NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

CUSTOMIZING APPLICATION HEADERS FOR IMPROVED WARFIGHTING COMMUNICATIONS

by

Kenneth J. Pittner

September 2021

Thesis Advisor: Geoffrey G. Xie
Second Reader: Vinnie Monaco

Approved for public release. Distribution is unlimited.

THIS PAGE INTENTIONALLY LEFT BLANK

<table>
<tr><td colspan="4">REPORT DOCUMENTATION PAGE</td><td>Form Approved OMB No. 0704-0188</td></tr>
<tr><td colspan="5">Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington, DC 20503.</td></tr>
<tr><td>1. AGENCY USE ONLY
(Leave blank)</td><td>2. REPORT DATE
September 2021</td><td colspan="3">3. REPORT TYPE AND DATES COVERED
Master's thesis</td></tr>
<tr><td colspan="4">4. TITLE AND SUBTITLE
CUSTOMIZING APPLICATION HEADERS FOR IMPROVED WARFIGHTING COMMUNICATIONS</td><td rowspan="2">5. FUNDING NUMBERS
RCQ5H</td></tr>
<tr><td colspan="4">6. AUTHOR(S) Kenneth J. Pittner</td></tr>
<tr><td colspan="4">7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-5000</td><td>8. PERFORMING ORGANIZATION REPORT NUMBER</td></tr>
<tr><td colspan="4">9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
NIWC PAC, San Diego, CA 92152</td><td>10. SPONSORING / MONITORING AGENCY REPORT NUMBER</td></tr>
<tr><td colspan="5">11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.</td></tr>
<tr><td colspan="4">12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release. Distribution is unlimited.</td><td>12b. DISTRIBUTION CODE
A</td></tr>
<tr><td colspan="5">13. ABSTRACT (maximum 200 words)
Currently, U.S. Navy shipboard communications have a great disadvantage: the data rates of satellite links are limited, typically below 4 Mbps for each link. Improving efficient utilization of these links while out to sea is paramount to maintaining our military advantage. Also, any improvement must be transparent to end user functionality. This thesis first explored implementing a free version of a commercial-off-the-shelf wide area network (WAN) optimizer, Artica, on a simulated shipboard network consisting of three local area networks (LAN). Artica works by performing auto-corrections on some web traffic and changing the transmission control protocol (TCP) window sizes. Results from browsing Alexa’s top 1,000 websites on the LANs show that Artica can speed up web traffic by 13–26% at link speeds between 1.544 and 8 Mbps. It then explored compressing Domain Name System (DNS) traffic by filtering out IPv6-related queries and removing unused fields of DNS queries and responses. Experimental results show that DNS compression did not significantly improve web traffic performance, which highlights the importance of selecting traffic-intensive applications to compress and control compression-induced processing overhead. Finally, the thesis explored whether Artica and the custom DNS compression program can be deployed together. In summary, this thesis shows that using WAN optimization techniques and saving bits over a slow data rate link can effectively speed up web traffic.</td></tr>
<tr><td colspan="4" rowspan="2">14. SUBJECT TERMS
network, networking, compression, efficiency, satellites, SATCOM, protocol compression</td><td>15. NUMBER OF PAGES
83</td></tr>
<tr><td>16. PRICE CODE</td></tr>
<tr><td>17. SECURITY CLASSIFICATION OF REPORT
Unclassified</td><td>18. SECURITY CLASSIFICATION OF THIS PAGE
Unclassified</td><td colspan="2">19. SECURITY CLASSIFICATION OF ABSTRACT
Unclassified</td><td>20. LIMITATION OF ABSTRACT
UU</td></tr>
</table>

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release. Distribution is unlimited.

CUSTOMIZING APPLICATION HEADERS FOR IMPROVED WARFIGHTING COMMUNICATIONS

Kenneth J. Pittner
Lieutenant, United States Navy
BS, U.S. Naval Academy, 2014

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
September 2021

Approved by: Geoffrey G. Xie
Advisor

Vinnie Monaco
Second Reader

Gurminder Singh
Chair, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Currently, U.S. Navy shipboard communications have a great disadvantage: the data rates of satellite links are limited, typically below 4 Mbps for each link. Improving efficient utilization of these links while out to sea is paramount to maintaining our military advantage. Also, any improvement must be transparent to end user functionality. This thesis first explored implementing a free version of a commercial-off-the-shelf wide area network (WAN) optimizer, Artica, on a simulated shipboard network consisting of three local area networks (LAN). Artica works by performing auto-corrections on some web traffic and changing the transmission control protocol (TCP) window sizes. Results from browsing Alexa's top 1,000 websites on the LANs show that Artica can speed up web traffic by 13–26% at link speeds between 1.544 and 8 Mbps. It then explored compressing Domain Name System (DNS) traffic by filtering out IPv6-related queries and removing unused fields of DNS queries and responses. Experimental results show that DNS compression did not significantly improve web traffic performance, which highlights the importance of selecting traffic-intensive applications to compress and control compression-induced processing overhead. Finally, the thesis explored whether Artica and the custom DNS compression program can be deployed together. In summary, this thesis shows that using WAN optimization techniques and saving bits over a slow data rate link can effectively speed up web traffic.

THIS PAGE INTENTIONALLY LEFT BLANK

Table of Contents

1 Introduction **1**
1.1 Problem Statement 2
1.2 Research Questions 2
1.3 Thesis Organization 3

2 Background **5**
2.1 Network Traffic Compression 5
2.2 Virtual Networks 6
2.3 Protocol to be Explored 10
2.4 DNS 11
2.5 Related Works 13

3 Experimental Design **15**
3.1 Testbed Design 15
3.2 Algorithmic Choices 18
3.3 Selection of Performance Metrics 21

4 Implementation and Integration **23**
4.1 Experimental Configurations 23
4.2 Experimental Flow 26
4.3 Results 27
4.4 Chapter Summary 30

5 Conclusion **35**
5.1 Future Work 36

Appendix A Network Setup **39**
A.1 Create the User Machines 40
A.2 Create the Routers 41

Appendix B Source Code **45**
B.1 create_traffic.py 45
B.2 for_all_start.sh 47
B.3 trials.sh 48
B.4 tc.sh 48
B.5 start.sh 49
B.6 control_run.sh 49
B.7 kill_all.sh 50
B.8 finish_check.sh 51
B.9 shore dns_shore_server.py 52
B.10 ship dns_server.py 54

Appendix C Artica Setup **59**

List of References **61**

Initial Distribution List **65**

List of Figures

Figure 2.1 WAN Notional Concept. Source: [7] 6
Figure 2.2 NAT Example. Source: [9] 8
Figure 2.3 SSH Communication Depiction. Source: [15] 10
Figure 2.4 DNS Process Illustration. Source: [16] 12
Figure 2.5 DNS Header Illustration. Source: [18] 12

Figure 3.1 Shipboard Topology Map 16
Figure 3.2 DNS Process Illustration. Adapted From: [16] and [18] 19

Figure 4.1 Baseline Topology Map 24
Figure 4.2 Artica Topology Map 25
Figure 4.3 Flow Diagram 26
Figure 4.4 2 Mbps Box Plot 28
Figure 4.5 300 Mbps Box Plot 28
Figure 4.6 Comparisons of Configurations by Network Speeds 32
Figure 4.7 CDF Comparisons by Experimental Configuration 33

Figure A.1 Basic Network Configuration 39
Figure A.2 Artica Network Configuration 40
Figure A.3 sysctl.conf Sample Image 42
Figure A.4 setup_iptables.sh 43
Figure A.5 setup_iptables.sh For ship_gateway_router 43

Figure C.1 Artica TCP Configuration 60

THIS PAGE INTENTIONALLY LEFT BLANK

List of Tables

Table 4.1 HW/SW Configuration Table 23

Table 4.2 Network Configuration by Experiment 24

Table 4.3 DNS Compression Ratios 29

THIS PAGE INTENTIONALLY LEFT BLANK

List of Acronyms and Abbreviations

ASCII American Standard Code for Information Interchange

CANES Consolidated Afloat Network and Enterprise Services

CDF cumulative distribution function

CDN content distributed network

COTS commercial-off-the-shelf

CPU central processing unit

DoD Department of Defense

DNS Domain Name System

DNSSEC Domain Name System (DNS) security

FQDN Fully Qualified Domain Name

GUI graphical user interface

HTTP Hypertext Transfer Protocol

HTTPS Hypertext Transfer Protocol Secure

ID identification

IDE integrated development environment

IP Internet Protocol

IPTables Internet Protocol (IP) Routing Tables

IPv4 IP version 4

IPv6 IP version 6

LAN local area network

MAC media access code

MB Megabytes

Mbps Megabits per second

NAT network address translation

OS operating system

RAM random access memory

RD recursion desired

SDN software defined network

SSH Secure Shell

TCP transmission control protocol

UDP user datagram protocol

URL uniform resource locator

US United States

USN United States (U.S.) Navy

VM virtual machine

WAN wide area network

Acknowledgments

Carol and Arya: Thank you for always supporting me. It has been a long process with many hardships, but there are no two other people I would rather go through them with than you two. I could not have done it without you. To the work here and work ahead: Fortis Fortuna Adiuvat.

Professor Xie: Thank you for all of the lessons going through this process and the mentoring. It has been a great experience!

Dan Lukaszewski: For helping me tweak programs, scripts, and always asking why; it drove me to be better and understand more about virtual networks. Thank you!

Mom and Dad: Without the foundations you taught me as a child, I would not be where I am today.

The Three Amigos: It has been great to go through NPS with great friends like you two. Thank you for all the laughs and cheer while here.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 1:
Introduction

Demands for faster networks to handle more traffic have been ever increasing for the past decade. While home- and shore-based facilities have been able to keep pace with demand, ships at sea have not. Current speeds for the United States (U.S.) Navy (USN) are still below 10 Megabits per second (Mbps) from a dedicated satellite link. Given the limited transmission capacity, an important technical question is how to maximize the performance of the link with respect to a chosen performance metric or metrics.

This thesis simulates a ship network in a virtual network and investigates the link performance question while focusing on application latency as the primary metric. For example, application latency for a web browser characterizes the delays the browser takes to download web pages. Broadly speaking, there are two ways to reduce application latency. First, by requiring fewer bits to retrieve the same information, transmission and receiving delays can be reduced. This method is commonly referred to as data compression. Second, by keeping information close by and easier to access, the number of communication hops and round trips can be reduced. This reduction method is commonly referred to as data caching.

Many applications are decades old using a high amount of bits to communicate information. As is observable in network traffic, most of these bits do not change or can be inferred by other information. These superfluous bits in the applications offer areas that can be cut, reduced, or compressed to save on the total number of bits moving over a given data link. Domain Name System (DNS) is one of these applications. DNS was originally built to accommodate a request with a varying number of queries: greater than or equal to one. However, according to Andersson and Montag, analyzing network traffic reveals that DNS requests typically only carry one query [1]. With this being the case, it is easily conceivable that many of these bits in a DNS packet can be compressed or cut, the packet sent, and then the packet reassembled at some distant end. This reassembly will ensure the packets are interoperable with the rest of the Internet when the expected information is returned. The answer to the query is almost a copy of the original request, with a little more information added, namely the Internet Protocol (IP) address. This close copy also means these answers are able to be compressed on their return; further increasing bit savings. In this thesis,

the virtual machines (VMs) do not cache DNS information so as to generalize to other applications that do not cache data.

Another solution to compressing network traffic is by utilizing a wide area network (WAN) Optimizer. At a high level, it sits near the edge of a private network, close to the gateway router, and exists to increase performance and speed of traffic. There are a number of ways they achieve this speed up. Mainly, they either do types of calculations for fastest routes, compress application information within the packet.

1.1 Problem Statement

With a majority of USN ships utilizing a satellite link with data rates less than 4 Mbps and their enterprise networks setup similarly to a business with an overwhelming amount of web traffic, more efficiency is needed across this link to enhance communications. The hypothesis of this thesis is that using WAN optimizers and reducing the amount of bits across a satellite link while still communicating the same information will yield an increase in efficiency to communication speed.

1.2 Research Questions

Currently, CISCO estimates 194.4 Exabytes of data move monthly on networks globally [2]. To counter this foreseen problem, a lot of research has been done in the past to make network traffic more efficient. Of note, researchers for delta encoding and transmission control protocol (TCP) compression show decent strides in reducing the amount of traffic across the network, yielding a higher throughput [3]. Also, research to improve DNS has had some success. For example, bundling multiple DNS requests into one message before transmission ultimately saves four to twelve bytes per message [4].

This technique proves useful for intranets and internal networks, and has great potential to help the USN's unclassified web browsing as it relies on some public architecture. The benefits of making these paths more efficient is to increase speed and throughput of a network by freeing space for other transmissions. Also, benefits made and discovered working with unclassified network traffic can easily be translated over to classified traffic with minor modifications.

Assumptions made in order to do this compression is that the entire transmission path must be owned by one owner before being forwarded to the rest of the public internet. Also, the compression work is done around the bottleneck to provide the maximum benefits. To this end, this thesis assumes no caching and no added packet loss outside of the congestion the three local area networks (LANs) will create. This thesis also assumes that all the LANs concentrate into one satellite link, consistent with enterprise networks currently deployed by the USN.

Outside of Hypertext Transfer Protocol (HTTP) traffic and the DNS technique mentioned, little has been done to make other network applications more efficient. For the researched applications, only one or two techniques have been tested leaving room for more research and testing. This thesis will develop a custom compression algorithm for DNS and will conduct a proof of concept for slower communications paths to specifically answer these two questions:

- Does compressing DNS traffic increase network performance and throughput for lower data rate network links?
- How does a commercial-off-the-shelf (COTS) WAN optimizer product perform to speed up network traffic?

Through our study of application-level protocols, we are going to answer the following additional questions:

- How does information get removed from a DNS, transmitted, and re-expanded to convey the same information invisible to the end user?
- Can the WAN optimizer and the custom DNS compression be combined and how do they perform together?

1.3 Thesis Organization

Chapter 2 begins by discussing IP Network Traffic Compression; specifically a WAN Optimizer. The chapter discusses Virtual Box, general information about it, and how it responds to setting up an overload network address translation (NAT) and IP Routing

Tables (IPTables) on an Ubuntu host virtual machine. Then, it goes into running Python 3, specifically running it inside a bash script and networking with it. It delves deeply into in depth protocol information to be studied and compressed. Finishing, it highlights related works and previous research completed on compressing network application protocols.

