



NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

CLUSTER COMPUTING FOR AUTOMATED NETWORK ANALYSIS AT SCALE

by

Benjamin J. Brida

June 2018

Thesis Advisor:

Frank E. Kragh

Co-Advisor:

James W. Scrofani

Approved for public release. Distribution is unlimited.

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			<i>Form Approved OMB No. 0704-0188</i>	
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.				
1. AGENCY USE ONLY (Leave blank)		2. REPORT DATE June 2018	3. REPORT TYPE AND DATES COVERED Master's thesis	
4. TITLE AND SUBTITLE CLUSTER COMPUTING FOR AUTOMATED NETWORK ANALYSIS AT SCALE			5. FUNDING NUMBERS W8A62	
6. AUTHOR(S) Benjamin J. Brida				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000			8. PERFORMING ORGANIZATION REPORT NUMBER	
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A			10. SPONSORING / MONITORING AGENCY REPORT NUMBER	
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.				
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release. Distribution is unlimited.			12b. DISTRIBUTION CODE A	
13. ABSTRACT (maximum 200 words) Conventional single node packet analyzers are unable to monitor network traffic at scale. In this thesis, elements of the Apache Hadoop ecosystem, including HBase, Spark, and MapReduce, are employed to conduct network traffic analysis on a large collection of network traffic. Limited analysis is conducted directly on packet capture next generation (pcapng) files on the Hadoop Distributed File System (HDFS) using MapReduce. Next, to allow for repeated analysis on the same dataset without reading all source files in their entirety for every calculation, pcapng files are parsed and relevant meta-data is bulk loaded into HBase, a Not Only Structured Query Language (NoSQL) database employing the HDFS for parallelization. This NoSQL database is then accessed via Apache Spark where pertinent data is loaded into DataFrames and additional analysis on the network traffic takes place. This research demonstrates the viability of custom, modular, automated analytics, employing open-source software to enable parallelization, to conduct traffic analysis at scale.				
14. SUBJECT TERMS big data, MapReduce, Hadoop, packet capture, traffic analysis, network analysis, Spark, HBase			15. NUMBER OF PAGES 105	
			16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UU	

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release. Distribution is unlimited.

**CLUSTER COMPUTING FOR AUTOMATED NETWORK ANALYSIS AT
SCALE**

Benjamin J. Brida
Captain, United States Marine Corps
BS, Johns Hopkins University, 2010

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

**NAVAL POSTGRADUATE SCHOOL
June 2018**

Approved by: Frank E. Kragh
Advisor

James W. Scrofani
Co-Advisor

R. Clark Robertson
Chair, Department of Electrical and Computer Engineering

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Conventional single node packet analyzers are unable to monitor network traffic at scale. In this thesis, elements of the Apache Hadoop ecosystem, including HBase, Spark, and MapReduce, are employed to conduct network traffic analysis on a large collection of network traffic. Limited analysis is conducted directly on packet capture next generation (pcapng) files on the Hadoop Distributed File System (HDFS) using MapReduce. Next, to allow for repeated analysis on the same dataset without reading all source files in their entirety for every calculation, pcapng files are parsed and relevant meta-data is bulk loaded into HBase, a Not Only Structured Query Language (NoSQL) database employing the HDFS for parallelization. This NoSQL database is then accessed via Apache Spark where pertinent data is loaded into DataFrames and additional analysis on the network traffic takes place. This research demonstrates the viability of custom, modular, automated analytics, employing open-source software to enable parallelization, to conduct traffic analysis at scale.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

I.	INTRODUCTION.....	1
A.	THESIS MOTIVATION.....	1
B.	BACKGROUND	1
1.	Apache Hadoop	2
2.	MapReduce.....	4
3.	HBase	7
4.	Spark	8
5.	Packet Capture Next Generation (pcapng)	8
C.	THESIS ORGANIZATION.....	13
II.	EXPERIMENTAL DESIGN	15
A.	ACADEMIC AND COMMERCIAL PROCESS REVIEW	15
B.	DATA ACQUISITION.....	16
C.	FRAMEWORK DESIGN	16
1.	MapReduce.....	17
2.	HBase	18
3.	Spark	18
III.	IMPLEMENTATION	19
A.	HADOOP ECOSYSTEM CONFIGURATION	19
B.	MAPREDUCE DATA ANALYSIS	19
1.	Driver Class and Job Configuration	20
2.	Parsing pcapng.....	22
3.	Mapper and Reducer Classes	25
C.	MAPREDUCE DATA PARSING AND BULK HBASE LOADING	28
D.	SPARK DATA ANALYSIS	29
IV.	RESULTS	33
A.	MAPREDUCE ANALYSIS RESULTS	33
1.	Network Usage in Bytes per IP Address	33
2.	Port Count	36
3.	Protocol Count	37
4.	Network Usage by Hour of Day and Day of Week per IP Address.....	38
B.	MAPREDUCE DATA PARSING AND BULK HBASE LOADING	41

C.	SPARK DATA ANALYSIS	43
V.	CONCLUSION	47
A.	SUMMARY	47
B.	FUTURE WORK	48
	APPENDIX A. MAPREDUCE ANALYTICS	51
A.	PARSING CODE FOR PCAPNG FILES:	51
B.	WHOLE FILE INPUT FORMAT AND WHOLE FILE RECORD READER	58
C.	COMPILE AND RUN SCRIPTS	60
D.	NETWORK USAGE PER IP CODE	61
1.	Driver Code	61
2.	Mapper Code for Network Usage per IP	62
3.	Reducer Code for Network Usage by IP	63
E.	PORT COUNT CODE	64
1.	Port Count Mapper Code	64
2.	Port Count Reducer Code	65
F.	PROTOCOL COUNT CODE	66
1.	Protocol Count Mapper Code	66
2.	Protocol Count Reducer Code	67
G.	NETWORK USAGE BY DAY OF WEEK AND TIME OF DAY PER IP CODE	67
1.	Network Usage by Day of Week and Time of Day per IP Mapper Code	67
2.	Network Usage by Day of Week and Time of Day per IP Reducer Code	68
	APPENDIX B. MAPREDUCE DATA PARSING AND BULK HBASE LOADING CODE	71
A.	DRIVER CODE	71
B.	MAPPER CODE	72
	APPENDIX C. SPARK ANALYSIS CODE	79
	LIST OF REFERENCES	83
	INITIAL DISTRIBUTION LIST	87

LIST OF FIGURES

Figure 1. Graphical Display of File Partition and Storage in HDFS. Adapted from [2].	3
Figure 2. Process Overview of Writing File to HDFS. Source: [5].	3
Figure 3. Word Count MapReduce Example. Source: [7].	5
Figure 4. Hadoop MapReduce Process Flow on Two Nodes. Source: [7].	6
Figure 5. Section Header Block Structure for pcapng Files. Source: [13].	9
Figure 6. Hexadecimal Representation of the First Sixteen Bytes of a pcapng File	10
Figure 7. Interface Description Block Structure for pcapng Files. Source: [13].	11
Figure 8. Enhanced Packet Block Structure for pcapng Files. Source: [13].	13
Figure 9. Hadoop MapReduce Job Configuration in Driver Code	20
Figure 10. Parser Code Excerpt with Custom Methods.	23
Figure 11. Mapper Code for Determining Bandwidth Use per IP in pcapng File	26
Figure 12. Reducer Code for Determining Bandwidth Use per IP in pcapng File	26
Figure 13. Example MapReduce Flow to Determine Bandwidth Use per IP Address	27
Figure 14. Catalog Defined for Mapping from HBase to Spark Table	30
Figure 15. TCP Three-Way Handshake with Capture Device. Adapted from: [29].	32
Figure 16. MapReduce Metrics for Calculating Total Bytes Sent/Received per IP address. One Input pcapng File with 1,176,590 Packets	34
Figure 17. Output Excerpt from MapReduce Job Determining Total Bytes Sent/Received per IP address	34
Figure 18. MapReduce Metrics for Calculating Total Bytes Sent/Received per IP Address—Six Input pcapng Files with 9,244,373 Packets	35
Figure 19. MapReduce Metrics for Calculating Port Count by IP Address - Single Input pcapng File with 4,152,080 Packets	37
Figure 20. Output Excerpt from MapReduce Job Determining Port Usage by IP	37

Figure 21. MapReduce Metrics for Ethernet Protocol Count - Six Input pcapng Files with 9,244,373 Packets	38
Figure 22. Output from MapReduce Job Determining Ethernet Protocol Counts – Sum of Values Corresponds to 9,244,373 Packets from Six Input pcapng Files	38
Figure 23. MapReduce Metrics for Calculating Network Usage by Day of Week and Time of Day per IP Address- One Input pcapng File with 1,176,590 Packets	40
Figure 24. Output Excerpt from Calculating Network Usage by Day of Week and Time of Day per IP Address	40
Figure 25. MapReduce Metrics for Loading pcapng Metadata into HBase – Six Input pcapng Files with 9,244,373 Packets	42
Figure 26. Scan of the First Five Row Keys in HBase Shell	42
Figure 27. Spark DataFrame Created from HBase Table with Composite Row Key as Single Column	43
Figure 28. Spark DataFrame Created from HBase Table with Composite Row Key Decomposed into Col0 – Local IP and Col1 – Timestamp, and Filtered by SYN Flag Set	44
Figure 29. Code Used to Determine Bytes Transmitted/Received per IP Address, IPv4 Protocol Distribution and Source Port Distribution	44
Figure 30. Output from Spark Queries Determining, from Left to Right, Bytes Transmitted/Received per IP Address, IPv4 Protocol Distribution, and Source Port Distribution.....	45
Figure 31. Excerpt of Spark Output, Calculating the Round Trip Time Pairs Comprising the Initial Round Trip Time of a TCP Three-Way Handshake	46

LIST OF TABLES

Table 1.	Method Name and Required Input, Type Returned and Purpose for Custom pcapng Class	24
----------	---	----

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF ACRONYMS AND ABBREVIATIONS

DBSCAN	density-based spatial clustering of applications with noise
HDFS	Hadoop Distributed File System
ICMP	Internet Control Message Protocol
IGMP	Internet Group Management Protocol
IP	Internet Protocol
iRTT	initial round trip time
JAR	Java ARchive
JVM	Java Virtual Machine
MCNEL	Marine Corps Network Efficiency Laboratory
NoSQL	Not Only Structure Query Language
OSI	Open Systems Interconnection
PCAPNG	Packet Capture Next Generation
JSON	JavaScript object notation
RDD	resilient distributed dataset
RIPE-NCC	Réseaux IP Européens Network Coordination Centre
RIR	regional Internet registry
RTT	round trip time
SQL	Structured Query Language
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
VM	virtual machine

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

Thanks to my loving wife and son, who made sure I had time to work on this thesis and still go to the park, respectively. I would also like to thank my advisor, Professor Frank Kragh, for his guidance and support, even when work came without spell-check.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. THESIS MOTIVATION

The motivation behind this thesis was a research request from the Marine Corps Network Efficiency Laboratory (MCNEL). It is projected that by 2021, annual global Internet protocol (IP) traffic will reach 3.3 Zettabytes (3.3×10^{21} bytes) per year, and there will be 3.5 networked devices per person connected to IP networks [1]. This reliance on interconnected devices, coupled with the massive growth in both volume and complexity of networks, will make network optimization and network security increasingly critical and increasingly challenging. In order to keep pace with the growth of networks, it is essential to automate analysis for network traffic monitoring. Additionally, this automation needs to handle and store large volumes of data, on the petabyte (10^{15} bytes) scale. For these reasons, MCNEL sought an automated data analysis solution on a big-data framework.

B. BACKGROUND

In the conduct of analysis on large volumes of data, parallelization is a key tenet that solves several problems. The first problem addressed is read/write access limitations. While disk storage capacity has increased one hundred-fold in the past two decades, read speeds on most drives have only increased by a factor of twenty-five [2]. Rather than storing all data on a single drive, we can instead partition and store data on hundreds of drives with shared access. By allowing for simultaneous reading of data on multiple drives, parallelization allows for tasks to be completed in a fraction of the time. Second, parallelization not only increases data reading speed, it also increases data reliability through redundancy. For example, by storing partitioned data in triplicate on physically separated hard drives, we greatly reduce the likelihood of a single, or even multiple, drive failures preventing access to data. Finally, we can greatly increase processing power through parallelization. By storing data on two, two hundred, or two thousand servers and allowing each server to individually do the processing of the data it stores, we not only reduce the required network traffic to move and manage data but also vastly increase effective processing power [2].

For MCNEL's purposes, we determined that using open source software on commodity hardware is most likely to be maximally scalable and financially supportable. Because of those considerations, the Hadoop ecosystem was selected as the large-data framework to be used. For increased functionality and ease of implementation, for this thesis specifically, MapR was employed. MapR is a third-party Hadoop distribution; however, any properly configured Hadoop cluster would be viable. Finally, packet capture next generation (pcapng) was selected as the expected file type for analysis since network collections conducted by MCNEL use this format.

1. Apache Hadoop

The ApacheTM Hadoop[®] ecosystem is a collection of open source software that facilitates distributed computing [3]. Hadoop is designed to run on commodity hardware, (i.e., affordable and obtainable). The benefit is that it is easily scalable, with inexpensive machines easily added for redundancy or increased capacity. For example, in 2010, Yahoo! had 25,000 servers comprising their Hadoop clusters storing 25 petabytes of data [4].

The backbone of Hadoop is the Hadoop Distributed File System (HDFS), which partitions and stores data across a cluster of servers (nodes). This is done by taking large files and splitting them into many smaller blocks, which are then stored on DataNodes, typically in triplicate. The average block size can vary, but the default is 128 MB. The metadata, documenting which blocks are stored on what DataNodes, is then stored on NameNodes [2]. The flow of splitting a file into smaller blocks and then allocating those blocks to separate nodes, with redundancy, is shown in Figure 1. Furthermore, how HDFS does this specifically is shown in Figure 2.

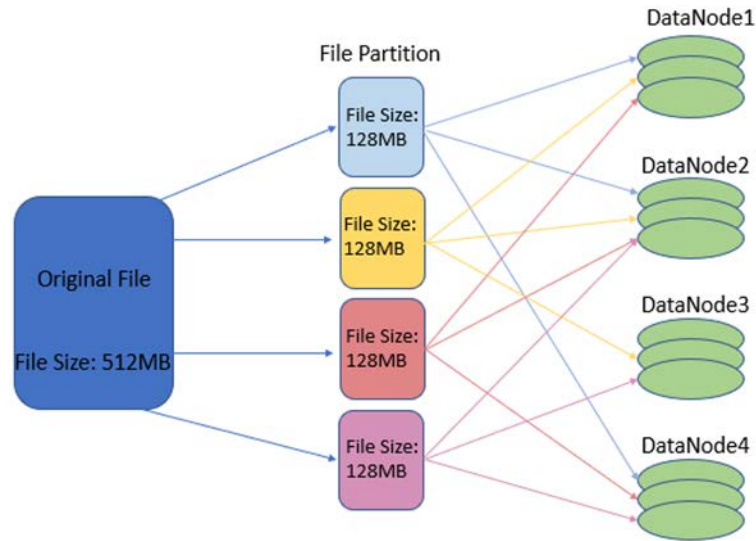


Figure 1. Graphical Display of File Partition and Storage in HDFS.
Adapted from [2].

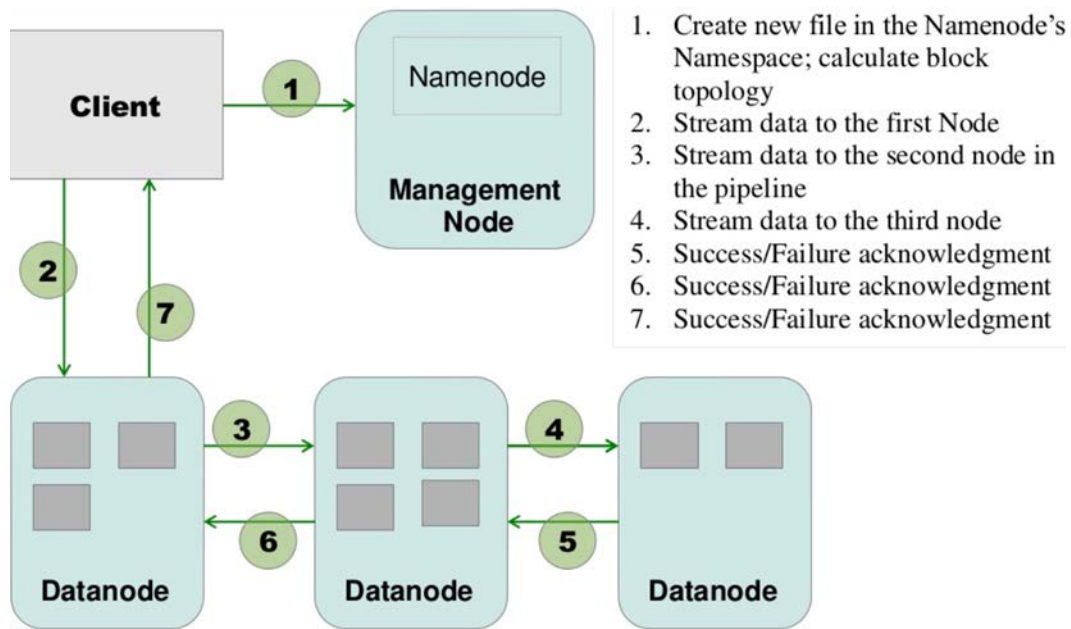


Figure 2. Process Overview of Writing File to HDFS. Source: [5].

Hadoop, and the various enterprise Hadoop distributions, all contain a variety of mechanisms for quality assurance, including data integrity and reliability; however, as they are not pertinent to the specific goals of this study, further discussion is omitted.

2. MapReduce

MapReduce is a parallel programming model for data processing and analysis which was originally published by Google engineers in 2004 [6]. MapReduce, as the name implies, works by separating the analysis into distinct map and reduce stages. As a broad overview example, we consider taking a book that we wish to analyze. Specifically, we want to know how many times each unique word occurs in the book. The word count problem is a common introductory example for MapReduce. One option to complete this task is to have a single person sit down and read through the entire book while tallying how many times each word occurs. This is analogous to conventional single-processor computing. If we want to speed this process up, we can perhaps find the fastest reader and assign the task to him; however, there will always be limitations to completion time with a single person or single processor completing the task. As an alternative to one person working on the problem, if we were to instead rip out every page of the book and give each page to a different person, there could theoretically be hundreds to thousands of people completing the first part of the task. If we had even more volunteers, we could further rip the pages in half so each “mapper” individual only had half a page or even just a single line. Each “mapper” can take the portion of the book they were assigned and write each unique word they see on an individual note, with the tally of how many times that word occurred.

Once all the “mappers” have finished writing the unique words and their tallies on a note, we then have an intermediary stage where we can sort all the outputs from each mapper, corresponding to every page. In this case, we can alphabetize and then group identical words. Finally, each person who was assigned a page as a “mapper” can now be assigned specific words to handle as a “reducer.” For example, one person may get all the notes with the word “the” and the corresponding tallies. They sum the tallies from every page and give the final count of the word as their output. For this reducer stage, we can have as many people acting as “reducers” up to the total number of unique words. The process flow of how the map, combine, shuffle, reduce, and final output stages work for a word count example is shown in Figure 3.

