A Hadoop-Based Packet Trace Processing Tool*

Yeonhee Lee, Wonchul Kang, and Youngseok Lee

Chungnam National University
Daejeon, 305-764, Republic of Korea
{yhlee06,teshi85,lee}@cnu.ac.kr

Abstract. Internet traffic measurement and analysis has become a significantly challenging job because large packet trace files captured on fast links could not be easily handled on a single server with limited computing and memory resources. Hadoop is a popular open-source cloud computing platform that provides a software programming framework called MapReduce and the distributed filesystem, HDFS, which are useful for analyzing a large data set. Therefore, in this paper, we present a Hadoop-based packet processing tool that provides scalability for a large data set by harnessing MapReduce and HDFS. To tackle large packet trace files in Hadoop efficiently, we devised a new binary input format, called `PcapInputFormat`, hiding the complexity of processing binary-formatted packet data and parsing each packet record. We also designed efficient traffic analysis MapReduce job models consisting of map and reduce functions. To evaluate our tool, we compared its computation time with a well-known packet-processing tool, CoralReef, and showed that our approach is more affordable to process a large set of packet data.

1 Introduction

Internet traffic measurement and analysis needs a lot of data storage and high-performance computing power to manage a large traffic data set. Tcpdump [1] is widely used for capturing and analyzing packet traces, and various tools based on the packet capture library, "libpcap" [1], such as wireshark [2], CoralReef [3], and snort [4] have been deployed to handle packets. For aggregated information of a sequence of packets sharing the same fields like IP addresses and port numbers, Cisco NetFlow [5] is extensively employed to observe traffic passing through routers or switches in the unit of a flow. However, these legacy traffic tools are not suited for processing a large data set of tera- or petabytes monitored at high-speed links. In analyzing packet data for a large-scale network, we often have to handle hundreds of giga- or terabyte packet trace files. When the outbreaks of global Internet worms or DDoS attacks occur, we also have to process quickly

* This research was partly supported by the MKE (Ministry of Knowledge Economy), Korea, under the ITRC (Information Technology Research Center) support program supervised by the NIPA (National IT Industry Promotion Agency) (NIPA-2011-(C1090-1031-0005)) and partly by the IT R&D program of MKE/KEIT [KI001878, "CASFI : High-Precision Measurement and Analysis Research"].

a large volume of packet data at once. Yet, with legacy packet processing tools running on a single high-performance server, we cannot perform fast analysis of large packet data. Moreover, with a single-server approach, it is difficult to provide fault-tolerant traffic analysis services against a node failure that often happens when intensive read/write jobs are frequently performed on hard disks.

MapReduce [6], developed by Google, is a software paradigm for processing a large data set in a distributed parallel way. Since Google's MapReduce and Google file system (GFS) [7] are proprietary, an open-source MapReduce software project, Hadoop [8], was launched to provide similar capabilities of the Google's MapReduce platform by using thousands of cluster nodes. Hadoop distributed filesystem (HDFS) is also an important component of Hadoop, that corresponds to GFS. Yahoo!, Amazon, Facebook, IBM, Rackspace, Last.fm, Netflix and Twitter are using Hadoop to run large-scale data-parallel applications coping with a large set of data files. Amazon provides Hadoop-based cloud computing services called Elastic Compute Cloud (EC2) and Simple Storage Service (S3). Facebook also uses Hadoop to analyze the web log data for its social network service. From the cloud computing environment like Hadoop, we could benefit two features of distributed parallel computing and fault tolerance, which could fit well for packet processing tools dealing with a large set of traffic files. With the MapReduce programming model on inexpensive commodity PCs, we could easily handle tera- or petabyte data files. Due to the cluster filesystem, we could provide fault-tolerant services against node failures.

In this paper, hence, we present a Hadoop-based packet trace processing tool that stores and analyzes the packet data on the cloud computing platform. Major features of our tool are as follows. First, it could write packet trace files in libpcap format on HDFS, with which the problem of archiving and managing large packet trace files is easily solved. Second, our tool could significantly reduce the traffic statistics computation time of large packet trace files with MapReduce-based analysis programs, when compared with the traditional tool. For these purposes, we have implemented a new binary file input/output module, `PcapInputFormat`, which reduces the processing time of packet records included in trace files, because text files are used for the conventional input file format in Hadoop. In addition, we have designed packet processing MapReduce job models consisting of map and reduce tasks.