Chapter 3 investigates the choices behind various implementations and network designs, and then details the one chosen for this thesis. First, a baseline is established for normalcy expectations and the various ways to generate traffic. Then, it details the psuedo-code for customized compressing of DNS queries and responses. Finishing Chapter 3, the algorithm for calculating results is given, specifically how to calculated the compression ratio for DNS runs.

Chapter 4 details the execution flows of the various experiments. It also gives the individual network changes that need to be made in between those experiments to do the minimal network reconfiguration. It concludes with the results of experiments to include: baseline, WAN Optimizer, DNS Compression, and the WAN Optimizer with the custom DNS Compression.

Lastly, Chapter 5 discusses this thesis's conclusion and potential future research projects in this area of networking.

CHAPTER 2:
Background

This chapter provides essential background details to software and protocols used in this thesis. It begins by delving into network traffic compression, more specifically compression as it relates to a WAN Optimizer. Then, the components of the employed virtual network, specifically including Virtual Box and Ubuntu. Moreover, technical methodology such as NATing, manipulating IPTables, Bash Scripting, and Secure Shell (SSH) will also be defined and their uses in these experiments will be described. The penultimate topic will be the DNS protocol under investigation by this thesis. This chapter concludes with an in-depth discussion related works in this area of research.

2.1 Network Traffic Compression

Network Traffic Compression is a key enabler to increasing throughput without increasing the data rate for them. There is a lot to be gained from compression techniques. Many studies, already, "have examined real-world network traffic and concluded the presence of considerable amounts of redundancy in the traffic data" [5]. This topic will only gain more recognition due to continuously growing demand for data movement across an internet with limits to data transfer at bottlenecks.

2.1.1 WAN Optimizer

A WAN is a network that has a router that utilizes IP addresses to route traffic. It is more advanced than a simple LAN that only uses media access code (MAC) addresses to local traffic. WAN Optimization is one of many areas to attempt to increase network performance and compress traffic. Ultimately, the optimizer's goal is to conduct complex network calculations to determine the fastest way to route IP traffic. Typically, it runs on a software defined network (SDN) technology for simplification purposes [6]. One team's approach ran an Optimizer on the SDN controller, thereby having the controller handle the complex calculations [6]. When implemented correctly, WAN Optimizers help increase bandwidth utilization, prioritize traffic, and increase throughput on a network via their heavy duty calculations and congestion control. Figure 2.1 depicts the notional concept of a WAN

Optimizer on a private network and illustrates the data flow on the network. The dotted line represents packets traversing networks from a client in a private network to some public server. Specifically, the image depicts a WAN Optimizer sitting near a gateway router, and performing work to increase speed of network traffic.

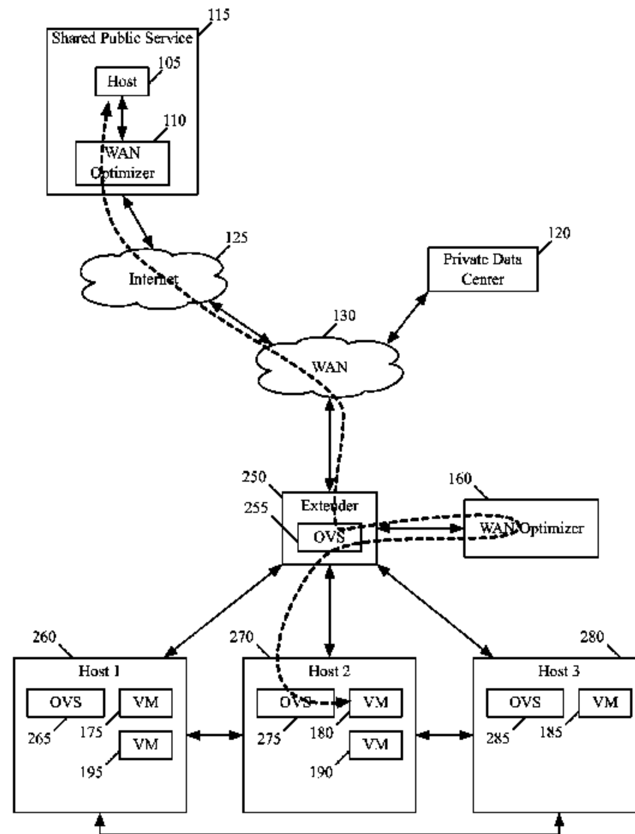


Figure 2.1: WAN Notional Concept. Source: [7]

2.2 Virtual Networks

Virtual networks have similar components to physical networks; however, multiple virtual servers will run on only one physical server. Thus making management of the network easier and faster to communicate changes across it. VirtualBox is an enabling application for virtual machines and networks chosen for this thesis.

2.2.1 VirtualBox

Virtual Box is an open source product owned by Oracle. It is completely free to use and easy to install on Mac. It enables running another operating system on the Mac computer. Additionally, multiple operating systems can be run simultaneously, up to the limits of the Mac hardware for a specific machine. VirtualBox works by allocating disk space on the host machine to load another operating system (OS) image file. Once the new OS is instantiated, a portion of the host computer's resources are allocated for the new virtual machine instance. While easy to use for simple uses, it is also easy to search on Google for answers or read their instruction manual for more complex set ups, such as this thesis' network.

Ubuntu OS

Ubuntu is an open source variation of Linux; it is designed to run on a multitude of platforms, easy to manipulate, and change settings to suit the needs of individual users. While it is possible to do everything using the graphical user interface (GUI) a user gets when they first log in to the OS, it is much preferable to use the command line terminal. The latter ensures changes are absolute and the user receives more detailed feedback of the OS actions. Throughout this thesis, instances are used as standard central processing unit (CPU)'s and routers to simulate a USN ship's network. Herein, instructions will be given in this thesis to set up these instances for the experiment.

2.2.2 NAT

NAT is used to change the IP address of a LAN packet to a common set of IP addresses apportioned to the WAN from a higher tier authority [8]. It is typically used to connect private networks to public, allowing for the re-use of IP addresses and assigning packets with a globally unique address, enabling more robust routing [8].

Figure 2.2 shows a schematic diagram of a router running NAT to translate the private IP to a public IP before sending the packet on to the rest of the Internet. From the client perspective, an IP source address is added to the packet it sends off. Once that packet makes it to the router, it is saved and the router puts its public IP address on it while remember where the packet came from. This way, from the public perspective, all the packets came from one IP address; thereby eliminating the need for multiple public IP addresses for one

network. Then, once the answer returns for the client, the router will put the client's address back on the packet.

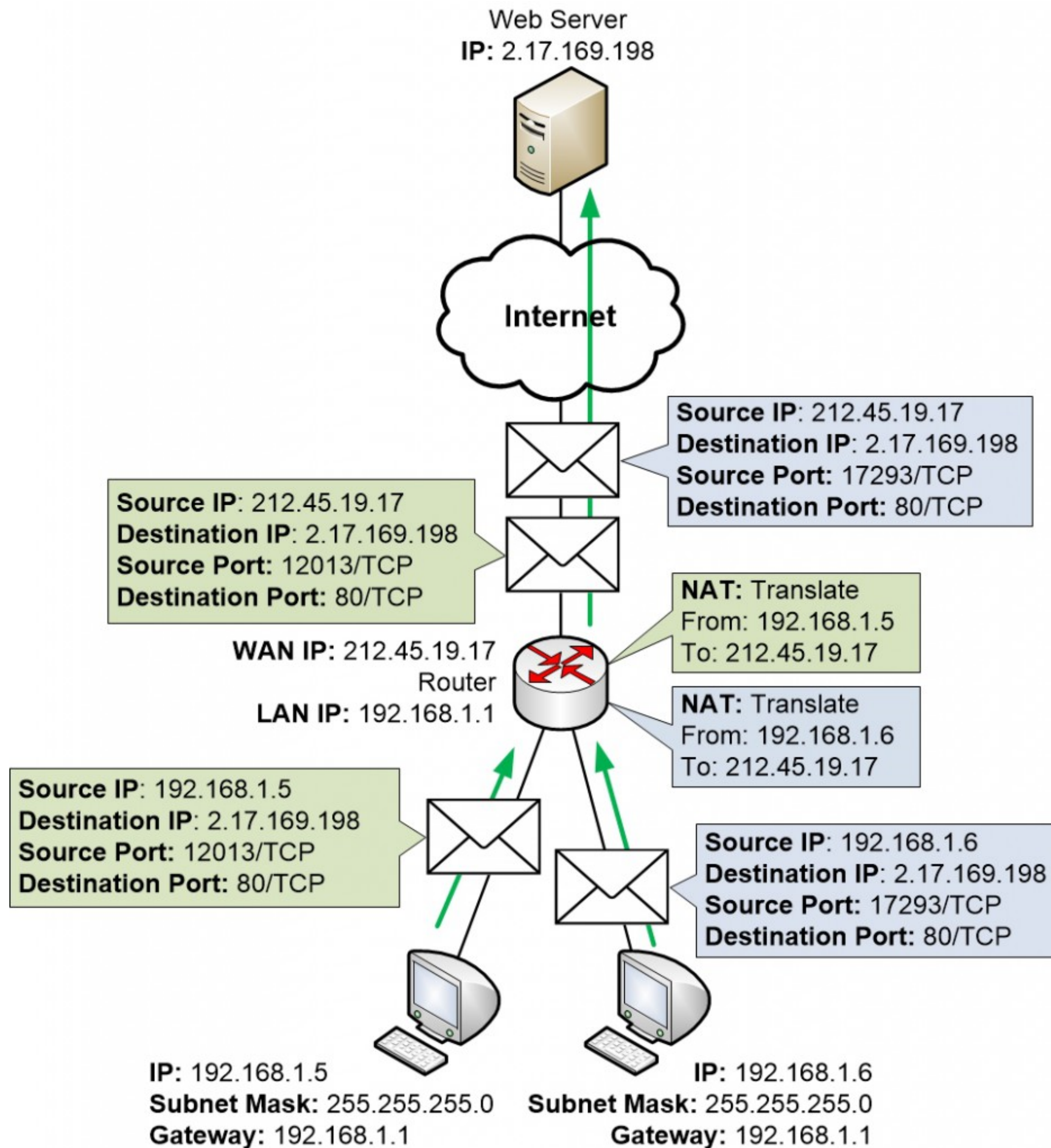


Figure 2.2: NAT Example. Source: [9]

2.2.3 IPTables

The command "iptables" is used to manipulate the firewall and OS filter rules on Linux flavored systems. It is an important enabler to change how the OS sends IP traffic [10]. Most applicably for this thesis, the iptables command provides the ability to explicitly state how to route IP traffic to the next hop in the virtual network. For this thesis, the iptables command is utilized to build the routes internal to the virtual network. Without this command, none of the machines would communicate with each other.

2.2.4 Python 3

Python 3 is an open source, interpreted, high-level, and object-oriented language for computer programming [11]. The language is very versatile, enabled by a plethora of libraries that can be loaded to work in any area on a machine or network. To further make it easier to code in Python 3, many integrated development environments (IDEs) exist now to handle Python 3 and error check the coder as they work.

Networking

Networking in Python can be handled in a multitude of ways. The first way is to import the OS library to get regular command line instructions to feed the computer through Python. While working for basic functions such as ping and traceroute, it does not succeed for deep packet inspections of traffic because most OS's are not built to access that protected memory very easily. To enable this capability through the command line OS functions is not feasible due to the level of difficulty.

A superior approach is to use libraries created for Python 3, such as IO and Scapy. Specifically, the IO library provides the Python library the commands to fetch inbound and outbound data streams [12]. Using the IO library, a programmer can access the network traffic coming in and out of the computer and move it to memory. The Scapy library allows for manipulation of network packets [13]. Together, Scapy and IO are powerful and allow for obtaining control, manipulating, and sending packets in and out of computers. For this thesis, these libraries will be critical to compressing and de-compressing the network traffic.

2.2.5 Bash Scripting

Bash scripting is crucial to this thesis. In order to generate sufficient network traffic for data and statistical trends, bash scripting must be employed. At its basic level, bash scripting utilizes the command line language interpreter, bash, and allows automating commands [14]. For this thesis, the bash script must be run with "super-user" level permissions, sudo; otherwise the network traffic will not be collected. The appendices of this thesis will provide the utilized bash code.

2.2.6 SSH

SSH is a protocol used for remote log-in from another computer. It utilizes logical port 22 on a machine to communicate and is established using the client-server model [15]. Once logged in, a user receives a command shell to execute commands on the remote machine. This capability is particularly useful for user access, and convenient operation.

Figure 2.3 illustrates the process for client log-on to an SSH Server. The term SSH server is generic in this instance, any machine can run an SSH server. For this thesis, all Ubuntu machines have an SSH server running on them to make it easy to log in to them via SSH.

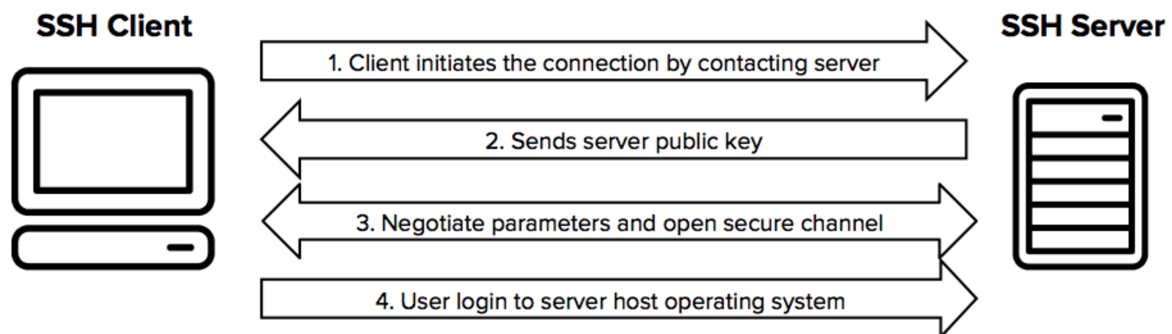


Figure 2.3: SSH Communication Depiction. Source: [15]

2.3 Protocol to be Explored

At their most basic level, networking protocols are an agreed upon ordering of bits in network traffic that have been standardized. This standardization enables communication and a list of rules to handle certain traffic. Users can bend or break those rules; however, they run the risk of the public Internet not understanding, deciphering, nor processing

unconventional data transmissions. Therefore, if users expect to communicate with the rest of the world, they must put packets back into standard form. This thesis will ensure that process is faithfully completed so the network can communicate with the public.