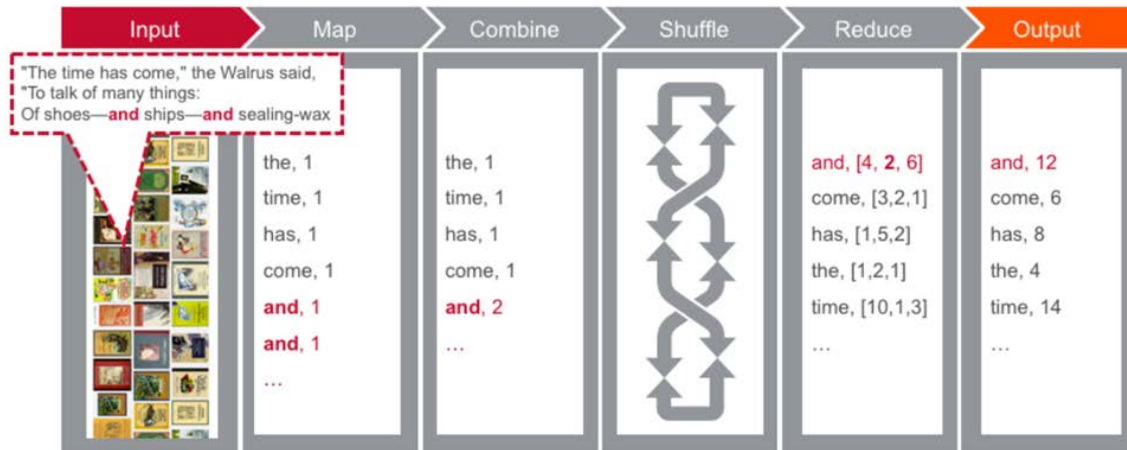


Figure 3. Word Count MapReduce Example. Source: [7].

From this example, we see there are several obvious support actions that must occur in addition to the map and reduce stages. First, there is the initial data management requirement, in this case ensuring all “mappers” have a non-overlapping block of text. Next, there is the requirement to shuffle and sort the output from all the “mappers,” which in this case meant grouping identical words. Finally, aggregating reducer outputs may be necessary for a single file analytic result.

If we dive even deeper into the example, we see there are other conceivable process management challenges. How do we choose the number of mappers/reducers and who assigns each one their task? What if instead of simply counting unique words, we want to count the number of paragraphs, and the page you are given is in the middle of a paragraph? What happens if a mapper or reducer is unable to complete their task for any reason?

These same considerations are necessary when using MapReduce in a cluster computing environment. This is where Hadoop MapReduce comes in. The Hadoop MapReduce framework manages many of the support functions considered above without any required user modification [2]. For example, the record reader is responsible for ingesting input splits and managing the logical boundaries that may exist in physically partitioned files. This ensures that each mapper is passed a complete individual record for analysis. Hadoop supplies native support for several commonly used varieties for file input format classes (e.g., text files). Additionally, Hadoop MapReduce automatically assigns

mappers, automatically shuffles and sorts the key/value pairs, and automatically assigns the sorted key/value pairs to reducers. These functions can be manually specified, but there is no requirement to do so. Also, default types of key/value pairs are provided. Example default key/value options are Int, Long, Bytes, Boolean, and Float. Hadoop MapReduce also has NodeManagers and ResourceManagers to monitor task assignment and completion status [2]. These are not unique nodes but are services run on existing data nodes. Finally, there is even further complexity in how Hadoop MapReduce manages memory and how information is spilled to disk or managed in case of memory errors. This is addressed later, as deviation from default settings was necessary.

The overview of the Hadoop MapReduce process is illustrated in Figure 4 and is to be discussed in further detail later, with several specific mappers, reducers, drivers, and the corresponding custom input format classes and record readers. Despite the additional requirements and considerations just listed, it is clear how some tasks in a MapReduce environment can be completed much faster by taking advantage of potentially hundreds of computers to complete a task.

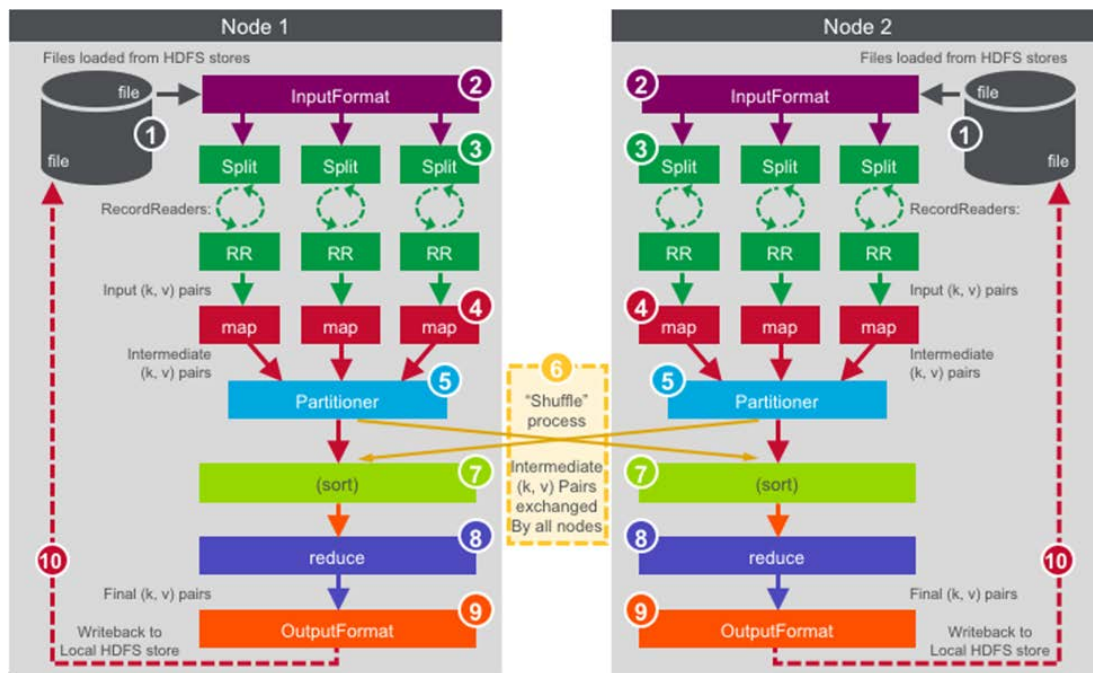


Figure 4. Hadoop MapReduce Process Flow on Two Nodes. Source: [7].

3. HBase

HBase is another capability that resides on Apache Hadoop. It is an open source variant modeled after Google's non-relational database BigTable. HBase is designed to function with HDFS and scale laterally, creating new blocks of storage as needed on many servers. Scaling laterally means that HBase can continue to grow by adding additional networked nodes. It can support billions of rows with millions of columns on commodity hardware [8].

The HBase column-based data storage model is structurally broken into three major components, consisting of the row key, column family, and column. The row key is a unique identifier used to locate a specific row. Row keys must be unique, are sorted lexicographically at the byte level, and determine where specific information is stored [9]. This makes row key selection an important task in constructing an efficient HBase table. Row key considerations for this thesis are discussed further in HBase implementation, presented in Chapter III.

Each row is broken into columns, which are grouped as column families. There can be an arbitrary number of individual columns under a column family, and they do not need to be pre-specified. While columns can be created during data assignment, column families are static for a table. Every row has the same number of column families, even if there are a differing number of columns. For example, an individual's full name could be a row key, and their street, city, state, and country could all be columns under a column family titled "Address." Some users could conceivably have no state column with a foreign country. HBase employs sparse storage, meaning any column which is not used does not have memory allocated for it [9].

Since row keys are unique, a duplicate entry is stored as a new version of the same row. HBase versioning is ordered in descending order (i.e., newest first). If not specified, a get request to HBase returns only the newest version [9].

Distributed storage for an HBase table is accomplished in two ways. First, blocks of rows are grouped into a row key range called a region. Each region is then assigned to a specific data node on the cluster. HBase automatically manages this process for the user.

Second, unique column families are stored in separate files so that they can be accessed independently. This method of storage enables data likely to be accessed together to be stored together and makes column family creation important based on anticipated access and use of the data [9]. HBase stores its data as HFiles in key/value pairs. The key is the coordinate to the value. At a minimum a key contains Table: Row Key: Column Family, and this yields a series of columns, their version, and values. Further specification is possible, including specific columns and version to yield just one value.

4. Spark

Apache Spark, like MapReduce, is another tool able to conduct analysis in a parallelized environment. Spark allows you to write programs in Scala, Python, or Java to build complex algorithms for interacting with data. Unlike MapReduce, which writes to disk after every job, Spark can provide in-memory performance. This allows Spark processing to operate up to 100 times faster than MapReduce jobs [10]. MapReduce may still outperform Spark in linear processing of very large datasets, where the output is much larger than available memory.

Spark can directly access data from HDFS and HBase. Additionally, Spark integrates with Spark SQL for interacting with structured data, MLlib, a machine learning library and GraphX for graphs and graph-parallel computation [10].

The backbone of Spark is interacting with resilient distributed datasets (RDD) and DataFrames. These are collections of elements which can be accessed in a parallelized environment. Both RDDs and DataFrames are immutable. DataFrames are organized into named columns. This allows users to apply structure and a schema to their data for ease of access and manipulation. RDDs lack of schema makes them better suited for unstructured data. DataFrames and their schema are discussed further in model design, as they relate to building a DataFrame from HBase data [11].

5. Packet Capture Next Generation (pcapng)

Packet Capture Next Generation, referred to as pcapng, is a file extension for traffic collection. Traffic collection consists of passively collecting and logging network packets.

The mechanism of collection can be either software on a host on the network (e.g., tcpdump or Wireshark) or through the use of a standalone packet capture appliance placed somewhere on the network.

Capture files are organized into blocks. The file begins with a global header called the section header block and an interface description block. The key elements of the section header block are the block type, block total length, byte-order magic, major version, minor version, section length, options and block total length again. Block total length is duplicated at the end of each block to allow for backward file navigation [12]. The layout of the section header block is depicted in Figure 5. In the figure, each column in the row represents a single bit, with a complete row representing four bytes, comprised of 32 bits numbered 0 to 31. The row count in the figure represent the starting byte (e.g., 0, 4, 8, 12, 16, etc).

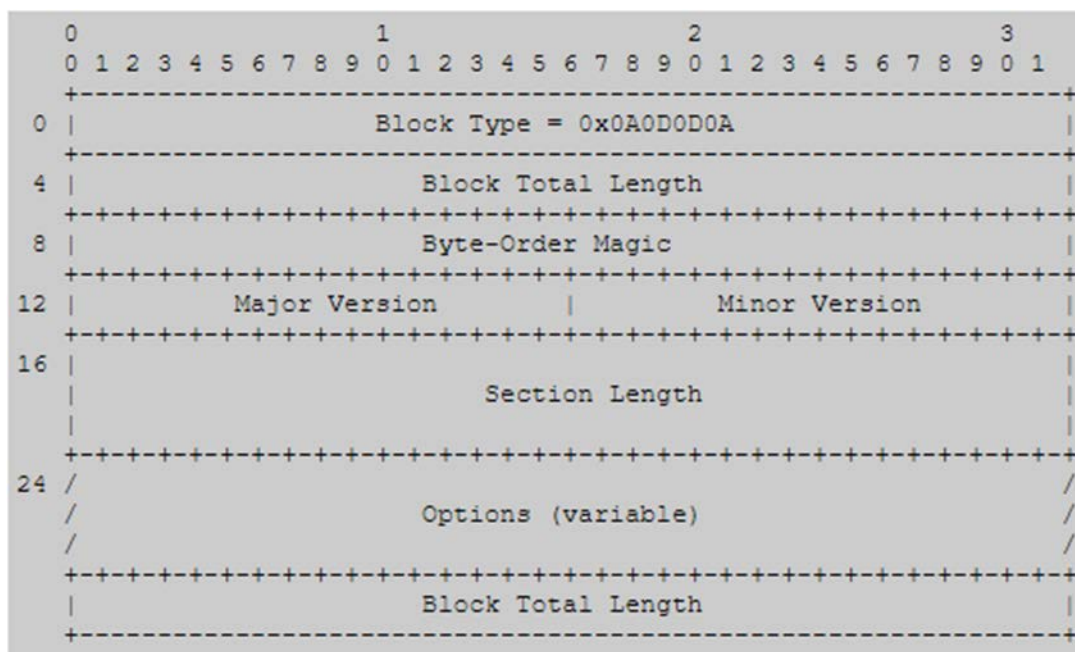


Figure 5. Section Header Block Structure for pcapng Files. Source: [13].

In the section header block, the first four bytes in hexadecimal, corresponding to the block type, are always the bytes 0x0A0D0D0A, corresponding to the ASCII four-character string “\r\n\n\r.” This value is used since it is palindromic, so it reads the same

regardless of endianness. Pcapng sections use the endianness of the native format from the generating host as this increases efficiency by avoiding the need to translate or modify data while capturing. Additionally, if a file has been transferred with inappropriate ASCII conversion, then this header will be an unexpected value, and a possibly corrupted file can be detected [13]. Next, the section header block total length is given. This is an integer corresponding to the number of bytes in the block. After block total length, the byte-order magic number is given. This is a hexadecimal value of 0x1A2B3C4D, which determines the endianness of the transmission. For example, all future analysis in this thesis was conducted using little endian interpretation. The first 16 bytes from a collected pcapng file used later in this thesis are shown in Figure 6. The bytes are represented in hexadecimal notation. The highlighted portion of this figure shows the byte-order magic as 0x4D3C2B1A, indicating the byte order inversion expected in little endianness. This hexadecimal byte representation also shows the block type is uncorrupted and is the expected 0x0A0D0D0A, and the block length, adjusted for being little endian, is 0x000000B4 or 180 bytes. Finally, as shown in Figure 6, the major version is currently 1 and the minor version is 0.

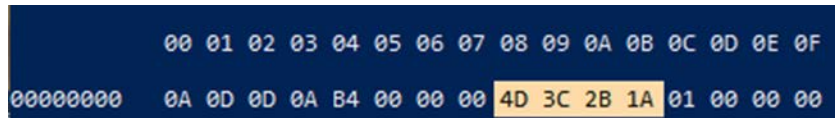


Figure 6. Hexadecimal Representation of the First Sixteen Bytes of a pcapng File

After the major and minor version in the section header block, the next four bytes are the section length, which is used to identify how long the options section is. Finally, there is the variable length options section, and then the block total length is repeated.

After the mandatory section header block there is the mandatory interface description block. Each interface description block is assigned an interface ID, which is a 32-bit number [12]. This assignment is unique per section, starts at zero, and increments. This identifier is referenced by packet blocks, which are discussed next, to identify to which interface the packet refers. The interface description block structure is depicted in

Figure 7. SnapLen in the interface description block allows the operator to set a maximum snap length, which truncates packets during collection if they exceed this value [12]. For this thesis, detailed study of the interface description block contents is less important and is omitted. Packets collected for this thesis had a link type of null, and the options used were to identify the capture device with a UTF-8 string. Of note, the options section here is where the timestamp resolution is set. The default setting, if unset, is microsecond resolution; that is, the timestamp represents the number of microseconds elapsed since January 1, 1970.

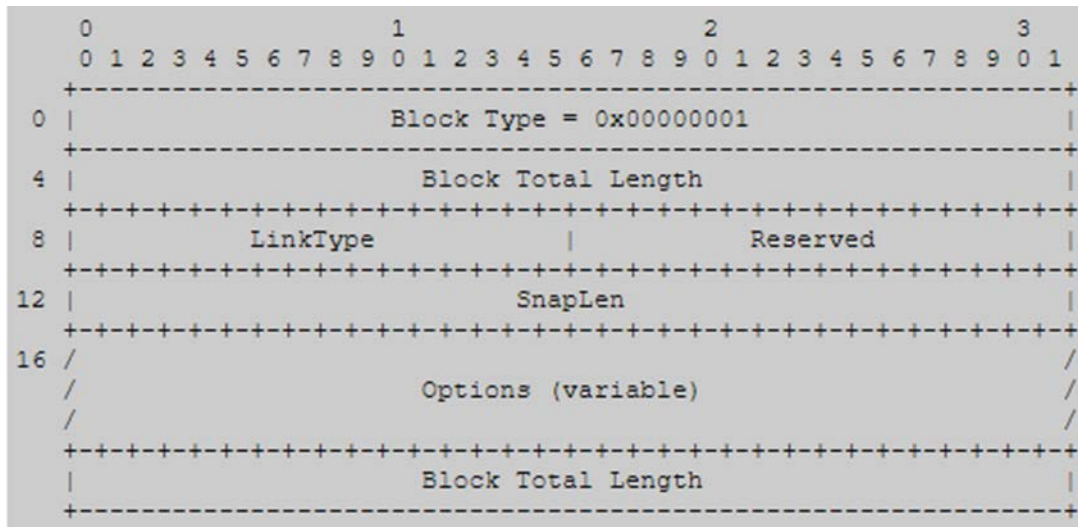


Figure 7. Interface Description Block Structure for pcapng Files.
Source: [13].

Finally, after the section headers, pcapng has the packet blocks. There is one block for each individual network packet. The packet blocks come in two types, the enhanced packet block and the simple packet block [13]. For this thesis, all collection utilized the enhanced packet blocks, and so discussion of the simple packet block is omitted. The structure of the enhanced packet block is shown in Figure 8. The enhanced packet block begins with the four-byte block type, 0x00000006. This is important, as it not only identifies the block as an enhanced packet block, but it can also be considered the start of a local header and aid in identifying the beginning of a new packet. After the block type, comes the block total length. Block total length is once again four bytes that represent the

total integer number of bytes in the block. The next four bytes are the interface ID, discussed previously. For collections used in this thesis, only one interface per section was used, and so this value was always 0x00000000 [12], [13].

After the four-byte block type, four-byte block total length, and four-byte interface ID, is the eight-byte timestamp, which is broken into two four-byte blocks, representing the high 32-bits and low 32-bits of a 64-bit unsigned integer. As previously mentioned, this 64-bit integer represents the time elapsed since January 1, 1970. The default resolution is in microseconds; however, this can be changed in the interface description block and can be as precise as nanosecond resolution [12], [13].

The next four bytes are the captured packet length followed by the original packet length. These can be different values as the captured packet length is the lesser of the SnapLen, assigned in the interface description block, and the original packet length [12]. Finally, after 24 bytes of header, we get to the variable length packet data. This is the actual network traffic data complete with link-layer headers. The packet data section is variable length but is always padded to be evenly divisible into 32-bit sections.

Next, after the packet data, there is a variable length options section, and finally, the four-byte block total length is repeated to complete the packet. This options section can contain things like packet drop count, packet hash information or link-layer flag information. There are a variety of options which can be used which are listed in both [12] and [13] but are not relevant to the analysis conducted in this thesis.

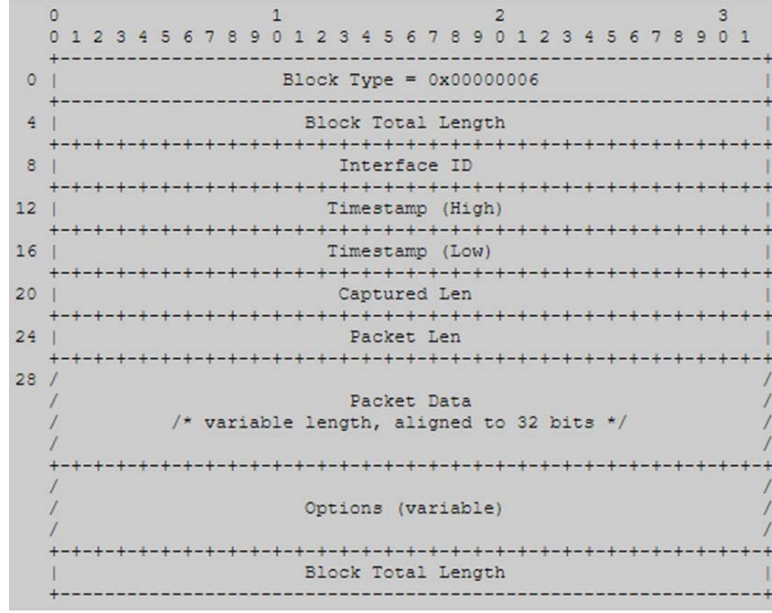


Figure 8. Enhanced Packet Block Structure for pcapng Files. Source: [13].