The remaining of this paper is organized as follows. In Section 2, we describe the related work on MapReduce and Hadoop. The architecture of our tool and its components are explained in Section 3, and the experimental results are presented in Section 4. Finally Section 5 concludes this paper.

2 Related Work

For Internet traffic measurement and analysis, there are a lot of packet-processing tools such as tcpdump, CoralReef, wireshark, and snort. Most of these packet-processing tools are run on a single host with the limited computing and storage resources.

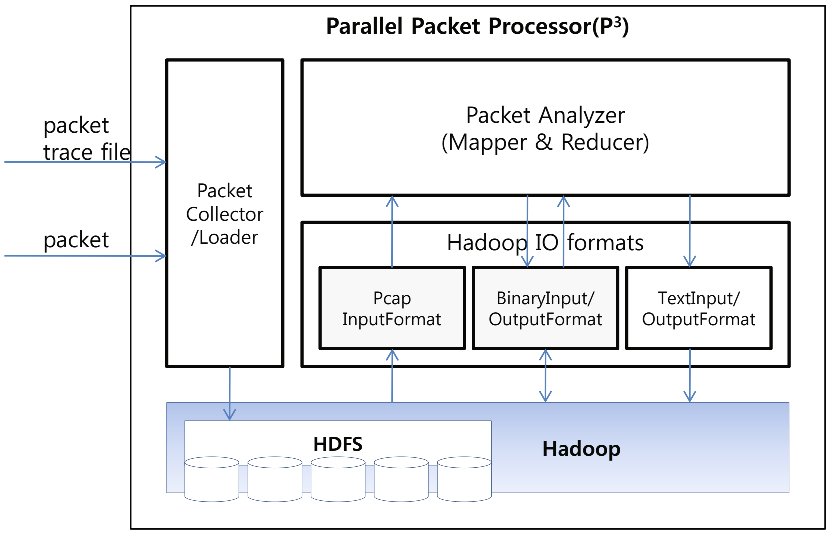

Fig. 1. The overview of Hadoop-based packet file processing tool, Parallel Packet Processor (P^3)

As the MapReduce platform has been popular, a variety of data mining or analytics applications are emerging in the fields of natural sciences or business intelligence. Typical studies based on Hadoop are text-data analysis jobs like web indexing or or log analysis. For the network management fields, snort log analysis was tried with Hadoop in [9]. In our previous work [10], we have devised a simple MapReduce-based NetFlow analysis method that could analyze text-converted NetFlow v5 files, which showed that MapReduce-based NetFlow analysis outperforms popular flow-tools. In this paper, we present a Hadoop-based packet trace analysis tool that could process enormous volumes of libpcap packet trace files.

On the other hand, there has been few work on dealing with non-text files coherently in Hadoop. As for extending the Hadoop API, Conner [11] has customized Hadoop's `FileInputFormat` for image processing, but has not clearly described its performance evaluation results. Recently, there have been a few studies [12, 13, 14] to improve the performance of Hadoop. Zaharia *et al.* [12] have designed an improved scheduling algorithm that reduces Hadoop's response time by considering a cluster node performing poorly. Kambatla *et al.* [13] proposed a Hadoop provisioning method by analyzing and comparing resource usage patterns of various MapReduce jobs. Condie *et al.* [14] devised a modified MapReduce architecture that allows data to be pipelined between nodes, which is useful for interactive jobs or continuous query processing programs.

3 Processing Packet Trace Files with Hadoop

Parallel Packet Processor (P^3)[1] consists of three main components: the *Packet Collector/Loader* that saves packets to HDFS from the online packet source using libpcap or the captured trace file; the *Packet Analyzer* implemented by map and reduce tasks processing packet data; and the *Hadoop IO formats* that read packet records from files on HDFS and return the analysis results. The overall architecture is illustrated in Fig. 1.