2.4 DNS

The DNS protocol is critical for network operations. Users unknowingly employ it whenever accessing the internet. It is the protocol responsible for associating Fully Qualified Domain Name (FQDN) and IP addresses. DNS allows users to remember something easy such as www.google.com, the FQDN, rather than 172.217.12.68 or 2607:f8b0:4000:80d::2004, the IP version 4 (IPv4) and IP version 6 (IPv6) addresses, respectively. One special item to note: when a user goes to www.cnn.com or www.foxnews.com, the DNS protocol is not only used to determine the IP addresses of those websites, but also used to determine all of those sites' embedded images and advertisements. This extended utilization quickly compounds and leads to network traffic congestion.

DNS relies on a user-server model, where the server's usage is hierarchical. Most enterprise networks employ a DNS server that receives unique query identifications (IDs) from the user's machine and then handle the request if the answer is unknown. Once the answer is obtained, the DNS server will send back an answer to the user's machine with the unique ID to match the query. Figure 2.4 provides and illustrates the process in which a client goes to a DNS server for query, requesting a recursive look up. The recursive lookup will ask the server to do any future lookups and simply pass the ultimate answer back to the user. Until the server receives the answer, it conducts the iterative lookups, whereby it goes to DNS servers where it thinks the answers are, asking every time and being redirected to another DNS server until it receives the ultimate answer.

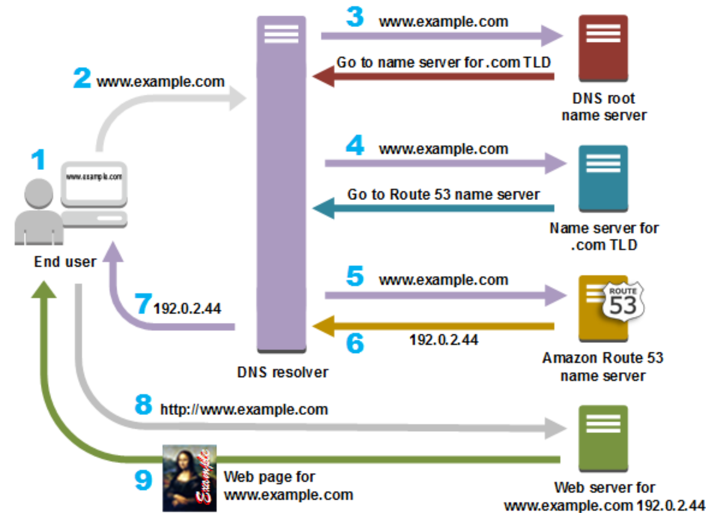


Figure 2.4: DNS Process Illustration. Source: [16]

While a very useful protocol, DNS's first RFC publication dates all the way back to September of 2000 [17]. As such, it has a high overhead number of bits and the queries are in American Standard Code for Information Interchange (ASCII) value, adding even more bits for plain text letters. To compound the issue and add congestion to the network, the vast majority of DNS queries and answers traversing the network only have one query. So, accessing that one website and all of its advertisements and embeds increases the number of queries, and increases the number of unnecessary bits moving on the network. Figure 2.5 is an illustration of the DNS Protocol Header.

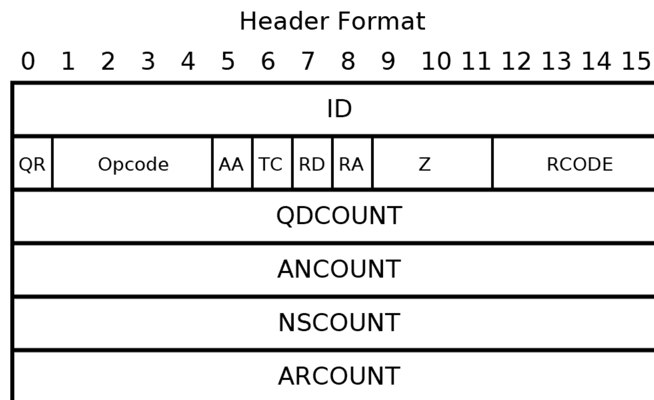


Figure 2.5: DNS Header Illustration. Source: [18]

There are many bits within the header that can be cut and compressed to save the total number of bits traversing networks. An example of the DNS header is shown in Figure 2.5. Many of the header fields are only required for either the query or the answer. An example is the recursion desired (RD) flag. It is only needed by the client asking the question to ask its DNS server to do all the iterative requests. Also, the counts of the various queries and answers can be inferred, so the bits to give those numbers is not needed in transmission. Similarly, the Opcode assignment is there for giving whether it is a query or the type of answer that is being sent back to the questioner [19]. These opcodes are also easily inferred and able to be cut to save bits.

2.5 Related Works

Much work has been done in compressing data traversing a network to attempt to increase efficiency for some protocols; HTTP being chief among them. In recent years, a compression scheme for HTTP, called HTTP/2 has been developed and started to be fielded [20]. It works by encoding the most common header field values in an exchange so as to reduce the amount of bits for the headers. For the uncommon, or not frequently used field values, the literal strings are sent [20]. The table that stores the most common header field values is dynamic; it keeps a count of the values and will ensure the most common values are continually up to date in the network exchange.

A team at Orange Labs in France researched how much HTTP/2 actually saves in traffic [21]. What they discovered is that for major websites, it has the potential to save a lot of time for the exchange. However, for smaller websites, it is not worth the overhead. Also, they discovered that on mobile networks, it actually hurts performance due to the instability of the physical network, i.e. cell phones going in and out of range to a cell tower [21].

However, not nearly as much has been done for the other protocols. Although past work to improve efficiency has not focused as much on other protocols, there have been some improvements in DNS and other general network traffic. Two of these improvements are briefly discussed in this section.

For DNS, Krische and Klauck have made the protocol more efficient by combining multiple queries into one packet, thereby reducing the amount of overhead if these were sent

individually [22]. They also reduced the number of bits in some fields in the transport and IP layers, such as the time to live value for the packet. On average, they saved roughly four to twelve bytes per message [22]. While initially that does not seem like much, if that is extrapolated over the hundreds of DNS queries that move across the network every second, that can be a lot of bytes saved and not sent on the network.

DNS security (DNSSEC) has also been making strides in not only security, but also speed of information. The original goal of DNSSEC was to provide security to DNS information being transmitted so attackers could not maliciously modify or deny it [23]. A new technique for it is to distribute the secure information to make it faster due to it being closer to the host [23]. However, a challenge here would be to compress that traffic further depending on the security choices made such as encryption. It would again require full ownership of the transmission path so the client and server would know the compression algorithms used and be able to understand each other.

For all of the protocols, Amdahl at F5 Networks, Inc developed a compression of all data transmitted on the network and patented it [24]. It is very efficient at saving bits by encoding many used portions into a few bits [24]. It achieves this highly compressed traffic at the cost of complexion. According to the patent, he created a middle-ware box that examines the network traffic on the wire, creates a fitness function to select chunks of data, and then sends specific chunks in data structures over the internet to be received by another middle-ware box for interpretation [24]. His invention requires heavy computing to create the fitness function and continually analyze it to ensure that correct data is going across the network. It also requires advanced data structures capable of holding a lot of information and searching through them quickly. While this solution is feasible for large enterprise networks, it is not feasible for USN ships; the processing power on the enterprise network for Navy ships is not present. Therefore, while this thesis will look similar to his network setup, it will not require or perform such heavy powered computing.

CHAPTER 3: Experimental Design

This chapter will detail high-level design decisions toward developing a testbed to test configurations, and to evaluate COTS WAN optimizers and DNS traffic compression. It will start by describing the setup of the network and the rationale for generating test traffic. It will then provide details for the various compression algorithms to be tested. This chapter will conclude with a detailed description of the performance metrics.

3.1 Testbed Design

3.1.1 Network Design

At the high level, the testbed of this thesis simulates a USN ship's network. From the networking perspective, USN ships typically have three major enclaves that feed traffic into one gateway router that has a reach-back link to shore. The three major enclaves are the enterprise networks to facilitate work at all security levels. They look and act similarly to any business network for any big corporation.

After leaving the ship, the signal relays through a satellite, goes to a ground station, and then goes to the shore side's router to be routed to the public internet. While the satellite-out path for the network is interesting, this thesis will focus more on the ship network and the satellite link itself.

Currently on the ship side of the network, the Navy is working on virtualizing more components of the ship's enterprise networks. The Navy started with and has rolled out Consolidated Afloat Network and Enterprise Services (CANES), moving a majority of the network core components to a virtual environment [25]. This thesis aligns with the Navy's intention to improve enterprise networks throughout the Fleet. Also, in a virtual environment, the command and control of this network is easier to execute experiments and move and save data for analysis.

The advantages of running this thesis on a testbed that is completely virtual is further

amplified by a complete simulation of OSs on hardware with enough CPU power and random access memory (RAM) to completely run all five OSs. The hardware, Mac computer, has the prerequisite capabilities to run this thesis.

To simulate a ship, this network has three LAN Ubuntu machines that represent the three major enclaves and two Linux machines, converted to routers, that represent the ship's gateway router and shore's router. This thesis also chose to utilize virtual technology to make the entire testbed run in a virtual environment (running on VirtualBox). While an alternate way to conduct this experiment may have been to implement every piece of gear in hardware, and execute the testing in that manner, it would not be aligned to the Navy's direction for the future. The setup is given in Figure 3.1 with a more in detail setup given in Appendix A.

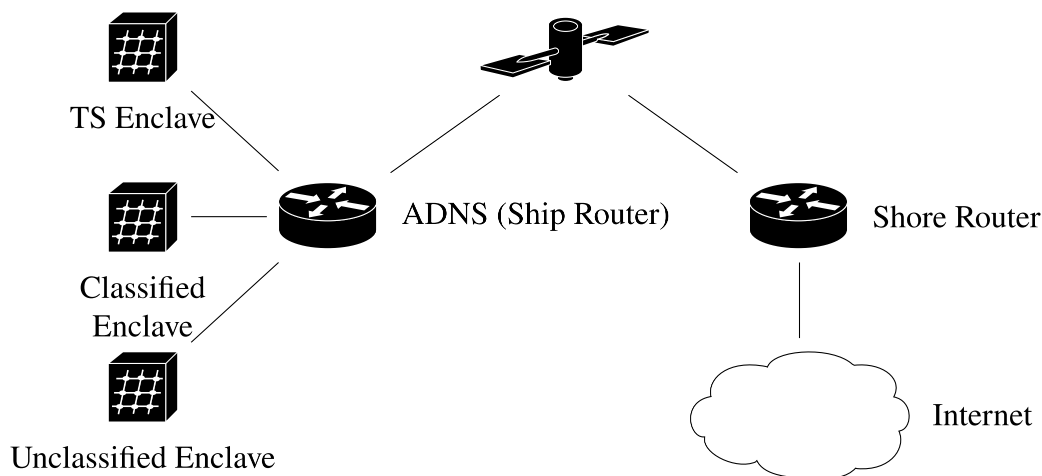


Figure 3.1: Shipboard Topology Map

3.1.2 Traffic Generation

In order to generate network traffic, the three LANs were visiting the Alexa's top 1000 websites on a top internet sites list under various conditions and configurations to test [26]. Of note, the list was slightly modified to remove many explicit websites that would not have been authorized on USN networks. This amount of network traffic is required for two reasons: accurate measurements and statistical power for these experiments. Without this amount of traffic, it would be difficult to determine whether the success or failure was a fluke, or if it was able to be repeated. This need is due to the constantly changing routing

environment on the Internet. Routes on the Internet are constantly being recalculated for better routes, so what may have been the route one time, may not be the route in the future. Therefore, it needs to be repeated multiple times to prove results.

Another way existed to generate traffic that was not plausible. Traffic captures on ships could have been taken, and then replayed on this test bed to give full fidelity to a ship's traffic. This choice presents a multitude of challenges. First, portions of that traffic on the ship are classified, so it could not successfully reach out to the public internet and return anything. This thesis does not simulate this traffic exactly. However, classified traffic routes similarly to unclassified traffic in that it has servers and applications run from the ship to the shore. In this case, the "public" servers are simply managed by the Department of Defense (DoD).

Second, the unclassified traffic is already accounted for. A majority of the traffic on a ship's enterprise network is what would exist on any business network: requests sent to news sites, Facebook, GMail, etc. This thesis has already accounted for it by utilizing the top websites list that would include these popular websites among Sailors anyway.

In this thesis, designing and running three LANs is a critical choice. It keeps true to the standard configuration on Navy ships where they have three major LANs. This design is beneficial because the three LAN VMs can generate traffic as if they had multiple hosts running on them; this constant visiting of the top websites. However, the drawback of this design is that these LANs do start to act similarly to hosts. They will cache previously visited, such as web and DNS information, for a limited time similar to any OS on currently on the market. This drawback is nullified because the time delay between visits is enough in that there should be enough of a change to cause the python get to behave as if it were going for the information the first time. Another drawback that is prevalent through this design choice is that all the VMs are running and generating traffic on the same host. This problem causes delays in processing throughout the experiment. However, the delays do not impact the times taken for the python get because of the way that python script is coded to mark the time, execute the python get, and then immediately mark the time again. The code is also marked to timeout after ten seconds for the python get. Ten seconds is used because in this author's observations, a majority of users will give up and either click refresh or give up entirely on getting to the website after that time period.

A final choice to fully simulate a network is in between every python get there is a random sleep time. This time simulates multiple users requesting information at various times. Seed files were created for each run on each LAN, and each file has 1000 random times.

3.2 Algorithmic Choices

There exist a range of choices to compress or eliminate network traffic. First and foremost, there exists tools to compress the TCP traffic between two points to reduce the amount of transmitted bits [27]. For example, there are various WAN optimizers, such as Artica [28], that achieve this compression. Secondly, another choice for compression is application specific compression, with HTTP traffic as the most preferred target for compression. Since most HTTP traffic is sent in plain-text (very costly on network resources), tools exist to compress the HTTP traffic being sent to again reduce the amount of bits being transmitted.

Caching is also a popular technique utilized to reduce the amount of traffic sent across a network. Every machine caches a small amount of data and most business networks have servers that cache as well, so they do not have to go out for information that has not changed for a short period of time. Examples of caches include DNS information and web pages for a limited time.

This thesis utilizes the WAN optimizer Artica. It was chosen for its ease of install and plug-n-play ability. It also has ample online support for install and troubleshooting issues, however not much was disclosed in how Artica worked. It was observed, though, to work by auto-correcting get requests made by the client to limit the number of round trips to get the same information, thereby making the traffic seem faster. Also, Artica is configurable to adjust the TCP window size to allow for the maximum amount of data to be requested from the server as possible. The benefit is all of Artica's work is transparent to the end user.

