed, the new policies are stored temporarily as a new local file. The new file is then renamed to match the original configuration file's name. This renaming of new file after first writing the policies to the file ensures that writing to the original local configuration file does not result in a conflict as the original file can be accessed by ALG at the same time as well.

ALG uses a system call for fetching the policies from the configuration file stored locally by the HTTP REST Client. During the initialization of ALG all the policies are loaded from the local configuration file. After the start of ALG's operation only the domain name policies of web servers are updated and the new connections are established based on the updated information. To prevent the policy updating function from interfering with other functions of ALG, the function for loading the updated policies is executed whenever a new client connection is accepted. The master process in ALG checks if the difference between the current time and the time at which the local configuration file was last accessed is greater than 10 seconds. If it is less than 10 seconds, ALG returns from the system call. Whereas if the difference between the times is greater than 10 seconds then ALG checks the time when the local configuration file was last modified. If the last modified time of the local file

is different than the global variable containing the previously stored value for the modified time, ALG updates the list of domain name policies for establishing new connections.

The reason that ALG uses a different approach for updating the policies using SPM is attributed to the differences in the architectural design of ALG and RGW. ALG uses a multi-process architecture and if every child process updates the policies by directly retrieving them from the database, it would increase the design complexity of ALG and also slow down the main operation of ALG concerned with connection establishment. It is easier if only the master process is responsible for updating the policies and as it is also responsible for accepting new connections, the method of retrieving directly from the policy database is best suited for a software that has an asynchronous architecture that operates in an event-driven manner. Also the policies in ALG are not as extensive as RGW, hence storing them locally has very insignificant impact on the system's resources. On the other hand, storing RGW's policies first locally and then accessing them would consume more time and fetching them directly using asynchronous call back function is compatible with RGW's design which relies on asynchronous co-routines for executing different functions.

5 Results and Evaluation

This chapter analyses the effect of the extensions incorporated in RGW. These extensions include integration of RGW with a custom ALG for handling web traffic and modifying the policy retrieval mechanism in RGW. At first we describe the environment used for carrying out various tests to evaluate the performance of RGW in the presence of ALG. This section also includes the details regarding software validation of the integrated RGW and ALG software package. Testing is done for HTTP and HTTPS application layer protocols by simulating various network conditions. Towards the end of each section, the results obtained for each testing scenario are evaluated.

5.1 Testing environment

The testing is carried out in a virtualized environment by using LXC containers to simulate different entities in the testing setup. The orchestration environment [6] comprises of five LXC containers representing the public clients, a public DNS server for resolving the DNS queries of the public clients, a container enacting SYN proxy, a container for simulating the local machine on which RGW and ALG's software is running and lastly a container for representing the private hosts served by the RGW. The containers are connected to each other using Linux bridges and the correct routing rules are added in each container for forwarding the packets. IP address assigned to each container are shown in Figure 22.

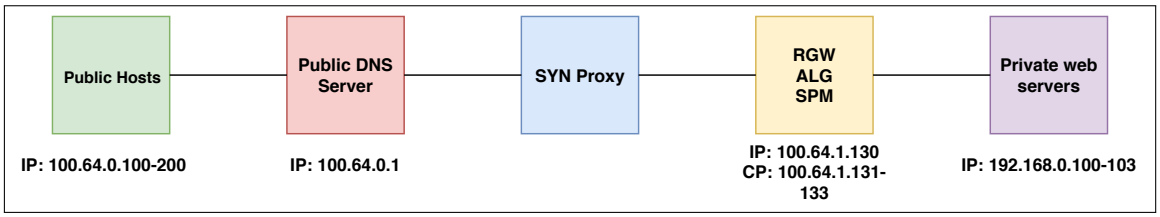

Figure 22: Orchestration environment used for testing

A range of IP addresses are assigned to the public hosts container and the private web servers container to simulate multiple clients and servers. Also the container running the RGW has three IP addresses for circular pool's operation while one IP address is assigned to RGW itself. SYN proxy operates on Layer 2 of the OSI model thus it does not have any IP address assigned to it. Tests are performed using ldpsynproxy [39] but in a few tests Linux kernel SYN proxy is also used to compare how the performance of the setup is affected by using a different SYN proxy. The orchestration environment is run on three different host machines having different hardware specifications indicated using Table 4. Host machine 1 is a laptop with

[6] The orchestration environment was created for the RGW software and is available in the software's github repository: https://github.com/Aalto5G/CustomerEdgeSwitching/tree/master/orchestration/lxc

Linux OS. Host machine 2 and 3 are servers available for testing purposes and accessed remotely using ssh connection.

System	Operating System	Processor Model	No. of cores	RAM
Host Machine 1	Ubuntu 16.04 (64 bit)	Intel(R) Core(TM) i5-6300U CPU @ 2.40GHz	4	8 GB
Host Machine 2	Ubuntu 16.04 (64-bit)	Intel(R) Xeon(R) CPU E5-2630 0 @ 2.30GHz	24	32 GB
Host Machine 3	Ubuntu 16.04 (64.bit)	Intel(R) Xeon(R) CPU E5-2630L v2 @ 2.40GHz	24	64 GB

Table 4: Specification of host machines used for testing

The performance of ALG is evaluated based on the latency of establishing a connection, the throughput of a connection and the scalability of the software under varied load conditions. Custom test scripts are used for evaluation in addition to using some benchmark tools designed for testing HTTP and HTTPS traffic. Siege [43] is an open-source benchmarking tool used for evaluating the performance of ALG. Weighttp [44] is also used for determining the concurrency level that can be achieved using ALG. To test the behaviour of ALG to an application layer DoS attack, SlowHTTPtest [45] is used. Tests to gauge the response time of RGW when using SPM are also explained in this section.

5.1.1 Software validation

Validation of the software is done to evaluate if the designed software fulfills the user requirements and operates correctly. The software validation involved verifying if the ALG can differentiate between HTTP/HTTPS requests and forward them to the correct back-end server. The setup involved having one public client send HTTP request to a web server hosted on 'www.test.gwa.demo' and another public client sending HTTPS request to a different web server 'www.test103.gwa.demo' which is running a web service on port 8000. The setup is represented in Figure 23. ALG relies on the policies stored for the domain name of web servers for forwarding the request to the correct upstream servers.

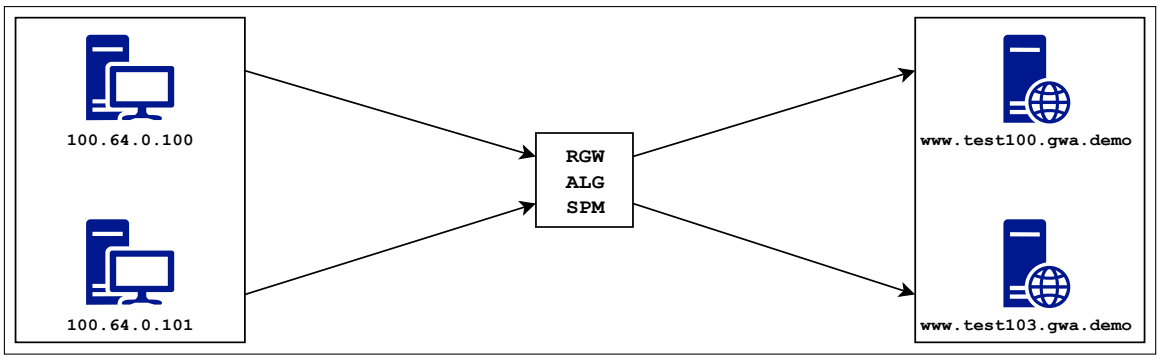

Figure 23: Test setup for Software Validation

The console information for public client requesting the web page from test103.gwa.demo is shown in Figure 24. The web page is requested using HTTP as the application layer protocol. A command line utility for retrieving the web page called cURL is used [47]. For implementing the HTTP Server, Python's HTTPServer module is used. The console reveals that the client is successfully able to retrieve the web page.

```
ubuntu@public:~$ curl --interface 100.64.0.101 www.test103.gwa.demo
<!doctype html>
<html>
  <head>
    <title>Test web page</title>
  </head>
  <body>
    <p>TTest web page for test103.gwa.demo <strong>p</strong> tag and its contents.</p>
  </body>
</html>
```

Figure 24: Retrieving web page using HTTP from test103.gwa.demo

The other public client sends an HTTPS request using cURL for retrieving the web page served by test.gwa.demo. Nginx web server is used for running the web service on test.gwa.demo and it supports both HTTP and HTTPS connections. Figure 25 indicates the content of the web page retrieved using HTTPS.

```
ubuntu@public:~$ curl --interface 100.64.0.100 --insecure https://www.test.gwa.demo
<!-- #Very very simple website for returning hostname of machine server is running on -->
<html>
 <head>
  <title>Hostname test</title>
 </head>
 <body>
 <p>My hostname is: test_gwa</p><h2>Aalto</h2>
<img src="aalto.jpg" alt="Aalto logo" style="width:175px;height:113px;">

<h2>Some code</h2>
<img src="code1.jpeg" alt="Some code" style="width:720px;height:1080px;">

<h2>More code</h2>
<img src="code2.jpeg" alt="More code" style="width:720px;height:480px;">

<h2>Even more code</h2>
<img src="code3.png" alt="Even more code" style="width:720px;height:487px;">


<p><a href="images.html">More images</a></p>
 </body>
</html>
```

Figure 25: Retrieving web page using HTTPS from test100.gwa.demo

Validation testing results have revealed that as long as the policies regarding the web servers are retrieved correctly in RGW and ALG from the policy database using SPM, ALG establishes the connection with the correct web servers. The same validation test was also performed by specifying an FQDN in the requested URL of cURL. The TCP connection was first handled by RGW using circular pool allocation and then the HTTP/HTTPS requests were processed by ALG.

5.2 Performance testing of ALG for HTTP

A series of tests are conducted to evaluate how ALG handles HTTP connections. The tests are carried on three different host machines specified in Table 4 using the orchestration environment explained in Section 5.1. Different Python test scripts are used for evaluating the performance in addition to some benchmarking tools.

5.2.1 Latency testing

To study the overhead added by integrating ALG to RGW, latency tests were conducted using one client and web server with a setup similar to Figure 23. An HTTP GET request is sent to the NGINX web server using TCP sockets and the response is received. The latency is first observed on the localhost running NGINX web server and then the test is performed from the public client using ALG. For comparison, the test is again performed replacing ALG with NGINX reverse proxy used previously with RGW. The time utility of Linux is used to measure the time taken for the execution of the test script for one successful HTTP request and response. The results were verified using Apache benchmark tool [42] that includes the connection latency as part of the output displayed on console. The cURL utility is not used for sending the request as it adds application latency and does not give an accurate measure of actual latency of ALG.

To get a better estimate, the test is performed by measuring the latency for establishing 1000 HTTP connections serially. The results obtained are used to calculate the latency measurement per connection and are indicated in the Table 5. Two different SYN proxies were used for measuring the latency and the ldpsynproxy [39] outperformed the kernel synproxy in establishing HTTP connection. While measuring the latency on the localhost, the measurement value includes the time taken to execute the test script in the OS kernel in addition to the connection latency. The system time and the user time have not been subtracted to maintain consistency among all the different latency measurements in different setups.

System	Latency on localhost (ms)	Latency using ALG (ms)	Latency using NGINX (ms)
Host machine 1 (with ldpsynproxy)	0.28	28.37	2.63
Host machine 2 (with kernel synproxy)	0.32	9.1	9.3
Host machine 2 (with ldpsynproxy)	0.32	6.43	1.54
Host machine 3 (with ldpsynproxy)	0.30	3.69	1.16

Table 5: Measuring latency for one HTTP connection using various setups

The client sent the HTTP request to retrieve a 626 bytes web page for the web domain www.test.gwa.demo. A graphical representation of the results is demonstrated in Figure 26. The graph reveals that NGINX reverse proxy is faster than ALG but the relative difference in performance is impacted by the hardware specification of the host machine running the orchestration. Using a 4-core laptop, ALG performs

10 times slower than the NGINX reverse proxy but the overhead added by ALG decreases significantly while using a system with 24 cores. A newer processor model also affects the latency as indicated by the difference in the results obtained using host machine 2 and host machine 3, where the latency of HTTP connection using ALG is decreased approximately by a factor of 2. The tests were repeated for different SYN proxies and the results indicated that kernel SYN proxy greatly affects the execution of NGINX's operation where NGINX actually takes more time for serving an HTTP client than ALG.