3.1 Packet Collector/Loader

P^3 analyzes packet trace files generated by either a packet sniffing tool such as tcpdump or our own packet-capturing module, *Packet Collector/Loader*. The *Packet Collector* captures packets from a live packet stream and save them to HDFS simultaneously, which enables P^3 to collect packets directly from packet probes. Given packet trace files in the libpcap format, *Packet Loader* reads files and splits them into the fixed-size of chunks, and save the chunks of files to HDFS. Network operators will irregularly copy captured packet trace files to HDFS in order to analyze detailed characteristics of the trace.

The *Packet Collector* uses libpcap and jpcap modules for capturing packets from a live source, and the HDFS stream writer for saving packet data. To use this stream writer, we developed a HDFS writing module in Java. The *Packet Collector* can capture packets by interacting with the libpcap module through jpcap. The *Packet Loader* just saves the packet trace file to HDFS by using the Hadoop stream writer. Within HDFS, each packet trace file larger than the specified HDFS block size (typically 64 MB) is split, and three replicas (by default) for every block are copied into HDFS for fault tolerance. The trace file contains binary-formed packet data, which has the non-fixed length of packet records.

3.2 New Binary Input/Output Format for Packet Records

In the Hadoop cluster environment, network bandwidth between nodes is a critical factor for the performance. To overcome the network bottleneck, MapReduce collocates the data with its computing node if possible. HDFS has concepts of a block which is the unit for writing a file and a split which is the unit for being processed by MapReduce task. As a block is in a large unit (64 MB by default), a file in HDFS will be divided into block-sized chunks that are stored as independent units. A split is a chunk of the input file that is processed by a single map task. Each split is divided into records, and passed to the map to process each record of a key-value pair in turn by an `InputFormat`.

The text file is a common format for input/output in Hadoop. `TextInputFormat` can create splits for the text file and parse each split into each line of records by carriage return. However, packet trace files are stored in the binary format defined by libpcap and have no carriage return. Thus, it is necessary to convert binary packet

[1] Source codes are available in [15].

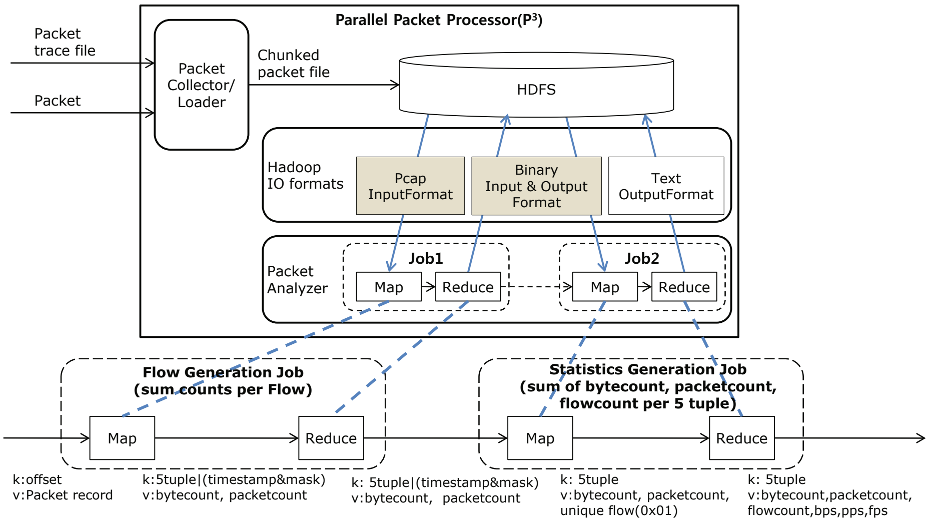
Parallel Packet Processor(P³)
Packet trace file
Packet
Packet Collector/ Loader
Chunked packet file
HDFS
Hadoop IO formats
Pcap InputFormat
Binary Input & Output Format
Text OutputFormat
Packet Analyzer
Job1
Map
Reduce
Job2
Map
Reduce
Flow Generation Job (sum counts per Flow)
Map
Reduce
Statistics Generation Job (sum of bytecount, packetcount, flowcount per 5 tuple)
Map
Reduce
k:offset v:Packet record
k:5tuple|(timestamp&mask) v:bytecount, packetcount
k: 5tuple|(timestamp&mask) v:bytecount, packetcount
k: 5tuple v:bytecount, packetcount, unique flow(0x01)
k: 5tuple v:bytecount,packetcount, flowcount,bps,pps,fps