C. THESIS ORGANIZATION

This thesis is organized into five chapters. In Chapter I, we focus on the thesis motivation and the background information necessary for understanding the tools used to conduct the research for this thesis. Experimental design is discussed in Chapter II, including an analysis of how industry solves problems like those MCNEL proposed, and the information review that lead to the selection of the tools described in Chapter I. Additionally, in Chapter II we cover how the data for this thesis was gathered. In Chapter III, we detail how the tools described in Chapter I, selected based on the assessments from Chapter II, were used to build a framework to address MCNEL’s research objectives. The results from the analysis in Chapter III are presented and discussed in Chapter IV. Finally, the summary of work completed and discussion on recommendations for future work is discussed in Chapter V.

THIS PAGE INTENTIONALLY LEFT BLANK

II. EXPERIMENTAL DESIGN

A. ACADEMIC AND COMMERCIAL PROCESS REVIEW

The motivation behind this thesis is also found in commercial industry. For that reason, while doing a literature review on the topic, it was necessary to consider sources in industry in addition to the traditional academic publications. Much of the direction for tool selection was based on considerations and best practices used by data scientist in industry. FloCon 2018, a conference hosted by the Carnegie Mellon University Software Engineering Institute, was a key resource in presenting dozens of cutting-edge techniques used in large scale data analysis for network security.

The initial academic literature review was focused on assessing how cluster computing had been used to analyze and classify networks historically. The preponderance of documentation in this stage was on Hadoop and MapReduce, as they are the more mature framework, with robust documentation and study. Lee and Kang [14] conducted analysis on NetFlow data using MapReduce. One specific experiment was to determine the port count from the traffic. A limitation observed while reviewing this study was that they converted all NetFlow data to text for ingestion into their mapper. Despite this, they still observed a 72% performance increase when their four-node cluster running Hadoop MapReduce was compared with traditional flow analysis tools.

We next sought studies specifically conducted using pcapng files, as NetFlow is neither as large nor as robust as MCNEL's pcapng data. Dhanotia, Kapoor, and Sharma [15] used a density-based spatial clustering of applications with noise (DBSCAN) algorithm executed with Hadoop MapReduce. In the study, the use of a single-node versus multi-node clusters was compared. They evaluated at what size of input the multi-node execution becomes superior in terms of performance. This study, despite being more recent, still preprocessed their input files. This time instead of text, they converted the pcapng files into csv files.

Finally, we sought out studies using both pcapng files and ingesting them in their native format. Here, we found fewer published articles and more blog entries or GitHub

contributions. Réseaux IP Européens Network Coordination Centre (RIPE-NCC) is the Regional Internet Registry (RIR) for Europe, the Middle East, Russia, and Central Asia. They manage the allocation and registration of IP addresses and autonomous system (AS) numbers. RIPE-NCC created a slide presentation and an article titled Large scale PCAP Analysis using Apache Hadoop [16]. Their assessment was that Hadoop worked well in the areas they anticipated; specifically, it scaled very well and was successful with over 100 CPU cores. They did note the processing overhead of using Hadoop MapReduce and observed that only 50% of computation time was on actual analysis. Additionally, while their GitHub contained the source code for ingesting PCAP files directly into MapReduce, the last update was in 2016 and includes no support for pcapng files.

B. DATA ACQUISITION

Initially, data for this research was to be provided by MCNEL, so that the research could also verify historic analysis previously conducted by MCNEL; however, due to the limitations of MCNEL's current tools and techniques, coupled with the typical operational support they provide, the data they had was neither robust enough nor well enough documented to be exclusively used. Some portions of the initial analysis using MapReduce on pcapng files, such as determining top bandwidth users on a network, was conducted using MCNEL provided data. Later work, where pcapng files were parsed and put into a distributed NoSQL database, required knowledge of what IPs were local, something not provided in MCNEL's data.

For this later analysis, standard household network traffic was collected in the author's home using Wireshark. These individual pcapng files ranged from 250 MB to 7.0 GB. While not ideal, the general structure of the collect is identical to that provided by MCNEL.

C. FRAMEWORK DESIGN

The framework design for this thesis was guided by a motivation to take the best practices of the work discussed in the literature review and build on it for increased performance and capacity. One repeated limitation observed was the tendency to convert traffic collection, either PCAP or pcapng files, into text, csv or JSON. There are several

motivations for this, including the increased structure of a JSON file and Hadoop having the native capacity to ingest text and csv files. While using these formats simplifies the work required of a researcher or developer since no additional coding is required to write a custom input format class and a custom record reader, serious performance consequences occur as a result. A pcapng file converted to a text file using Wireshark, with no loss or gain of information, results in a file size increase of 700%. As a major motivation for this research was to enable analysis on terabyte-sized datasets, that inefficiency is unsupportable outside of the laboratory. At best, assuming ASCII encoding with one byte per character, taking an unstructured hexadecimal representation of a pcapng file saved as text, there is a size increase of 100%. This is expected since a single byte has two hex characters, each requiring 1-byte encoding.

Another limitation of many previous studies is that they exclusively used MapReduce. As discussed in Chapter I, Spark provides a performance increase of up to 1000% by operating exclusively in memory without the need to write and then reread files after every job. Since MCNEL is interested in many types of network analysis, including network top talkers, top protocols, top ports, error flag breakdown, and network latency, in this thesis we sought to create a framework where all analysis could be conducted in one job or, at a minimum, in several jobs without the read/write to disk limitation of MapReduce.

The compilation of the listed considerations, specifically evaluating pcapng files in their native format with Hadoop MapReduce and then applying Spark for increased performance over Hadoop MapReduce were at the core of framework design, which consists of three stages.

1. MapReduce

As mentioned, the first goal was to be able to do basic network analysis using MapReduce directly on the pcapng files saved to the HDFS. This had to be accomplished with an arbitrary number of individual pcapng files, which only had to be identified at runtime.

2. HBase

While altering the storage format of pcapng to something with greater structure increases the disk space required for storage, the benefits of employing a NoSQL database designed for horizontal scaling, compared to a flat file, lix text, outweighed the cost. The second stage of the framework development was to bulk ingest the metadata from an arbitrary number of pcapng files into HBase. This is done using the parsing capability for pcapng files, developed for analysis in step-1, and the built-in capability in Hadoop MapReduce to create HFiles with the HFileOutputFormat class. Ultimately, this provides the capability to use our entire cluster to create an HBase table from pcapng files and drastically increases parsing capacity.

3. Spark

The final step of the framework is to access desired pcapng metadata from the HBase table in the HDFS to create specific DataFrames and conduct analysis with no more than a single disk read/write, providing drastic performance increase.

III. IMPLEMENTATION

A. HADOOP ECOSYSTEM CONFIGURATION

For this thesis, the MapR Sandbox virtual machine was used as a single node setup to execute MapReduce and Spark code. MapR is a converged data platform that comes with many of the Hadoop Ecosystem tools pre-installed and configured [17]. While MapR provides its own file system and database, since in this thesis we sought to maximize portability and usability across all open-source Hadoop clusters, the MapR file system and database were not used. The MapR Sandbox comes with the CentOS Linux operating system, providing a command line interface to the pre-installed Hadoop Ecosystem. Additionally, HDFS, HBase, and job tracking can be accessed through Hue, a browser-based interface for the cluster [18].

VMware Workstation 14 Player with MapR Sandbox for Hadoop version 5.2.1 was used to run the virtual machine (VM). Hadoop version 2.7.0 and Spark version 2.1.0 were the software versions within the VM. All standard properties were used except the heap-size for child Java Virtual Machines (JVM) of map tasks and the resource limit for maps. These values were overridden to provide more memory to the map task. This override process is described in [19], and for this thesis 2.0 GB were provided to child JVMs of maps and 2.0 GB was the resource limit for maps.

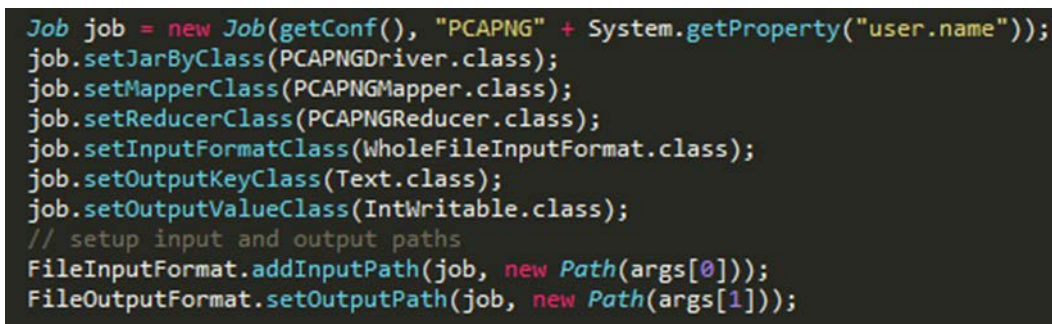
B. MAPREDUCE DATA ANALYSIS

First, while Hadoop MapReduce jobs do not need to be written in Java, it is executed using a (JVM) [20], [21]. For this thesis, all MapReduce code was written in Java. To use MapReduce to analyze pcapng files, there are many elements required beyond the mapper and reducer algorithms. From Figure 4, we see that we need an input format class capable of ingesting pcapng files and a corresponding record reader. These are all part of the job configuration for a MapReduce execution. A `Job` object in Java represents a MapReduce job configuration [20]. For this thesis, in addition to the mapper and reducer code, a driver code was used to create and configure a new `Job`. This driver code is where

the `Job` is created, and the `Job` parameters are set. This driver is submitted by the job client as a JAR file [22] to the resource manager, which results in the execution of the `Job`.

At a minimum, the driver class written for the MapReduce applications in this thesis included setting the file input location and format class, the file output format, Jar by class, mapper, reducer, and finally, the input/output key and value classes. An excerpt of the driver code with job configuration used for a MapReduce job on pcapng files is shown in Figure 9. We dissect each specification and the motives for how the MapReduce job was configured.

In this example, we see that we have `PCAPNGDriver`, `PCAPNGMapper`, and a `PCAPNGReducer` classes. The code for the mapper and reducer, shown in their entirety in Appendix A, comprise the core methodology used for analysis in this stage and are discussed later in this section. The driver is where we initiate the job and set the job configuration. In Figure 9 we see a screen capture from the driver. Additionally, we also see the input format class is `WholeFileInputFormat`, the output key class is `Text`, and the output value class is `IntWritable`. Finally, we see that our file input path and file output path are set at run time by arguments 0 and 1, respectively.



```
Job job = new Job(getConf(), "PCAPNG" + System.getProperty("user.name"));
job.setJarByClass(PCAPNGDriver.class);
job.setMapperClass(PCAPNGMapper.class);
job.setReducerClass(PCAPNGReducer.class);
job.setInputFormatClass(WholeFileInputFormat.class);
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(IntWritable.class);
// setup input and output paths
FileInputFormat.addInputPath(job, new Path(args[0]));
FileOutputFormat.setOutputPath(job, new Path(args[1]));
```

Figure 9. Hadoop MapReduce Job Configuration in Driver Code

1. Driver Class and Job Configuration

The default input file format is text [2]. The file input format class is responsible for determining if and how a file is to be split and to call the record reader for managing logical boundaries [2], [20]. As discussed previously, the goal was to avoid preprocessing

our pcapng files into text or csv, as these files are much larger, and the preprocessing requires a substantial increase in computation time. As a result, we required a custom input format class and record reader to support pcapng files.

The original PCAP file format does not support random access within a file. It must be read sequentially to identify individual packet sizes and where new packet begins. For this reason, you cannot split a PCAP file without reading it, and Hadoop must treat each individual file as its own record. For pcapng files, as we show in Chapter I, there is an enhanced block identifier at the start of each new packet, 0x00000006. This can theoretically be used to identify individual packets and to allow pcapng files to be splittable; however, it is possible that this value could occur in a capacity outside of a block identifier. Additionally, it is simple to set a capture device to create a new pcapng file every 256 MB, and 256 MB is the default split size in MapR HDFS and can be set to be a split size in any Hadoop distribution. This means that every individual pcapng file is its own record and occupies its own complete split. Using that as a basis, we chose to use the whole file input format class and record reader presented in [2] and in Appendix A. This ingests whole files with null key and BytesWritable as the value with isSplittable set to false and ensures each input file is treated as one complete record. No output format is specified in this stage, as the default output format is text with key/value pairs, which is what we desire.

Next, in Figure 9, we see the output key class is set to Text, and the output value class is set to IntWritable. No custom keys or custom values were used for this thesis, and so only a brief discussion on the necessity of these objects and the default options selected is presented here. Hadoop reads and writes objects in a serialized form for transmission [23]. Writable is the Java interface which allows for this. For example, the Text object is serialized by writing its length followed by the UTF-encoded string. Defaults include Text, IntWritable, LongWritable, FloatWritable, BytesWritable, and BooleanWritable. Furthermore, default key and value classes in Hadoop use an even stricter interface, WritableComparable. This is necessary for any object which is the output key from a mapper. The comparable interface allows for the hashing and sorting of identical keys so that they may be aggregated and sent to a single reducer.

The output key and output value shown in Figure 9 correspond to a driver that was coupled with a mapper and reducer used to determine total network usage from each unique IP collected. The output key type of Text corresponds to a string notation of IP (e.g., 192.168.0.1), and the output value of IntWritable corresponds to the total number of bytes that IP sent/received during all captures analyzed.

2. Parsing pcapng

Once the MapReduce driver has been configured and we are able to ingest complete pcapng files as BytesWritable, which is created by a byte array, we still must parse the file. An objective identified early on by MCNEL was to have flexibility in the analytics conducted so that not only is our solution scalable, by using a big data framework, but also customizable, as analytic requirements may vary from mission to mission. For these reasons, we opted to build a Java class for pcapng files complete with methods able to identify individual packets, move sequentially through packets, and extract key bytes pertaining to specific information from the pcapng file. The complete code used to do this is included in Appendix A.

Several methods that required unique considerations are presented as examples here. The first two methods presented are designed to check if there are additional packets in the pcapng file and to logically select the next packet. These can be seen in Figure 10. The `isNextPacket()` method, seen at the top of Figure 10, conducts two checks to see if there are any additional packets in the pcapng file. First, it checks to see if there are any bytes remaining after completing the current packet. This check, in some cases, proved to be insufficient by itself. Specifically, if we access the raw bytes from our BytesWritable using the speed-optimized Hadoop BytesWritable method `getBytes()`, the returned byte array will not be the precise length of the data [24]. The extra bytes exist but contain no packets. The size check will fail, and we will think there are more packets when there are not. This problem can be addressed by using the Hadoop BytesWritable method `copyBytes()` to access the underlying data. The method `copyBytes()` returns a copy of the bytes exactly the length of the data; however, this method is slower than `getBytes()`. To be able to successfully use either data access methods, a second check condition was added to

the code where we evaluate the block type, as discussed in the pcapng file structure presented in Chapter I and ensure that it matches the packet block type.

The second method, getNextPacket(), shown in Figure 10, adjusts the pointer so that we have logically isolated the next packet. Since the entire pcapng file is ingested and held in memory, to create a new object to hold individual packets is redundant, and so instead a custom pointer, currentloc, is created and incremented to move through the complete file. getNextPacket() increments this location variable by the total size of the current packet, including all pcapng file format headers/footers and the entire packet size, which brings our pointer to the next start of the next packet.

```
public static boolean isNextPacket(){
    int temploc = currentloc; //temploc used to not permanently increment currentloc
    int nextpacket = ByteBuffer.wrap(Pcap, temploc + indextolength, 4).order(ByteOrder.LITTLE_ENDIAN).getInt();
    temploc = temploc + nextpacket;

    if(temploc >= Pcap.length){
        return false;
    }

    int packettypeint = ByteBuffer.wrap(Pcap, temploc, 4).order(ByteOrder.LITTLE_ENDIAN).getInt();

    if(packettypeint == 3 || packettypeint == 6){
        return true;
    }
    else
        return false;
}

public static void getNextPacket(){
    int nextpacket = 0;

    nextpacket = ByteBuffer.wrap(Pcap, currentloc + indextolength, 4).order(ByteOrder.LITTLE_ENDIAN).getInt();
    currentloc = currentloc + nextpacket;
}
```

Figure 10. Parser Code Excerpt with Custom Methods

In Table 1, the 22 methods that comprise our pcapng class at the time of this writing are shown and briefly described. These methods collectively represent the functionality developed for reading and parsing the pcapng files.