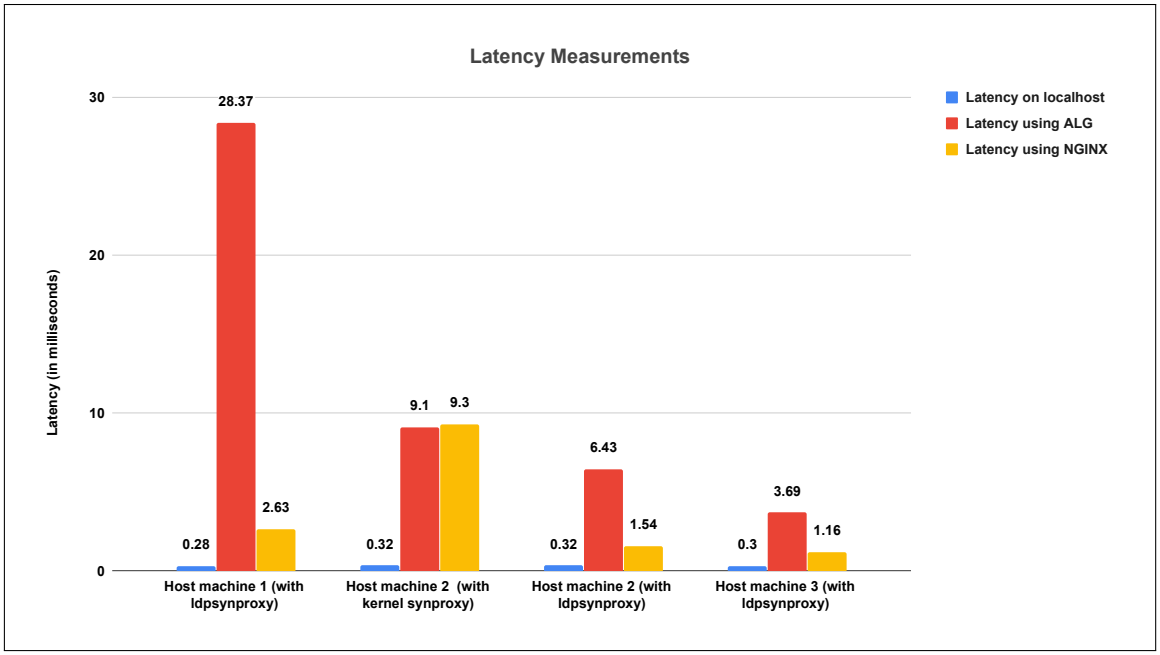

Figure 26: Measuring latency for one HTTP connection

The reason for the reduced latency on host machine 2 and host machine 3 is attributed to the fork system call. It takes less time to fork the child processes with a more powerful system. The forking time reduced by a factor of 10 in host machine 3 in comparison to host machine 1. The same test was also performed using FQDN in the HTTP request whereby the TCP connection is established using circular pool IP address and then handled by ALG as indicated by Table 6. We observed that when the client does not specify the service when sending the initial DNS query, the HTTP request is first received by the kernel space and then forwarded to ALG after releasing the IP address to the circular pool thus adding some delay in the completion of one HTTP request. This difference is minimized by using a better CPU for execution of the software.

ALG written using Python is the bottleneck in the overall setup. NGINX server does not have the ability to handle an HTTP request with an FQDN. It should be noted that the latency measurements can vary slightly depending on the load on the host machine. Looking at the results of the latency testing it can

be deduced that ALG's performance can be improved by using a system with a powerful processor and having greater number of cores. RGW and ALG are not developed to be run on client machines, ISP's or other intended customers employ better dedicated systems for their operation.

System	Latency using ALG with SFQDN (ms)	Latency using ALG with FQDN (ms)
Host machine 2	6.43	7.32
Host machine 3	3.69	4.22

Table 6: Comparison between latency measurements requesting FQDN and SFQDN

5.2.2 Throughput testing

Throughput testing was done to evaluate how the data transfer rate of ALG varied with increasing the number of concurrent clients. It is a measure of how much data ALG can handle at a particular instance of time. Multiple clients requested a large file from the back-end NGINX web server and the time it took to complete the request was observed. This test also evaluates the performance of the SYN proxy by analyzing how quickly the TCP packets are processed by the different SYN proxies. The file used for downloading is an image of a virtual machine and has a size of 1.8GB (gigabytes). Wget [48] tool was used for fetching the file from the web server and the experiment was conducted on host machine 1 and host machine 2.

Table 7 demonstrates the total throughput obtained when downloading the file with different number of clients. In case of multiple clients, the requests are sent concurrently and they are handled using multiple processes in ALG. The time taken by each client to download the file was noted and then the average throughput per connection was computed based on the average time taken to download the file. A significant relative deviation was observed in the values of time taken to download the file using NGINX when there were 10 clients being served concurrently on host machine 2. To get a better approximation of the total throughput using NGINX, the transmitted bytes on the container running RGW and NGINX were observed after the client starts downloading the file for calculating the average throughput.

Setup	Total throughput using ALG (MB/s)			Total throughput using ALG through circular pool (MB/s)			Total throughput using NGINX (MB/s)		
	c=1	c=4	c=10	c=1	c=4	c=10	c=1	c=4	c=10
Host machine 2 (with kernel synproxy)	54.09	183.38	261.34	197.75	315.12	470.0	251.9	298.72	156.96
Host machine 1 (with ldpsynproxy)	55.37	76.40	73.80	54.09	56.92	61.67	49.60	61.56	63.04
Host machine 2 (with ldpsynproxy)	49.60	72.04	63.85	63.42	68.31	62.90	65.30	80.76	80.00

Table 7: Measuring throughput of downloading 1.8 GB file using HTTP

Figures 27, 28, 29 present a graphical comparison of the throughput measurements obtained using different setups. In the throughput results it was seen that kernel SYN proxy can handle large chunks of data at a faster rate than the ldpsynproxy when performing the test on same host machine.

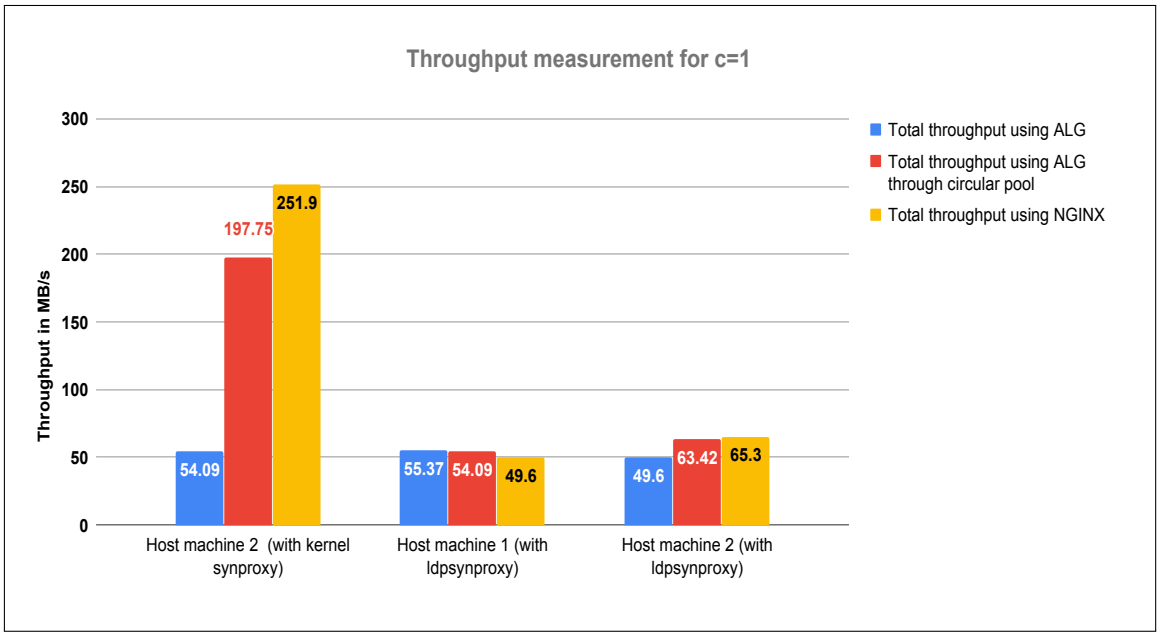

Figure 27: Measuring throughput with one client

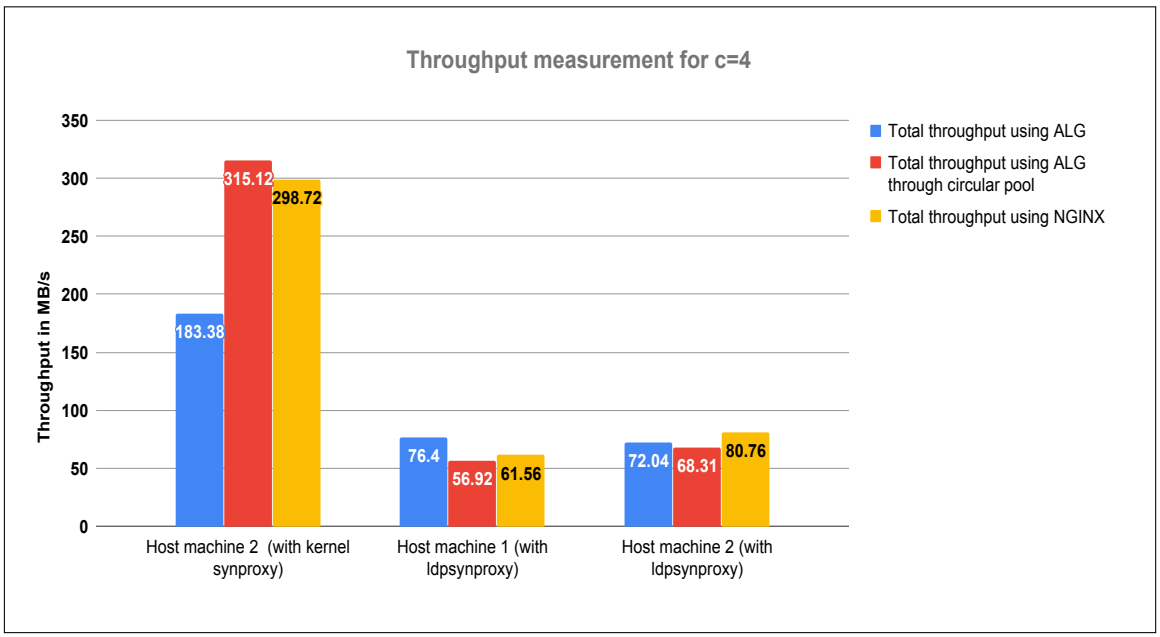

Figure 28: Measuring throughput with 4 clients

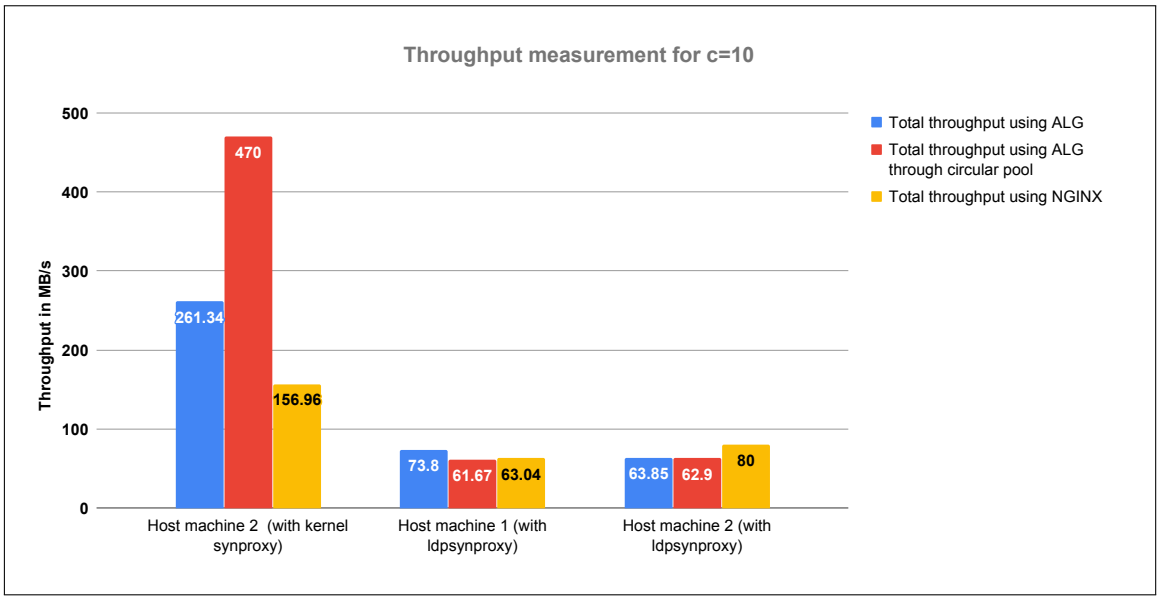

Figure 29: Measuring throughput with 10 clients

The throughput obtained with NGINX reverse proxy is approximately the same as that using ALG when ldpsynproxy is used in the setup. Though some inconsistencies were seen in the behaviour of NGINX when serving the request of 10 files simultaneously. Some of the clients finished downloading the complete file in a few seconds while it took around 180 seconds for the rest of them to download the file. The average throughput per connection remained the same with ALG.