Fig. 2. New Hadoop input/output formats for processing packet trace. This example shows how the periodic flow statistics is computed. `PcapInputFormat` is used for reading packet records to generate flows from packet trace files. `BinaryOutputFormat` is for writing binary-form of flow records to HDFS, and `BinaryInputFormat` for reading flow records from HDFS to generate flow statistics.

trace files to text ones for the input to Hadoop using `TextInputFormat`. This conversion requires sequential processing which consists of reading variable-length of binary packet records by using the packet length field specified in each packet header, converting them to text-formed ones, and writing to text ones. Thus, this causes additional computing time and storage resources, and does not correspond with a parallel manner of Hadoop.

The process of a Hadoop job consists of several steps. First, before starting a job, we save large packet trace files on HDFS cluster nodes in the unit of fixed-length of blocks. When a job is started, the central job controller, `Jobtracker`, assigns map tasks to each node to process blocks, and then, map tasks in each node reads records from the assigned block in parallel. At this point, both the start location and end location of `InputSplits` can not be clearly detected within a fixed-size HDFS block, because each packet record has variable length of binary data without any record boundary character like a carriage return. That is, there is no way for each map to know the location of the staring position of the first packet in the block until another map task running on the previous block finishes reading the last packet record.

In order to solve this boundary detection issue in manipulating the binary packet trace files on HDFS in parallel, we have devised `PcapInputFormat` as a new packet input module. `PcapInputFormat` could support parallel packet processing by finding the first and last packet location in the HDFS blocks, and by parsing variable-length of packet records. We also devised `BinaryInputForamt` as the input module of fixed-sized binary records, and `BinaryoutputFormat`, the output module of fixed-sized binary records to handle flow statistics using packet data. Figure 2 shows how the native libpcap binary files are processed with binary packet input/output formats in the modified Hadoop. There are two continuous jobs. In the first job, `PcapInputFormat` is used for reading packets from packet trace files to generate flows. `BinaryOutputFormat` is responsible for writing binary-form of flow records to HDFS. In the second job, we compute the flow statistics with `BinaryInputFormat` by reading flow records. Hadoop in itself could support binary data as inputs and outputs with sequence file format, but we have to transform packet traces to sequence files by parsing every packet record sequentially. Therefore, it will not be efficient to use the built-in sequence file format for handling packet trace files.

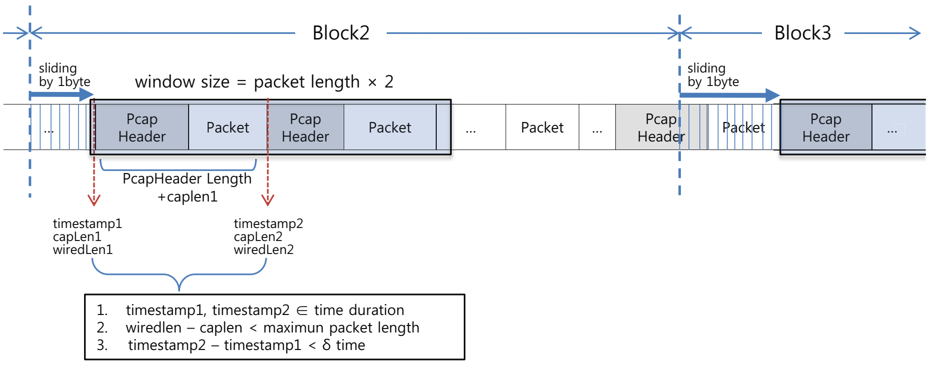
Block2
Block3
sliding by 1byte
window size = packet length × 2
sliding by 1byte
...
Pcap Header
Packet
Pcap Header
Packet
...
Packet
...
Pcap Header
Packet
Pcap Header
...
PcapHeader Length +caplen1
timestamp1 capLen1 wiredLen1
timestamp2 capLen2 wiredLen2
1. timestamp1, timestamp2 ∈ time duration
2. wiredlen – caplen < maximun packet length
3. timestamp2 – timestamp1 < δ time

Fig. 3. Our heuristic method of `PcapInputForamt` for finding the first record from each block using sliding-window