Table 1. Method Name and Required Input, Type Returned and Purpose for Custom pcapng Class

Method Name	Return Type	Method Purpose
getFirstPacket()	void	Moves pointer to the start of the first packet, including pcapng headers
getFirstPacketData()	byte array	Returns a byte array of the first packet in pcapng file
getNextPacket()	void	Moves pointer to the packet immediately following the current packet
getNextPacketData()	byte array	Returns a byte array of the packet immediately following the current packet
isNextPacket()	boolean	Returns true if there is another packet immediately following the current packet. Returns false otherwise
getPacketSize()	int	Returns an integer value representing the total packet size of the current packet, including pcapng headers and options
getDate()	string	Returns a string with date and time of the current packets collection
getLongDate()	long	Returns the recombined 64-bit timestamp of the current packet represented as a long
getEtherType()	byte array	Returns a byte array indicating the protocol encapsulated in packet payload. E.g. IPv4 as 0x0800
getSrcMac()	byte array	Returns a byte array of the source MAC address of the current packet
getDstMac()	byte array	Returns a byte array of the destination MAC address of the current packet
getIPVersion()	char	Returns a hexadecimal character indicating the IP version of the current packet. E.g. version 4 or version 6
getIPv4HeaderLength()	int	Returns the integer value indicating the length of the IPv4 header of the current packet
getIPv4Protocol()	string	Returns a string representation of the hexadecimal code for the IPv4 protocol of the current packet
getIPv4srcIP()	string	Returns the string representation of the IPv4 source IP of the current packet. E.g. "192.168.0.1"
getIPv4dstIP()	string	Returns the string representation of the IPv4 destination IP of the current packet. E.g. "192.168.0.16"
getSrcPort(int ipheaderlength)	int	Returns the integer value of the source port used in the current packet

Method Name	Return Type	Method Purpose
getDstPort(int ipheaderlength)	int	Returns the integer value of the destination port used in the current packet
getSeqNum(int ipheaderlength)	long	Returns a 64-bit signed long representing the current packet's TCP sequence number, a 32-bit unsigned int
getAckNum(int ipheaderlength)	long	Returns a 64-bit signed long, representing the current packet's TCP acknowledgement number, a 32-bit unsigned int
getFlags(int ipheaderlength)	string	Returns a string of the hexadecimal representation of set flags for the current packet
getTCPHeaderLength	int	Returns the integer value of the TCP header length of the current packet

3. Mapper and Reducer Classes

Using the driver class and job configuration described in Section 1 and the pcapng parsing described in Section 2 with the specific methods shown in Table 1, implementing algorithms in mappers and reducers becomes easy for an analyst. The mapper and reducer classes comprise the core of the analytics for our process. One example of MapReduce analysis which was done for this thesis was determining the top talker on the network, or more generally, how much bandwidth is every individual IP using. The mapper, which is given a BytesWritable value corresponding to a byte array of one complete pcapng file, iterates through all packets, and for each packet, outputs two key/value pairs of Text and IntWritable. The text key corresponds to the source and destination IP address, and the IntWritable value corresponds to the total packet size. The code of this mapper is shown in Figure 11. We see in Figure 11 how little code is necessary for execution as a result of the methods shown in Table 1.

```

public class PCAPNGMapper extends Mapper<NullWritable, BytesWritable, Text, IntWritable> {
    public void map(NullWritable key, BytesWritable value, Context context) throws IOException, InterruptedException {

        Pcapng data = new Pcapng(value);
        data.getFirstPacket();

        while(Pcapng.isNextPacket()==true){
            context.write(new Text(data.getIPv4dstIP()), new IntWritable(data.getPacketSize()));
            context.write(new Text(data.getIPv4srcIP()), new IntWritable(data.getPacketSize()));
            data.getNextPacket();
        }
    }
}

```

Figure 11. Mapper Code for Determining Bandwidth Use per IP in pcapng File

The reducers receive a key and all values associated with that key. In this case, each reducer receives an IP and the number of bytes from each individual packet that IP has sent or received. The IP then iterates through each byte count and sums them, finally outputting the IP and the total number of bytes sent and received by that IP from all corresponding input files. The code for this reducer is shown in Figure 12. As we see in the code, the input key is Text, which is the IP address, and the input value is Iterable<IntWritable>, indicating that there is an array of IntWritable values corresponding to the byte count of each individual packet. In both the driver example and this reducer, the output key is set to IntWritable; however, it is possible that we may need to switch to LongWritable depending on the anticipated byte count per IP.

```

public class PCAPNGReducer extends Reducer<Text, IntWritable, Text, IntWritable> {
    public void reduce(Text key, Iterable<IntWritable> values, Context context) throws IOException, InterruptedException {
        int sum = 0;
        int temp = 0;

        for (IntWritable value: values) {
            temp = value.get();
            sum+=temp;
        }

        context.write(new Text(key.toString()), new IntWritable(sum));
    }
}

```

Figure 12. Reducer Code for Determining Bandwidth Use per IP in pcapng File

The logic flow of the map, shuffle/sort and reduce stages, corresponding to the code presented in Figures 11 and 12, is shown in Figure 13. In this flow, only a limited number of key/value pairs from the map task are shown; in an actual process there would be many millions or potentially billions of mapper key/value outputs. Additionally, for this thesis, since a single node VM was employed, only a single reducer is used, and only a single final output file occurs. If a larger cluster were to be used, the number of reducers would be the lesser of either the number of unique keys or number of DataNodes.

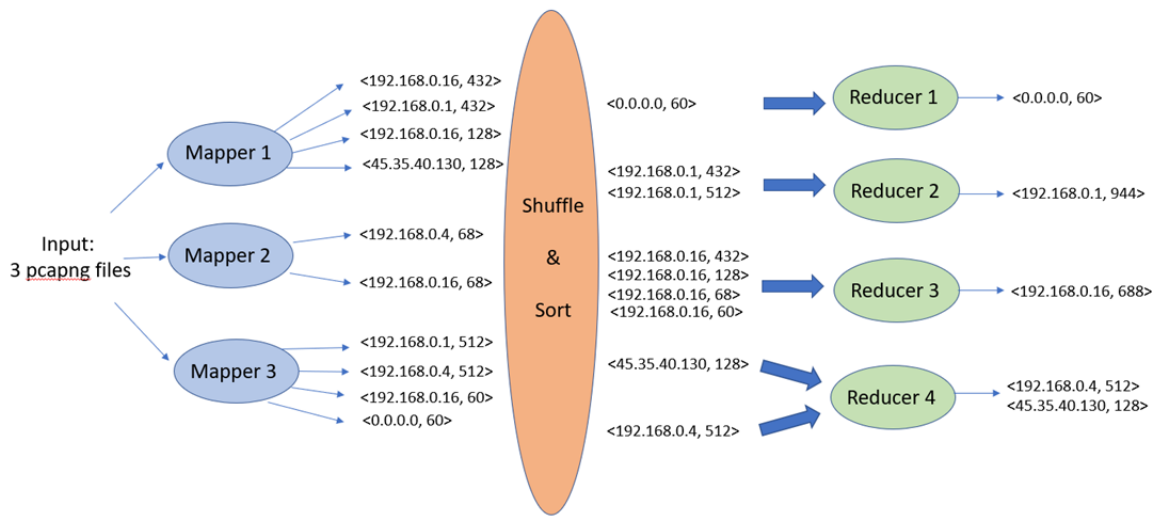


Figure 13. Example MapReduce Flow to Determine Bandwidth Use per IP Address

The logic flow shown in Figure 13 can be used to determine many standard network metrics. For example, in our mapper, by writing our key/value as port number and 1, respectively, then we can reduce and sum how many times each individual port was used. We can do the same process replacing the port number with protocol as key and determine the protocol distribution. This can be done by either network usage if we make our mapper output value the total packet size for each key or by total count by making our mapper output value a 1 for each key. It is trivial to extend this analysis to top protocols, port usage, top talkers, top error flags, or any other simple aggregate and sum network statistic; however, as can likely be discerned and as discussed in Chapter I when providing

background on Spark, MapReduce is ill suited to more complex analysis that may require repeated iterations through the data. This necessitated additional big data frameworks to conduct more sophisticated analysis.

C. MAPREDUCE DATA PARSING AND BULK HBASE LOADING

Based on the desire to calculate more sophisticated analytics, and to also conduct multiple analytics with only one read for a single job, we sought to use Spark. While accessing the data directly from a pcapng file is possible, it can be inefficient. For example, we have to potentially read many complete files to extract just a single IP we wish to analyze. Even more critically, we also have to read all the actual data in the packet that we may not care about if we are simply doing metadata analysis. To address these issues, an intermediary step between receiving raw pcapng files and conducting Spark analysis was created. This step was to load the pcapng into the NoSQL database HBase.

MapReduce provides native support to ingesting files into HBase with the `HFileOutputFormat` class. Using the methods shown in Table 1, we are able to extract relevant metadata and write it to HFiles, where the mapper output key is an `ImmutableBytesWritable` row key, and the value is a `Put` operation [25] with the column family, column name, and column value for each column. The driver and mapper code for bulk loading the data into HBase is given in Appendix B.

Before bulk loading the pcapng files into HBase using MapReduce, an HBase table was created in the HBase shell. This creation requires naming the table and identifying column families. Only one column family was used for this thesis; however, for operational use, the intention is to create two column families, one for metadata and one for the packet data. This method of storage means that there is no necessity to retain the original pcapng files, and when doing analysis, we can access just the column family of metadata if that is all that is required, without reading the packet data.

The next important consideration is that of row key design. As discussed in Chapter I, row keys are unique identifiers for a row. This uniqueness requirement means we cannot simply use the IP address from our local network as our row key since each packet creates a new version of the same row. One practice is to append a timestamp to the

row key to make it unique. Just a time stamp for a row key is not optimal, since the values are all sequential it results in “hot spots,” which is when one region is assigned all the data rather than individual regions each receiving a portion [26]. After attempting using IP_timestamp for a row key, we observed that when multiple pcapng files were ingested with millions of packets, some overlap occurred. Out of approximately 1.5 million packets, exactly six had identical timestamps down to the microsecond resolution. While pcapng supports nanosecond resolution, it is not feasible to assume that all collects have this precision. The solution employed in this thesis was to append an additional digit to the end of each row key. This digit increments between 0 and 9 and then reset to 0, ensuring that so long as no ten consecutive packets share the same microsecond resolution timestamp, all row keys are unique. The final row key used was IP_timestamp_digit. This technique was successful in ensuring unique row keys for all pcapng files parsed.

D. SPARK DATA ANALYSIS

Once the pcapng files have been put into an HBase table, we can access the data via Spark to calculate more sophisticated analytics, calculate multiple metrics without rereading the data from disk multiple times, and take advantage of Spark SQL for easy analysis and rapid aggregate and sum results. Data can be ingested into Spark from HBase in multiple ways. The primary technique employed in this thesis was to use the HBase-Spark Connector [27]; however, this method was employed to load the entire HBase table into a DataFrame. If instead we wanted to isolate specific groups of row keys, we could use the HBaseContext and the bulkGet or hBaseRDD methods to create an RDD in Spark with just the sought row key(s) [27]. The Spark-Shell can use Java or Scala as the programming languages. For this thesis, Scala was used with the Spark-Shell. There are also PySpark (python) and SparkR (R) shells which can be used [28].

To load data from HBase into Spark using the HBase-Spark Connector, we define a catalog which outlines the mapping from HBase to Spark tables [27]. This mapping consists of defining which Spark column the row key is mapped to and what Spark column each desired HBase column is mapped to. The complete code for importing HBase data into Spark and calculating network metrics is presented in Appendix C. The catalog used

for creating a table to calculate the initial round trip time (iRTT) for TCP connections is shown in Figure 14. TCP iRTT is discussed later in this section. The general syntax for mapping columns consists of providing the spark column name we want, then providing the address to the corresponding HBase column given by column family, column, and object type. As discussed in Section B of this chapter, our row key consists of the local IP_timestamp_digit. Another column specifically imported was external or destination IP; if both IPs are local, then the row key uses the sender IP. Additionally, columns were imported for the total packet size, the ethernet type, the sequence and acknowledgement numbers for TCP connections, the TCP flags, TCP header size, and the total TCP data size. The last values are important in incrementing sequence and acknowledgement numbers to follow a TCP stream, which is discussed more with an evaluation of iRTT.

```
val cat = s"""{
  "table":{"namespace":"default", "name":"/user/user01/hbasetest/table/packet"},
  "rowkey":"key",
  "columns":{
    "key":{"cf":"rowkey", "col":"key", "type":"string"},
    "ipdst":{"cf":"cf", "col":"ipdst", "type":"string"},
    "size":{"cf":"cf", "col":"size", "type":"int"},
    "ethtyp":{"cf":"cf", "col":"ethtyp", "type":"binary"},
    "seq":{"cf":"cf", "col":"seq", "type":"long"},
    "ack":{"cf":"cf", "col":"ack", "type":"long"},
    "flgs":{"cf":"cf", "col":"flgs", "type":"string"},
    "tcphdr":{"cf":"cf", "col":"tcphdr", "type":"int"},
    "tcpsze":{"cf":"cf", "col":"tcpsze", "type":"int"}
  }
}""".stripMargin
```

Figure 14. Catalog Defined for Mapping from HBase to Spark Table

It is not necessary to import all columns from the HBase table into a Spark table. If a column is omitted from the catalog it is simply not included. Alternately, if a column is identified in our catalog and does not exist in HBase, a null value is put there in Spark. This is very useful since HBase is flexible and capable of handling an arbitrary number of columns within a column family, we are able to accommodate many different network protocol structures and need not assign columns until run time. Additionally, with the sparse storage mechanism of HBase, we do not store columns that have no value [9]; however, for Spark we are using an SQL structure for speed and rows have a fixed length.

Combining these mechanisms gives us the storage flexibility and efficiency of run-time dynamic column assignment and sparsity coupled with the analytic efficiency of SQL, even when different columns corresponding to different protocols exist.

Calculating standard network metrics becomes trivial once we have the data loaded into a Spark DataFrame. Using filters, group by and sum tools, we can determine the top talkers, protocols, ports, etc. discussed in this chapter's MapReduce section. However, more sophisticated analysis can also be done. This thesis assumes the reader's familiarity with the Open Systems Interconnection (OSI) model and has not gone in depth in explaining the different layers and the structure of network traffic at each layer. However, a brief explanation of the TCP three-way handshake, which occurs to establish a connection, is warranted here. This process is also described in detail in [29]. TCP conversations between two hosts have sequence and acknowledgement numbers to ensure both the successful arrival of packets and the order of the packets, so even if packets arrive out of order they can be properly reconstructed. This is done by incrementing the sequence and acknowledgement numbers of each host by the size of the data transmitted and received, respectively.

To start this TCP conversation and set the sequence and acknowledgement numbers, a three-way handshake is used to establish a connection. This three-way handshake, with a collection device placed between the two hosts, is depicted in Figure 15. The host requesting a connection, host A, sends a packet with the SYN(chronize) flag set and an initialization sequence number. The receiving host, host B, responds with a packet with the SYN and ACK(knowledge) flags set. Host B sets its acknowledgement number to the sequence number plus one from the first packet sent by host A, and host B generates its own sequence number to respond with. Finally, host A uses the sequence number plus one from host B as its acknowledgement number and responds with the ACK flag set and its original sequence number plus one [29]. At this stage the acknowledgement and sequence number of host B and the sequence and acknowledgement number of host A are aligned. This setting, incrementing and aligning of sequence and acknowledgement numbers can be seen in Figure 15.

Now, if a capture device is placed between the two hosts establishing a TCP connection, we can use the timestamps from the captured packets to determine the initial round trip time from when a packet is sent to when the response is received. In Figure 15, we can see highlighted in blue the amount of time that elapses from when the first packet with a SYN flag is collected, and the second packet with SYN/ACK flag is collected. In green we see the amount of time that passes from when the SYN/ACK packet is collected to when the ACK packet is collected. By summing these two measurements from the initial three packets we get our iRTT [30]. The iRTT is a good indication of latency between hosts because the three-way handshake packets are small and uniform. This means that with large and numerous network captures we have a robust and consistent source of latency measurement that can be passively calculated, without introducing additional traffic into the network.

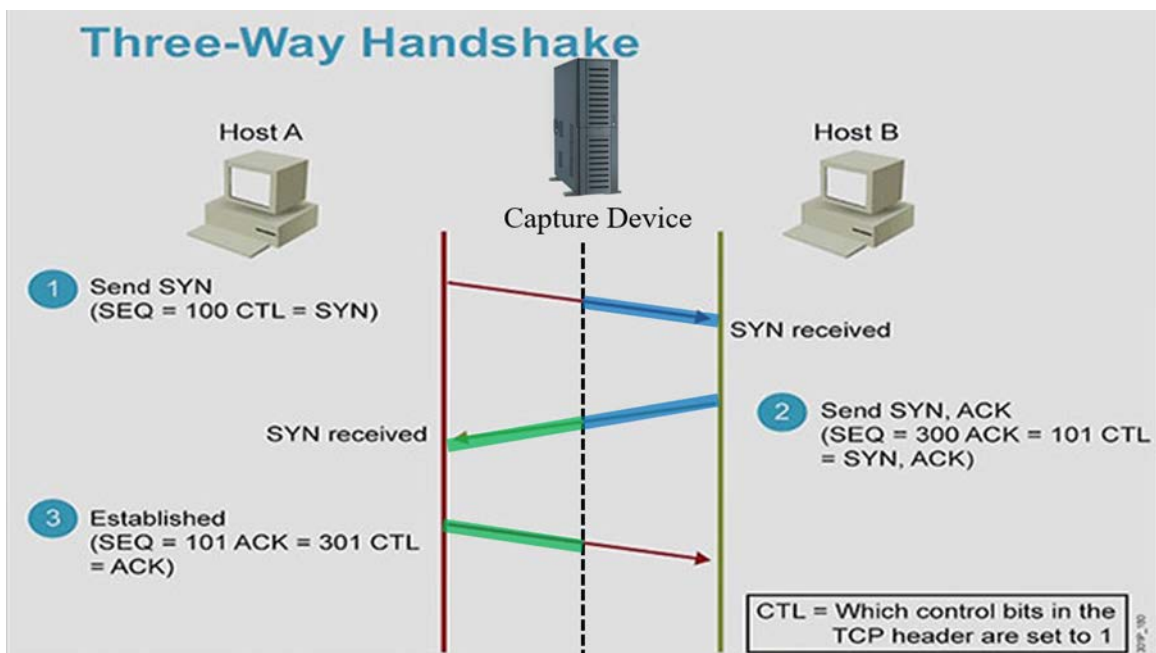


Figure 15. TCP Three-Way Handshake with Capture Device.
Adapted from: [29]

IV. RESULTS

Results were measured both in binary terms; that is, in terms of success or failure and in terms of performance when compared against the conventional packet analysis freeware Wireshark. Additionally, when evaluating individual pcapng files, and when comparisons are made against Wireshark, all results in this thesis were corroborated by Wireshark's calculations. No discrepancies between Wireshark and the Hadoop analytics occurred for any calculations. Additionally, while all Hadoop ecosystem analysis in this thesis was performed exclusively on a single-node VM, the solutions were designed with scalability in mind. The successful lateral scaling of the Hadoop ecosystem has been repeatedly verified, including in [14], [15], [16].

Finally, while almost all MCNEL data was IPv4 and all analyses are conducted using only IPv4 packets, a large portion of the household pcapng collection (see Chapter II, Section B) included IPv6. For these reasons, some code includes checks to ensure packets are IPv4. Results are broken down and presented in the same order as the thesis methodology presented in Chapter III.

A. MAPREDUCE ANALYSIS RESULTS

MapReduce analysis was done to determine top talkers, top ports, top protocol, and initially, bandwidth per IP address by day of week and hour of day. None of the packet traces collected were greater than three days in duration, and so day of week/hour of day did not prove useful and was not continued. The code for each of these is presented in Appendix A.