Also it was seen that the throughput of ALG and NGINX was better when running the orchestration environment on host machine 2 owing to the better hardware specification of the system. The difference in performance was considerably greater when kernel synproxy was used in the test setup. This is because kernel synproxy uses TCP Segmentation offloading which allows it to process packets larger than the usual 1500 bytes thus making it faster than ldpsynproxy. The throughput when the file was downloaded by sending an FQDN in the DNS query was notably greater than directly downloading the file from ALG using SFQDN with kernel synproxy in the setup. Whereas with ldpsynproxy, downloading the file using SFQDN required approximately the same time as requesting it using circular pool IP address.

5.2.3 Scalability testing

We performed a series of experiments to assess how much load ALG can handle without affecting its performance. This test helped us to analyze the operating capacity of ALG and the number of concurrent connections that ALG can support without crashing. The public container in the orchestration setup in Figure 22 is used to simulate 100 clients having different IP addresses sending multiple HTTP requests to the web server test.gwa.demo behind the RGW. Each of these clients was used to send multiple requests to the web server using different source ports. The

connections were opened serially and then kept alive even after serving the request to simulate load on the ALG. This test is performed only for SFQDN queries sent by the clients as the main objective of this test is gauge the scalability of ALG.

For evaluating the performance, we gradually increased the number of connections handled by ALG at a given instance of time and analyzed the memory of the system used in establishing the connections. Since ALG's design is based on a multi-process architecture, memory used for forking processes is one of the limiting factors in ALG's support for multiple connections. However, since each of the host machines was handling several other processes for executing other tasks, the memory consumption is only an approximation. The results could vary if the orchestration is run on a dedicated server only running the orchestration setup. ALG has a rate limiting algorithm that allows it to set a maximum limit on the number of connections it can handle simultaneously. For analyzing the maximum load ALG can tolerate, the maximum connection count in the algorithm was changed to a significantly large number.

Table 8 presents an analysis of memory consumed in handling varied number of HTTP connections. The symbol 'c' indicates the connection count while 'x' indicates that the respective host machine did not have enough system resources to handle the specified connection count. Opening any more connections than the limit resulted in either slowing down or momentary hanging of the system.

System	**Memory used by ALG (in GB)**					
	c=100	c=1000	c=2000	c=5000	c=10,000	c=20,000
Host Machine 1 (8GB RAM)	0.18	1.54	3.08	x	x	x
Host machine 2 (32 GB RAM)	0.15	1.54	2.96	7.74	14.75	x
Host machine 3 (64 GB RAM)	0.15	1.53	3.04	7.78	15.47	34.9

Table 8: Memory consumption for multiple HTTP connections in ALG

The main goal of the test was finding the upper limit of the connections that can be handled by ALG before it depletes the system's resources. The graphical representation of the data is shown in Figure 30. Using host machine 1, ALG was able to handle 2000 connections simultaneously. The maximum count increased significantly upto 10,000 connections on host machine 2. However, the maximum number of connections were handled by host machine 3 accounting to 20,000 HTTP connections as it had the most RAM available for forking multiple processes.

While conducting this experiment, it was also verified that ALG closes all the connections properly that are closed by the client but for some reason were not

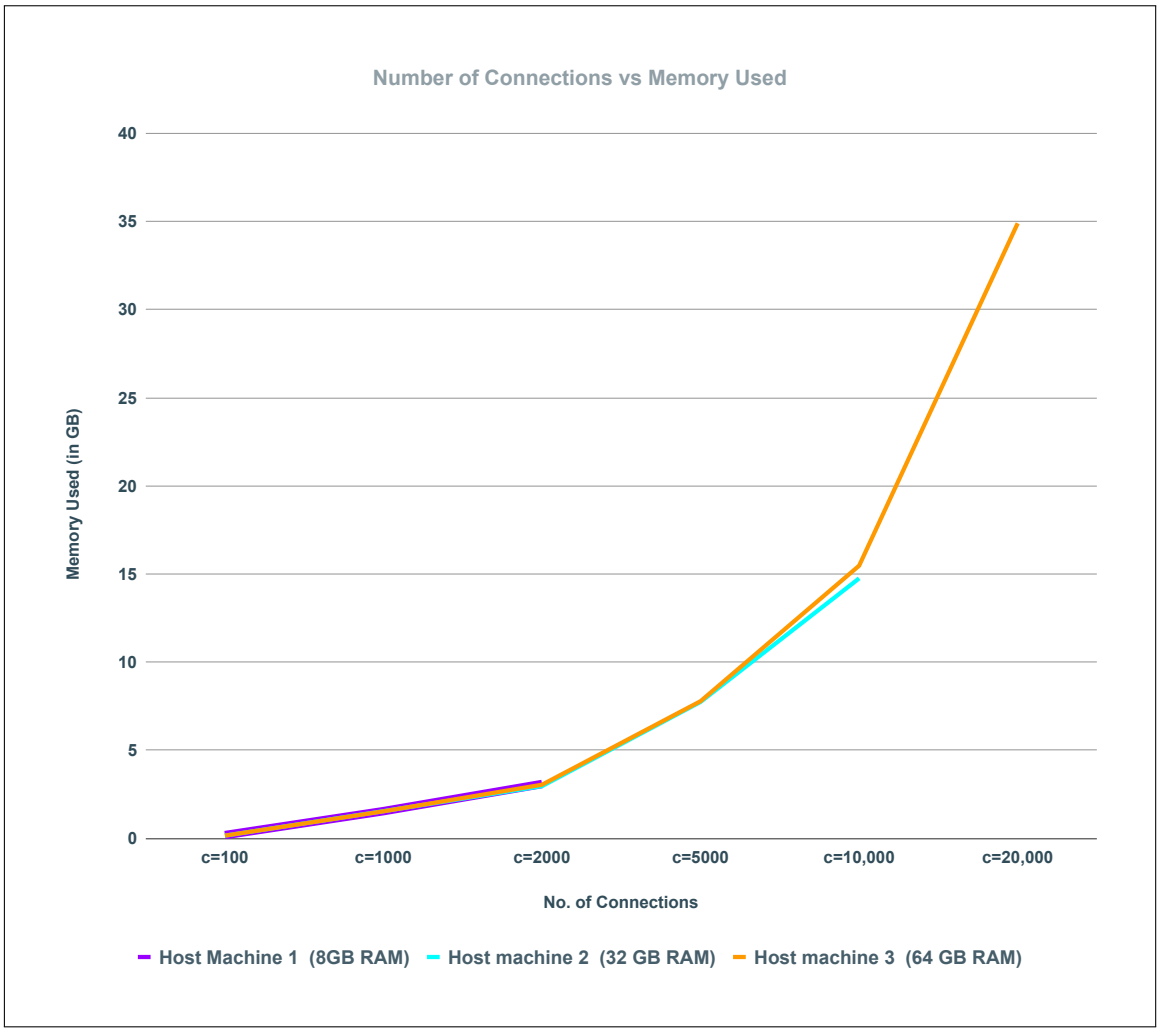

Figure 30: Number of connections vs Memory utilized for HTTP connections

terminated properly by the back-end NGINX server. After sending keep-alive probes for a specified duration set using a configuration parameter in ALG, all the connections closed by the clients are also closed by ALG.

In the above mentioned experiment, the requests were sent by the clients serially to find out how many connections ALG handles before the system's resources are exhausted. Another test was performed to evaluate how ALG behaves when receiving large number of concurrent requests. This test helped in finding the maximum number of requests ALG can handle in one second successfully. Two benchmark tools were used to carry out the experiments, namely Siege [43] and weighttp [44]. Wieghttp uses multi-threading to create concurrent requests and thus it is faster than Siege. Siege has a limitation where it cannot handle more than 100 concurrent users and the HTTP requests start failing after that. The reason for using Siege is that weighttp does not support TLS with the SNI extension and thus can not be used for testing the results for HTTPS. Siege is used mainly to draw a

comparison between the performance of ALG for HTTP and HTTPS connections.

The scalability test using benchmark tools can be considered a stress test where the system is tested to its limits to identify which component fails under high load conditions. During the test, NGINX reverse proxy is used with the default configuration parameters while the configuration parameters on the back-end NGINX server are tuned for performance. The maximum number of connections that one worker process in NGINX can handle is set to 10,000 and each worker process is allowed to accept multiple connections at one instant of time. It should be noted that although the test is performed for upto 1000 concurrent clients, a medium e-commerce website has around 100 to 200 requests per second. Table 9 indicates the results obtained performing a series of tests with different number of concurrent clients sending multiple requests. The 'x' in the table indicates that the test was not successfully completed as all the requests were not served by the upstream NGINX server. Consequently the clients received an error message with a status code 502 which indicates that the back-end server was unable to handle the request and thus the gateway relayed this information to the client.

Test parameters		**Performance using ALG**		**Performance using NGINX**	
Concurrency level (c)	Number of requests (N)	Time (in seconds)	Requests/second	Time (in seconds)	Requests/second
10	1000	0.081	12302	0.19	5057
100	1000	0.29	3488	0.19	5311
1000	1000	2.53	395	x	x
10	10000	0.69	12088	1.23	8071
100	10000	0.93	8313	1.11	8997
1000	10000	2.9	3456	x	x
10	100000	6.1	16388	9.4	10628
100	100000	5.69	17567	5.83	17142
1000	100000	x	x	x	x

Table 9: Stress Testing using weighttp for HTTP connections

Figure 31 represents the concurrency level achieved for different number of requests using ALG and NGINX. The graph illustrates that NGINX reverse proxy cannot handle 1000 concurrent clients and the requests start failing. It can be seen from Figure 31 that for the same number of total requests, increasing the concurrency level results in less requests being handled in one second by the ALG or NGINX reverse proxy. Whereas for the same number of concurrent clients, increasing the total number of requests results in an increase in the requests per second processed by the ALG or NGINX reverse proxy. The latter holds true only until the maximum concurrency level handled by ALG or NGINX is attained. After reaching the maximum number of simultaneous clients ALG or NGINX reverse proxy can handle,