The DNS compression algorithm chosen for these experiments herein is due to the overwhelming unmet need to improve this application; recent improvements are not as plentiful as those for HTTP/Hypertext Transfer Protocol Secure (HTTPS). This application is challenging because of the communications required as discussed in Chapter 2. If a DNS server on the internet gets a message in an improper format, then the request will not be answered, thereby causing a communication failure and unsuccessful fulfillment of the end

host request. Likewise, if a client gets an improperly formatted message, it will not accept the information contained thereby causing a communication failure. Therefore, the easiest way to save bits for the application is to create a technique for this unmet need.

Figure 3.2 illustrates the compression algorithm. The DNS compressor operates by receiving a DNS request from a client on the ship gateway router, verifying it is asking for the “A” record (IPv4), reading the FQDN and transaction ID, and then sending those two pieces of information along in a user datagram protocol (UDP) packet to the shore gateway router. If the request was for the “AAA” record (IPv6), it immediately creates a "service not available packet and sends that to the client so it stops asking for that record and clogging up the in-pipe. The client requests both records separately because if the DNS server does not handle “AAAA” records, it will return an error for the whole set of requests to that one transaction ID, including the A record [29]. By sending these requests separately, it guarantees at least one will be answered for the client.

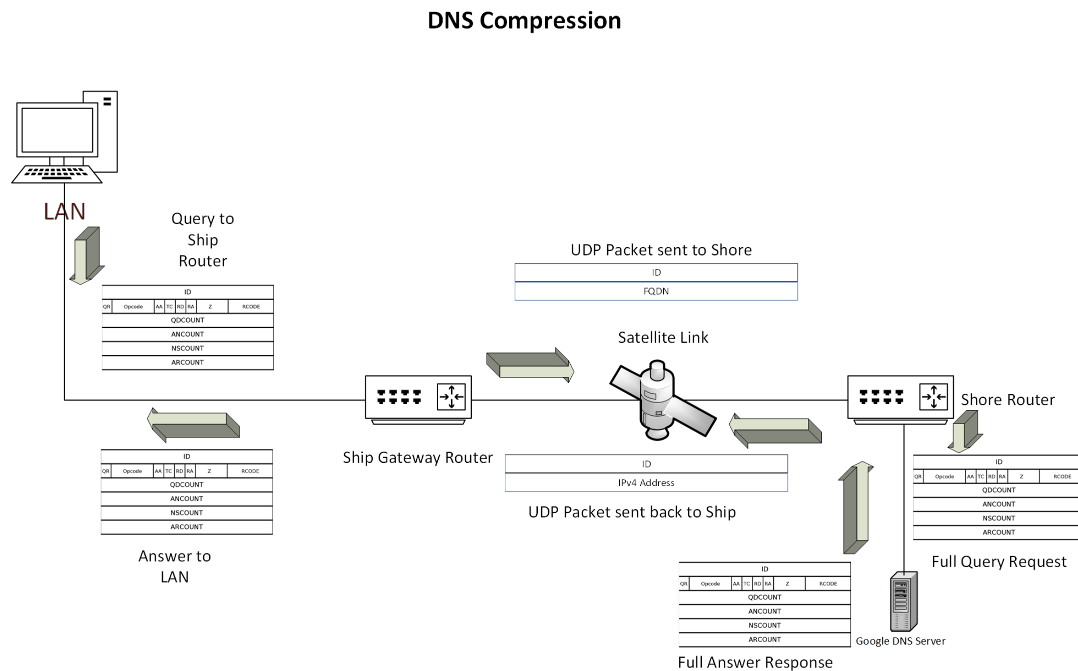


Figure 3.2: DNS Process Illustration. Adapted From: [16] and [18]

While waiting for a response from the shore gateway, the ship saves the pertinent information from the client’s request into a dictionary (using the transaction ID as the key) for lookup later: port sent from by the client, IP address for the request, and the FQDN.

The reason only the transaction ID and FQDN are sent from the ship to the shore is because they are the only unique pieces to DNS Queries. The remainder of the bits, seen in Figure 2.5, are not needed in the query. The Opcode field can be ignored, along with the other flags that come after the OPCODE. They all can be standardized to always ask for an authoritative answer and perform how the overwhelming majority of DNS queries perform in Internet traffic.

The shore, after receiving the transaction ID and the FQDN, performs the IPv4 lookup of the FQDN using Google's public DNS server. After receiving the information, sends a UDP packet back to the ship with only the transaction ID and the IP address of the FQDN. The rest of the information in the answer can be cut out. First, the ship only needs one IP address to connect to for the website, not the plethora that usually come back in an answer. Also, with the information stored on the ship's DNS compressor, the query can already be reassembled by the ship, so that information can be cut out of the response from the shore to the ship. The Opcode is not needed because the end user will not read it unless there is an error, and the remainder of the flags can be set by the ship since they are standard with any successful DNS answer.

Once the ship receives the two pieces of information from the shore, the transaction ID and IP address, it does a lookup in its dictionary for the previously saved information, and then takes that information as well as what was the UDP packet and creates a properly formatted DNS response packet containing the answer to the original query, and sends that back to the LAN. The advantage of this method is it strips out all of the unneeded information and flag bits along the link that is the most resource constrained.

This thesis will also explore combining the two previous techniques: TCP compression and DNS compression. These two techniques will be combined to explore if there is any benefit in combining techniques. If there is a benefit, it will also explore the net increase in traffic speed. This dual-targeted approach is being explored because it will ultimately serve as a proof of concept for these combinations.

3.3 Selection of Performance Metrics

There are many ways to measure performance for this thesis. One simple method is to compute a running average of web browsing delays for each simulated LAN. Once the experiments are finished, then the average could be stored and plotted for later use. This method does not provide enough fidelity for how much the network performance may vary based on times of day, and change during the experiment. Therefore, more measurements need to be taken throughout the experiment. This thesis records the time for every "python get" out to the website to provide accuracy for every single outreach. This way, more accurate trends can be established and calculated. More formally, let S denote the set of distinct `python get` instances executed in an experiment. For each $w \in S$, we derive the running round trip time of w based on $t_1(w)$ and $t_2(w)$ where w is the python get command, two timestamps collected right before and after the `python get` execution. The timestamps have a resolution out to six decimal places for accuracy.

$$r(w) = t_2(w) - t_1(w) \tag{3.1}$$

Additionally, we collect data to measure traffic reduction of the DNS and HTTP compression algorithms. More formally, for each $w \in S$, we record $b_1(w)$ and $b_2(w)$, the total byte counts of all targeted control messages (i.e., DNS queries or HTTP/HTTPS GETs) pertaining to w before and after compression, respectively. We derive a compression ratio for w as follows.

$$c(w) = \frac{b_1(w)}{b_2(w)} \tag{3.2}$$

There is more on the network than just the targeted DNS and HTTP messages. To be an honest broker of information, all traffic should be captured since the WAN optimizer will compress all TCP traffic. This thesis does this by using "tcpdump" to collect every packet traversing the network at the ship gateway router for two reasons: that particular router has visibility for every packet on the network and utilizes NAT, which hides the specific LAN the packet came from after it, so it needs to be captured there. With that visibility, the calculations will be more comprehensive and include all the traffic seen during these experiments.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 4:
Implementation and Integration

This chapter will first present configuration details for the experiments. Then it will explain the flow of the experimental execution order. It will conclude with the results of the experiments: baseline, Artica (WAN Optimizer), customized DNS compression algorithm, and a joint deployment of Artica and the DNS compression algorithm.

4.1 Experimental Configurations

There are four major experiments executed for this thesis. A detailed, step-by-step instruction set on exactly how to set up the VMs and the baseline network can be found in Appendix A. All of the supporting code, bash scripts, and python programs to run the experiments can be found in Appendix B.

Table 4.1 summarizes the hardware and software configurations for each of the VMs per experimental configuration. The differences will be highlighted in sections 4.1.1 - 4.1.4. Of note, 2048 Megabytes (MB) were used for the LANs' memory due to machine limitations. Extending above this VM MB limit would have exceeded the physical machine's capacity to simultaneously handle all of the required processes and its own OS.

Table 4.1: HW/SW Configuration Table

VM	OS	Memory (MB)	Bridged Network Adapters	Processors
Shore Router	Ubuntu 20.04	4096	2	2
Ship Router	Ubuntu 20.04	4096	4	2
LAN1	Ubuntu 20.04	2048	1	1
LAN2	Ubuntu 20.04	2048	1	1
LAN3	Ubuntu 20.04	2048	1	1
Artica	Debian 64-bit	4096	2	2

Table 4.2 summarizes the network configurations for VMs that change for each of the experiments in sub-sections 4.1.1-4.1.4.

Table 4.2: Network Configuration by Experiment

Experiment	LANs Gateway Router (DNS Server)	Ship Gateway Router (DNS Server)
Baseline	Ship Router (8.8.8.8)	Shore Router (8.8.8.8)
Artica	Ship Router (8.8.8.8)	Artica Interface .3 (8.8.8.8)
DNS	Ship Router (Ship Router)	Shore Router (8.8.8.8)
Artica w/ DNS	Ship Router (Ship Router)	Artica Interface .3 (8.8.8.8)

For each configuration, there are 12 runs completed where each run consists of each LAN visiting 1000 websites at the throttled link speeds by utilizing the "tc" commands: 1.544 Mbps, 2 Mbps, 4 Mbps, 8 Mbps, 300 Mbps.

4.1.1 Baseline Experiment

The baseline network is set up in accordance with Figure 4.1. Its purpose is to establish a baseline for the hardware and software this experiment is running on; it will give a control so the differences in the configurations can be discovered.

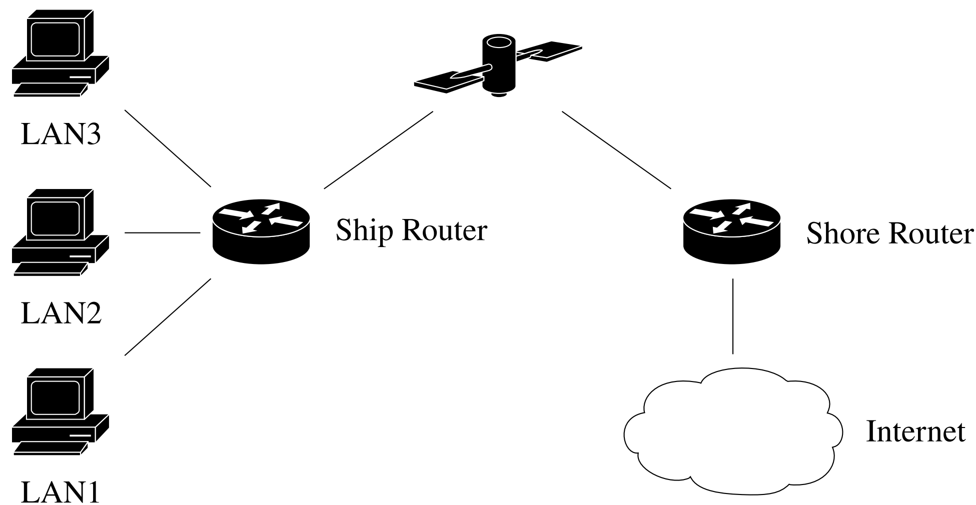


Figure 4.1: Baseline Topology Map

4.1.2 Artica WAN Optimizer Experiment

The Artica WAN Optimizer configuration is in accordance with Figure 4.2. Of note, its major difference is that the Artica WAN Optimizer is set up between the ship and shore router. In this experiment, Artica is simulated as being on the ship, so the link between Artica and the shore is the satellite link and is throttled. The ship router also uses Artica as its gateway router.

This setup is the best approach because any WAN optimizer will have the most impact on the ship. It is closest to the machines and servers working, so bits can be saved before the satellite link.

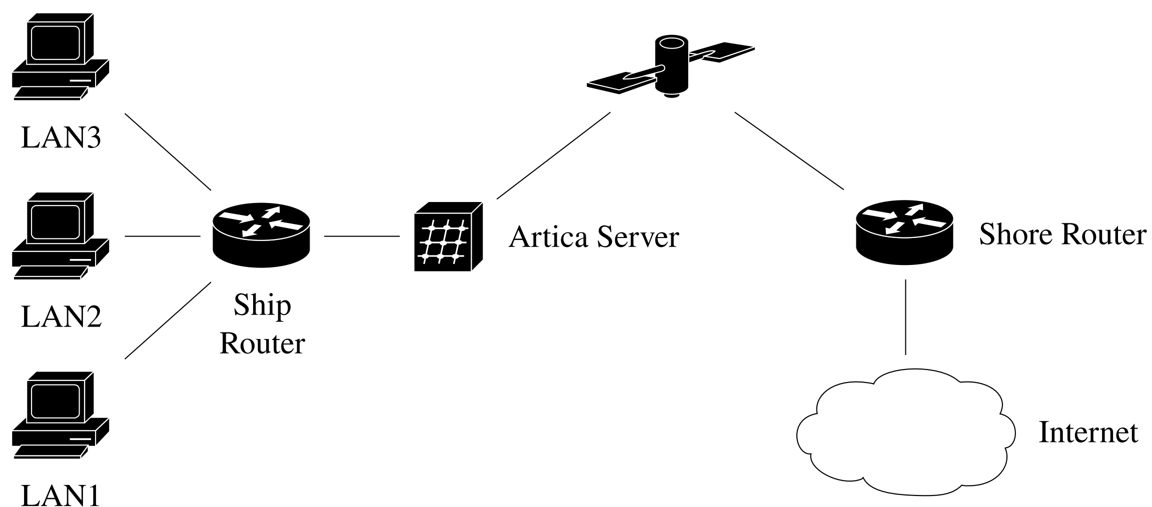


Figure 4.2: Artica Topology Map

4.1.3 DNS Compression Experiment

The DNS Compression Experiment network is set up similarly to the baseline network. The only difference is that the LANs use the ship gateway router as their DNS servers, and as such, send their queries there. The DNS code running on the ship gateway router will then catch and handle the uniform resource locator (URL) queries.

4.1.4 DNS Compression with Artica Experiment

This experimental network is set up similarly to the Artica Experiment configuration, and has the LANs utilizing the ship gateway router as their DNS server. This configuration is

utilized to see how much, if any more savings can be seen with these two working together.

4.2 Experimental Flow

The logic of each experiment follows the flow described in Figure 4.3.