1. Network Usage in Bytes per IP Address

First, corresponding to the code shown in Figures 11 and 12, we calculated the total bandwidth use by IP address. This was initially done for a single, 393-MB pcapng file consisting of 1,176,590 packets. In Figure 16, we see the job information output from MapReduce. Of note, we see that we have a single input record with the Bytes Read as 393,029,132 and 2,353,180 output records. This output record count is two times the

number of input packets, as expected, since the mapper writes an output for both the source and destination IP address for each packet. The total CPU time spent to complete the analysis was 8.370 s with an output file size of 29,373 bytes written. An example of the output, showing nine of 1382 unique IP addresses, is presented in Figure 17. The output is lexicographically sorted by IP address.

```

Map-Reduce Framework
  Map input records=1
  Map output records=2353180
  Map output bytes=42486548
  Map output materialized bytes=0
  Input split bytes=144
  Combine input records=0
  Combine output records=0
  Reduce input groups=1382
  Reduce shuffle bytes=47192910
  Reduce input records=2353180
  Reduce output records=1382
  Spilled Records=7059540
  Shuffled Maps =1
  Failed Shuffles=0
  Merged Map outputs=2
  GC time elapsed (ms)=135
  CPU time spent (ms)=8370
  Physical memory (bytes) snapshot=1483218944
  Virtual memory (bytes) snapshot=10077450240
  Total committed heap usage (bytes)=1430781952
Shuffle Errors
  IO_ERROR=0
File Input Format Counters
  Bytes Read=393029132
File Output Format Counters
  Bytes Written=29373

```

Figure 16. MapReduce Metrics for Calculating Total Bytes Sent/Received per IP address. One Input pcapng File with 1,176,590 Packets

192.156.13.121	301252.0
192.156.37.171	3098.0
192.156.37.99	290817.0
192.156.7.120	312404.0
192.156.7.121	314329.0
192.156.71.120	312172.0
192.156.71.121	308827.0
192.156.74.104	1892.0
192.156.74.115	84099.0

Figure 17. Output Excerpt from MapReduce Job Determining Total Bytes Sent/Received per IP address

This analysis was then repeated for both three and six pcapng input files. The MapReduce driver, presented in Appendix A, was configured for directory inputs. For the six input files, the MapReduce metrics are presented in Figure 18. Shown in the figure, we see six Map input records with a total of 18,488,746 Map output records corresponding to 9,244,373 individual packets. The total bytes read was 2,418,083,096. A total input size larger than 2.0 GB was deliberately chosen so that we would exceed the maximum memory allocated to the Map task and Map child JVMs (see Chapter III). This confirms that lateral scaling would be successful since the tasks must be conducted individually per input. Finally, we see the CPU time spent was 57.6 s. The largest pcapng file of the six was 635.6 MB. For comparison, using Wireshark on the same machine that was executing the MapReduce code, it took 35.69 s to open the largest file and 44.0 s to run the IPv4 statistic to calculate IP address packet counts. While the time required to open the file includes calculating many additional metrics and presents much more data, considering only the Wireshark statistics and filtering time, Wireshark is still consistently outperformed by the Hadoop framework, even with just a single node.

```
Map-Reduce Framework
  Map input records=6
  Map output records=18488746
  Map output bytes=326091154
  Map output materialized bytes=0
  Input split bytes=841
  Combine input records=0
  Combine output records=0
  Reduce input groups=2489
  Reduce shuffle bytes=363068658
  Reduce input records=18488746
  Reduce output records=2489
  Spilled Records=54694372
  Shuffled Maps =6
  Failed Shuffles=0
  Merged Map outputs=7
  GC time elapsed (ms)=1156
  CPU time spent (ms)=57600
  Physical memory (bytes) snapshot=8586321920
  Virtual memory (bytes) snapshot=36647677952
  Total committed heap usage (bytes)=7908360192
Shuffle Errors
  IO_ERROR=0
File Input Format Counters
  Bytes Read=2418083096
File Output Format Counters
  Bytes Written=53065
```

Figure 18. MapReduce Metrics for Calculating Total Bytes Sent/Received per IP Address—Six Input pcapng Files with 9,244,373 Packets

2. Port Count

Several slightly varying techniques were employed to present the results for the total number of times each port was used by each IP address to send and received TCP and UDP packets. The mapper outputs a key composed of the IP address, whether the packet was sent or received represented by an R or an S, and the port number. The mapper output value for each packet is 1. After reducing, we are left with the key composed of the IP address, whether the packets were sent or received, the port number and the total count of how many total packets matched the criteria in the key. The MapReduce metrics for this job, conducted on a MCNEL-provided single pcapng file, are presented in Figure 19. As seen in Figure 19, the single input file at 666,477,436 bytes took a total of 30.28 s to process. A similar metric can be calculated in Wireshark under the IPv4 Statistic - Destinations and Ports. This Wireshark statistic presents only destination IP addresses and ports with no inclusion of sending IP addresses and ports. Even with turning off the extra burst rate calculations, it took 25 minutes and 15 s for Wireshark to finish determining only the received “R” results calculated in the MapReduce analysis. This factor of 50 difference in calculation speed is perhaps more indicative of the inability of free-ware packet analyzers to manage large pcapng files, but the scalability of this framework is nevertheless supported. The top ten ports from the output file, sorted by count, are presented in Figure 20. All of the top ten port usages were on port 53, a common Domain Name System (DNS) query port. The total output consists of 142,978 unique combinations of IP addresses, send/receive, and port number.

```

Map-Reduce Framework
  Map input records=1
  Map output records=8304160
  Map output bytes=196295381
  Map output materialized bytes=0
  Input split bytes=128
  Combine input records=0
  Combine output records=0
  Reduce input groups=142978
  Reduce shuffle bytes=212903703
  Reduce input records=8304160
  Reduce output records=142978
  Spilled Records=24912480
  Shuffled Maps =1
  Failed Shuffles=0
  Merged Map outputs=2
  GC time elapsed (ms)=376
  CPU time spent (ms)=30280
  Physical memory (bytes) snapshot=2541649920
  Virtual memory (bytes) snapshot=10099077120
  Total committed heap usage (bytes)=2429026304
Shuffle Errors
  IO_ERROR=0
File Input Format Counters
  Bytes Read=666477436
File Output Format Counters
  Bytes Written=3290006

```

Figure 19. MapReduce Metrics for Calculating Port Count by IP Address - Single Input pcapng File with 4,152,080 Packets

157.153.3.240 R 53	2189883
157.153.3.240 S 53	1183391
214.3.160.20 R 53	107350
215.65.40.37 R 53	69423
215.65.40.38 R 53	67942
214.3.160.21 R 53	67662
207.133.239.155 R 53	67354
207.133.239.156 R 53	67337
152.229.110.236 R 53	66777
205.56.178.157 S 53	36245

Figure 20. Output Excerpt from MapReduce Job Determining Port Usage by IP

3. Protocol Count

Ethernet Type Protocol count was one of the easier methods to implement as it does not require going deeper into OSI model/packet structure than layer 2. The MapReduce metrics for this job, conducted on six pcapng files, are shown in Figure 21. As shown in the figure, six input files totaling approximately 2.4 GB of data were evaluated in 27.64 s.

The output file is shown in its entirety in Figure 22. The output keys are the hexadecimal representation of the protocol, with 0x0800 representing IPv4, 0x0806 representing Address Resolution Protocol (ARP), and 0x86dd representing IPv6. The corresponding value is the number of packets of the respective protocol that were transmitted.

```
Map-Reduce Framework
  Map input records=6
  Map output records=9244373
  Map output bytes=83199357
  Map output materialized bytes=0
  Input split bytes=841
  Combine input records=0
  Combine output records=0
  Reduce input groups=3
  Reduce shuffle bytes=101688115
  Reduce input records=9244373
  Reduce output records=3
  Spilled Records=27347186
  Shuffled Maps =6
  Failed Shuffles=0
  Merged Map outputs=7
  GC time elapsed (ms)=1199
  CPU time spent (ms)=27640
  Physical memory (bytes) snapshot=8316690432
  Virtual memory (bytes) snapshot=36617711616
  Total committed heap usage (bytes)=7789871104
Shuffle Errors
  IO_ERROR=0
File Input Format Counters
  Bytes Read=2418083096
File Output Format Counters
  Bytes Written=34
```

Figure 21. MapReduce Metrics for Ethernet Protocol Count - Six Input pcapng Files with 9,244,373 Packets

0800	9137459
0806	106620
86dd	294

Figure 22. Output from MapReduce Job Determining Ethernet Protocol Counts
– Sum of Values Corresponds to 9,244,373 Packets from
Six Input pcapng Files

4. Network Usage by Hour of Day and Day of Week per IP Address

A secondary objective of this thesis was to work toward anomaly detection in network behavior. With that in mind, categorizing network traffic per IP address by time of day and day of week was the next objective attempted. While the code was able to

accomplish the desired effect, no network collects provided from MCNEL spanned more than three days, and so results had no chance of being meaningful since months of collect are required to set a realistic baseline with such specific time windows.

Slight changes to the driver had to be made for this analysis, as the mapper output key is text, where all previous analytics output keys were integers. Additionally, for this analysis, we are only interested in IP addresses on the local network, not external IP addresses. This consideration could easily be applied to all previous analyses as well with a single line of code confirming the IP address is local. For the example presented here and the code in Appendix A, we assumed all IP addresses containing “214.43.86.X” were local. All packets are iterated through by the mapper, and the output key from the mapper is a text key which is the local IP address, regardless of whether it was source or destination. The output value is a string which concatenates the day of the week the packet was sent, presented as a three-character string (e.g. “Mon,” “Tue,” “Wed”), , the hour of day of transmission, presented as a value between 0–23, and the packet size. Based on the local IP address key, these are passed to the reducer which sums all packet sizes with the same day of week and hour of day, yielding a text key with the local IP address and day of week and a text value with 24 values corresponding to total bytes sent in each hour block. The MapReduce metrics are shown in Figure 23, and an excerpt from the job output is presented in Figure 24. As shown in Figure 23, analyzing a single file with 393 MB took 8.3 s. The output excerpt shown in Figure 24 contains two unique IP addresses and the three days over which there was collection. The total output contained 151 IP addresses on the 214.43.86.0/24 each with three days analyzed, for a total of 453 rows, each with IP address, day of week, and a vector of length 24 corresponding to the 24 hours in the day and the total bytes sent in each hour.

```

Map-Reduce Framework
  Map input records=1
  Map output records=1041669
  Map output bytes=25496181
  Map output materialized bytes=0
  Input split bytes=144
  Combine input records=0
  Combine output records=0
  Reduce input groups=151
  Reduce shuffle bytes=27579521
  Reduce input records=1041669
  Reduce output records=453
  Spilled Records=2083338
  Shuffled Maps =1
  Failed Shuffles=0
  Merged Map outputs=2
  GC time elapsed (ms)=188
  CPU time spent (ms)=8300
  Physical memory (bytes) snapshot=1838600192
  Virtual memory (bytes) snapshot=10091479040
  Total committed heap usage (bytes)=1916272640
Shuffle Errors
  IO_ERROR=0
File Input Format Counters
  Bytes Read=393029132
File Output Format Counters
  Bytes Written=33864

```

Figure 23. MapReduce Metrics for Calculating Network Usage by Day of Week and Time of Day per IP Address- One Input pcapng File with 1,176,590 Packets

```

214.43.86.136 Mon 0 0 0 0 0 0 0 0 0 0 0 0 0 545394 279702 223211 205526 232662 415718 218874 173471 179957 169170 187363 136167
214.43.86.136 Tue 0 0 0 0 0 0 56357 84046 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
214.43.86.136 Wed 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
214.43.86.137 Mon 0 0 0 0 0 0 0 0 0 0 0 0 0 310301 419523 0 109676 12638 0 0 0 0
214.43.86.137 Tue 0 0 0 0 0 0 0 236764 52611 0 0 0 0 0 0 0 0 0 0 0 0 0
214.43.86.137 Wed 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

```

Figure 24. Output Excerpt from Calculating Network Usage by Day of Week and Time of Day per IP Address

B. MAPREDUCE DATA PARSING AND BULK HBASE LOADING

After exploring the capability of MapReduce to conduct some network analytics and realizing many of its limitations, the next step was to use the parsing capability developed for MapReduce analytics to bulk load packet metadata from pcapng files into HBase.

First, a table was created using the HBase shell. As discussed in Chapter I and in Chapter III, only the column family needs to be prespecified. For the purposes of this thesis, only one column family was used for all metadata columns; however, the expectation is that a second column family would be used to store the packet data. The MapReduce code used to ingest the pcapng files is presented in Appendix B. The MapReduce metrics from the job to create the HFiles is shown in Figure 25. With a directory input containing six pcapng files, totaling approximately 2.4 GB, it took 29.75 s to complete the job. For this stage we treat any IP address starting with 192.168 as a local IP address. A scan of the first five rows from the table in the HBase shell is presented in Figure 26. Remembering that HBase is logically a table but physically stored as HFiles (discussed in Chapter I), we interpret the HBase shell output as every identical row key belonging to the same row. In Figure 26 we see the first four rows have an identical row key, and in fact the first 12 rows are made up of only three unique row keys, each with four columns. The last two row keys seen in Figure 26 have eleven columns each; this is because they have an IPv4 protocol of TCP, and so additional metadata is extracted.

```

Map-Reduce Framework
  Map input records=6
  Map output records=385632
  Map output bytes=102562479
  Map output materialized bytes=0
  Input split bytes=841
  Combine input records=0
  Combine output records=0
  Reduce input groups=385632
  Reduce shuffle bytes=103863739
  Reduce input records=385632
  Reduce output records=2552992
  Spilled Records=1156896
  Shuffled Maps =6
  Failed Shuffles=0
  Merged Map outputs=7
  GC time elapsed (ms)=901
  CPU time spent (ms)=30100
  Physical memory (bytes) snapshot=9431703552
  Virtual memory (bytes) snapshot=36796481536
  Total committed heap usage (bytes)=8811708416
Shuffle Errors
  IO_ERROR=0
File Input Format Counters
  Bytes Read=2418083096

```

Figure 25. MapReduce Metrics for Loading pcapng Metadata into HBase – Six Input pcapng Files with 9,244,373 Packets

```

hbase(main):001:0> scan "/user/user01/hbasetest/table/packet", {'LIMIT' => 5}
ROW                                COLUMN+CELL
192.168.0.10_1503779553357209_6    column=cf:ethtyp, timestamp=1526678838204, value=\x00\x00
192.168.0.10_1503779553357209_6    column=cf:ipdst, timestamp=1526678838204, value=192.168.0.16
192.168.0.10_1503779553357209_6    column=cf:ipv4prctl, timestamp=1526678838204, value=01
192.168.0.10_1503779553357209_6    column=cf:sze, timestamp=1526678838204, value=\x00\x00\x00b
192.168.0.10_1503779558658981_5     column=cf:ethtyp, timestamp=1526678838204, value=\x00\x00
192.168.0.10_1503779558658981_5     column=cf:ipdst, timestamp=1526678838204, value=192.168.0.16
192.168.0.10_1503779558658981_5     column=cf:ipv4prctl, timestamp=1526678838204, value=11
192.168.0.10_1503779558658981_5     column=cf:sze, timestamp=1526678838204, value=\x00\x00\x01X
192.168.0.10_1503779559815314_4     column=cf:ethtyp, timestamp=1526678838204, value=\x00\x00
192.168.0.10_1503779559815314_4     column=cf:ipdst, timestamp=1526678838204, value=192.168.0.16
192.168.0.10_1503779559815314_4     column=cf:ipv4prctl, timestamp=1526678838204, value=11
192.168.0.10_1503779559815314_4     column=cf:sze, timestamp=1526678838204, value=\x00\x00\x01a
192.168.0.10_1503779560301254_9     column=cf:ack, timestamp=1526678838204, value=\x00\x00\x00\x00~l'
192.168.0.10_1503779560301254_9     column=cf:dstprt, timestamp=1526678838204, value=\x00\x00\x1A\xD0
192.168.0.10_1503779560301254_9     column=cf:ethtyp, timestamp=1526678838204, value=\x00\x00
192.168.0.10_1503779560301254_9     column=cf:flgs, timestamp=1526678838204, value=12
192.168.0.10_1503779560301254_9     column=cf:ipdst, timestamp=1526678838204, value=192.168.0.16
192.168.0.10_1503779560301254_9     column=cf:ipv4prctl, timestamp=1526678838204, value=06
192.168.0.10_1503779560301254_9     column=cf:seq, timestamp=1526678838204, value=\x00\x00\x00\x00Yu\
192.168.0.10_1503779560301254_9     column=cf:srcprt, timestamp=1526678838204, value=\x00\x00 \x08
192.168.0.10_1503779560301254_9     column=cf:sze, timestamp=1526678838204, value=\x00\x00\x00B
192.168.0.10_1503779560301254_9     column=cf:tcphdr, timestamp=1526678838204, value=\x00\x00\x00
192.168.0.10_1503779560301254_9     column=cf:tcpsze, timestamp=1526678838204, value=\x00\x00\x00\x00
192.168.0.10_1503779560303854_3     column=cf:ack, timestamp=1526678838204, value=\x00\x00\x00\x00~la
192.168.0.10_1503779560303854_3     column=cf:dstprt, timestamp=1526678838204, value=\x00\x00\x1A\xD0
192.168.0.10_1503779560303854_3     column=cf:ethtyp, timestamp=1526678838204, value=\x00\x00
192.168.0.10_1503779560303854_3     column=cf:flgs, timestamp=1526678838204, value=10
192.168.0.10_1503779560303854_3     column=cf:ipdst, timestamp=1526678838204, value=192.168.0.16
192.168.0.10_1503779560303854_3     column=cf:ipv4prctl, timestamp=1526678838204, value=06
192.168.0.10_1503779560303854_3     column=cf:seq, timestamp=1526678838204, value=\x00\x00\x00\x00Yu\
192.168.0.10_1503779560303854_3     column=cf:srcprt, timestamp=1526678838204, value=\x00\x00 \x08
192.168.0.10_1503779560303854_3     column=cf:sze, timestamp=1526678838204, value=\x00\x00\x00B
192.168.0.10_1503779560303854_3     column=cf:tcphdr, timestamp=1526678838204, value=\x00\x00\x00\x14
192.168.0.10_1503779560303854_3     column=cf:tcpsze, timestamp=1526678838204, value=\x00\x00\x00\x08
5 row(s) in 0.1140 seconds

```

Figure 26. Scan of the First Five Row Keys in HBase Shell

C. SPARK DATA ANALYSIS

Now that the data was successfully in HBase, we wanted to pull it from the logical table of HBase into a DataFrame in Spark to analyze. The complete code used to do this is presented in Appendix C.

To start a Spark session and load the data into a DataFrame from HBase took 15 s on the single node VM used for evaluation. The top 20 rows of the DataFrame are shown in Figure 27. There are a total of 385,632 rows corresponding to the network traffic that contained an IP address starting with 192.168. We can see in Figure 27 that the column labeled “key” contains the row key. While Spark supports splitting composite row keys during ingest from HBase and breaking them into individual columns for the DataFrame, this was not attempted in this thesis; however, using the map function on the DataFrame we are able to break out the local IP address and the timestamp into columns “col0” and “col1,” respectively. The new DataFrame is shown in Figure 28. Additionally, the display in Figure 28 shows only the top 20 rows and is filtered to only TCP transmissions with the SYN flag set (i.e., flgs = 02).

tcpsze	ipdrptcl	seqidstprt	key	eth	type	srcprt	flgs	site	phdr	szel	ipdst	ack	ipsrc
01	01	01	192.168.0.10_1503...[00 00]	00	00	00	00	00	00	00	192.168.0.16	01	01
01	01	01	192.168.0.10_1503...[00 00]	00	00	00	00	00	00	00	192.168.0.16	01	01
01	01	01	192.168.0.10_1503...[00 00]	00	00	00	00	00	00	00	192.168.0.16	01	01
01	06	06	1508900981	6864	192.168.0.10_1503...[00 00]	8200	12	32	66	192.168.0.16	2121032042	01	01
10	06	06	1508900982	6864	192.168.0.10_1503...[00 00]	8200	10	20	64	192.168.0.16	2121032042	01	01
1460	06	06	1508900982	6864	192.168.0.10_1503...[00 00]	8200	10	20	1514	192.168.0.16	2121032042	01	01
960	06	06	1508902442	6864	192.168.0.10_1503...[00 00]	8200	10	20	1014	192.168.0.16	2121032042	01	01
10	06	06	1508903482	6864	192.168.0.10_1503...[00 00]	8200	11	20	64	192.168.0.16	2121032042	01	01
10	06	06	1508903483	6864	192.168.0.10_1503...[00 00]	8200	10	20	64	192.168.0.16	2121032043	01	01
0	06	06	1505658029	6865	192.168.0.10_1503...[00 00]	8200	12	32	66	192.168.0.16	4195999645	01	01
10	06	06	1505658030	6865	192.168.0.10_1503...[00 00]	8200	10	20	64	192.168.0.16	4195999842	01	01
1460	06	06	1505658030	6865	192.168.0.10_1503...[00 00]	8200	10	20	1514	192.168.0.16	4195999842	01	01
960	06	06	1505659490	6865	192.168.0.10_1503...[00 00]	8200	10	20	1014	192.168.0.16	4195999842	01	01
10	06	06	1505660450	6865	192.168.0.10_1503...[00 00]	8200	11	20	64	192.168.0.16	4195999842	01	01
10	06	06	1505660451	6865	192.168.0.10_1503...[00 00]	8200	10	20	64	192.168.0.16	4195999843	01	01
0	06	06	1493627421	6866	192.168.0.10_1503...[00 00]	8200	12	32	66	192.168.0.16	1901184335	01	01
10	06	06	1493627422	6866	192.168.0.10_1503...[00 00]	8200	10	20	64	192.168.0.16	1901184467	01	01
710	06	06	1493627422	6866	192.168.0.10_1503...[00 00]	8200	10	20	772	192.168.0.16	1901184467	01	01
10	06	06	1493628140	6866	192.168.0.10_1503...[00 00]	8200	11	20	64	192.168.0.16	1901184467	01	01
10	06	06	1493628141	6866	192.168.0.10_1503...[00 00]	8200	10	20	64	192.168.0.16	1901184468	01	01