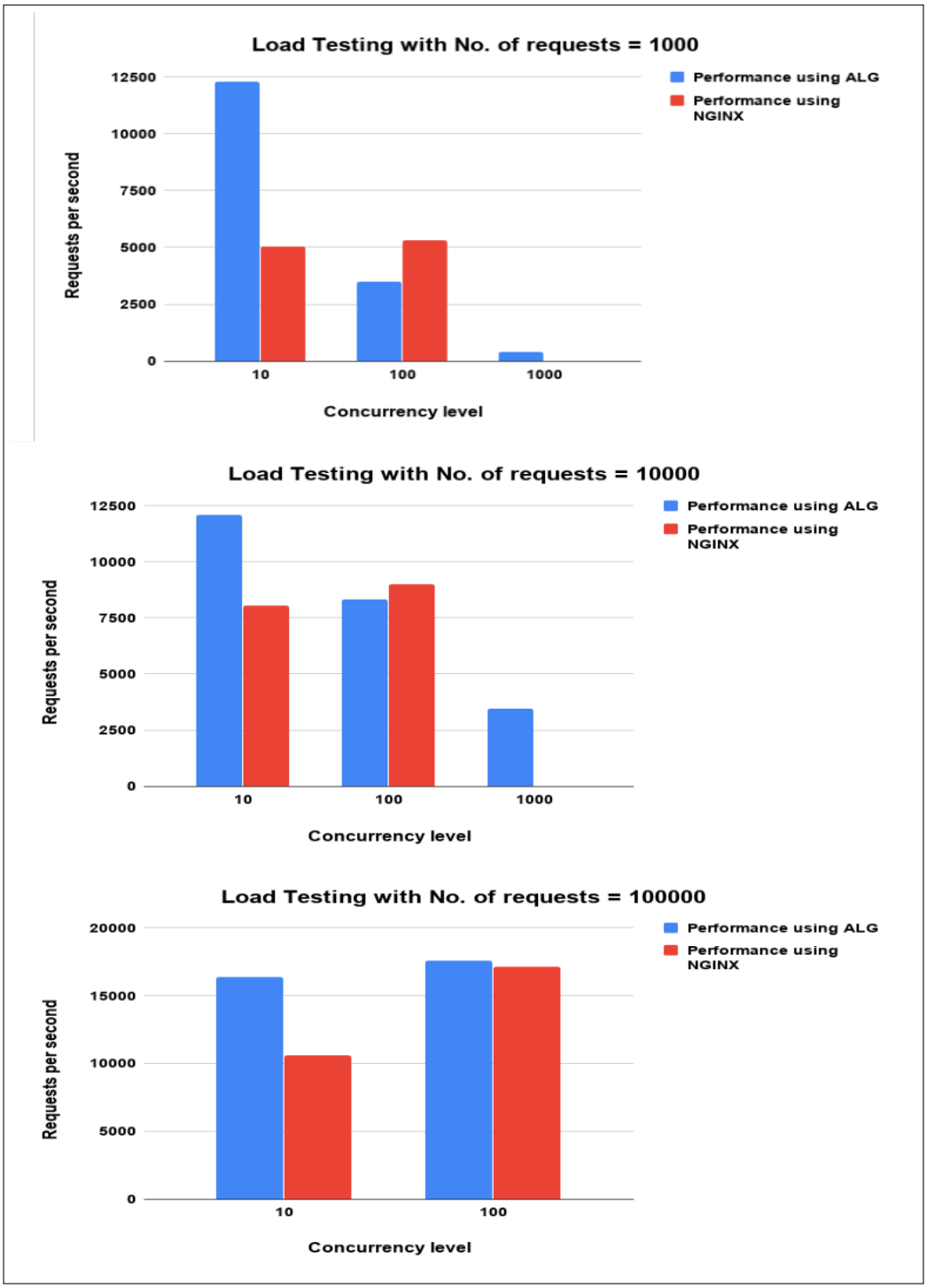

Figure 31: Testing the scalabiity using weighttp

the served request rate starts decreasing as indicated by concurrency level of 1000 in Table 9. The error logs of the upstream NGINX server hosted at test.gwa.demo reveal that it is the PHP-FastCGI module, used by NGINX web server to handle dynamic web pages, that is unable to process the multiple requests and results in

the requested page becoming unavailable. The error is indicated as follows:

```
[error] 3038030380: *1510994 connect() to unix:/run/php/php7.0-fpm.sock
failed (11: Resource temporarily unavailable) while connecting to
upstream, client: 192.168.0.1, server: *.gwa.demo, request: "GET /
HTTP/1.1", upstream: "fastcgi://unix:/run/php/php7.0-fpm.sock:", host:
"www.test.gwa.demo"
```

To compare the load testing results obtained using HTTP with HTTPS, another benchmark tool was used called Siege [43]. Siege retrieved the web page with all the embedded content and as a result the response time for one HTTP connection was greater than the response time obtained in other tests. To make the setup comparable to other tests, the client requested only a static web page of 323 bytes. Siege was used in the benchmarking mode in which there is no delay between the different clusters of requests. One cluster of requests has the number of requests equal to the concurrency level set by the user.

Table 10 lists the results of stress testing done using Siege. The same trend was observed using Siege as weighttp, where increasing the number of requests by maintaining a constant concurrency level, resulted in an increase in the rate at which requests were handled by the ALG or NGINX reverse proxy. This holds true only when the total number of requests sent are equal to a reasonable value after which the performance starts degrading.

Test parameters		**Performance using ALG**			**Performance using NGINX**		
Concurrency level (c)	**No. of requests (N)**	**Failed requests**	**Time (s)**	**Requests per second**	**Failed requests**	**Time (s)**	**Requests per second**
10	1000	0	0.15	6666	0	0.19	5263
100	1000	0	0.34	2941	0	0.16	6250
1000	1000	184	65.23	12.51	163	70.12	11.92
10	10000	90	1.13	8769	90	1.31	7564
100	10000	0	1.07	9345	0	1.30	7692
10	100000	990	8.79	11263	990	11.86	8342
100	100000	900	8.79	11274	900	7.67	12920

Table 10: Stress Testing using Siege for HTTP connections

Requests failed in two different scenarios. First, if the concurrency level was set to a very high value, NGINX back-end server was unable to process the requests. In the second scenario, if a very large number of requests are sent using a few concurrent clients, the benchmark tool established a lot of TCP connections with RGW but was unable to send all the HTTP requests before the socket timed out. The second reason for failing of the requests is attributed to the design of Siege.

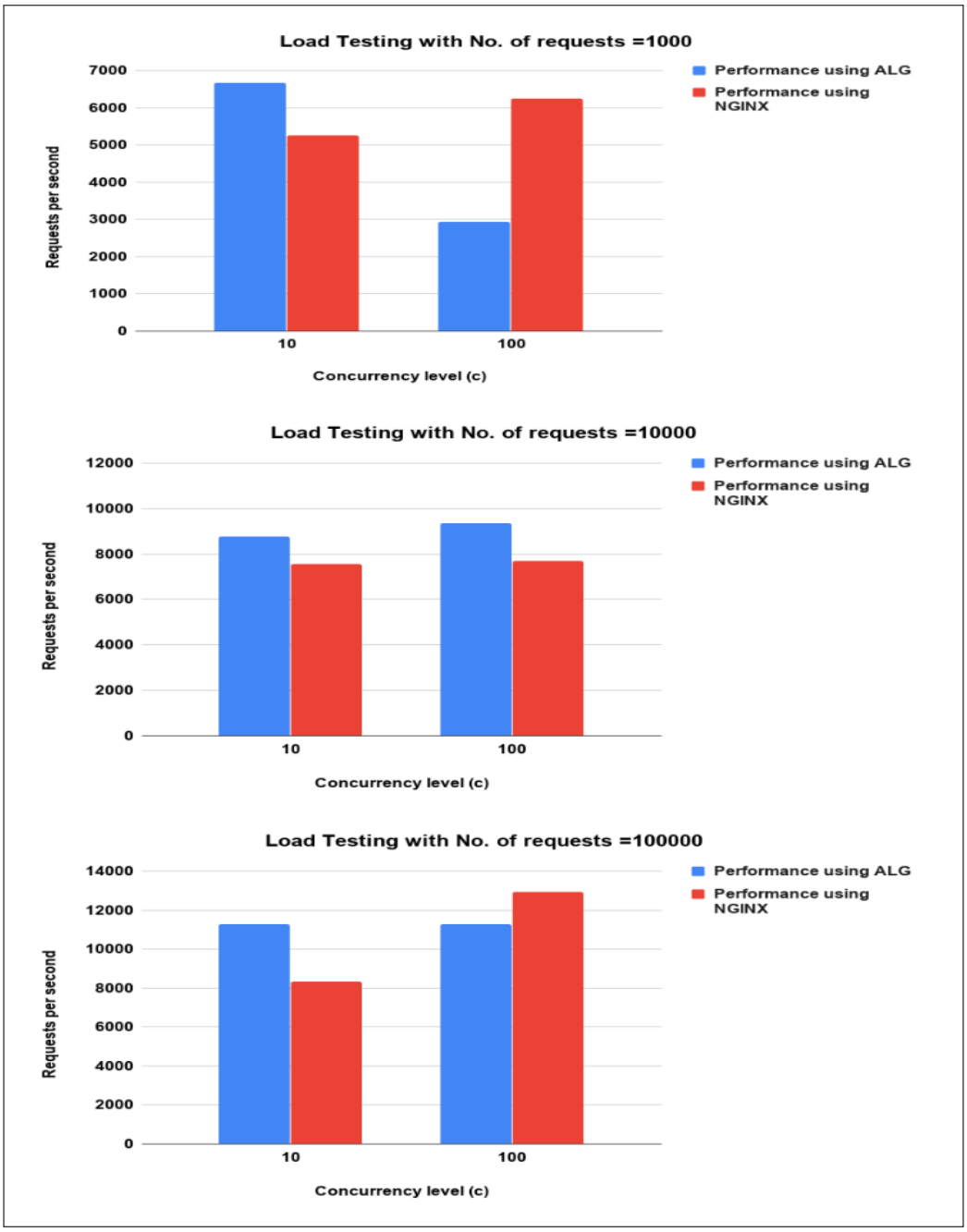

Figure 32: Testing the scalability for HTTP using Siege

Figure 32 gives a pictorial representation of the obtained results using Siege. It can be seen that for a smaller concurrency level, ALG outperforms NGINX reverse proxy while the results vary for concurrency level of 100 for different total requests sent. Also it should be noted that the results are comparable for smaller concurrency value with the results obtained using weighttp. However using weighttp, the requests per second handled by ALG or NGINX were considerably higher for different values of N and c.

Scalability results performed using weighttp show that ALG is better than NGINX reverse proxy at handling concurrent clients though the back-end server has a limit of concurrency after which the requests start failing.

5.3 Performance testing of ALG for HTTPS

For analyzing the performance overhead of ALG when dealing with HTTPS connections, a number of different experiments were conducted. Three different host machines were used for the purpose of setting up the test environment having different hardware specifications listed in Table 4. The tests are centered around measuring the latency and throughput of an HTTP connection and analyzing the impact of increasing the connection count on ALG's performance.

5.3.1 Latency testing

During the latency test, one client sent the HTTPS request to the web server 'test.gwa.demo' and the response time for the corresponding request is recorded. The HTTP GET method is used for sending the request to retrieve the default page served by the back-end web server. OpenSSL library is used for wrapping the TCP sockets with SSL (Secure Sockets Layer) wrapper to send the encrypted request from the client. Similar to the latency test conducted for HTTP in Section 5.2.1, first the latency of retrieving the page using a local web client is measured. The experiment is then repeated from the public client running on the public container and ldpsynproxy is used in the test setup shown in Figure 22. A comparison is done between the performance of ALG and NGINX reverse proxy. Python's time module is used for measuring the execution time of the test script.

To get a more accurate measurement, the latency test is performed by one client sending 1000 requests serially and then the results obtained are scaled down by a factor of 1000 to get the latency of serving one HTTPS request by the web server in different setups. The latency measurement obtained for the local host also includes the CPU time used by the process running the test script in addition to the time taken for completing one HTTP request. CPU's execution time has not been subtracted for maintaining consistency in the latency values achieved in various setups. Table 11 illustrates the latency for one HTTPS connection using various setups.

System	**Latency on localhost (ms)**	**Latency using ALG (ms)**	**Latency using NGINX (ms)**
Host machine 1	2.3	19.11	5.89
Host machine 2	3.52	11.72	9.17
Host machine 3	3.34	6.79	7.90

Table 11: Measuring latency for one HTTPS connection using various setups

The graphical representation of the latency measurements is presented in Figure 11. Using a 4-core laptop, ALG is approximately 3 times slower than NGINX reverse proxy server. The difference between the latency measurements of ALG and NGINX reverse proxy on host machine 1 is 13 ms whereas on host machine 2 the difference reduces significantly to 2.5 ms. While using host machine 3, it can be seen that ALG outperforms NGINX reverse proxy server by 1 ms. The reason for this improvement is that ALG utilizes the multi-core structure of the CPU whereas NGINX handles multiple connections using a single worker process. These results are an indication that ALG's performance is directly dependent on the system on which it is run.