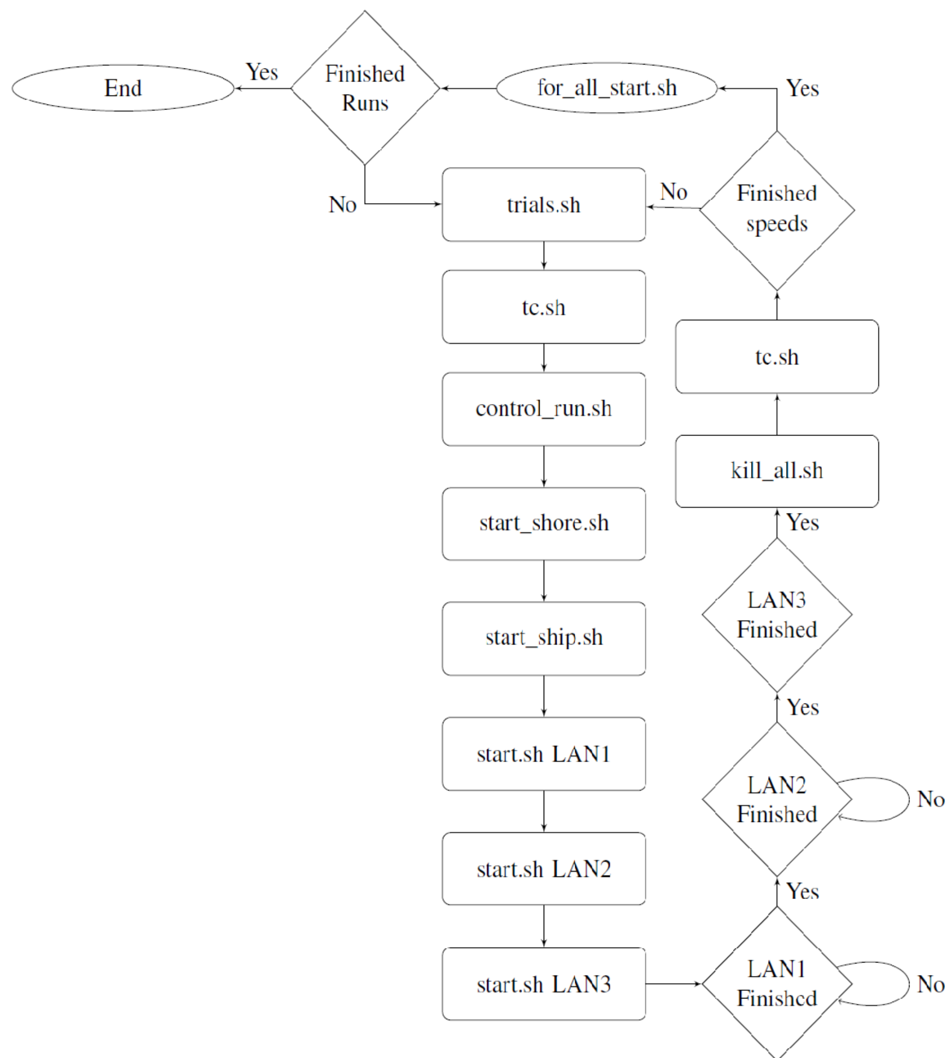


Figure 4.3: Flow Diagram

1. for_all_start.sh is executed on the Ubuntu ship gateway router. This program exists to start the whole experiment run and call trials.sh.

2. trials.sh controls each run through the experiment. It is ultimately called 12 times per experiment and calls control_run.sh and grab_files.sh as it moves through each of the network speeds.

3. control_run.sh starts tcpdump on the ship and shore routers, and then calls start.sh on the LANs.

4. start.sh starts create_traffic.py on each LAN which will generate all the of the "python get" requests and eventually writing the metrics.

5. After control_run.sh sees that all three LAN have finished generating their traffic, kill_all.sh is called to kill all the tcpdump commands running.

4.3 Results

The experimental configurations produced some expected and unexpected results. Overall, the data show that link performance, whether compression is through a COTS WAN optimizer or a self-made algorithm, is worth the effort. Also, the 95% confidence interval for each of the data sets is +/- 0.03 seconds. This confidence interval shows that the data collected is highly reliable.

4.3.1 Artica Results

As can be seen from the speed comparisons in Figures 4.4, 4.5, and 4.6 the Artica configuration consistently retrieved web pages faster than the baseline configuration for speeds less than 300 Mbps, ranging from 13-26% faster. At 300 Mbps they performed about the same, only about 3% slower, which makes it clear that it is no longer worth the processing time to compress the traffic. It is much faster to send the bit at that speed: the break over point was reached. After this break over point, it could potentially hinder speed and slow down the overall network.

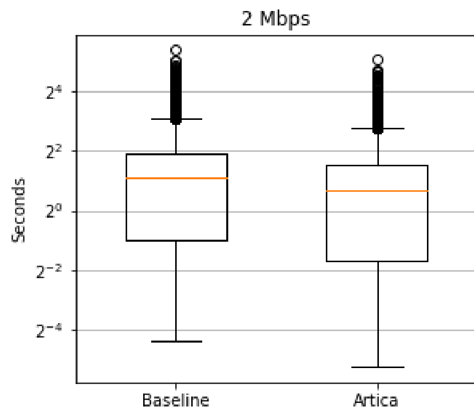


Figure 4.4: 2 Mbps Box Plot

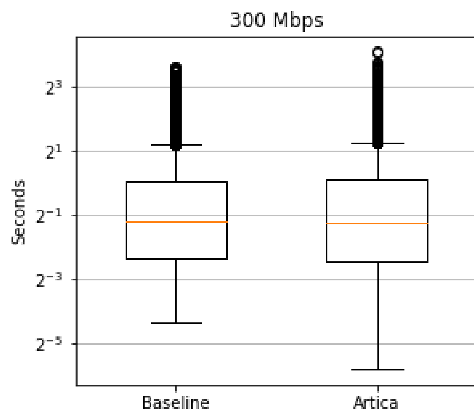


Figure 4.5: 300 Mbps Box Plot

The speedup in Artica can be attributed to their geographically dispersed infrastructure, similarly to a content distributed network (CDN), in order to provide timely auto-correction of HTTP URLs and adjustment of the TCP window size. Upon studying the network traffic transiting across Artica, it revealed that Artica analyzes the URLs, and if there is any error with the URL, it automatically corrects it in the packet and then sends the updated packet along, hence the auto-correct feature. An example is changing a simple GET www.youtube.com, which would trip an correction question from the server, to GET www.youtube.com/index.html to ensure no confusion on the server's end. This lack of confusion and explicit direction prevents multiple round trips, thereby shortening the overall get time for that website.

Figure 4.7 provides the cumulative distribution function (CDF) plots for the various experimental configurations. As these plots show, Artica consistently has a higher probability of having a majority of its web browsing faster than the other configurations. Also, the DNS configuration probability lines match closely to the baseline configuration lines, therefore showing that DNS compression helps, but not significantly.

As expected, Figure 4.7 shows a correlation that the faster the network data rate, the faster the python get time is. This point helps show that the network did perform as expected and none of the configurations caused any major slowdown that would cripple a network.

4.3.2 DNS Compression

First and foremost, the proof of concept was a success for compressing DNS traffic. The results prove that it successfully compressed DNS queries and responses all the while staying invisible to the simulated users. This success also proves that customization could be accomplished for any application protocol so long as it is uncompressed before leaving the owned network heading to the rest of the public Internet. Furthermore, it shows that self-made customization can start to compete with commercial optimization products, especially at slower data rates that the USN still utilizes.

Table 4.3 summarizes the compression ratios and does not take IPv6 savings into account for the queries and answers.

Queries were consistently compressed by 16 bytes if they were requesting the A record, or not accepted if they were requesting the AAAA record. The only variance in the query is how long the FQDN was which accounts for the variance in the compression ratios. The longer the FQDN, the less that was technically saved since the 16 bytes accounted for less in the network packet.

Table 4.3: DNS Compression Ratios

DNS Compression Ratio	Queries	Responses
Min	1.16	1.35
Median	1.36	2.05
Mean	1.36	2.24
Max	1.46	8.44

For the response ratios, there is a much higher variance. This variance is due to multiple number of IP addresses that could be contained in the original answer. While this thesis only sent back the first IP address received at the shore router, there usually were multiple contained in the response. The major byte savings did result in fewer bytes being sent across the satellite link.

Unexpectedly, the DNS custom configuration performed on par with the baseline at slower speeds, and then at faster speeds, performed worse. It only had a 6-8% increase for speeds 1.544-4 Mbps and performed 8-30% worse for higher speeds compared to the baseline. These results clearly show that the slower the speed, customizing to compress traffic pays dividends. After the breakover point is reached, it is not longer worth the processing delay to compress the traffic; it is faster to simply send the original packet.

The results also show that targeting a more traffic intensive application protocol, such as HTTP or HTTPS could potentially yield much better results. DNS may represent too small of a portion of the overall traffic to significantly impact network performance.

As more evidence that DNS traffic may be too small a portion to significantly impact the overall performance, the Artica with DNS configuration shows a speed up comparatively to the regular DNS configuration, 3-6% for speeds less than 8 Mbps. Artica, and its performance, appears to be the dominant factor to speed gains.

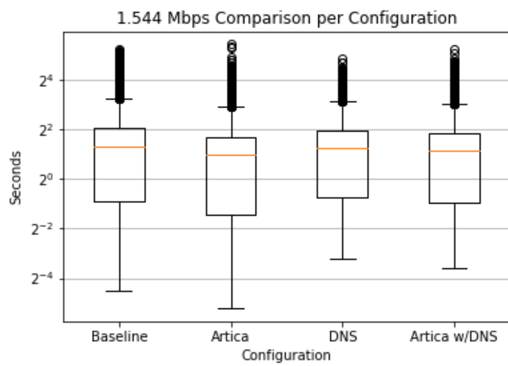
4.3.3 Artica with Custom DNS Compression

The Artica server and custom DNS compression worked together at slower speeds. As another proof of concept, the results show that it is possible to combine a COTS WAN optimizer and customized compression algorithms. Figure 4.6 also shows that it performed better than the baseline at slower speeds. The extra processing incurred by the DNS algorithm resulted in no significant speedup of web browsing, and worse performance at link rates above 4 Mbps.

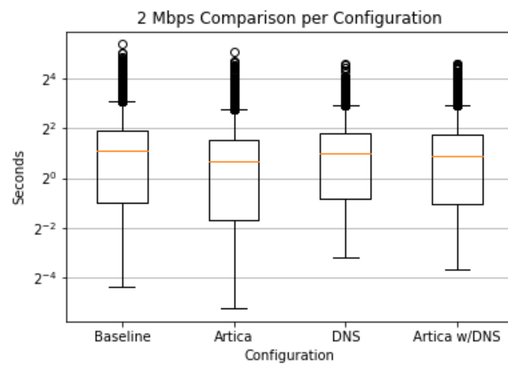
4.4 Chapter Summary

In this chapter, the experimental configurations and main logic flow were discussed. Specifically, it detailed how the network changed from one configuration to the next before more

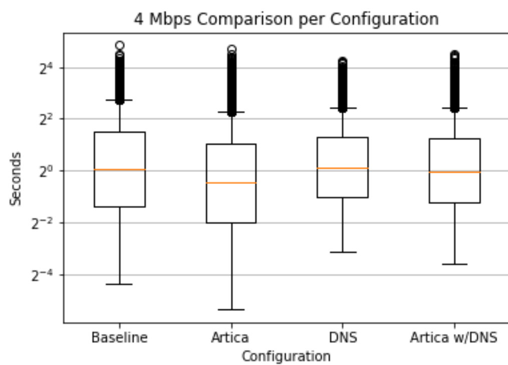
experiments were run. Then, the results were discussed. First, Artica performed very well for web traffic speeds. Second, the DNS compression algorithm succeeded at slow speeds and was able to work with Artica. The results show that customizing a broader range of protocols have the potential to further enhance the link performance.



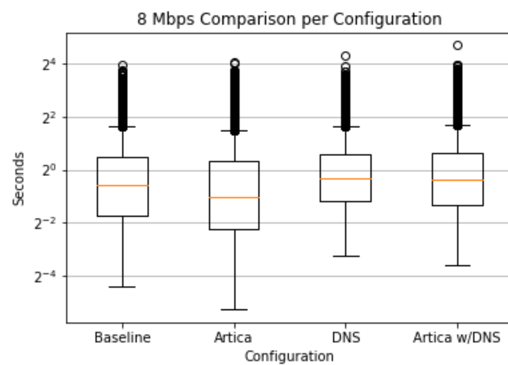
(a) 1.544 Mbps



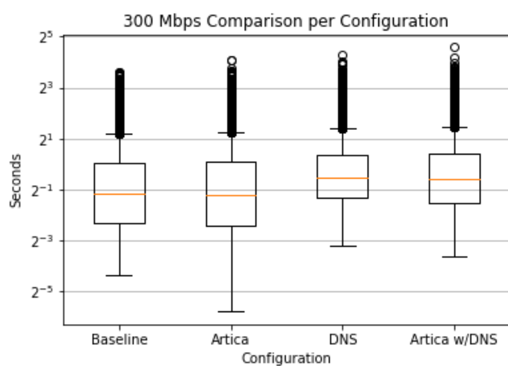
(b) 2 Mbps



(c) 4 Mbps

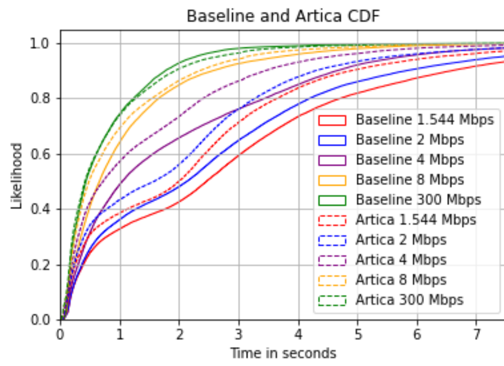


(d) 8 Mbps

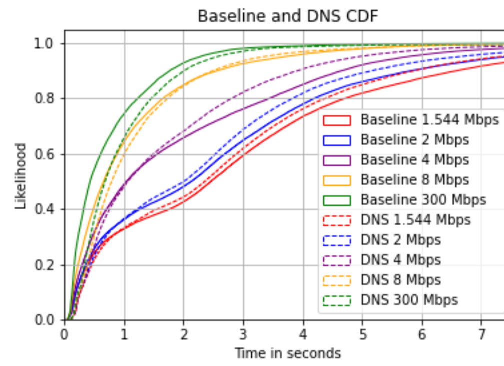


(e) 300 Mbps

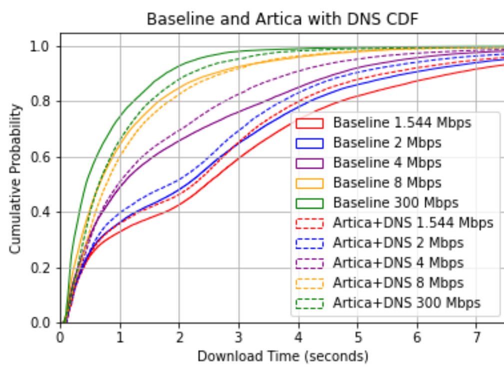
Figure 4.6: Comparisons of Configurations by Network Speeds