Figure 27. Spark DataFrame Created from HBase Table with Composite Row Key as Single Column

tcpsze	ip	iprtcl	seqid	stprt	eth	typ	srcprt	if	lgs	tcph	drisze	ipdst	ack	ipsrc	col0	col1
0	06	185422813	8443	[08 00]		6632	02	32	66	198.31.193.211	0	null	192.168.0.16	1583779554589393		
0	06	184842588	881	[08 00]		6638	02	32	66	192.168.0.1	0	null	192.168.0.16	1583779556217868		
0	06	1525735432	881	[08 00]		6817	02	32	66	192.168.0.1	0	null	192.168.0.16	1583779556285448		
0	06	3147957628	881	[08 00]		6818	02	32	66	192.168.0.1	0	null	192.168.0.16	1583779556323588		
0	06	3553478644	881	[08 00]		6828	02	32	66	192.168.0.1	0	null	192.168.0.16	1583779556381578		
0	06	2292586658	881	[08 00]		6821	02	32	66	192.168.0.1	0	null	192.168.0.16	1583779556415486		
0	06	1591828241	881	[08 00]		6822	02	32	66	192.168.0.1	0	null	192.168.0.16	1583779556445886		
0	06	2778929847	881	[08 00]		6823	02	32	66	192.168.0.1	0	null	192.168.0.16	1583779556489324		
0	06	3882784213	881	[08 00]		6824	02	32	66	192.168.0.1	0	null	192.168.0.16	1583779556553241		
0	06	2953613241	881	[08 00]		6825	02	32	66	192.168.0.1	0	null	192.168.0.16	1583779556685888		
0	06	3495958514	881	[08 00]		6826	02	32	66	192.168.0.1	0	null	192.168.0.16	1583779556629554		
0	06	1672369879	881	[08 00]		6827	02	32	66	192.168.0.1	0	null	192.168.0.16	1583779556982855		
0	06	3344191727	881	[08 00]		6828	02	32	66	192.168.0.1	0	null	192.168.0.16	1583779557862542		
0	06	1728975865	443	[08 00]		6829	02	32	66	48.77.224.254	0	null	192.168.0.16	1583779557229882		
0	06	669539165	443	[08 00]		6830	02	32	66	65.52.188.229	0	null	192.168.0.16	1583779557249652		
0	06	4831868775	443	[08 00]		6831	02	32	66	64.4.54.36	0	null	192.168.0.16	1583779557288123		
0	06	185422813	8443	[08 00]		6632	02	32	66	198.31.193.211	0	null	192.168.0.16	1583779557518851		
0	06	4111134695	881	[08 00]		6631	02	32	66	13.187.4.52	0	null	192.168.0.16	1583779558858895		
0	06	2769646388	3911	[08 00]		6839	02	32	66	192.168.0.21	0	null	192.168.0.16	1583779559921264		
0	06	1337755512	3911	[08 00]		6840	02	32	66	192.168.0.21	0	null	192.168.0.16	1583779559325762		

Figure 28. Spark DataFrame Created from HBase Table with Composite Row Key Decomposed into Col0 – Local IP and Col1 – Timestamp, and Filtered by SYN Flag Set

Now that the data is in a DataFrame, it is easier to perform the same network analysis done in Part A of this chapter using Spark SQL. The simple code used to determine total network usage per IP, the total count of each IPv4 protocol, and the total count of source ports is shown in Figure 29. Since empty fields are filled with null values, there is no need to check for protocol type before performing an operation like source port count. Additionally, with the speed and efficiency of SQL queries, these all execute in under 1.0 s. The output from these three analytics is presented in Figure 30. For the IPv4 protocol distribution shown in the center of Figure 30, 11 corresponds to UDP, 01 to Internet Control Message Protocol (ICMP), 02 to Internet Group Management Protocol (IGMP), and 06 to TCP. Finally, for the source port distribution shown on the far right of Figure 30, only the top 20 source ports are displayed.

```
df1.groupBy("col0").agg(sum("size")).show
df1.groupBy("ipv4prtcl").count.show
df1.groupBy("srcprt").count.show
```

Figure 29. Code Used to Determine Bytes Transmitted/Received per IP Address, IPv4 Protocol Distribution and Source Port Distribution

col0	sum(size)
192.168.0.19	622
192.168.0.22	3764
192.168.0.12	920835
192.168.0.10	271368
192.168.0.20	458712
192.168.0.13	23220
192.168.0.18	27430
192.168.0.1	718900
192.168.0.15	39344
192.168.0.23	181680
192.168.0.21	78405
192.168.0.16	163681446
192.168.0.17	196

ipv4prctl	count
11	240184
01	179
02	917
06	144352

srcprt	count
443	70683
80	24410
11215	6105
2099	3866
17254	2831
7903	2556
7083	1400
7169	985
17269	841
17257	714
7856	595
7815	570
16884	570
5223	558
7842	504
15049	504
7889	452
7087	326
7043	295
9197	293

Figure 30. Output from Spark Queries Determining, from Left to Right, Bytes Transmitted/Received per IP Address, IPv4 Protocol Distribution, and Source Port Distribution

Lastly, now that we are able to operate with repeated access to the data, we calculated the iRTT for all TCP three-way handshakes. Once again, the code for this is shown in Appendix C. Filtering for an IP-IP pair, specifically 192.168.0.16 -> 192.168.0.1, all of the SYN flag-only packets are identified and their sequence numbers collected as a list. This list is then iterated through, and for each we find the corresponding acknowledgement number and the SYN/ACK and ACK packets. For this specific test, there were 101 SYN flag only packets between those two IP addresses. The output is a printed list of the two RTTs from the three-way handshake. An excerpt of seven of the 101 RTT pairs is presented in Figure 31. Since the capture device for this traffic was the local host itself, it is expected that the first RTT will be substantially larger than the second since the second RTT only corresponds to the time from when the SYN/ACK is received by the local machine to when the local machine responds with an ACK.

```
5059 63
4294 86
4927 44
5842 41
6465 40
6660 38
8434 48
```

Figure 31. Excerpt of Spark Output, Calculating the Round Trip Time Pairs
Comprising the Initial Round Trip Time of a TCP Three-Way
Handshake

V. CONCLUSION

A. SUMMARY

The purpose of this research was to provide the Marine Corps Network Efficiency Laboratory with prototype capability to conduct automated, large scale packet analysis. This was accomplished employing open-source software designed for use on commodity hardware, specifically, the Apache Hadoop ecosystem. In this thesis, we took an incremental approach, first seeking to conduct network analytics on Packet Capture Next Generation (pcapng) files using MapReduce. After a successful evaluation on the strengths and limitations of MapReduce for analysis, a more structured storage mechanism than pcapng was sought.

HBase, a NoSQL database employing the HDFS, was selected as a preferred storage mechanism, where columns can be dynamically assigned based on packet protocol and packet data can be separated from metadata. This separation allows for access to data without sequentially reading every file in its entirety. HBase data was then mapped to a Spark DataFrame to conduct multiple types of analysis and more sophisticated analysis than is practical in just MapReduce.

The types of analysis conducted were traditional network metrics, most of which are already performed by the Marine Corps Network Efficiency Laboratory. The emphasis was on evaluating these metrics in a big data framework, allowing for scaling to many nodes simultaneously and completing the work on terabyte and petabyte size data sets. While only a single node virtual machine was used for this thesis research, input data sizes larger than allocated memory were used to validate the concept of scaling, and the code produced is viable for use on an arbitrarily large Hadoop cluster.

The metrics calculated in MapReduce were network usage per IP address, port counts, protocol counts, and network usage by hour of day and day of week per IP address. The network usage by IP address and port count metrics were repeated in Spark. Additionally, IP protocol count and TCP initial round-trip time metrics were also calculated in Spark. Even operating in a single-node virtual machine, the code executed for this thesis

research was able to often outperform a free-ware packet analyzer in terms of speed. In some cases, a substantial performance increase was shown, e.g., port count.

Ultimately, in this thesis, we demonstrated the viability of the Apache Hadoop ecosystem for automated bulk data analytics on pcapng files. In doing so, the whole file input format class and record reader class from [2] were used with a custom class developed to directly ingest pcapng files into Hadoop without requiring an intermediary file format (e.g., text, csv, JSON). This capability was coupled with custom analytics developed to determine network metrics in both MapReduce and Spark, using HBase as a long-term storage format.

B. FUTURE WORK

There are four immediate goals which should be pursued as future work on this project, both at NPS and at MCNEL.

a. Increasing Functionality and Supported Protocols

The first goal is to increase the capability of the pcapng class used for parsing the files. At a minimum, IPv6 supportability should be added and IPv4 protocol support expanded. As IPv6 continues to expand, this becomes increasingly relevant. Once this support is added to the pcapng parsing methods, the analytic code needs to be revised to check for protocols and extract metadata as needed, similar to what was done for this thesis research by confirming IPv4 and TCP in the HBase bulk loading.

b. Verify capacity with Large Cluster and Large Collect

The second goal is to verify the capability explored in this thesis with a large dataset, either on a local Hadoop cluster or employing a cloud service like Amazon Web Services (AWS). An additional requirement that accompanies this future work is to obtain a large-scale persistent collect from a network. Ideally, several weeks or month of data will be gathered and terabytes of network traffic analyzed.

c. *Employ Machine Learning to Establish Baselines and Automatically Identify Anomalous Network Behavior*

An original goal of this thesis, which went unmet, was to automatically identify what is normal and abnormal behavior. For example, if we know the average network usage of every IP address organized by time of day and day of week, can we identify if that IP address is acting abnormally? Similar questions can be asked for latency. If we are aware of the iRTT distribution from each IP to IP TCP connection, can we employ some form of support vector machine, Gaussian mixture model or clustering algorithm to classify abnormal latency? In this thesis, we effectively established the data framework for the data preparation necessary to answer these questions. The limiting factor that prevented any follow-on machine learning for anomaly detection was the absence of robust and persistent collection. Months of data are needed for evaluation and ingestion to have reliable output. Accordingly, once the second goal of verifying capacity with a large collect is satisfied, it is conceivable machine learning for network anomaly detection can be explored.

d. *Increase Ease of Analyst Use and Improve Metric Output Formatting*

The final goal for future work is to create a user interface for an analyst to easily run the underlying algorithms. An interface where queries can be easily submitted, specifically controlling what data is ingested into Spark from HBase, what metrics are calculated, and how they are presented, drastically increases the utility of this thesis capability development. Additionally, currently metrics are produced exclusively in table format. By exploring a dashboard, graphical outputs can easily be included and automatically generated, increasing the utility and interpretability of the analytics being conducted.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A. MAPREDUCE ANALYTICS

A. PARSING CODE FOR PCAPNG FILES

```
package PCAPNG;
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.TimeZone;
import org.apache.commons.codec.binary.Hex;
import java.util.Arrays;
import java.nio.ByteBuffer;
import java.nio.ByteOrder;
import org.apache.hadoop.io.BytesWritable;

public class Pcapng {

    public static byte[] Pcap = null;
    public static int currentloc = 0;
    public static int ipheaderlength = 0;
    public static final int indextolength = 4;

    public Pcapng(BytesWritable value){
        Pcap = value.getBytes();
    }

    //Cycle through packet methods

    public static byte[] getFirstPacketData() {

        int nextpacket = 0;
        //global header
        nextpacket = ByteBuffer.wrap(Pcap, currentloc + indextolength,
4).order(ByteOrder.LITTLE_ENDIAN).getInt();
        currentloc = currentloc + nextpacket;
        //serial header
        nextpacket = ByteBuffer.wrap(Pcap, currentloc + indextolength,
4).order(ByteOrder.LITTLE_ENDIAN).getInt();
        currentloc = currentloc + nextpacket;
        //first data packet
        nextpacket = ByteBuffer.wrap(Pcap, currentloc + indextolength,
4).order(ByteOrder.LITTLE_ENDIAN).getInt();
        return Arrays.copyOfRange(Pcap, currentloc, currentloc +
nextpacket);
    }
}
```

```

}

public static void getFirstPacket(){

    int nextpacket = 0;
    //global header
    nextpacket = ByteBuffer.wrap(Pcap, currentloc + indextolength,
4).order(ByteOrder.LITTLE_ENDIAN).getInt();
    currentloc = currentloc + nextpacket;
    //serial header
    nextpacket = ByteBuffer.wrap(Pcap, currentloc + indextolength,
4).order(ByteOrder.LITTLE_ENDIAN).getInt();
    currentloc = currentloc + nextpacket;
}

public static void getNextPacket(){

    int nextpacket =0;
    nextpacket = ByteBuffer.wrap(Pcap, currentloc + indextolength,
4).order(ByteOrder.LITTLE_ENDIAN).getInt();
    currentloc = currentloc + nextpacket;
}

public static byte[] getNextPacketData(){

    byte[] totalpacketlength = null;
    int nextpacket =0;
    nextpacket = ByteBuffer.wrap(Pcap, currentloc + indextolength,
4).order(ByteOrder.LITTLE_ENDIAN).getInt();
    currentloc = currentloc + nextpacket;
    nextpacket = ByteBuffer.wrap(Pcap, currentloc + indextolength,
4).order(ByteOrder.LITTLE_ENDIAN).getInt();
    return Arrays.copyOfRange(Pcap, currentloc, currentloc +
nextpacket);
}

public static boolean isNextPacket(){

    int temploc = currentloc; //temploc used to not permanently
increment currentloc
    int nextpacket = ByteBuffer.wrap(Pcap, temploc + indextolength,
4).order(ByteOrder.LITTLE_ENDIAN).getInt();
    temploc = temploc + nextpacket;
    if(temploc>=Pcap.length){

```

```

        return false;
    }
    int packettypepoint = ByteBuffer.wrap(Pcap, temploc,
4).order(ByteOrder.LITTLE_ENDIAN).getInt();
    if(packettypepoint == 3 || packettypepoint == 6){
        return true;
    }
    else
        return false;
}

public static int getPacketSize(){

    int packetsizeint =
ByteBuffer.wrap(Pcap, currentloc+24, 4).order(ByteOrder.LITTLE_ENDIAN).ge
tInt();
    //byte[] packetsizeint =
Arrays.copyOfRange(Pcap, currentloc+24, currentloc+28);
    return packetsizeint;
}

public static Long getLongDate(){

    byte[] hightime = Arrays.copyOfRange(Pcap, currentloc+12,
currentloc+12+4);
    byte[] lowtime = Arrays.copyOfRange(Pcap, currentloc+16,
currentloc+16+4);
    byte[] totaltime = new byte[8];

    for(int x=7;x>=4;x--){
        totaltime[x] = lowtime[7-x];
    }

    for(int x=3;x>=0;x--){
        totaltime[x] = hightime[3-x];
    }
    long windowsTime = ByteBuffer.wrap(totaltime, 0, 8).getLong();
    long unixTime = (windowsTime / 1000);
    return windowsTime;
}

public static String getDate(){

```

```

    byte[] hightime = Arrays.copyOfRange(Pcap, currentloc+12,
currentloc+12+4);
    byte[] lowtime = Arrays.copyOfRange(Pcap, currentloc+16,
currentloc+16+4);
    byte[] totaltime = new byte[8];

    for(int x=7;x>=4;x--){
        totaltime[x] = lowtime[7-x];
    }

    for(int x=3;x>=0;x--){
        totaltime[x] = hightime[3-x];
    }
    long windowsTime = ByteBuffer.wrap(totaltime, 0, 8).getLong();
    long unixTime = (windowsTime / 1000);

    SimpleDateFormat sdf = new SimpleDateFormat("EEE HH:mm:ss:SSS");
    //SimpleDateFormat sdf = new SimpleDateFormat("MM/dd/yyyy
HH:mm:ss:SSS z");
    //sdf.setTimeZone(TimeZone.getTimeZone("UTC-7"));
    String date = sdf.format(new Date(unixTime));
    return date;
}

//Ethernet Frame Methods
//Pcapng packet header 28 bytes to data
//Ethernet Frame: 6 byte dest -> 6 byte src -> 2 byte type

public static byte[] getEtherType(){

    byte[] ethertypebytes = Arrays.copyOfRange(Pcap, currentloc+28+12,
currentloc+40+2);
    return ethertypebytes;
}

public static byte[] getSrcMac(){

    byte[] srcmac = Arrays.copyOfRange(Pcap, currentloc +34,
currentloc+34+6);
    return srcmac;
}

public static byte[] getDstMac(){

```

```

        byte[] dstmac = Arrays.copyOfRange(Pcap, currentloc + 28,
currentloc+28+6);
        return dstmac;
    }

    //
    //IP methods
    public static String getIPv4Protocol(){

        String ipv4protocol = "";
        byte[] ipv4version = Arrays.copyOfRange(Pcap, currentloc + 28 + 14
+ 9, currentloc+28+14+10);
        ipv4protocol = (Hex.encodeHexString(ipv4version));
        return ipv4protocol;
    }

    public static char getIPVersion(){

        char ipversionchar = 'a';
        byte[] ipversion = Arrays.copyOfRange(Pcap, currentloc + 28 + 14,
currentloc+28+14+1);
        ipversionchar = (Hex.encodeHexString(ipversion)).charAt(0);
        return ipversionchar;
    }

    public static int getIPv4HeaderLength(){

        char holding =
Hex.encodeHexString(Arrays.copyOfRange(Pcap,currentloc+28 + 14,
currentloc + 28+14+1)).charAt(1);
        int ipheaderint = Character.digit(holding,16);
        return ipheaderint * 4;
    }

    public static String getIPv4srcIP(){

        String ipv4srcstring = "";
        int[] ipv4srcipint = new int[4];
        byte[] ipv4src =
Arrays.copyOfRange(Pcap,currentloc+28+14+12,currentloc+28+14+12+4);

        ipv4srcipint[0] = ipv4src[0] & 0xFF;
        ipv4srcipint[1] = ipv4src[1] & 0xFF;
        ipv4srcipint[2] = ipv4src[2] & 0xFF;

```

```

        ipv4srcipint[3] = ipv4src[3] & 0xFF;

        ipv4srcstring = Integer.toString(ipv4srcipint[0])
+ "." + Integer.toString(ipv4srcipint[1]) + "." + Integer.toString(ipv4srcipin
t[2]) + "." + Integer.toString(ipv4srcipint[3]);
        return ipv4srcstring;
    }

    public static String getIPv4dstIP(){

        String ipv4srcstring = "";
        int[] ipv4srcipint = new int[4];
        byte[] ipv4src =
Arrays.copyOfRange(Pcap, currentloc+28+14+16, currentloc+28+14+16+4);

        ipv4srcipint[0] = ipv4src[0] & 0xFF;
        ipv4srcipint[1] = ipv4src[1] & 0xFF;
        ipv4srcipint[2] = ipv4src[2] & 0xFF;
        ipv4srcipint[3] = ipv4src[3] & 0xFF;

        ipv4srcstring = Integer.toString(ipv4srcipint[0])
+ "." + Integer.toString(ipv4srcipint[1]) + "." + Integer.toString(ipv4srcipin
t[2]) + "." + Integer.toString(ipv4srcipint[3]);
        return ipv4srcstring;
    }

    //
    //tcp methods
    public static int getSrcPort(int ipheaderlength){

        String srcport =
Hex.encodeHexString(Arrays.copyOfRange(Pcap, currentloc+28 + 14 +
ipheaderlength, currentloc+28 + 14 + ipheaderlength + 2));
        int srcportint = Integer.parseInt(srcport, 16);
        return srcportint;
    }

    public static int getDstPort(int ipheaderlength){

        String dstport =
Hex.encodeHexString(Arrays.copyOfRange(Pcap, currentloc+28 + 14 +
ipheaderlength+2, currentloc+28 + 14 + ipheaderlength + 4));
        int dstportint = Integer.parseInt(dstport, 16);
        return dstportint;
    }