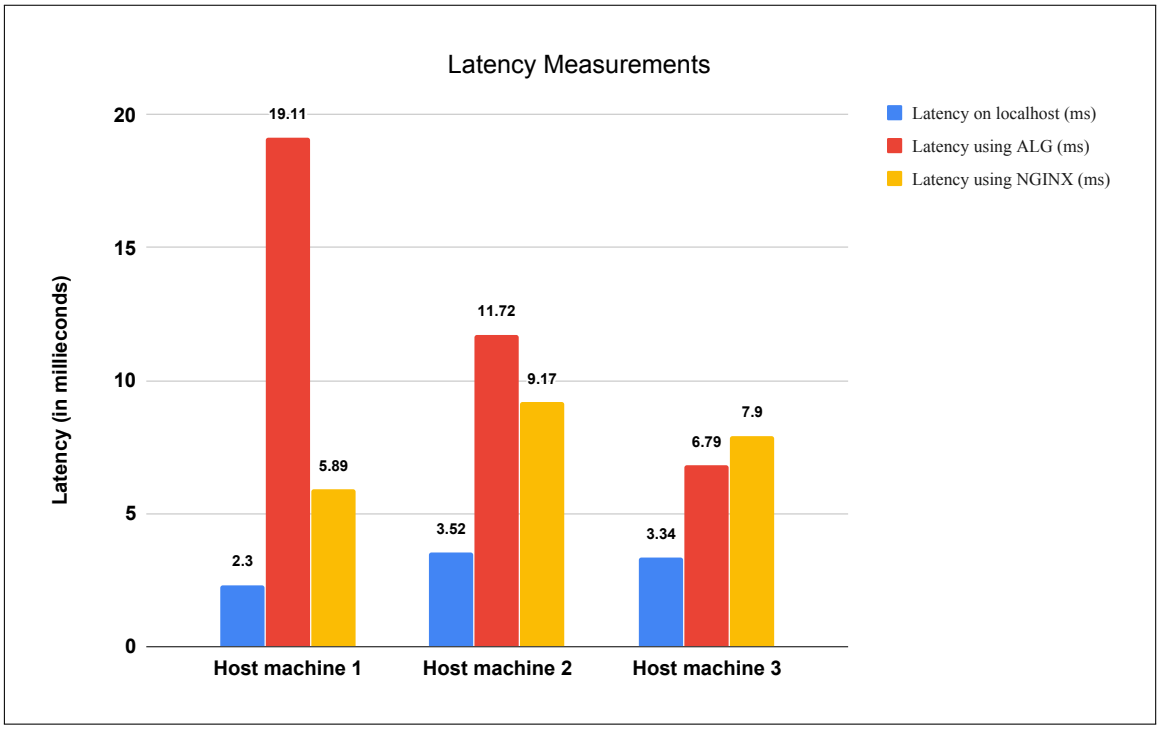

Figure 33: Latency Measurements for HTTPS connection

As explained in Section 5.2.1, the main reason for the lower latency is attributed to the time taken by the operating system in forking a process. The time for forking 1000 processes in host machine 1 is 7.4 seconds while it is reduced to a significantly lower value of 0.76 seconds in host machine 3. The latency test was also performed with the client requesting the web page using FQDN to see how the circular pool affects the latency measurements. NGINX reverse proxy server doesn't support HTTPS connections using FQDN in the HTTPS request thus the test was executed only using ALG on host machine 3. While performing the test using FQDN on host machine 1 and host machine 2, it was observed that the DNS server in RGW was reserving the circular IP address for allocation but the reserved connection was not being used. The problem could be related to the DNS caching taking place at the

public client's DNS resolver or it could be some timing issue related to execution of different functions in RGW as a result of which RGW cannot detect the incoming connection from the client and thus lowers its reputation. This problem needs to be further investigated to know the real cause of the problem.

Figure 34 illustrates the overhead added when RGW's circular pool is used for the initial connection establishment. The difference is approximately 0.5 ms on host machine 3. Looking at the results of latency when ALG is directly handling the connections in different host machines, this difference could be larger when using a system with less CPU cores and processing power. Summarizing the test results, it is observed that ALG works better when the circular pool IP address is not used for the HTTPS connection and a better performance can be achieved using powerful host machine.

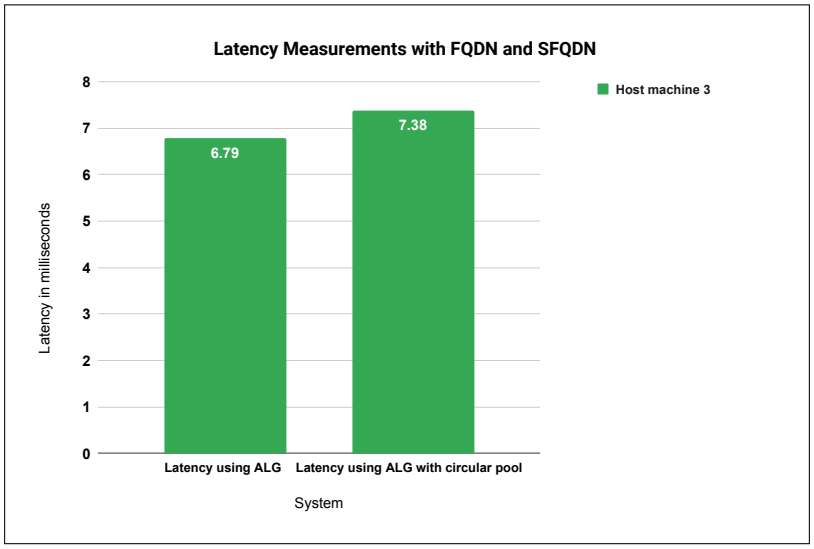

Figure 34: Latency Measurements for HTTPS connection with FQDN vs SFQDN

5.3.2 Throughput testing

This test was conducted to evaluate how much data can be processed by ALG in a given amount of time and analyze the impact of concurrent connections on the throughput achieved. The test setup is indicated in Figure 22 where the public container was used to simulate multiple clients sending HTTPS requests to retrieve a file of size 1.8 GB from the back-end web server hosted at test.gwa.demo. Two different SYN proxies are used in the test setup to gauge which SYN proxy is better suited for the design of RGW. Similar to the test performed in Section 5.2.2, Wget[48] was used for retrieving the web page content from test.gwa.demo. The orchestration environment was set up on host machine 1 and host machine 2 for this experiment.

Table 12 indicates the throughput achieved using different host machines where c

refers to the number of clients. When multiple clients were requesting the same file, parallel connections were established using different Linux processes. The time taken for each client to download the same file was recorded. The total throughput was calculated by computing the average time to download the file and multiplying it with the number of clients. This total throughput is a good estimate when the time taken by various clients is approximately the same. When the concurrency level was 10 and NGINX reverse proxy was used for downloading the file from the back-end server, significant deviation was observed in the download time of multiple clients. Hence, the total throughput was measured by looking at the bytes received in 5 seconds on the network interface in the public container after running the test script that sends the file retrieval request using multiple clients. The value obtained was divided by 5 to give the total throughput achieved using NGINX with a concurrency level of 10. This is an an approximation but the average is not a good mathematical measure when the data values are far apart.

Setup	**Total throughput using ALG (MB/s)**			**Total throughput using ALG through circular pool (MB/s)**			**Total throughput using NGINX (MB/s)**		
	c=1	**c=4**	**c=10**	**c=1**	**c=4**	**c=10**	**c=1**	**c=4**	**c=10**
Host machine 1 (with ldpsynproxy)	57.47	70.04	67.7	57.47	67.64	66.2	57.47	78.4	58.62
Host machine 2 (with ldpsynproxy)	63.41	74.40	67.6	65.6	71.24	66.3	73.56	85.32	80
Host machine 2 (with kernel synproxy)	167.19	230	229.93	224.28	297.66	294.75	141.47	159.4	119.01

Table 12: Measuring throughput of downloading 1.8 GB file using HTTPS

Figure 35 gives a visual representation of the measured throughput. It was observed that the highest throughput in all the test setups was obtained using kernel SYN proxy. The reason for the inferior performance of ldpsynproxy is that it is customarily designed for operating in a real network environment and thus gives poor results when used in a virtualized environment where it is not directly connected to the NIC. When downloading the file using 10 concurrent clients, NGINX showed an anomalous behavior where some clients were able to download the file in a few seconds while it took several minutes to download the file completely by the other clients. Thus the total throughput was not calculated based on the average throughput per connection rather directly from the network interface.

The total throughput obtained using NGINX reverse proxy was comparable with ALG but ALG performed slightly better in case of multiple clients. Also using kernel synproxy, the throughput obtained when using the circular pool for connection establishment is significantly higher than ALG alone. This suggests that kernel synproxy is compatible with the circular pool of RGW which uses Linux kernel iptables for the NAT rules. When using ldpsynproxy in the test setup, the throughput obtained using ALG directly or using ALG through the circular pool is approximately the same. This indicates that once the connection is established, there is no difference in the throughput obtained using ALG directly or with RGW.

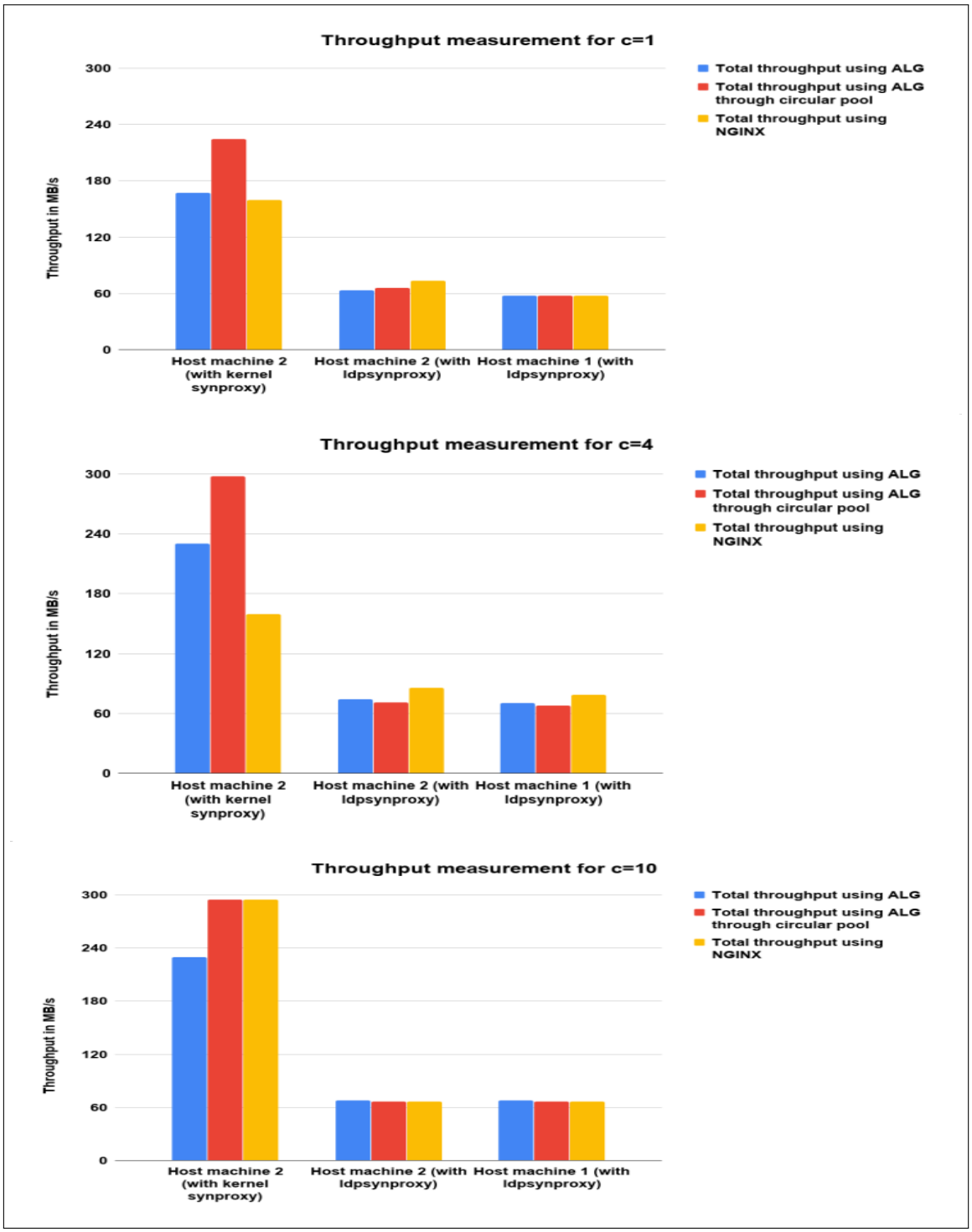

Figure 35: Measuring throughput for multiple clients using HTTPS

5.3.3 Scalability testing

In this section we present an analysis on the performance of ALG when tested under high load conditions. The main goal of the test was to find the maximum capacity of

ALG for establishing HTTPS connections. The test was performed using 100 clients on the public container in the test setup illustrated in Figure 22. Each of these 100 clients sends multiple requests to the web server test.gwa.demo behind RGW for simulating high load conditions. In the test, the connections are established serially and then kept alive for a duration of 300 seconds.