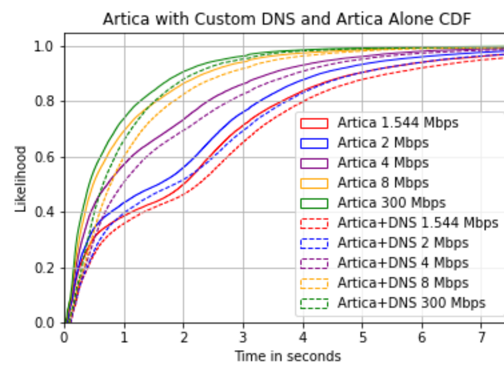
(a) 1.544 Mbps



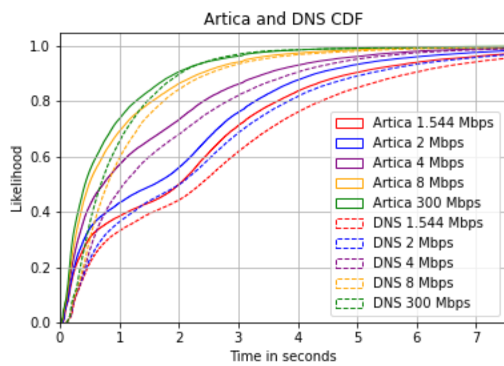
(b) 2 Mbps



(c) 4 Mbps



(d) 8 Mbps



(e) 300 Mbps

Figure 4.7: CDF Comparisons by Experimental Configuration

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 5: Conclusion

This thesis investigated ways to make USN satellite links more efficient for web browsing through experimenting with a free version of a WAN optimizer, compressing DNS traffic, and understanding how the two techniques are interoperable. The main conclusions from the investigation are as follows:

1. Artica sped up the web browsing for link speeds between 1.544 and 8 Mbps, by 13-26% on average. The speedups are significant and can be attributed to its geographically dispersed infrastructure that provides timely auto-correction of HTTP URLs and its adjustment of the TCP window size.

2. The DNS algorithm developed by this thesis was able to achieve average compression ratios of 1.36 and 2.24 for DNS queries and answers, respectively. However, it was observed that the compression did not cause significant speedup at higher speeds compared to baseline. This lack of speedup can be attributed to two primary factors. First, DNS traffic volumes, measured by bits, represented a small percentage of overall web traffic. Second, the compression algorithm introduced non-trivial processing delays at the ship and shore routers. At higher speeds, there was a slight performance degradation due to the extra processing delays.

3. Artica and the custom DNS program were shown to be interoperable at slower speeds (1.544 – 4 Mbps) and did not cause further performance degradation beyond what DNS compression introduced. However, at higher speeds, it degraded the performance of Artica because the extra DNS processing had a greater impact than the speedup Artica was able to achieve.

Ultimately, compression and bit transmission saving measures will increase performance on a network. However, as learned from this thesis, customized written code should target a broad range of protocols, not just one, for the most effective outcome. The USN should look into adopting non-patented measures to quickly increase its own efficiency, such as auto-correcting URL's and compressing traffic of widely used applications.

5.1 Future Work

Much of the future work identified in this section involves more customization of different protocols or testing other WAN optimizers; there are still a plethora of them to test with a variety of features. The proof of concept in this thesis showed it is possible to attempt to optimize any protocol, so long as the transmission path is controlled by the same owner.

1. Develop custom programs to optimize highly used protocols, such as HTTP or TCP. While TCP is not an application layer protocol, it is still responsible for a massive amount of data being moved on the internet and as such should be investigated.

2. Develop custom software to be pushed to clients, instead of a pseudo-proxy design, to cut out the middle-ware box and still be invisible to the user. This saving of the middle-ware would hopefully yield better times with fewer processing stops in the traffic path.

3. Develop a bit encoding scheme for DNS to represent the top 1000 websites so only a few bits are sent, not the whole FQDN. This method would save even more space in the queries. This thesis showed averages compression ratios of 1.36 and 2.24 for DNS queries and responses, respectively. With a bit encoding scheme, these ratios could potentially be even higher. Also, a potential security benefit exists with this method. If this traffic were intercepted, it would not give away what was queried without the list corresponding the numbers to the FQDNs.

4. Experiment with other WAN optimizers to infer additional techniques to reduce application latency. No single WAN optimizer is the same, and as such, each uniquely compresses or modifies network traffic. This yields the question of which combination of techniques works best for the USN's environment.

With any of these future research topics chosen, the way to test would be very similar to this thesis: establish a baseline and run them against the baseline to measure performance. All of these should be run against a baseline on the machine they are running to be an honest broker of performance.

The interoperability between COTS and customized protocol work has potential for a huge cost savings measure in future work. Having shown in this thesis that the two techniques

are complementary, it is clear that more savings could be done if implemented correctly and broadly enough. Combining these two is key for the future of USN satellite communications.

There is also potential for future work in researching the security implications in the customized application protocol software. This work raises a concern of the security risk of only passing the first IP address back as the DNS answer. There potentially exists a way to poison a DNS cache, or only the first address in the cache, and that address would be used. The other addresses would not be included in the answer for verification purposes.

Another security field to research with this implementation are threats of side-channel attacks. By paying attention to the only IP address sent back to the ship, it could potentially give away its geographic location since that IP address is likely the closest server to the ship. While security was not heavily researched for this thesis, it should not be completely ignored before implementing any of this thesis's work.

A final question for future work is how to handle compressing network traffic through encryption. Is it possible to compress specific portions through encryption? Achieving this modular approach would allow for security to co-exist with traffic compression and optimization. This capability would guarantee that upgrading network compression methods could continue to be a module on the network and not impact the security of the network.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A:
Network Setup

While easy to set up individual virtual machines on VirtualBox, it is much more difficult to setup multiples of them and have them communicate with each other. This appendix provides the steps to set up the testing network. First, the steps to make the individual User machines will be given. Following the user machines, instructions will be given to set up the two routers. These instructions assume Virtual Box is already downloaded.

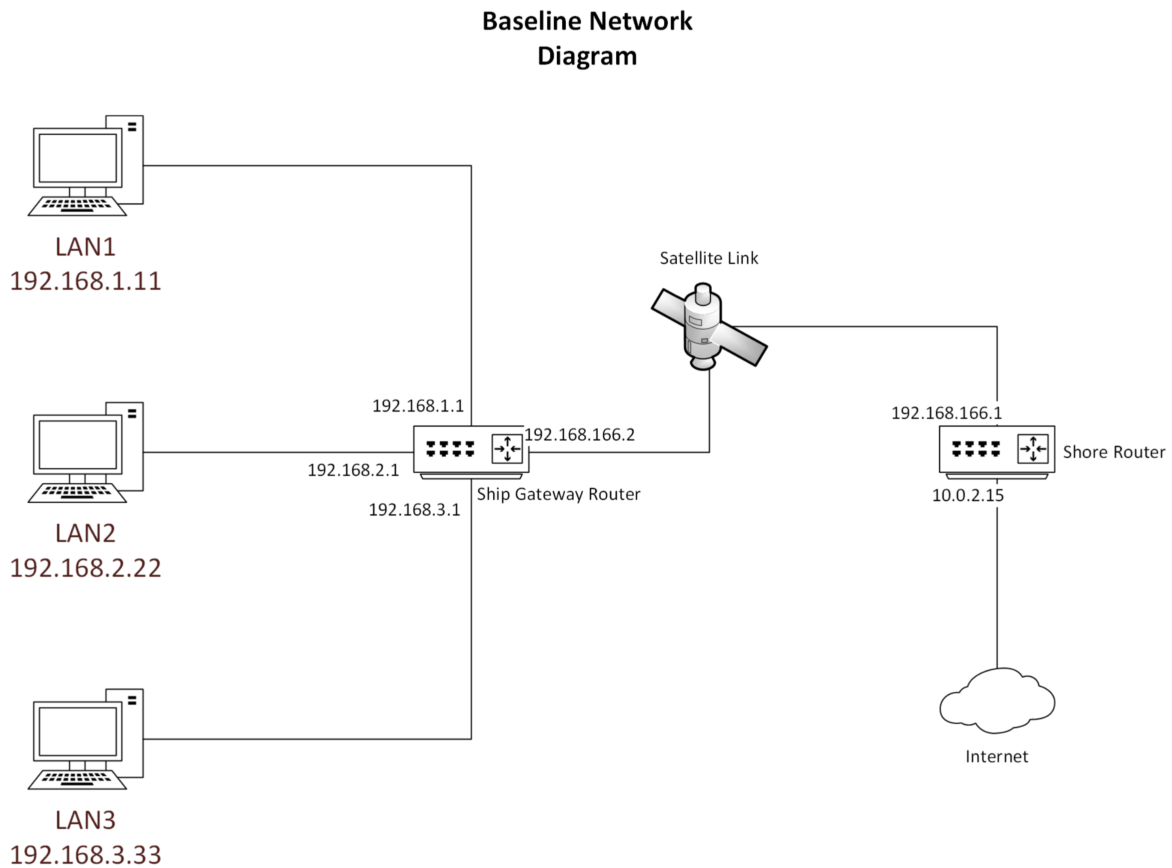


Figure A.1: Basic Network Configuration

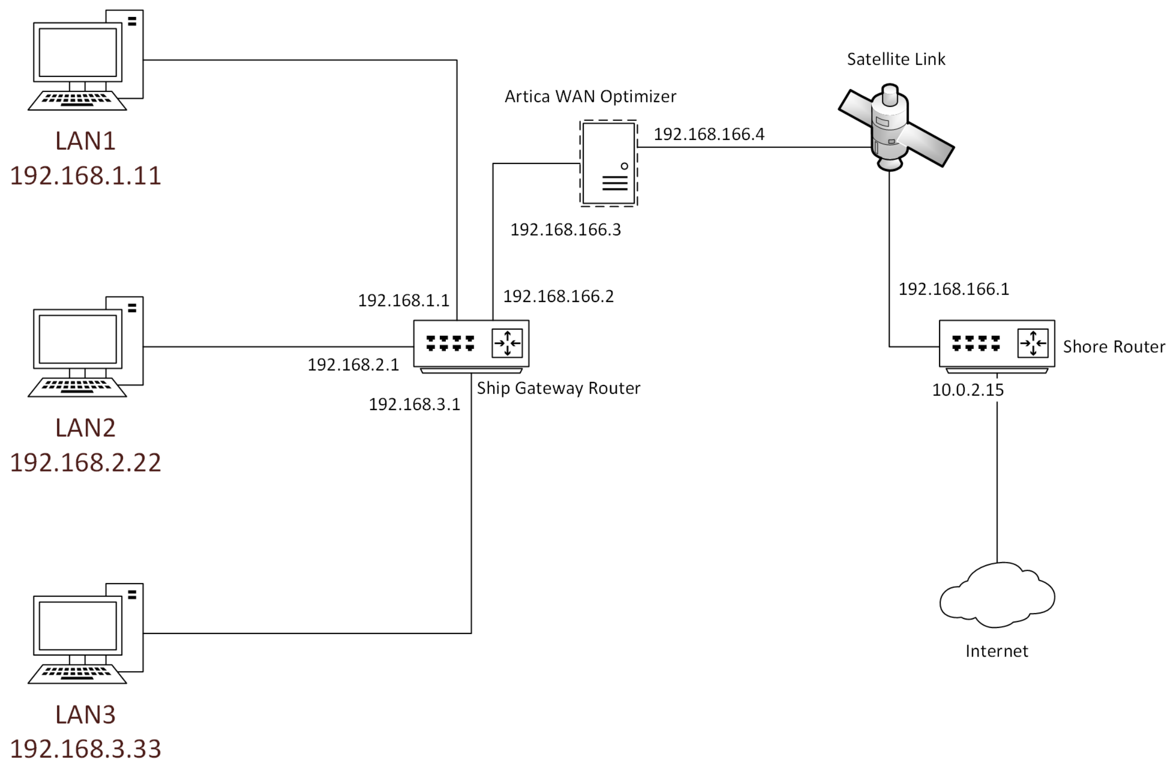


Figure A.2: Artica Network Configuration

A.1 Create the User Machines

1. Begin by going to https://ubuntu.com/download/desktop and downloading a copy of the Ubuntu 20 Desktop .iso image file.
2. Open Virtual Box and select create new operating system and select the Ubuntu .iso file.
3. Install using the default selections. Name it LAN1 Ubuntu.
4. Boot the machine.
5. Log in and install the SSH Server with the command: "sudo apt-get install openssh-server".
6. install scapy with the command "sudo apt-get install scapy"
7. Power down the machine.
8. Navigate to the Network Adapter Settings under properties on the machine and select

it as bridged only.

9. Boot the machine.
10. Select the "Ethernet Connected" label from the top right triangle of options on the machine.
11. Select Wired Settings.
12. Select the Ethernet Settings Gear on the row for the connected network connection.
13. Select the IPv4 tab.
14. Click manual for IPv4 method and DNS and add the following to the table underneath:
    (a) Address: 192.168.1.11
    (b) Netmask: 255.255.255.0
    (c) Gateway: 192.168.166.4 (Will be set up later as a router).
    (d) DNS: 8.8.8.8 (Google's Public DNS Server)
15. Close those windows and shut down the machine.
16. Create a shared directory for the VM to write all of its files to during the experiment one [30]. This thesis created it on the Desktop of the Mac for simplicity.
17. On the Virtual Box list of Virtual Machines will be LAN 1 Ubuntu.
18. Create Full Clone. Note, it must be a full clone so data collected during the experiments remain separated.
19. Rename the new clone LAN2 Ubuntu. Repeat to create LAN3 Ubuntu.
20. Boot LAN2 Ubuntu and LAN3 Ubuntu machines.
21. Navigate to their Address table under IPv4 insert these IP Addresses (the other settings will be there):
    (a) User 2 Ubuntu: 192.168.2.22
    (b) User 3 Ubuntu: 192.168.3.33

A.2 Create the Routers

1. Create a new Ubuntu Machine using the same .iso file used to create the user machines, and name this one "shore_router". Create with the same defaults as well.
2. Before booting the machine, select the "Network Adapter Settings" from the top row network drop down menu. In the upper right hand corner select "Add Device.
3. Select "Network Adapter" and click "Add...".
4. Select "Bridged" for this network connection. There should be one network connec-

tion that has NAT selected and a second one for the hosts.