```

```

}

public static long getSeqNum(int ipheaderlength){

    byte[] bytes =
Arrays.copyOfRange(Pcap,currentloc+28+14+ipheaderlength+4,
currentloc+28+14+ipheaderlength+8);

    long result = 0x00FF & bytes[0];
    result <= 8;
    result += 0x00FF & bytes[1];
    result <= 8;
    result += 0x00FF & bytes[2];
    result <= 8;
    result += 0x00FF & bytes[3];

    return result;
}

public static long getAckNum(int ipheaderlength){

    byte[] bytes =
Arrays.copyOfRange(Pcap,currentloc+28+14+ipheaderlength+8,
currentloc+28+14+ipheaderlength+12);

    long result = 0x00FF & bytes[0];
    result <= 8;
    result += 0x00FF & bytes[1];
    result <= 8;
    result += 0x00FF & bytes[2];
    result <= 8;
    result += 0x00FF & bytes[3];

    return result;
}

public static String getFlags(int ipheaderlength){

    String flags =
Hex.encodeHexString(Arrays.copyOfRange(Pcap,currentloc+28+14+ipheaderle
ngth+13, currentloc+28+14+ipheaderlength+14));
    return flags;
}

```

```

public static int getTCPHeaderLength(int ipheaderlength){

    char holding =
Hex.encodeHexString(Arrays.copyOfRange(Pcap, currentloc+28 +
14+ipheaderlength+12, currentloc + 28+14+ipheaderlength+13)).charAt(0);
    int tcpheaderint = Character.digit(holding,16);
    return tcpheaderint * 4;

}
}

```

B. WHOLE FILE INPUT FORMAT AND WHOLE FILE RECORD READER

```

package PCAPNG;

import java.io.IOException;
import org.apache.hadoop.fs.*;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapreduce.InputSplit;
import org.apache.hadoop.mapreduce.JobContext;
import org.apache.hadoop.mapreduce.RecordReader;
import org.apache.hadoop.mapreduce.TaskAttemptContext;
import org.apache.hadoop.mapreduce.lib.input.*;

//vv WholeFileInputFormat
public class WholeFileInputFormat
    extends FileInputFormat<NullWritable, BytesWritable> {

    @Override
    protected boolean isSplittable(JobContext context, Path file) {
        return false;
    }

    @Override
    public RecordReader<NullWritable, BytesWritable> createRecordReader(
        InputSplit split, TaskAttemptContext context) throws IOException,
        InterruptedException {
        WholeFileRecordReader reader = new WholeFileRecordReader();
        reader.initialize(split, context);
        return reader;
    }
}

package PCAPNG;
import java.io.IOException;

```

```

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FSDataInputStream;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.BytesWritable;
import org.apache.hadoop.io.IOUtils;
import org.apache.hadoop.io.NullWritable;
import org.apache.hadoop.mapreduce.InputSplit;
import org.apache.hadoop.mapreduce.RecordReader;
import org.apache.hadoop.mapreduce.TaskAttemptContext;
import org.apache.hadoop.mapreduce.lib.input.FileSplit;

//vv WholeFileRecordReader
class WholeFileRecordReader extends RecordReader<NullWritable,
BytesWritable> {

    private FileSplit fileSplit;
    private Configuration conf;
    private BytesWritable value = new BytesWritable();
    private boolean processed = false;

    @Override
    public void initialize(InputSplit split, TaskAttemptContext context)
        throws IOException, InterruptedException {
        this.fileSplit = (FileSplit) split;
        this.conf = context.getConfiguration();
    }

    @Override
    public boolean nextKeyValue() throws IOException,
InterruptedException {
        if (!processed) {
            byte[] contents = new byte[(int) fileSplit.getLength()];
            Path file = fileSplit.getPath();
            FileSystem fs = file.getFileSystem(conf);
            FSDataInputStream in = null;
            try {
                in = fs.open(file);
                IOUtils.readFully(in, contents, 0, contents.length);
                value.set(contents, 0, contents.length);
            } finally {
                IOUtils.closeStream(in);
            }
        }
    }
}

```

```

        processed = true;
        return true;
    }
    return false;
}

@Override
public NullWritable getCurrentKey() throws IOException,
InterruptedException {
    return NullWritable.get();
}

@Override
public BytesWritable getCurrentValue() throws IOException,
    InterruptedException {
    return value;
}

@Override
public float getProgress() throws IOException {
    return processed ? 1.0f : 0.0f;
}

@Override
public void close() throws IOException {
}
}

```

C. COMPILE AND RUN SCRIPTS

```

#!/bin/bash

export HADOOP_HOME=/opt/mapr/hadoop/hadoop-2.7.0
export LD_LIBRARY_PATH=$HADOOP_HOME/lib/native/Linux-amd64-64
export
CLASSPATH=$HADOOP_HOME/share/hadoop/mapreduce/*:$HADOOP_HOME/share/hado
op/mapreduce/lib/*:$HADOOP_HOME/share/hadoop/common/lib/*:$HADOOP_HOME/
share/hadoop/common/*:$HADOOP_HOME/share/hadoop/tools/lib/*:$HADOOP_HOM
E/share/hadoop/yarn/*:$HADOOP_HOME/share/hadoop/yarn/lib/*
export HADOOP_CLASSPATH=$CLASSPATH

javac -d classes WholeFileRecordReader.java WholeFileInputFormat.java
Pcapng.java PCAPNGMapper.java PCAPNGReducer.java PCAPNGDriver.java

jar -cvf PCAPNG.jar -C classes/ .

```

```
#!/bin/bash
export HADOOP_HOME=/opt/mapr/hadoop/hadoop-2.5.1
export LD_LIBRARY_PATH=$HADOOP_HOME/lib/native/Linux-amd64-64
export
CLASSPATH=$HADOOP_HOME/share/hadoop/mapreduce/*:$HADOOP_HOME/share/hadoop/mapreduce/lib/*:$HADOOP_HOME/share/hadoop/common/lib/*:$HADOOP_HOME/share/hadoop/common/*:$HADOOP_HOME/share/hadoop/tools/lib/*:$HADOOP_HOME/share/hadoop/yarn/*:$HADOOP_HOME/share/hadoop/yarn/lib/*
export HADOOP_CLASSPATH=$CLASSPATH

USER=`whoami`

rm -rf /user/$USER/mapred/OUT

hadoop jar PCAPNG.jar PCAPNG.PCAPNGDriver /user/$USER/data/
/user/$USER/mapred/OUT
```

D. NETWORK USAGE PER IP CODE

1. Driver Code

```
package PCAPNG;

import java.util.*;
import org.apache.hadoop.conf.Configured;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;

public class PCAPNGDriver extends Configured implements Tool {

    public int run(String[] args) throws Exception {
        // check the CLI
        if (args.length != 2) {
```

```

        System.err.println("usage: hadoop jar PCAPNG.jar
PCAPNG.PCAPNGDriver <inputfile> <outputdir>");
        System.exit(1);
    }
    // setup the Job

    Job job = new Job(getConf(), "pcapng" +
System.getProperty("user.name"));
    job.setJarByClass(PCAPNGDriver.class);
    job.setMapperClass(PCAPNGMapper.class);
    job.setReducerClass(PCAPNGReducer.class);
    job.setInputFormatClass(WholeFileInputFormat.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);

    // setup input and output paths
    FileInputFormat.setInputDirRecursive(job, true);
    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));

    return job.waitForCompletion(true) ? 0 : 1;
}

public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    System.exit(ToolRunner.run(new PCAPNGDriver(), args));
}
}

```

2. Mapper Code for Network Usage per IP

```

package PCAPNG;

import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.BytesWritable;
import org.apache.hadoop.io.NullWritable;
import java.io.IOException;

public class PCAPNGMapper extends Mapper
<NullWritable, BytesWritable, Text, IntWritable> {

```

```

    public void map(NullWritable key, BytesWritable value, Context
context) throws IOException, InterruptedException {

        Pcapng data = new Pcapng(value);
        data.getFirstPacket();
        while(Pcapng.isNextPacket()==true){
            context.write(new Text(data.getIPv4dstIP()), new
IntWritable(data.getPacketSize()));
            context.write(new Text(data.getIPv4srcIP()), new
IntWritable(data.getPacketSize()));
            data.getNextPacket();
        }
        //repeated for the last packet
        context.write(new Text(data.getIPv4dstIP()), new
IntWritable(data.getPacketSize()));
        context.write(new Text(data.getIPv4srcIP()), new
IntWritable(data.getPacketSize()));
    }
}

```

3. Reducer Code for Network Usage by IP

```

package PCAPNG;

import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.io.FloatWritable;
import org.apache.hadoop.io.IntWritable;
import java.io.IOException;

public class PCAPNGReducer extends Reducer
<Text,IntWritable,Text,FloatWritable> {

    public void reduce(Text key, Iterable<IntWritable> values, Context
context) throws IOException, InterruptedException {
        int sum = 0;
        // counter used for calculated average in experimentation
        // int count = 0;
        int temp = 0;
        for (IntWritable value: values) {
            temp = value.get();
            sum+=temp;
        }
    }
}

```

```

        context.write(new Text(key.toString()), new FloatWritable(sum));
        //other metrics calculated including min packet size per IP, max,
        average and sum
        //context.write(new Text(key.toString() + "_" + "min"), new
        FloatWritable(min));
        //context.write(new Text(key.toString() + "_" + "sum"), new
        FloatWritable(sum));
    }
}

```

E. PORT COUNT CODE

1. Port Count Mapper Code

```

package PCAPNG;

import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.BytesWritable;
import org.apache.hadoop.io.NullWritable;
import java.util.Arrays;
import java.io.IOException;

public class PCAPNGMapper extends Mapper
<NullWritable,BytesWritable,Text,IntWritable> {

    public void map(NullWritable key, BytesWritable value, Context
context) throws IOException, InterruptedException {

        int count = 1;
        Pcapng data = new Pcapng(value);

        data.getFirstPacket();
        while(Pcapng.isNextPacket()==true){

            //confirm IPv4, only relevant for some of the later
            testing when using local data with IPv6 included
            if(data.getEtherType().contains("08")) {
                //confirm protocol is TCP or UDP
                if(data.getIPv4Protocol().contains("6") ||
data.getIPv4Protocol().contains("11")){

```

```

                                context.write(new
Text(data.getIPv4dstIP() + " R " +
Integer.toString(data.getDstPort(data.getIPv4HeaderLength()))), new
IntWritable(count));

                                context.write(new
Text(data.getIPv4srcIP() + " S " +
Integer.toString(data.getSrcPort(data.getIPv4HeaderLength()))), new
IntWritable(count));

                                data.getNextPacket();

                        }      }

    }

//repeated for the last packet
    if(data.getEtherType().contains("08")) {
        if(data.getIPv4Protocol().contains("6") ||
data.getIPv4Protocol().contains("11")){
            context.write(new Text(data.getIPv4dstIP() + " R
" + Integer.toString(data.getDstPort(data.getIPv4HeaderLength()))), new
IntWritable(count));
            context.write(new Text(data.getIPv4srcIP() + " S
" + Integer.toString(data.getSrcPort(data.getIPv4HeaderLength()))), new
IntWritable(count));
        }      }
}}

```

2. Port Count Reducer Code

```

package PCAPNG;

import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.io.FloatWritable;
import org.apache.hadoop.io.IntWritable;
import java.io.IOException;

public class PCAPNGReducer extends Reducer
<Text,IntWritable,Text,IntWritable> {

    public void reduce(Text key, Iterable<IntWritable> values, Context
context) throws IOException, InterruptedException {
        int sum = 0;
        int temp = 0;
    }
}

```

```

    for (IntWritable value: values) {
        temp = value.get();
        sum+=temp;
    }
    context.write(new Text(key.toString()), new IntWritable(sum));
}
}

```

F. PROTOCOL COUNT CODE

1. Protocol Count Mapper Code

```

package PCAPNG;

import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.BytesWritable;
import org.apache.hadoop.io.NullWritable;
import java.util.Arrays;
import java.io.IOException;

public class PCAPNGMapper extends Mapper
<NullWritable,BytesWritable,Text,IntWritable> {

    public void map(NullWritable key, BytesWritable value, Context
context) throws IOException, InterruptedException {

        int count = 1;
        Pcapng data = new Pcapng(value);
        data.getFirstPacket();
        while(Pcapng.isNextPacket()==true){
            context.write(new Text(data.getEtherType()), new
IntWritable(count));
            data.getNextPacket();
        }
        //repeated for the last packet
        context.write(new Text(data.getEtherType()), new
IntWritable(count));
    }
}

```

2. Protocol Count Reducer Code

```
package PCAPNG;

import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.io.FloatWritable;
import org.apache.hadoop.io.IntWritable;
import java.io.IOException;

public class PCAPNGReducer extends Reducer
<Text, IntWritable, Text, IntWritable> {

    public void reduce(Text key, Iterable<IntWritable> values, Context
context) throws IOException, InterruptedException {
        int sum = 0;
        int temp = 0;

        for (IntWritable value: values) {
            temp = value.get();
            sum+=temp;
        }
        context.write(new Text(key.toString()), new IntWritable(sum));
    }
}
```

G. NETWORK USAGE BY DAY OF WEEK AND TIME OF DAY PER IP CODE

1. Network Usage by Day of Week and Time of Day per IP Mapper Code

```
package PCAPNG;

import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.BytesWritable;
import org.apache.hadoop.io.NullWritable;
import java.io.IOException;

public class PCAPNGMapper extends Mapper
<NullWritable, BytesWritable, Text, Text> {
```

```

    public void map(NullWritable key, BytesWritable value, Context
context) throws IOException, InterruptedException {

        Pcapng data = new Pcapng(value);

        int[] montimeslots = new int[24];
        int[] tuestimeslots = new int[24];
        int[] wedtimeslots = new int[24];
        int[] thurtimeslots = new int[24];
        int[] fritimeslots = new int[24];
        int[] sattimeslots = new int[24];
        int[] suntimeslots = new int[24];

        data.getFirstPacket();
        while(Pcapng.isNextPacket()==true){
            String srcIP = data.getIPv4srcIP();
            String dstIP = data.getIPv4dstIP();
            if( srcIP.contains("214.43.86")){
                String date = data.getDate();
                context.write(new Text(srcIP), new
Text(date.substring(0,3) +" " + date.substring(4,6) + " " +
Integer.toString(data.getPacketSize())));
            }

            if( dstIP.contains("214.43.86")){
                String date = data.getDate();
                context.write(new Text(dstIP), new
Text(date.substring(0,3) +" " + date.substring(4,6) + " " +
Integer.toString(data.getPacketSize())));
            }

            data.getNextPacket();
        }
    }
}

```

2. Network Usage by Day of Week and Time of Day per IP Reducer Code

```

package PCAPNG;

import org.apache.hadoop.mapreduce.Reducer;

```

```

import org.apache.hadoop.io.Text;
import org.apache.hadoop.io.FloatWritable;
import org.apache.hadoop.io.IntWritable;
import java.io.IOException;

public class PCAPNGReducer extends Reducer <Text,Text,Text,Text>{

    public void reduce(Text key, Iterable<Text> values, Context context)
    throws IOException, InterruptedException {

        String temp = "";
        String[] brokentemp = new String[3];
        int[] monhourslots = new int[24];
        int[] tuehourslots = new int[24];
        int[] wedhourslots = new int[24];
        int[] thuhourslots = new int[24];
        int[] frihourslots = new int[24];
        int[] sathourslots = new int[24];
        int[] sunhourslots = new int[24];
        for (Text value: values) {
            //used to show the mapper outputs
            //context.write(new Text(key.toString()), value);
            temp = value.toString();
            brokentemp = temp.split("\\s");
            switch(brokentemp[0]){

                case "Mon":

                    monhourslots[Integer.parseInt(brokentemp[1])] +=
Integer.parseInt(brokentemp[2]);
                    break;

                case "Tue":

                    tuehourslots[Integer.parseInt(brokentemp[1])] +=
Integer.parseInt(brokentemp[2]);
                    break;

                case "Wed":

                    wedhourslots[Integer.parseInt(brokentemp[1])] +=
Integer.parseInt(brokentemp[2]);
                    break;

```

```

        ////!!FILE ANALYZED ONLY WENT TO WED SO CODE LEFT
INCOMPLETE!!
    }

}

StringBuilder monday = new StringBuilder();
StringBuilder tuesday = new StringBuilder();
StringBuilder wednesday = new StringBuilder();
for(int i=0;i<24;i++){
    monday.append(Integer.toString(monhourslots[i]) + " ");
    tuesday.append(Integer.toString(tuehourslots[i]) + " ");
    wednesday.append(Integer.toString(wedhourslots[i]) + " ");
}

context.write(new Text(key.toString() + " Mon"), new
Text(monday.toString()));
context.write(new Text(key.toString() + " Tue"), new
Text(tuesday.toString()));
context.write(new Text(key.toString() + " Wed"), new
Text(wednesday.toString()));
}}

```

APPENDIX B. MAPREDUCE DATA PARSING AND BULK HBASE LOADING CODE

A. DRIVER CODE

```
package HBase;

import java.util.*;
import org.apache.hadoop.conf.Configured;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.hbase.HBaseConfiguration;
import org.apache.hadoop.hbase.client.HTable;
import org.apache.hadoop.hbase.mapreduce.HFileOutputFormat;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.hbase.io.ImmutableBytesWritable;
import org.apache.hadoop.hbase.client.Put;

public class HBaseDriver extends Configured implements Tool {

    public int run(String[] args) throws Exception {
        // check the CLI
        if (args.length != 3) {
            System.err.println("usage: hadoop jar HBase.jar
HBase.HBaseDriver <inputfile> <outputdir> <table name>");
            System.exit(1);
        }
        // setup the Job

        Job job = new Job(getConf(), "Hbase_bulk_load");

        HTable table = new HTable(getConf(), args[2]);
        job.setJarByClass(HBaseDriver.class);
        job.setMapperClass(HBaseMapper.class);
        job.setInputFormatClass(WholeFileInputFormat.class);
        job.setMapOutputKeyClass(ImmutableBytesWritable.class);
        job.setMapOutputValueClass(Put.class);
        job.setSpeculativeExecution(false);
    }
}
```

```

    job.setReduceSpeculativeExecution(false);
    job.setOutputFormatClass(HFileOutputFormat.class);
    HFileOutputFormat.configureIncrementalLoad(job, table);

    // setup input and output paths
    FileInputFormat.setInputDirRecursive(job, true);
    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));

    return job.waitForCompletion(true) ? 0 : 1;
}

public static void main(String[] args) throws Exception {
    Configuration conf = HBaseConfiguration.create();
    conf.set("hbase.mapred.outputtable", args[2]);
    System.exit(ToolRunner.run(new HBaseDriver(), args));
}
}