ALG is gradually subjected to higher load by increasing the number of HTTPS requests sent by the clients. The memory utilization is observed on the host machine running the orchestration environment. ALG has a multi-process architecture and relies on forking for establishing multiple connections. Since each HTTPS connection requires two processes for its operation, forking processes seems to take the most memory in ALG's operation. Each of the host machines had many other processes running in parallel thus the results shown in Table 13 are only an approximation. The symbol 'x' in the table represents that the specific host machine did not have the capability to handle that many connections. The tables lists the memory used by the system for establishing the number of connections represented as 'c'.

System	**Memory used by ALG (in GB)**					
	c=100	c=1000	c=2000	c=5000	c=10,000	c=20,000
Host Machine 1 (8GB RAM)	0.17	1.61	2.58	x	x	x
Host machine 2 (32 GB RAM)	0.16	1.57	3.14	7.59	15.35	x
Host machine 3 (64 GB RAM)	0.16	1.60	3.27	8.02	16.21	32.8

Table 13: Memory consumption for multiple HTTPS connections in ALG

The results obtained in the test indicate that ALG's ability to handle multiple connections is directly impacted by the hardware specification of the host machine running it. The upper bound on the number of HTTPS requests using each of the host machines is represented in Figure 36. Host machine 1 has the least amount of memory out of the three machines and hence can only send 2000 HTTPS requests after which the system freezes temporarily until the connections are closed freeing up the system's memory. 10,000 HTTPS connections are established using host machine 2 while the maximum connection count of 20,000 is achieved using host machine 3. Comparing the memory usage in HTTP and HTTPS connections, it can be seen that since the operation of ALG is the same for both HTTP and HTTPS protocols apart from the hostname detection in the beginning, the memory used for opening HTTP or HTTPS connections is almost identical.

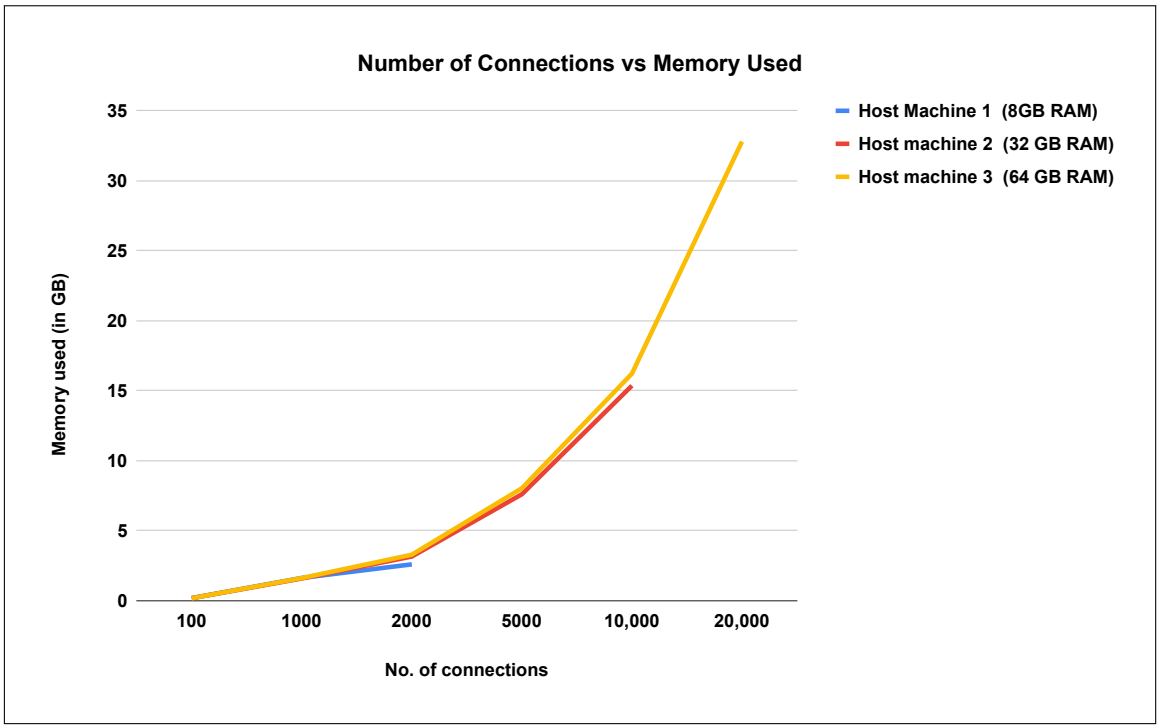

Figure 36: Number of connections vs Memory utilized for HTTPS connections

The above mentioned experiment was performed to analyze the effect of opening multiple connections on the system's resources running ALG's software. This test does not estimate the concurrency level that can be handled by the ALG. Therefore to test the concurrency level that can be achieved with a reasonable performance a stress test was conducted. In this test, the system was subjected to a large burst of traffic in a short duration of time to evaluate the breaking point of various software components in the orchestration environment although the focus was on ALG.

Since ALG was to be subjected to a large burst of traffic, the afore mentioned test was executed only on host machine 3. The stress test was carried out using a benchmark tool known as Siege [43]. Choosing a benchmark tool for HTTPS was fairly difficult as most of the widely known tools either do not have the support for HTTPS or do not use the SNI extension during the TLS handshake essential to the operation of ALG. Siege uses the SNI extension and was thus used for the purpose of testing. Although the tool had a limitation where sending a large number of requests resulted in a few failed requests. The upstream NGINX server hosted on test.gwa.demo is tuned for handling larger traffic loads by altering the number of connections handled by each worker process of NGINX to 10000 from the default 768 connections. Also the worker process was allowed to accept multiple simultaneous connections rather than handling them one at a time. The NGINX reverse proxy was used with the default configuration parameters as increasing the number of connections handled by a worker process had no impact on the results obtained.

Table 14 summarizes the results of stress testing performed using ALG and

NGINX reverse proxy respectively. The 'x' in the table is an indication that with the specified test parameters, the failure rate of the test was extremely high and thus is considered unsuccessful. The reason for the failure is attributed to either the back-end NGINX server or the benchmark tool. The reverse proxy or ALG were not the point of failure in the test. Failed requests due to NGINX web server were due to the failure of PHP-FastCGI module in NGINX server which is used for serving dynamic web content. The benchmark tool also had an inherent design limitation where generating multiple requests over a certain threshold resulted in socket operation error. The error due to the benchmark tool is shown as follows:

```
[error] socket: unable to connect sock.c:249: Operation already in progress.
[alert] socket: polled(60) and discovered it's not ready sock.c:310: Connection
timed out.
```

From the table it can be seen that increasing the concurrency level for the same number of total requests resulted in a decreased rate of requests handled by ALG and NGINX reverse proxy. For the concurrency level of 1000, the performance of ALG and the reverse proxy is very poor, resulting in failure of requests by the back-end NGINX web server. For a normal web page, the number of requests per second is around 100 to 200. Also for the tests in which the requests fail due to a higher concurrency level, the time to complete the test is considerably higher than the other tests with the same number of total requests. The requests handled by ALG or NGINX reverse proxy server reduce from 4546 to 16 and 632 to 13 by increasing the concurrency level from 10 to 1000 respectively,

Test parameters		**Performance using ALG**			**Performance using NGINX**		
Concurrency level (c)	**No. of requests (N)**	**Failed requests**	**Time (s)**	**Requests per second**	**Failed requests**	**Time (s)**	**Requests per second**
10	1000	0	0.22	4546	0	1.58	632
100	1000	0	0.39	2564	0	0.54	1851
1000	1000	61	60.29	16	100	70.36	13
10	10000	90	1.26	7865	90	9.88	1003
100	10000	0	1.08	9259	0	4.62	2164
1000	10000	393	70.25	137	x	x	x
10	100000	990	11.84	8362	990	95.16	1040
100	100000	900	10.49	9447	900	34.14	2903
1000	100000	1191	65.28	123	x	x	x

Table 14: Stress Testing using Siege for HTTPS connections

A visual representation of the tabular results is shown in in Figure 37. Analyzing the 3 different bar graphs it can be deduced that increasing the concurrency level by

keeping the total number of requests constant, increases the requests per second. This is applicable only until the concurrency limit of the backend server is reached. After the maximum concurrency level, reverse behaviour is observed. Also it can be seen that using Siege as the benchmark tool, ALG can handle more requests than NGINX reverse proxy.

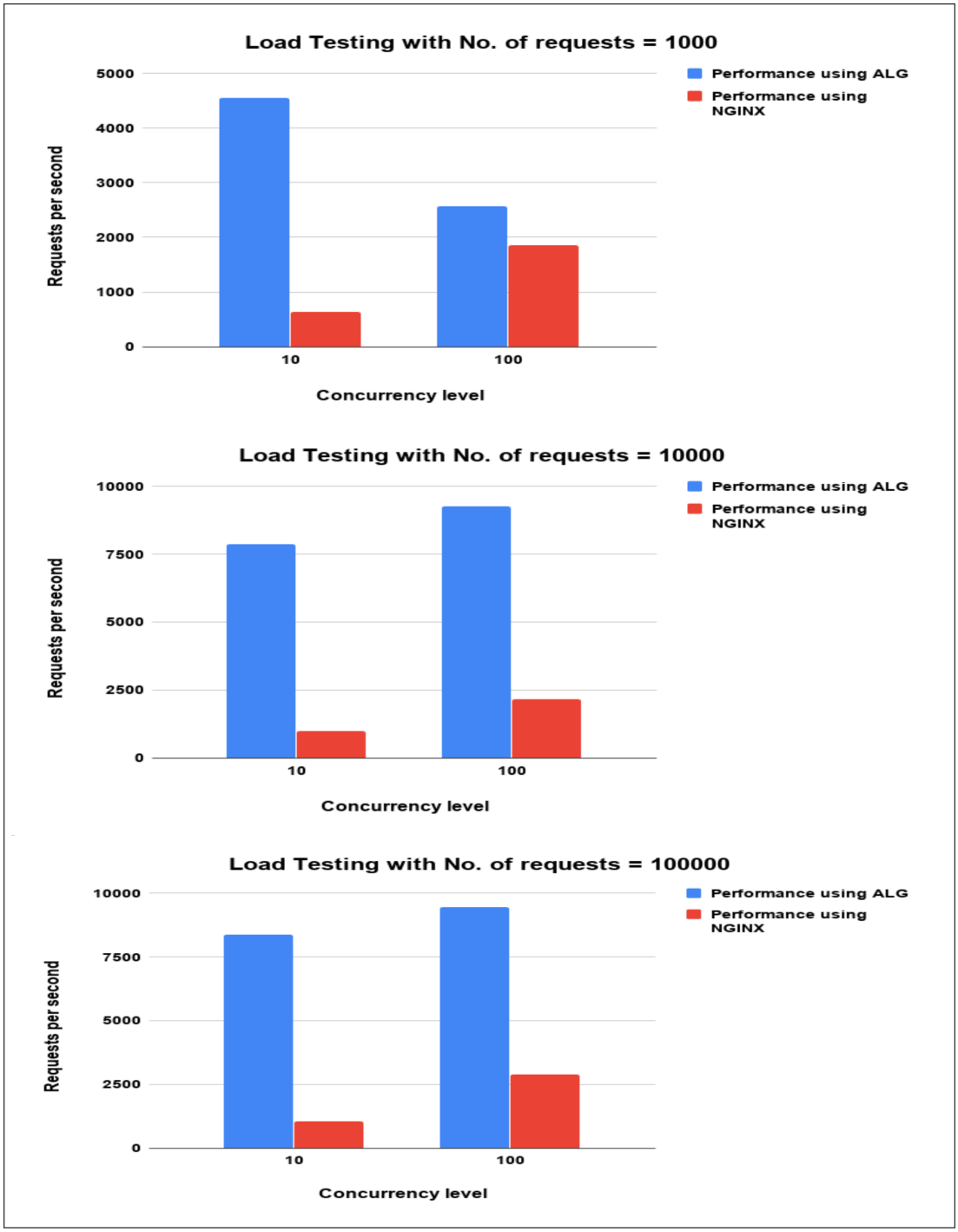

Figure 37: Testing the scalability for HTTPS using Siege

Comparing the performance results achieved using HTTP with HTTPS for ALG and NGINX reverse proxy, it can be seen that HTTP requests are handled at a much faster rate than the HTTPS requests due to the additional overhead associated with establishing an HTTPS connection. Since ALG does not decrypt the TLS traffic, it handles the requests at a much faster rate than NGINX reverse proxy hence the performance degradation when using HTTPS instead of HTTP is less in ALG.