5. Power on the virtual machine.
6. After logging in, open a terminal.
7. Type: "sudo nano /etc/sysctl.conf" [31].
8. In that file, change the line "#net.ipv4.ip_forward=0" to "net.ipv4.ip_forward=1". Figure A.3 gives an example. You must save and close it [31]. Be very careful, an error in this file will be detrimental to this virtual machine.

```
# Uncomment the next two lines to enable Spoof protection (reverse-path filter)
# Turn on Source Address Verification in all interfaces to
# prevent some spoofing attacks
#net.ipv4.conf.default.rp_filter=1
#net.ipv4.conf.all.rp_filter=1

# Uncomment the next line to enable TCP/IP SYN cookies
# See http://lwn.net/Articles/277146/
# Note: This may impact IPv6 TCP sessions too
#net.ipv4.tcp_syncookies=1

# Uncomment the next line to enable packet forwarding for IPv4
net.ipv4.ip_forward=1

# Uncomment the next line to enable packet forwarding for IPv6
#  Enabling this option disables Stateless Address Autoconfiguration
#  based on Router Advertisements for this host
#net.ipv6.conf.all.forwarding=1


###################################################################
# Additional settings - these settings can improve the network
# security of the host and prevent against some network attacks
# including spoofing attacks and man in the middle attacks through
# redirection. Some network environments, however, require that these
# settings are disabled so review and enable them as needed.
#
--More--(63%)
```

Figure A.3: sysctl.conf Sample Image

9. Install SSH Server with the command "sudo apt-get install openssh-server".
10. Install scapy with the command "sudo apt-get install scapy"
11. Restart the virtual machine and log in.
12. Create a file called setup_iptables.sh with the commands in Figure A.4 [32]. Be sure to keep it in an easy place, this will have to be run every time the VM is restarted. These series of commands force the VM to forward IP traffic and NAT it as if it were a router. You may also set it up to run on startup [33].
13. Reboot the machine if you choose to have it automatically run or run the script with command "sudo bash setup_iptables"

```bash
#! /bin/bash
sudo iptables -t nat -A POSTROUTING -d 0/0 -s 192.168.166.0/24 -j MASQUERADE
sudo iptables -A FORWARD -s 192.168.166.0/24 -d 0/0 -j ACCEPT
sudo iptables -A FORWARD -s 0/0 -d 192.168.166.0/24 -j ACCEPT
sudo route -n add -net 192.168.0.0/16 enp0s8
```

Figure A.4: setup_iptables.sh

14. Shutdown the VM.
15. Create a copy of the VM and name it ship_gateway_router.
16. Change both network interfaces to be "bridged" and add two more interfaces as bridged as well.
17. Boot ship_gateway_router.
18. One interface should have the IP address of 192.168.166.2, another should have 192.166.1.11, another should have 192.168.2.22, and the final one should have 192.168.3.33. All of them should have the DNS server as 8.8.8.8. The interface with the IP address of 192.168.166.2 should have a gateway of 192.168.166.1 to start and the others should have a gateway of 192.168.166.2 to point them to the simulated satellite link.
19. change the setup_iptables.sh file to look like Figure A.5:

```bash
#! /bin/bash
sudo iptables -t nat -A POSTROUTING -d 0/0 -s 192.168.166.0/24 -j MASQUERADE
sudo iptables -t nat -A POSTROUTING -d 0/0 -s 192.168.1.0/24 -j MASQUERADE
sudo iptables -t nat -A POSTROUTING -d 0/0 -s 192.168.2.0/24 -j MASQUERADE
sudo iptables -t nat -A POSTROUTING -d 0/0 -s 192.168.3.0/24 -j MASQUERADE

sudo iptables -A FORWARD -s 192.168.0.0/16 -d 0/0 -j ACCEPT
sudo iptables -A FORWARD -s 0/0 -d 192.168.0.0/16 -j ACCEPT

sudo iptables -A FORWARD -s 192.168.1.0/24 -d 0/0 -j ACCEPT
sudo iptables -A FORWARD -s 0/0 -d 192.168.1.0/24 -j ACCEPT

sudo iptables -A FORWARD -s 192.168.2.0/24 -d 0/0 -j ACCEPT
sudo iptables -A FORWARD -s 0/0 -d 192.168.2.0/24 -j ACCEPT

sudo iptables -A FORWARD -s 192.168.3.0/24 -d 0/0 -j ACCEPT
sudo iptables -A FORWARD -s 0/0 -d 192.168.3.0/24 -j ACCEPT
```

Figure A.5: setup_iptables.sh For ship_gateway_router

20. Reboot the machine so the script applies or run it so the changes are made.

Now, your network should be configured. All the user Ubuntu machines will route traffic through the ship_gateway_router to the shore_router and from there to the rest of the Internet using your host machines internet connection to its network.

APPENDIX B:
Source Code

This is the source code for programs and scripts executed during this thesis.

B.1 create_traffic.py

```
"""
Created on Tue Oct 20 09:10:40 2020

create_traffic.py generates all of the network traffic
by conducting a series of python gets with random wait times
in between each get.

It takes 4 arguments:
1: seed_number to know which seed file to read for waits
2. the run folder to save the run information to
3. the total_number of websites to go to
4. the speed that is currently being test to save as a part
   of the file name

@author: kenne
"""

import pandas as pd
import numpy
import requests
from datetime import datetime
from pathlib import Path
import time
import sys

df = pd.read_csv("top-1m.csv")
i = 0
```

```
#change for sys argv
seed_number = sys.argv[1]
run_folder = sys.argv[2]
total_websites = int(sys.argv[3])
string_to_add = sys.argv[4]
write_location = "/media/sf_share/run_scripts/" + run_folder +
    "/Run" + seed_number
string_to_open = write_location + "/" + string_to_add +
    "_LAN1_run_times.txt"

seed_open = "./Seeds/seed_run" + seed_number + ".txt"
f_seed = open(seed_open, "r")

delta_calc = datetime.now()
visit_time_dict = {}

for index, row in df.iterrows():
    web_page = "http://www." + row[0]
    success = True
    try:
        now1 = datetime.now()
        receive = requests.get(web_page, params={"Cache-Control":
        "no-cache"}, timeout=5)
        now2 = datetime.now()
        #print("Webpage: ", web_page, "\tReceived now:", receive)
        delta_calc = now2 - now1
        value_str = str(abs(delta_calc.total_seconds())) + "," +
        str(success) + "," + "LAN1" + "," + str(receive)[11:14]
        visit_time_dict.update({web_page:value_str})
    except Exception as e:
        #print(e)
        now2 = datetime.now()
        success = False
        delta_calc = now1 - now2
        value_str = str(abs(delta_calc.total_seconds())) + "," +
```

```
        str(success) + "," + "LAN1" + "," + "Failed"
        visit_time_dict.update({web_page:value_str})
    time.sleep(float(f_seed.readline()))
    if(i == (total_websites-1)):
        f = open(string_to_open, "w")
        f.write("FQDN,Wget_time_sec,Success,LAN,Response_code\n")
        for k,v in visit_time_dict.items():
            string_to_write = k + "," + str(v) + "\n"
            f.write(string_to_write)
        f.close()
        f_seed.close()
        break

    i+=1
```

B.2 for_all_start.sh

```
#! /bin/bash
#$1 number of trials
#$2 run folder save to
#$3 the number of websites to visit

folder=$2
websites=$3

for ((i=1; i<=$1; i++))
do
   date
   echo "Starting Run$i seed $i"
   ./trials.sh $i $folder $websites
done

date
```

B.3 trials.sh

```
#! /bin/bash
#$1 Run number and seed number
#$2 run folder save to
#$3 the number of websites to visit

mkdir /media/sf_share/run_scripts/$2/Run$1

for speed in 300000 1544 2000 4000 8000
do
  echo "starting $speed"
  ./tc.sh add $speed $2
  ./control_run.sh $1 $2 $3 $speed
  ./tc.sh del $speed $2
  sleep 5

done
```

B.4 tc.sh

```
#! /bin/bash
#$1 = add or del
#$2 = speed setting in kbit
#$3 is folder argument

test_var="artica"
folder=$3

sshpass -p defaultpassword ssh newuser@10.0.2.15
"sudo tc qdisc $1 dev enp0s8 root tbf rate $2kbit latency
    50ms burst 1540"

sleep 5

if [[ "$folder" == "$test_var" ]]; then
```

```
    echo "doing artica tc"
    sshpass -p root ssh root@192.168.166.3
    "tc qdisc $1 dev eth0 root tbf rate $2kbit latency
    50ms burst 1540"

else
    echo "doing regular tc"
    sudo tc qdisc $1 dev enp0s3 root tbf rate $2kbit latency
    50ms burst 1540

fi
```

B.5 start.sh

```
#start.sh
#!/bin/bash

nohup python3 create_traffic.py $1 $2 $3 $4 & > /dev/null&
```

B.6 control_run.sh

```
#! /bin/bash
#$1 Run number and seed number
#$2 run folder save to
#$3 the number of websites to visit
#$4 is test speed in kbits

echo "starting shore"

sshpass -p defaultpassword ssh newuser@10.0.2.15
"./start_shore.sh $4kbit Run$1 $2 >& /dev/null && exit"

sleep 5
```

```
echo "starting ship gateway tcp dump"
bash /home/newuser/start_ship.sh $4kbit Run$1 $2 >& /dev/null

sleep 5

echo "starting 1"
sshpass -p defaultpassword ssh newuser@192.168.1.11
"nohup ./start.sh $1 $2 $3 $4kbit >& /dev/null && exit"

sleep 5

echo "starting 2"
sshpass -p defaultpassword ssh newuser@192.168.2.22
"nohup ./start.sh $1 $2 $3 $4kbit >& /dev/null && exit"

sleep 5

echo "starting 3"
sshpass -p defaultpassword ssh newuser@192.168.3.33
"nohup ./start.sh $1 $2 $3 $4kbit >& /dev/null && exit"


sleep 600

./finish_check.sh Run$1 $2 $4kbit

echo "Killing all"
./kill_all.sh

sleep 5
```

B.7 kill_all.sh

```
#! /bin/bash
```

```
sshpass -p defaultpassword ssh newuser@10.0.2.15
'sudo pkill "tcpdump" && exit'
sleep 5
sudo pkill "tcpdump"
```

B.8 finish_check.sh

```
#! /bin/bash
#$1 is Runnumber
#$2 is folder number
#$3 is speed in kbit

echo "LAN11 check"
while [ ! -f /media/sf_share/run_scripts/$2/$1/
    $3_LAN1_run_times.txt ]
do
    echo "still waiting LAN11"
    sleep 120
done

echo "LAN22 check"
while [ ! -f /media/sf_share/run_scripts/$2/$1/
    $3_LAN2_run_times.txt ]
do
    echo "still waiting LAN22"
    sleep 60
done

echo "LAN33 check"
while [ ! -f /media/sf_share/run_scripts/$2/$1/
    $3_LAN3_run_times.txt ]
do
    echo "still waiting LAN33"
    sleep 30
```

```
done
```

B.9 shore dns_shore_server.py

```
import socket
import dns.resolver
import sys

#declarations for socket connections
localIP      = "192.168.166.1"
localPort    = 55000
bufferSize   = 8000
ship_address = "192.168.166.2"
ship_port    = 53
comms_port   = 60000
timeout_time = 10.0
nameservers  = ["8.8.8.8"]
#alt DNS servers for NPS
#nameservers  = ["172.20.20.11", "172.20.20.12"]

# Create a datagram socket
UDPServerSocket = socket.socket(family=socket.AF_INET,
    type=socket.SOCK_DGRAM)

# Bind to address and ip
UDPServerSocket.bind((localIP, localPort))

#create UDPServerSocket for Communicating to shore router
UDPServerSocket_send_ship = socket.socket(family=socket.AF_INET,
    type=socket.SOCK_DGRAM)
UDPServerSocket_send_ship.bind((localIP, comms_port))

#setup dns resolver
```

```
resolver = dns.resolver.Resolver()
resolver.timeout = timeout_time
resolver.lifetime = timeout_time
resolver.nameservers=nameservers

print("DNS up and listening")

# Listen for incoming datagrams
while(True):
    bytesAddressPair = UDPServerSocket.recvfrom(bufferSize)
    message = bytesAddressPair[0]
    address = bytesAddressPair[1]

    #handle IP requests from LANs
    #Transaction ID is always first 2 bytes of DNS header
    xID = message[0:2]

    #FQDN is the remainder of message
    url_str = message[2:].decode('utf-8')

    try:
        #do the look up for the IPv4 record
        record = resolver.query(url_str,"A")
        #since xID is in bytes, IPv4 address must be in bytes
        bytes_to_send = xID + bytes(map(int,
            str(record[0]).split(".")))
        #send back to the ship
        UDPServerSocket_send_ship.sendto(bytes_to_send,
            (ship_address,53))

    except Exception as e:
        pass
        #for t/s
        #print_string = "URL:" + url_str + "\t" + str(e)
        #print(print_string)
```