```

B. MAPPER CODE

```

package HBase;

import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.hbase.io.ImmutableBytesWritable;
import org.apache.hadoop.hbase.client.Put;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.BytesWritable;
import org.apache.hadoop.io.NullWritable;
import java.io.IOException;
import org.apache.hadoop.hbase.util.Bytes;
import java.util.Arrays;
import java.nio.ByteBuffer;
import java.nio.ByteOrder;

public class HBaseMapper extends Mapper
<NullWritable, BytesWritable, ImmutableBytesWritable, Put> {
    public void map(NullWritable key, BytesWritable value, Context
context) throws IOException, InterruptedException {

        Pcapng data = new Pcapng(value);
        int count = 0;
    }
}

```

```

byte[] cf = Bytes.toBytes("cf");
byte[] ipdst = Bytes.toBytes("ipdst");
byte[] ipsrc = Bytes.toBytes("ipsrc");
byte[] sze = Bytes.toBytes("sze");
byte[] ethtyp = Bytes.toBytes("ethtyp");
byte[] ipv4prctl = Bytes.toBytes("ipv4prctl");
byte[] seq = Bytes.toBytes("seq");
byte[] ack = Bytes.toBytes("ack");
byte[] ipv4tester = {(byte)0x08, (byte)0x00};
byte[] flgs = Bytes.toBytes("flgs");
byte[] tcpsze = Bytes.toBytes("tcpsze");
byte[] tcphdr = Bytes.toBytes("tcphdr");
byte[] srcprt = Bytes.toBytes("srcprt");
byte[] dstprt = Bytes.toBytes("dstprt");
data.getFirstPacket();
while(Pcapng.isNextPacket()==true){
    //confirm IPv4
    if(Arrays.equals(data.getEtherType(), ipv4tester)){
        //check if source IP is local
        if(data.getIPv4srcIP().contains("192.168")){

            String newkey = data.getIPv4srcIP() + "_"
+ Long.toString(data.getLongDate()) + "_" + Integer.toString(count);
            if(count == 9){
                count =0;
            }
            count = count+1;
            ImmutableBytesWritable Hkey = new
ImmutableBytesWritable(Bytes.toBytes(newkey));
            Put Hput = new
Put(Bytes.toBytes(newkey));

            Hput.add(cf, ipdst, Bytes.toBytes(data.getIPv4dstIP()));

            Hput.add(cf, sze, Bytes.toBytes(data.getPacketSize()));
            Hput.add(cf, ethtyp, data.getEtherType());

            Hput.add(cf, ipv4prctl, Bytes.toBytes(data.getIPv4Protocol()));
            //check if TCP
            if(data.getIPv4Protocol().contains("6")){
                int ipv4header =
data.getIPv4HeaderLength();

                Hput.add(cf, seq, Bytes.toBytes(data.getSeqNum(ipv4header)));

```

```

Hput.add(cf, ack, Bytes.toBytes(data.getAckNum(ipv4header)));

Hput.add(cf, flgs, Bytes.toBytes(data.getFlags(ipv4header)));

Hput.add(cf, tcphdr, Bytes.toBytes(data.getTCPHeaderLength(ipv4header)));

                                int tcpszeval =
data.getPacketSize() - 14 - ipv4header -
data.getTCPHeaderLength(ipv4header);

Hput.add(cf, tcpsze, Bytes.toBytes(tcpszeval));

Hput.add(cf, srcprt, Bytes.toBytes(data.getSrcPort(ipv4header)));

Hput.add(cf, dstprt, Bytes.toBytes(data.getDstPort(ipv4header)));
                                }

                                context.write(Hkey, Hput);
                                }

                                //check if destination IP is local
                                else if(data.getIPv4dstIP().contains("192.168")){
                                    String newkey = data.getIPv4dstIP() + "_"
+ Long.toString(data.getLongDate()) + "_" + Integer.toString(count);
                                    if(count == 9){
                                        count = 0;
                                    }
                                    count = count+1;
                                    ImmutableBytesWritable Hkey = new
ImmutableBytesWritable(Bytes.toBytes(newkey));
                                    Put Hput = new
Put(Bytes.toBytes(newkey));

Hput.add(cf, ipsrc, Bytes.toBytes(data.getIPv4srcIP()));

Hput.add(cf, sze, Bytes.toBytes(data.getPacketSize()));
                                Hput.add(cf, ethtyp, data.getEtherType());

Hput.add(cf, ipv4prctl, Bytes.toBytes(data.getIPv4Protocol()));
                                //check if TCP
                                if(data.getIPv4Protocol().contains("6")){
                                    int ipv4header =
data.getIPv4HeaderLength();

```

```

Hput.add(cf, seq, Bytes.toBytes(data.getSeqNum(ipv4header)));

Hput.add(cf, ack, Bytes.toBytes(data.getAckNum(ipv4header)));

Hput.add(cf, flgs, Bytes.toBytes(data.getFlags(ipv4header)));

Hput.add(cf, tcphdr, Bytes.toBytes(data.getTCPHeaderLength(ipv4header)));

                                int tcpszeval =
data.getPacketSize() - 14 - ipv4header -
data.getTCPHeaderLength(ipv4header);

Hput.add(cf, tcpsze, Bytes.toBytes(tcpszeval));

Hput.add(cf, srcprt, Bytes.toBytes(data.getSrcPort(ipv4header)));

Hput.add(cf, dstprt, Bytes.toBytes(data.getDstPort(ipv4header)));
    }

    context.write(Hkey, Hput);

}}

data.getNextPacket();
}
//repeated code for the last packet
if(Arrays.equals(data.getEtherType(), ipv4tester)){

    if(data.getIPv4srcIP().contains("192.168")){
        String newkey = data.getIPv4srcIP() + "_" +
Long.toString(data.getLongDate()) + "_" + Integer.toString(count);
        if(count == 9){
            count = 0;
        }
        count = count+1;
        ImmutableBytesWritable Hkey = new
ImmutableBytesWritable(Bytes.toBytes(newkey));
        Put Hput = new Put(Bytes.toBytes(newkey));

Hput.add(cf, ipdst, Bytes.toBytes(data.getIPv4dstIP()));

Hput.add(cf, sze, Bytes.toBytes(data.getPacketSize()));
        Hput.add(cf, ethtyp, data.getEtherType());

```

```

        Hput.add(cf, ipv4prctl, Bytes.toBytes(data.getIPv4Protocol()));
        if(data.getIPv4Protocol().contains("6")){
            int ipv4header =
data.getIPv4HeaderLength();

        Hput.add(cf, seq, Bytes.toBytes(data.getSeqNum(ipv4header)));

        Hput.add(cf, ack, Bytes.toBytes(data.getAckNum(ipv4header)));

        Hput.add(cf, flgs, Bytes.toBytes(data.getFlags(ipv4header)));

        Hput.add(cf, tcphdr, Bytes.toBytes(data.getTCPHeaderLength(ipv4hea
der)));
            int tcpszeval =
data.getPacketSize() - 14 - ipv4header -
data.getTCPHeaderLength(ipv4header);

        Hput.add(cf, tcpsze, Bytes.toBytes(tcpszeval));

        Hput.add(cf, srcprt, Bytes.toBytes(data.getSrcPort(ipv4header)));

        Hput.add(cf, dstprt, Bytes.toBytes(data.getDstPort(ipv4header)));
        }
        context.write(Hkey, Hput);
    }

    else if(data.getIPv4dstIP().contains("192.168")){
        String newkey = data.getIPv4dstIP() + "_" +
Long.toString(data.getLongDate()) + "_" + Integer.toString(count);
        if(count == 9){
            count =0;
        }
        count = count+1;
        ImmutableBytesWritable Hkey = new
ImmutableBytesWritable(Bytes.toBytes(newkey));
        Put Hput = new Put(Bytes.toBytes(newkey));

        Hput.add(cf, ipsrc, Bytes.toBytes(data.getIPv4srcIP()));

        Hput.add(cf, sze, Bytes.toBytes(data.getPacketSize()));
        Hput.add(cf, ethtyp, data.getEtherType());
    }
}

```

```

        Hput.add(cf, ipv4prctl, Bytes.toBytes(data.getIPv4Protocol()));
        if(data.getIPv4Protocol().contains("6")){
            int ipv4header =
data.getIPv4HeaderLength();

            Hput.add(cf, seq, Bytes.toBytes(data.getSeqNum(ipv4header)));

            Hput.add(cf, ack, Bytes.toBytes(data.getAckNum(ipv4header)));

            Hput.add(cf, flgs, Bytes.toBytes(data.getFlags(ipv4header)));

            Hput.add(cf, tcphdr, Bytes.toBytes(data.getTCPHeaderLength(ipv4hea
der)));
            int tcpszeval = data.getPacketSize() - 14
- ipv4header - data.getTCPHeaderLength(ipv4header);

            Hput.add(cf, tcpsze, Bytes.toBytes(tcpszeval));

            Hput.add(cf, srcprt, Bytes.toBytes(data.getSrcPort(ipv4header)));

            Hput.add(cf, dstprt, Bytes.toBytes(data.getDstPort(ipv4header)));
        }
        context.write(Hkey, Hput);

    }}

}}

```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX C. SPARK ANALYSIS CODE

```
import org.apache.spark.sql.SparkSession
import org.apache.spark.sql.types._
import org.apache.spark._
import org.apache.spark.sql.{ DataFrame, SQLContext }
import org.apache.spark.sql.functions._
import org.apache.spark.SparkConf
import org.apache.spark.sql.datasources.hbase.HBaseTableCatalog
import java.nio.ByteBuffer
import scala.collection.JavaConversions._
import scala.collection.mutable.ListBuffer
import org.apache.spark.sql.functions.countDistinct

val cat = s"""{
    |"table":{"namespace":"default",
"name":"/user/user01/hbasetest/table/packet"},
    |"rowkey":"key",
    |"columns":{
    |"key":{"cf":"rowkey", "col":"key", "type":"string"},
    |"ipdst":{"cf":"cf", "col":"ipdst", "type":"string"},
    |"ipsrc":{"cf":"cf", "col":"ipsrc", "type":"string"},
    |"size":{"cf":"cf", "col":"size", "type":"int"},
    |"ethtyp":{"cf":"cf", "col":"ethtyp", "type":"binary"},
    |"ipv4prtcl":{"cf":"cf", "col":"ipv4prtcl",
"type":"string"},
    |"seq":{"cf":"cf", "col":"seq", "type":"long"},
    |"ack":{"cf":"cf", "col":"ack", "type":"long"},
    |"flgs":{"cf":"cf", "col":"flgs", "type":"string"},
    |"tcphdr":{"cf":"cf", "col":"tcphdr",
"type":"int"},
    |"tcpsze":{"cf":"cf", "col":"tcpsze", "type":"int"},
    |"srcprt":{"cf":"cf", "col":"srcprt", "type":"int"},
    |"dstprt":{"cf":"cf", "col":"dstprt", "type":"int"}
    |}
    |}"""

import spark.implicits._

val tableName = "/user/user01/hbasetest/table/packet"

val sqlContext = new SQLContext(spark.sparkContext)
```

```

def withCatalog(cat: String): DataFrame = {
  sqlContext
    .read
    .options(Map(HBaseTableCatalog.tableCatalog -> cat))
    .format("org.apache.hadoop.hbase.spark")
    .load()
}

val df = withCatalog(cat)
println("show DataFrame")
df.show
df.count
df.schema

val df1 = df.withColumn("temp", split(col("key"), "_")).select(
  col("*") +: (0 until 2).map(i =>
col("temp").getItem(i).as(s"col$i")): _*
).drop("key").drop("temp")

df1.filter($"flgs" === "02").show
val syndf = df1.filter($"col0" === "192.168.0.16").filter($"ipdst"
=== "192.168.0.1").filter($"flgs" ===
"02").select("seq").collectAsList()
// println(syndf)
syndf.size

//iterate through all syn seq numbers
for (x <- syndf){
  //set seq numb
  val seqnum = x.getLong(0)
  //println(seqnum)

  //find ack number
  if( df1.filter($"ack" === seqnum+1).count == 1){
    df1.filter($"ack" === seqnum+1).filter($"flgs" ===
"12").select("seq").count
    val acknum = df1.filter($"ack" === seqnum+1).filter($"flgs" ===
"12").select("seq").first().getLong(0)
    //println(acknum)

    //calculate syn - synack rtt
    val syntime = df1.filter($"seq" === seqnum).filter($"flgs"
=== "02").select("col1").first().getString(0)
    val longsytme = syntime.toLong

```

```

    val synacktime = df1.filter($"seq" === acknum).filter($"flgs"
=== "12").select("col1").first().getString(0)
    val longsynacktime = synacktime.toLong
    //println(longsynacktime - longsyntime)

    val rttList = new ListBuffer[Long]()
    rttList += (longsynacktime - longsyntime)

    //calculate synack - ack rtt
    if(df1.filter($"seq" === seqnum+1).filter($"flgs"
=== "10").count > 0){
        val acktime = df1.filter($"seq" ===
seqnum+1).filter($"flgs" === "10").select("col1").first().getString(0)
        val longacktime = acktime.toLong
        //println(longacktime - longsynacktime)
        rttList += (longacktime - longsynacktime)
    }
    println(rttList.mkString(" "))
}
}

df1.groupBy("col0").agg(sum("size")).show
df1.groupBy("ipv4prtcl").count.show
df1.groupBy("srcprt").count.orderBy($"count".desc).show

```

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

- [1] Cisco, “The Zettabyte era: Trends and analysis.” 2017. [Online]. Available: <https://www.cisco.com/c/en/us/solutions/service-provider/visual-networking-index-vni/white-paper-listing.html>.
- [2] T. White. *Hadoop The Definitive Guide*. Beijing: O’Reilly Media, 2015.
- [3] “Welcome to Apache™ Hadoop®!,” Hadoop.apache.org, 2018. [Online]. Available: <http://hadoop.apache.org/index>. [Accessed: 06- May- 2018].
- [4] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, “The Hadoop distributed file system,” in *Proc. IEEE 26th Sym. Mass Storage Sys. Tech.*, 2010, pp. 1-10.
- [5] D. May, “Hadoop Distributed File System (HDFS) overview.” 2012. [Online]. Available: <http://www.coreservlets.com/hadoop-tutorial/>
- [6] J. Dean and S. Ghemawat, “MapReduce: Simplified data processing on large clusters,” *OSDI*, 2004.
- [7] MapR Technologies, “Developing Hadoop applications.” 2017. [Online]. Available: MapR Technologies.
- [8] “Apache HBase™,” Hbase.apache.org, 2018. [Online]. Available: <http://hbase.apache.org/>. [Accessed: 06- May- 2018].
- [9] MapR Technologies, “Apache HBase applications design and build.” 2017. [Online]. Available: MapR Technologies.
- [10] MapR Technologies, “Developing Spark applications.” 2016. [Online]. Available: MapR Technologies.
- [11] J. Damji, “A tale of three Apache Spark APIs: RDDs, DataFrames, and Datasets,” *Databricks*, 2018. [Online]. Available: <https://databricks.com/blog/2016/07/14/a-tale-of-three-apache-spark-apis-rdds-dataframes-and-datasets.html>. [Accessed: 06- May- 2018].
- [12] “PCAP Next Generation (pcapng) capture file format,” Pcapng.github.io, 2018. [Online]. Available: <https://pcapng.github.io/pcapng/>. [Accessed: 06- May- 2018].
- [13] “PCAP Next Generation dump file format,” Winpcap.org, 2018. [Online]. Available: <https://www.winpcap.org/ntar/draft/PCAP-DumpFileFormat.html>. [Accessed: 06- May- 2018].
- [14] Y. Lee and W. Kang, “An Internet traffic analysis method with MapReduce,” in *Network Operations and Management Symposium Workshops*, 2010.

- [15] P. Dhanotia, V. Kapoor and M. Sharma, “A clustering approach for network traffic classification in Hadoop distributed computing environment,” *Int. J. Comp. Sci. Inform. Sec.*, vol. 14, no. 8, 2016.
- [16] W. Nagele, “Large-scale PCAP data analysis using Apache Hadoop,” RIPE NCC, 2011. [Online]. Available: <https://labs.ripe.net/Members/wnagele/large-scale-pcap-data-analysis-using-apache-hadoop>.
- [17] “Platform Services | MapR,” Mapr.com, 2018. [Online]. Available: <https://mapr.com/products/platform-services/>. [Accessed: 06- May- 2018].
- [18] “Hue, the self service open source analytics workbench for browsing, querying and visualizing data interactively,” Gethue.com, 2018. [Online]. Available: <http://gethue.com/>. [Accessed: 06- May- 2018].
- [19] “Hadoop 2.x mapred-site.xml,” Doc.mapr.com, 2018. [Online]. Available: <http://doc.mapr.com/display/MapR3/Hadoop+2.x+mapred-site.xml>. [Accessed: 10- May- 2018].
- [20] “Apache Hadoop 2.9.0—MapReduce tutorial,” Hadoop.apache.org, 2018. [Online]. Available: <https://hadoop.apache.org/docs/stable/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html>. [Accessed: 06- May- 2018].
- [21] B. Venners, *Inside the Java Virtual Machine*. New Delhi: Tata McGraw-Hill, 2000.
- [22] “JAR file overview,” Docs.oracle.com, 2018. [Online]. Available: <https://docs.oracle.com/javase/8/docs/technotes/guides/jar/jarGuide.html>. [Accessed: 06- May- 2018].
- [23] “Hadoop tutorial—YDN,” Developer.yahoo.com, 2018. [Online]. Available: <https://developer.yahoo.com/hadoop/tutorial/module5.html>. [Accessed: 07- May- 2018].
- [24] Hadoop.apache.org. (2018). “BytesWritable (Apache Hadoop Main 3.1.0 API).” [Online] Available at: <https://hadoop.apache.org/docs/r3.1.0/api/org/apache/hadoop/io/BytesWritable.html#getBytes--> [Accessed 9 May 2018].
- [25] “Put (Apache HBase 3.0.0-SNAPSHOT API),” Hbase.apache.org, 2018. [Online]. Available: <https://hbase.apache.org/apidocs/org/apache/hadoop/hbase/client/Put.html>. [Accessed: 12- May- 2018].

- [26] “IBM Knowledge Center—Best practices for loading data in HBase,” Ibm.com, 2018. [Online]. Available: https://www.ibm.com/support/knowledgecenter/en/SSWSR9_11.4.0/com.ibm.swg.im.mdmhs.pmebi.doc/topics/bestpractice_loadhbase.html. [Accessed: 12- May- 2018].
- [27] “Apache HBase TM reference guide,” Hbase.apache.org, 2018. [Online]. Available: <https://hbase.apache.org/book.html#spark>. [Accessed: 12- May- 2018].
- [28] “Spark SQL and DataFrames—Spark 2.3.0 documentation,” Spark.apache.org, 2018. [Online]. Available: <https://spark.apache.org/docs/latest/sql-programming-guide.html>. [Accessed: 12- May- 2018].
- [29] “What is a three-way handshake in TCP?” Learningnetwork.cisco.com, 2018. [Online]. Available: <https://learningnetwork.cisco.com/docs/DOC-30061>. [Accessed: 04- May- 2018].
- [30] “Determining TCP initial round trip time,” Blog.packet-foo.com, 2018. [Online]. Available: <https://blog.packet-foo.com/2014/07/determining-tcp-initial-round-trip-time/>. [Accessed: 24- May- 2018].

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
Ft. Belvoir, Virginia
2. Dudley Knox Library
Naval Postgraduate School
Monterey, California