5.4 Attack testing of ALG

This section focuses on evaluating the robustness of ALG against different kinds of attacks. It is seen that the proliferation of different firewalls and application layer gateways is accompanied by the increase in sophistication of different attack methods. ALG is designed to complement the operation of RGW and thus does not cater to attacks already mitigated by RGW such as TCP flooding. Using a series of tests we demonstrate the effectiveness of ALG against three different attacks namely DoS attack against the web server protected by RGW, resource exhaustion attack against the system running ALG and finally an attack against the hostname detection functionality of ALG. We explain the protection mechanism used for mitigating these attacks in each of the subsections.

5.4.1 HTTP DoS attack test

This section explains an application layer DoS attack carried out against the web server behind ALG. A Denial of Service attack is a type of attack in which the attacker aims to make the system or the network component unavailable to the legitimate users. This can be done by consuming all the resources of the system or by disrupting its operation. For the purpose of our testing, we used an application layer DoS simulator called SlowHTTPTest [49]. The tool is used to carry out a low-bandwidth HTTP DoS attack against the back-end server. The attack is used to exhaust a web server's resources by opening multiple connections towards the web server and keeping them alive for as long as possible. The attacker achieves this by sending HTTP headers periodically but never completing the HTTP request. As a result, the targeted web server is unable to serve other legitimate clients once its maximum connection count is reached.

The test setup consisted of a public client sending multiple HTTP requests using a range of ephemeral ports to the NGINX web server test.gwa.demo sitting behind RGW. The ALG is used for establishing the HTTP connections between the back-end server and the client. The attack exploits the design of HTTP protocol whereby the web server needs the complete request before it can be processed. Opening a large number of such connections can temporarily make the web service running on the back-end server unavailable for other users resulting in a denial of service. The command for initiating the attack is shown in Listing 1. The parameter c indicates the total number of connections initiated, i represents the time interval in seconds between subsequent partial HTTP requests for one connection, the

parameter x specifies the size of the partial HTTP request size while p indicates the time in seconds after which the probe connection determines the status of the availability of web service.

```
slowhttptest -c 20000 -H -i 10 -r 2000 -t GET -u http://www.test.
    gwa.demo -x 24 -p 2
```

Listing 1: SlowHTTPTest command

During the test, the number of connections was increased to 20000 at a rate of 1000 connections per second. The maximum window size advertised by the client was 24 bytes and it sent the partial HTTP request headers after 10 seconds. NGINX had a connection timeout of 60 seconds and if a complete HTTP request was not received within the specified time, the connection was closed. The results obtained from the test are shown in Figure 38.

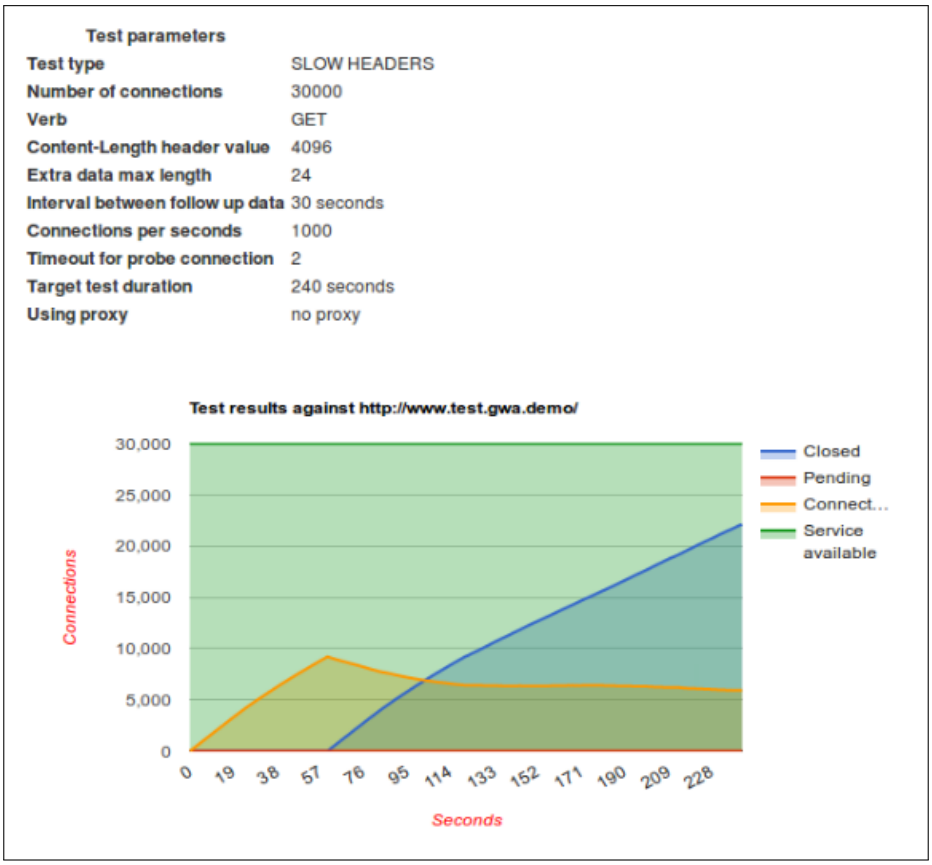

Figure 38: Results of the SlowHTTPTest tool

ALG's operation was verified in this test in two different aspects. First it was checked if ALG closes both halves of the connections properly after NGINX server has closed the connection. Second, the rate limiting functionality of ALG was verified where it doesn't accept more connections than the the connection limit specified in its configuration file. The maximum number of connections was set to 30000 as indicated by the graph. Since NGINX server closed all the those connections for

which the 60 second timeout was reached, the load on ALG was never more than 10,000 connections and hence the DoS attack was not successful.

It should be noted that ALG alone does not provide protection against the slow-loris attack. The back-end web server should have a default HTTP request timeout. To exhaust the resources of ALG, it was subjected to HTTP flooding using GET requests to verify its operating capacity in section 5.2.3. During the stress test mentioned in Section 5.2.3, some clients were unable to access the resource requested from the backend server but it was due to the back-end NGINX web server's inability to process the requests at a high concurrency level.

5.4.2 Disk exhaustion testing

Disk exhaustion attacks also considered as denial of service attacks, exploit the design vulnerabilities of the system and are a result of resource leakages. In order to test the log rotation functionality of ALG a simple test was performed. A log rotation scheme was employed in which a new log file was created if the size of the previous log file became larger than a specified size. The log file size was defined using a configurable parameter in ALG. The test verified if the log rotation was working perfectly and if the log file was available to different processes for logging.

For the purpose of testing, the maximum size of log file was changed to 104 KB and 10,000 HTTP requests were sent using 100 clients to retrieve the default web page hosted on test.gwa.demo. The log rotation worked as the file descriptor was accessed by different processes during the test and the size of the newly created log file was less than 104 KB as shown in Listing 2.

```
1 ubuntu@gwa:~$ ls -ahl Logging.txt*
2 -rw-r--r-- 1 ubuntu ubuntu  90K Sep  5 14:26 Logging.txt
3 -rw-r--r-- 1 ubuntu ubuntu 103K Sep  5 14:25 Logging.txt.old
```

Listing 2: Console showing log rotation

The current implementation of ALG does not store the backup of log files before overwriting them and hence there is no mechanism to retrieve the old log files that get overwritten. It is very important in servers that run over a lnog duration of time that the log is rotated to ensure that finding information related to a particular event is done easily.

5.4.3 Testing using idle connections

In this section we analyze the behaviour of ALG when the clients establish a TCP connection but do not send any application data. We consider a client connection whereby no HTTP request is sent after completing the TCP handshake as an idle connection. For N idle connections, ALG forks N processes and thus use system's resources. By initiating multiple idle connections, the attacker might try to deplete the resources of the ALG so that the maximum connection count handled by ALG is

reached. Consequently, the legitimate clients would be unable to access the web pages hosted on the web servers behind RGW and ALG. Thus a security mechanism is enforced to protect ALG against such an attack.

ALG relies on the hostname sent in the client request for establishing a connection with the back-end server. It parses the HTTP/HTTPS request and once the hostname is detected it uses the domain name for finding information about the IP address and port on which the web server is hosted. Thus for each TCP connection established between the client and ALG, one process is forked. To avoid exhausting system's resources, ALG has a connection timeout that is defined in ALG's configuration file. This timeout specifies the time ALG waits for the client to send application data. If no data is sent during the specific interval the connection times out and ALG closes the associated process.

To test if the connection timeout in ALG is working adequately, a public client was used for initiating a TCP connection on port 80 with RGW's public IP address. The connection timeout in ALG was specified as 2 seconds. Once ALG forked a new process to handle the connection, it waited for a duration of 2 seconds for receiving the application data. When no data was sent, ALG closed the connection. To verify the value of connection timeout, the clock time was noted when the client initiated the TCP connection with RGW and the time was again recorded when the client socket received an empty byte indicating the end of the response received by the ALG. Listing 3 is used to verify the connection timeout of ALG while the output of the console is shown using Listing 4.

```
import random
import socket
import datetime


sock= socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect(('100.64.1.130',80))
print('Time when initiating the connection\n', datetime.datetime.
    now())
sock.send(b'')
data=sock.recv(1024)
print('Time when socket closed\n', datetime.datetime.now())
sock.close()
print('Received', data)
```

Listing 3: Code to check the connection timeout in ALG

SPM

```
ubuntu@public:~$ python3 test_connection_timeout.py
Time when initiating the connection
 2019-09-05 19:59:46.725726
Time when socket closed
 2019-09-05 19:59:48.747545
Received b''
```

Listing 4: Console output for connection timeout

To simulate an attack, 100 clients on the public container initiated 40000 connections with RGW's public IP address on port 80 which were handled by ALG. The ALG closed all the processes when the connection timeout was reached. The process count never exceeded 2500 indicating that before opening new connections, ALG closed the previous connections. The size of the log file was increased to 100 MB and the log records for the test indicate that ALG closed the connection properly.

5.5 Policy Database Testing

In this section, we discuss how the integration to SPM has affected the performance of ALG and RGW. In the original design of RGW, the policies were stored using local files but the current design fetches the RGW's policies from SPM. Also previously there was no mechanism for updating the policies once loaded at the initialization of RGW. In the current design, an AIOHTTP client polls SPM every 10 seconds to retrieve the policies and updates the local copy of policies stored in the RGW as Python's dictionary object. A series of tests have already been performed to verify the operation and scalability of SPM [37] thus the purpose of this test was to analyze how ALG and RGW's operation is affected by the integration to SPM, particularly observing the time it takes to load the policies from SPM.

As mentioned in Section 4.5.3, another process is used for fetching the policies for ALG and storing them in a local file. ALG only checks the local file for updated policies after 10 seconds if the file has been modified thus integrating SPM with ALG does not add any additional overhead in the connection establishment process of ALG. Also only the policies regarding the hostname of the web servers are updated while the values of configuration parameters are not changed. However, RGW's policies are quite extensive and the time for retrieving the policies needs to be taken into account. Test environment constituted of the setup shown in Figure 22. SPM, ALG and RGW were all running as separate processes in the same container. The orchestration environment was run on host machine 3. A total of 463 firewall policies for 16 different hosts were retrieved for RGW. Additionally 181 policies regarding RGW's circular pool and NAT operation were also fetched from the MySQL database using SPM. All the configuration policies of ALG were grouped together into 1 policy and added in the database.