B.10 ship dns_server.py

```
import socket
from scapy.all import DNS, DNSQR, DNSRR, IP, send, sr1, UDP
import dpkt, dpkt.dns
import sys

#sys.argv[1] is expected to be speed i.e. speed1
#sys.argv[2] is expected to be the Run folder # i.e. Run1
#iplookup for testing if want to make this the
#primary dns servery for t/s
def ip_lookup(fqdn):
    #print("Starting lookup")
    resolver = dns.resolver.Resolver()
    resolver.timeout = 10.0
    resolver.lifetime = 10.0
    resolver.nameservers=["8.8.8.8"]
    #resolver.nameservers=["172.20.20.11","172.20.20.12"]
    try:
        record = resolver.query(url_to_look,"A")
    except:
        return "8.8.8.8"

    return record[0]

def dns_parser(pkt):
   dns = dpkt.dns.DNS(pkt)
   return [dns.id,dns.qd[0].name,dns.qd[0].type]

#declared variables for use sending and receiving sockets
localIP     = "192.168.166.2"
localPort   = 53
bufferSize  = 8000
shore_address="192.168.166.1"
shore_port = 55000
interface_connection = "enp0s10"
```

```
out_port = 60000
ipv6_number = 28
file_location = "/media/sf_share/dns_runs_test/
    dns_ship_request_compression.txt"
#for testing
#file_location = "testrun.txt"

#create Dictionary as data structure for requests
requests_dict = {}

# Create a datagram socket
UDPServerSocket = socket.socket(family=socket.AF_INET,
    type=socket.SOCK_DGRAM)

# Bind to address and ip
UDPServerSocket.bind((localIP, localPort))

#create UDPServerSocket for Communicating to shore router
UDPServerSocket_send_shore = socket.socket(family=socket.AF_INET,
    type=socket.SOCK_DGRAM)
UDPServerSocket_send_shore.bind((localIP, out_port))

f = open(file_location,"w")
f.write("FQDN,full_request_size,new_request_size\n")

print("DNS up and running")

while(True):
    #variables for receiving from buffer
    bytesAddressPair = UDPServerSocket.recvfrom(bufferSize)
    message = bytesAddressPair[0]
    address = bytesAddressPair[1]

    #if this is a response from the shore router/DNS server there
    if address[0] == shore_address:
```

```
    try:
        #handle response from server
        xID = message[0:2]
        ip_address = message[2:]
        #saves is "ipv4_address,port,fqdn"
        saved = requests_dict.pop(int.from_bytes(xID,"big"))
        #print("Got from dictionary: ", saved)
        saved_items = saved.split(",")

        #get the full fqdn
        fqdn_response= saved_items[2]
        #response_pkt, src must be the .2 and same port
        #since that is where request went to
        #the id must be in base 16 for the message back
        #the query being answered must be there as well
        #so the requesting machine knows how to process
        response_pkt = IP(src=localIP,dst=saved_items[0])/
        UDP(dport=int(saved_items[1]),sport=53)/
        DNS (id=int.from_bytes(xID,"big"), qr=1, qd=DNSQR (qtype="A",
        qname=saved_items[2]),
         ancount=1,an=DNSRR (rrname=saved_items[2],
         rdata=socket.inet_ntoa(ip_address)))
        #must specify the interface so it comes from
        #the correct IP address
        send(response_pkt,verbose=0,iface=interface_connection)
    except Exception as e:
        print(e)

#this is a query from one of the LANs
else:
    #handle IP requests from LANs
    #Transaction ID is always first 2 bytes of DNS header
    xID = message[0:2]

    #get the important parts from the dns message
```

```
dns_list_items = dns_parser(message) #id, name
if dns_list_items[2] == ipv6_number:
    #send client dns packet saying not handling AAAA
    response_pkt = IP(src=localIP,dst=address[0])/
    UDP (dport=int(address[1]), sport=53)/
    DNS (id=int.from_bytes(xID,"big"), qr=1, qd= DNSQR
    (qtype="AAAA",qname="unavailable"))
    #must specify the interface so it comes
    #from the correct IP address
    send(response_pkt,verbose=0,iface=interface_connection)
    continue
else:
    #send the ID and fqdn to the shore side, cutting
    #everything else out
    bytes_to_send = xID + dns_list_items[1].encode()
    UDPServerSocket_send_shore.sendto(bytes_to_send,
    (shore_address,shore_port))

    #save info in data structure for response from
    #shore side later
    save_string = address[0] + "," + str(address[1]) + "," +
    dns_list_items[1]
    requests_dict.update({dns_list_items[0]:save_string})

    string_to_write = dns_list_items[1] + "," +
    str(len(message)) + "," + str(len(bytes_to_send)) + "\n"
    f.write(string_to_write)
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX C:
Artica Setup

While Artica is easy to setup, here are some guiding steps on configurations changes this thesis did to get the Artica working on the network and operating as it would on a ship.

1. Download a copy of Artica 4 and the manual from their wiki [28].
2. Install using Virtual Box with two network interfaces that are bridged. Also, the OS flavor is Debian 64-bit.
3. Upon boot up, navigate to the network option, top selection, and assign values to the interfaces for the virtual network. In this case, its DNS server should 8.8.8.8. One interface, for this thesis, was 192.168.166.3 with its netmask 255.255.255.0 and its default gateway as 192.168.166.4. The other interface for this thesis was 192.168.166.4 with its netmask as 255.255.255.0 and its default gateway as 192.168.166.1.
4. Navigate back to the home screen, and then navigate to the Change Web Interface selection. There, change the SuperAdmin credentials for the online GUI interface. Of note, default admin credentials for this command line view are login:root password:root. Those are not the credentials for the GUI interface, hence the need to change them.
5. Reboot Artica.
6. Once rebooted, go to another VM and navigate to the web address for the GUI interface listed at the top of home page for Artica's boot up.
7. Once at the GUI, log in with the new credentials created in step 4.
8. On the navigation pane on the left hand side, select Your Proxy, and then listen ports from the drop down.
9. Once the window appears, select Transparent Ports from the top navigation selection.
10. For proper management of the network, successful communications for the DNS servers, and successful communications for the LANs out to get the websites, ports 80, 443, 53, 22, and 55000 must be opened. 55000 (the only port greater than 1023), must be opened because that was the port chosen for the DNS servers to utilize for communications.
11. Select Apply Configuration.

12. Select Global Settings under Your Proxy. Navigate to timeouts from the navigation pane across the top.
13. Match settings to Figure C.1.

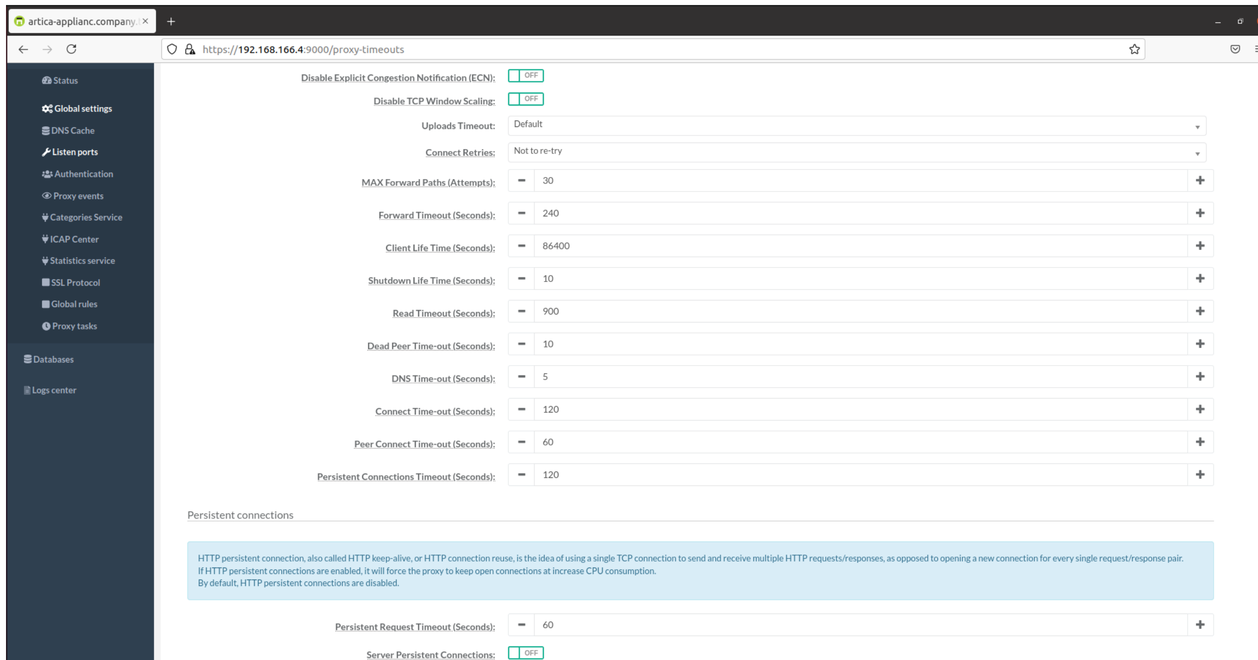


Figure C.1: Artica TCP Configuration

14. Select Apply at the bottom of the page.

List of References

[1] K. Andersson and D. Montag. (2007). Development of DNS security, attacks and countermeasures. [Online]. Available: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.120.7896&rep=rep1&type=pdf

[2] CISCO. (2016). VNI complete forecast highlights. [Online]. Available: https://www.cisco.com/c/dam/m/en_us/solutions/service-provider/vni-forecast-highlights/pdf/Global_2020_Forecast_Highlights.pdf

[3] J. Mogul, F. Douglis, A. Feldmann, and B. Krishnamurthy, "Potential benefits of delta encoding and data compression for HTTP," *Proceedings of ACM SIGCOMM '97 conference*, pp. 181–194, 1997.

[4] R. Klauck and M. Kirsche, "Enhanced DNS message compression - optimizing mDNS/DNS-SD for the use in 6LoWPANs," in *2013 IEEE International Conference on Pervasive Computing and Communications Workshops (PERCOM Workshops)*, 2013, pp. 596–601.

[5] A. Beirami, M. Sardari, and F. Fekri, "Packet-level network compression: Realization and scaling of the network-wide benefits," *IEEE/ACM Transactions on Networking*, vol. 24, no. 3, pp. 1588–1604, 2016.

[6] L. Vdovin, P. Likin, and A. Vilchinskii, "Network utilization optimizer for SD-WAN," in *2014 International Science and Technology Conference (Modern Networking Technologies) (MoNeTeC)*, 2014, pp. 1–4.

[7] A. Padmanabhan, R. Zhang, P. Thakkar, T. Kopenen, and M. Casado. (2015, October). United States Patent WAN Optimizer for Logical Networks. [Online]. Available: https://patents.google.com/patent/US9172603B2/en

[8] CISCO. (2020, November). Network address translation (NAT) FAQ. [Online]. Available: https://www.cisco.com/c/en/us/support/docs/ip/network-address-translation-nat/26704-nat-faq-00.html

[9] Homenet Howto. (2021). Address translation, complete picture. [Online]. Available: https://www.homenethowto.com/advanced-topics/address-translation-complete-picture/

[10] R. Russel, M. Boucher, J. Morris, J. Kadlecsik, and H. Welte. (2020). iptables(8) - linux man page. [Online]. Available: https://linux.die.net/man/8/iptables

[11] python.org. (2021). Python:about. [Online]. Available: https://www.python.org/about/

[12] P. S. Foundation. (2021). io - core tools for working with streams. [Online]. Available: https://docs.python.org/3/library/io.html

[13] P. Biondi and the Scapy community. (2021). Scapy: Packet crafting for python2 and python3. [Online]. Available: https://scapy.net/

[14] L. Rendek. (2021). Bash scripting tutorial for beginners. [Online]. Available: https://linuxconfig.org/bash-scripting-tutorial-for-beginners

[15] T. Ylonen, "SSH - secure login connections over the internet," in *6th USENIX Security Symposium*, vol. 6, 1996, pp. 37–42.

[16] A. W. Services. (2021). What is DNS? [Online]. Available: https://aws.amazon.com/route53/what-is-dns/

[17] D. E. 3rd. (2000, September). Domain Name System (DNS) IANA Considerations. [Online]. Available: https://www.hjp.at/doc/rfc/rfc2929.html

[18] L. V. Winkle. (2021). Hands-on network programming. [Online]. Available: https://www.oreilly.com/library/view/hands-on-network-programming/9781789349863/812dd5c5-0d22-4ccd-8faf-f339b416bb2e.xhtml

[19] D. E. 3rd. (2013, April). Domain Name System (DNS) IANA Considerations. [Online]. Available: https://www.hjp.at/doc/rfc/rfc6895.html

[20] R. Peon and H. Ruellan. (2015, May). HPACK: Header Compression for HTTP/2. [Online]. Available: https://www.hjp.at/doc/rfc/rfc7541.html

[21] H. de Saxcé, I. Oprescu, and Y. Chen, "Is HTTP/2 really faster than HTTP/1.1?" in *2015 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, 2015, pp. 293–299.

[22] R. Klauck and M. Kirsche, "Enhanced DNS message compression - Optimizing mDNS/DNS-SD for the use in 6LoWPANs," in *2013 IEEE International Conference on Pervasive Computing and Communications Workshops (PERCOM Workshops)*, 2013, pp. 596–601.

[23] G. Ateniese and S. Mangard, "A New Approach to DNS Security (DNSSEC)," in *Proceedings of the 8th ACM Conference on Computer and Communications Security*, ser. CCS '01. New York, NY, USA: Association for Computing Machinery, 2001, p. 86–95. [Online]. Available: https://doi.org/10.1145/501983.501996

[24] S. C. Amdahl. (2011, February). Compression of data transmitted over a network. [Online]. Available: https://patents.google.com/patent/US7882084B1/en

[25] Northrop Grumman. (2021). Consolidated Afloat Networks And Enterprise Services (CANES). [Online]. Available: https://www.northropgrumman.com/what-we-do/sea/consolidated-afloat-networks-and-enterprise-services-canes/

[26] Amazon.com. (2021). Alexa top sites. [Online]. Available: https://aws.amazon.com/alexa-top-sites/

[27] M. B. Nirmala, "WAN Optimization Tools, Techniques and Research Issues for Cloud-Based Big Data Analytics," in *2014 World Congress on Computing and Communication Technologies*, 2014, pp. 280–285.

[28] Artica Tech. (2021). Artica wiki. [Online]. Available: https://wiki.articatech.com/

[29] S. Hogg. (2017, June). Why Should We Separate A and AAAA DNS Queries? [Online]. Available: https://blogs.infoblox.com/ipv6-coe/why-should-we-separate-a-and-aaaa-dns-queries/

[30] estorgio. (2021, July). Mounting VirtualBox shared folders on Ubuntu Server 18.04 LTS (Bionic Beaver). [Online]. Available: https://gist.github.com/estorgio/0c76e29c0439e683caca694f338d4003

[31] networkinghowtos.com. (2013, July). Enable IP Forwarding on Ubuntu 13.04. [Online]. Available: https://www.networkinghowtos.com/howto/enable-ip-forwarding-on-ubuntu-13-04/

[32] askubuntu.com. (2018). Route all traffic of a machine through another within a subnet? [Online]. Available: https://askubuntu.com/questions/907972/route-all-traffic-of-a-machine-through-another-within-a-subnet

[33] L. Rendek. (2020, February). How to run script on startup on Ubuntu 20.04 Focal Fossa Server/Desktop. [Online]. Available: https://linuxconfig.org/how-to-run-script-on-startup-on-ubuntu-20-04-focal-fossa-server-desktop

THIS PAGE INTENTIONALLY LEFT BLANK

Initial Distribution List

1. NIWC PAC
   San Diego, CA

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California