The time taken to retrieve the policies of ALG and RGW was recorded. RGW's policies were retrieved by RGW itself but ALG's policies were retrieved by another Python program executed as a separate process. The time taken to retrieve the policies was recorded in two different scenarios. During the first case, only one of the two software components was running and fetching the policies from SPM. In the second case, RGW and ALG were both started and the affect on the policy retrieval time was observed. The time was noted when the function call was made to fetch the policies from SPM. The time was again recorded after the policies were retrieved. The difference between the two values indicated the time taken to fetch the policies for ALG and RGW. Since the policies of RGW were stored in two

different databases of SPM to complement its original design, two AIOHTTP clients were used for accessing the data stored in different databases of SPM. The results are demonstrated in the Table 15. It must be noted that the time to retrieve the policies is an estimate as it uses an asynchronous system call to fetch the policies. Moreover, ALG's policies have less attributes which led to less retrieval time for the policies from SPM in comparison to RGW's policies. The retrieval time increased as the number of HTTP clients accessing SPM increased. New HTTP clients are used for each policy retrieval call and then terminated after retrieving the policies. For every call to the SPM using an AIOHTTP client, the SPM server uses a REST HTTP client to retrieve the policies from the MySQL database.

Setup	**Time taken (ms)**			
	All policies (ALG)	**Host policies (RGW)**	**CP policies (RGW)**	**All policies (RGW)**
Stand-alone software	9.85	102.08	23.47	126.44
RGW+ALG combined	10.63	146.09	25.94	173.55

Table 15: Time taken to fetch policies from SPM

To verify if the current design choices have affected the performance of RGW, the round trip time taken for one HTTP request was measured from the public client using sockets and datetime module of Python. When the RGW was not integrated with SPM and the policies were retrieved locally, it took 5.13 ms for the client to receive the HTTP response. Integrating the RGW and ALG with SPM and running the new source code, the measured response time for one request was 5.18 ms. The difference in response times was negligible and can be attributed to some random system variation; hence it proves that RGW's integration to SPM does not degrade its performance. Looking at the results obtained in Table 15, it was observed that retrieving ALG's policies took considerably less time than RGW's policies owing to their simplicity.

6 Conclusion

In this thesis, we extended the functionality of an existing software known as Realm Gateway. The main objective was to provide better support for application layer protocols particularly HTTP and HTTPS in RGW. Furthermore, the usability of RGW was improved by integrating it with an existing policy management system.

RGW addressed the NAT traversal problem by using a circular pool of publically accessible IP addresses for establishing end-to-end communication but it encountered connectivity issues when dealing with web traffic. To solve the connectivity problem, a reverse proxy was integrated with RGW that handled all the traffic for the web servers. When end hosts communicated using HTTPS, the reverse proxy required certificates and private keys of the back-end web servers for forwarding the connection correctly. Also the application data was decrypted and then re-encrypted before it was sent to the upstream web server protected by RGW. The decryption of traffic raised a security concern in addition to increasing the latency of the associated HTTPS connection.

To overcome the shortcomings experienced by the reverse proxy we developed an ALG. Our solution combined the functionality of a reverse proxy server and a parser-lexer to establish the HTTPS connection without decrypting the application data. Light deep packet inspection was done for the initial web traffic to determine the domain name of the destined web server. The inspection involved extraction of the hostname from the HOST header in HTTP while the hostname in TLS connection was detected from the SNI extension. On successful identification of the hostname, the connection was forwarded to the requested web server in the private network by the ALG. A multi-process architecture was employed in ALG for connection establishment whereby each connection was handled by two processes. The multi-process approach reduced the design complexity but the scalability of the solution is dependent on the hardware specification of the host machine, in particular the amount of memory which affects the maximum process count.

Moreover, to improve the usability of the RGW we integrated it to an existing policy management system called SPM. In our solution, the policies of RGW were retrieved from SPM and updated after every 10 seconds. Any changes in the policies were reflected in the operation of RGW. The configuration policies of ALG were also integrated with SPM to bring uniformity in the integrated software package that comprised of ALG and RGW. ALG was designed to be used in conjunction with RGW. However, it can be used as a standalone software independent of RGW.

Our evaluation of the performance of RGW with regard to the proposed extensions demonstrated that for HTTP connections the latency using reverse proxy was less than ALG. However, the difference in the measured latency could become negligible by using a more powerful processor. With regards to HTTPS connections, our solution outperformed the NGINX reverse proxy when using a powerful processor.

Furthermore, our experimental results showed that the scalability of our solution in terms of handling HTTP and HTTPS connection was significantly improved when using a system with higher processing capabilities in comparison with NGINX reverse proxy.

6.1 Future Work

Further studies should involve carrying out additional tests for evaluating the performance of ALG. Particularly, the concurrency level handled by ALG for HTTP and HTTPS connections should be verified by extensive experimentation. The performance of ALG should be observed in non-virtualized environment by simulating real network conditions. Future work could involve re-implementing ALG using a faster language to solve the performance issues encountered because of Python's single-threaded nature.

As discussed in Chapter 5, our solution is heavily dependent on the hardware specifications particularly the CPU power. For this reason, we believe our proposed solution could be further improved by changing the architectural design to become more independent. The multi-process architecture can be modified to handle one connection using one process or handle multiple connections using one process. Load balancing can be implemented among the different processes for handling multiple connections which can improve the scalability of the software. Another improvement could involve leveraging the RGW's reputation system for HTTP and HTTPS connections in ALG for allocating the resources to the web clients.

Future work could also involve studying the security aspect of the ALG. In our current solution, there is no provision for detecting web application attacks thus a possible extension could involve adding another component to mitigate the application layer attacks but that would require considerable time and effort.

References

[1] J. Stryjak, and M. Sivakumaran, "The Mobile Economy 2019," *GSMA Intelligence*, 2019. [Online]. Available: https://www.gsmaintelligence.com/research/2019/02/the-mobile-economy-2019/731/. Accessed: April 03, 2019].

[2] P. Richter, M. Allman, R. Bush, and V. Paxson, "A Primer on IPv4 Scarcity," *ACM SIGCOMM Computer Communication Review* **45**, no. 2, 21 – 31, 2015.

[3] S. Deering and R. Hinden. *Internet Protocol, Version6 (IPv6) Specification* RFC1883 (Proposed Standard) 1995. Obsoleted by RFC2460.

[4] K. Egevang, and P. Francis. *The IP Network Address Translator (NAT)* RFC1631, 1994.

[5] V. Fuller, T. Li, J. Yu, and K. Varadhan. *Classless Inter-Domain Routing (CIDR): An Address Assignment and Aggregation Strategy* RFC1519, 1993.

[6] J. Rosenberg, R. Mahy, and P. Matthews. *Session Traversal Utilities for NAT (STUN),* RFC5389, 2008.

[7] J. Rosenberg, R. Mahy, and P. Matthews. *Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN),* RFC5766, 2010.

[8] J. Rosenberg. *Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols,* RFC5245, 2010.

[9] H. Kabir, J. L. Santos, and R. Kantola. Securing the Private Realm Gateway, 2016, In *2016 IFIP Networking Conference and Workshops,* IEEE, 243 – 251, 2016.

[10] J. L. Santos, *Communicating Globally Using Private IP Addresses.* M.Sc. Thesis, Aalto University, Department of Communications and Networking, 1 – 80, 2012.

[11] I. Recommendation, *Information technology-Open Systems Interconnection-Basic Reference Model: The basic model*, 200 (1994)1 ISO/IEC 7498-1: 1994, 1994.

[12] M. Rose. *On the Design of Application Protocol.* RFC 3117, 2001.

[13] P. Waher, *Learning Internet of Things*, PACKT Publishing Ltd., 35, 2015. ISBN-10: 1783553537.

[14] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach and T. Berners-Lee. *Hypertext transfer protocol–HTTP/1.1 ,* RFC 2616, 1999.

[15] E. Casey, *Handbook of Digital Forensics and Investigation*, Elsevier Academic Press, 450, 2009.

[16] R. Braden. *Requirements for Internet Hosts – Application and Support,* RFC1123, 1989.

[17] M. Belshe, R. Peon and M. Thomson.*Hypertext Transfer Protocol Version 2 (HTTP/2),* RFC7540, 2015.

[18] "Usage Statistics of Default protocol https for Websites, June 2019", W3techs.com, 2019. [Online]. Available: https://w3techs.com/technologies/details/ce-httpsdefault/all/all. Accessed: June 27, 2019.

[19] R. Khare and S. Lawrence. *Upgrading to TLS Within HTTP/1.1 ,* RFC2817, 2000.

[20] J. Kennedy, M. Jacobs and M. Satran, "Public Key Infrastructure - Windows applications," Docs.microsoft.com, 2018. [Online]. Available: https://docs.microsoft.com/en-us/windows/desktop/seccertenroll/public-key-infrastructure. Accessed: June 27, 2019.

[21] J. M. Tilli, *YaLe. Software*, 2019. Available: https://github.com/Aalto5G/yale.

[22] Y. Rekhter, B. Moskowitz, D. Karrenberg, G. J. Groot and E. Lear, *Address allocation for private internets,* RFC1918, 1996.

[23] D. Wing, S. Cheshire, M. Boucadair, R. Penno and P. Selkirk.*Port Control Protocol (PCP),* RFC6887, 2013.

[24] H. Elgebaly and K. Phomsopha, "Communication protocols operable through network address translation (NAT) type devices," U.S. Patent 7 272 650, 2007.

[25] J. M. Tilli, *ldpairwall Software.* Available: https://github.com/Aalto5G/ldpairwall/commit/05f1d43f57fcb5487760b06c0563eb98269935cb.

[26] G. Tsirtsis and P. Srisuresh. *Network Address Translation - Protocol Translation (NAT-PT),* RFC2766, 2000.

[27] M. Allman, S. Ostermann and C. Metz. *FTP Extensions for IPv6 and NATs.* RFC2428, 1998.

[28] P. Srisuresh. *Security Model with Tunnel-mode IPsec for NAT Domains,* RFC2709, 1999.

[29] *Squid. Software*, Available: http://www.squid-cache.org/.

[30] J. M. Tilli. *CG-HCPCLI* Available: https://github.com/Aalto5G/cghcpcli.

[31] I. Liusvaara. *btls Software.* Available: https://gitlab.com/ilari_l/btls.git.

[32] "Policy," *Merriam-Webster.* [Online]. Available: https://www.merriam-webster.com/dictionary/policy. Accessed: July 4, 2019.

[33] B. Moore, E. Ellesson, J. Strassner and A. Westerinen. *Policy Core Information Model – Version 1 Specification.* RFC3060, 2001.

[34] Cisco, *Cisco Security Manager*, Cisco Systems, USA, 2018. [Online] Available: https://www.cisco.com/c/en/us/products/security/security-manager/index.html.

[35] A. Lara, B. Ramamurthy, "OpenSec: Policy-based security using software-defined networking", *IEEE transactions on network and service management* **13**, 30 – 42, 2016.

[36] S. Calo, I. Manotas, G. de Mel, D. Cunnington, M. Law, D. Verma, A. Russo and E. Bertino. "AGENP: an ASGrammar-based GENerative policy framework," *In Policy-Based Autonomic Data Governance* **11550**, Springer, Cham, 3 – 20, 2019.

[37] M. H. Mohsin, *Security Policy Management for a Cooperative Firewall.* M.Sc. Thesis, Aalto University, Department of Communications and Networking, 2 – 80, 2018.

[38] J. L. Santos, *Realm Gateway. Software*, 2018. Available: https://github.com/Aalto5G/RealmGateway.

[39] J. M. Tilli. *nmsynproxy Software.* Available: https://github.com/jmtilli/nmsynproxy.

[40] I. Sysoev, *"Configuring HTTPS servers,"* Nginx Inc, 2019. [Online]. Available: http://nginx.org/en/docs/http/configuring_https_servers.html Accessed: Aug. 16, 2019.

[41] M. H. Mohsin, *Security Policy Management . Software*, 2018. Available: https://github.com/Aalto5G/SecurityPolicyManagement.

[42] Apache HTTP Server Project. Available: http://www.apache.org.

[43] Siege. Software. Available: https://github.com/JoeDog/Siege.

[44] weighttp. Available: https://github.com/lighttpd/weighttp.

[45] SlowHTTPtest. Available: https://github.com/shekyan/slowhttptest .

[46] Slowloris. Available: https://github.com/gkbrk/slowloris.

[47] cURL. Available: https://curl.haxx.se/.

[48] G. Scrivano et al. , GNU Wget. Software, 2017. Available: https://www.gnu.org/software/wget/.

[49] SlowHTTPTest. Available: https://github.com/shekyan/slowhttptest