PcapInputFormat. `PcapInputFormat` includes a sub-class, called `PcapVlenRecordReader`, that reads packet records from a block on Hadoop, and it is responsible for parsing records from the split across blocks. When each map task reads packet records from blocks stored on HDFS through the `InputFormat` concurrently, `PcapInputFormat` has to find the first packet in the assigned block, because the last packet record in the previous block might be stored across two continuous HDFS blocks. However, it is not easy to pinpoint the position of the first packet in a block without searching records sequentially due to variable packet size without boundary character of each packet record. Hence, how to identify the boundary of a split from distributed blocks is a main concern of MapReduce tasks for packet processing. That is, the map task has to know the starting position and ending position of splits with only partial fragments of the trace files. Thus, we employ a heuristic to find the first packet of each block by searching a timestamp field included in the packet, assuming that the timestamps of two continuous packets stored in the libpcap trace file will not be much different. The threshold of timestamp difference might be configured to consider the characteristics of packet traces on various link speeds.

The libpcap packet header consists of four kinds of byte arrays: *timestamp of seconds*, *timestamp of microseconds*, *captured length*, and *wired length. captured length* is the number of octets of a packet saved in the file and *wired length* is the actual length of a packet record flowed into. Our heuristic to find the first/last packet records needs two input timestamp parameters, which limits a permissible range of *timestamp* for each packet.

Using these values, we perform a sliding-window pattern matching algorithm with the candidate timestamp field on the block. Figure 3 illustrates this method. First, `PcapVlenRecordReader` takes in a sample byte window of 2 $\times$ the maximum packet length from the beginning portion of the block. To find the beginning point of the split, we suppose that first four bytes consist of the *timestamp* of the first packet. According to pcap header format, another four bytes later, it might be followed by four bytes of *captured length* and *wired length* fields. This assumption let us know that the next packet is followed by the length of bytes as noticed in *captured length* of the first packet. Thus, we can extract its header information for validation.

To validate the assumption that the first four-byte field represents a timestamp of the first packet in an assigned block, we conduct three intuitive verification procedures. First, we make sure that the first timestamp and the second one are within the time duration for capturing packets given by user. Second, we examine that the gap between *captured length* and *wired length* of each packet is smaller than the maximum packet length. Third, we investigate that the gap between the first timestamp and the second one is less than δ *time*, which might be acceptable as an elapsed time for two consecutive packets. We set this value as 60 by default. `PcapVlenRecordReader` repeats this pattern matching function by moving a single byte until it finds the position that meets these conditions.

An extracted record is a byte array that contains the binary-formed packet information and it is passed to the map task as a key-value pair of the `LongWritable` byte offset of the file and a `BytesWritable` byte array of a packet record. Through experiments of a large data set, we confirmed that this practical heuristic works well and the performance is affordable at the same time.

BinaryInputFormat and BinaryOutputFormat. The `PcapInputFormat` module in the modified Hadoop manages libpcap files in a binary format. The *Packet Analyzer* generates basic statistics such as the total IPv4 byte count or packet count. Besides, it also computes periodic statistics results such as bit rate, packet rate, or flow rate for a fixed-length of a time interval. To create these statistics, we have to produce flows from raw packets. In Fig. 2, `PcapInputFormat` is used for the first MapReduce job as a carrier of variable length of packet records from packet trace files. After flows are formulated at the first MapReduce job, they are stored to HDFS through the `BinaryOutputFormat` in the fixed-length of byte arrays. To calculate the statistics of these flows, we deliver flows to the next MapReduce job with `BinaryInputFormat`.

`BinaryInputFormat` and `BinaryOutputFormat` need one parameter, the size of the record to read. In the case of the periodic flow statistics computation job, the input parameter will be the size of a flow record. `BinaryInputFormat` parses a flow record of a byte array, and passes it to the map task as a key-value pair of a `LongWritable` byte offset of the file and a `BytesWritable` byte array of a flow record. Likewise, `BinaryOutputFormat` saves the record consisting of a `BytesWritable` key and a `BytesWritable` value produced by the job to HDFS. Our new binary input/output formats could enhance the speed of loading the libpcap packet trace files on HDFS, and reading/writing packet records in the binary file format in HDFS.

3.3 Packet Analyzer

For the *Packet Analyzer*, we have developed four MapReduce analysis commands.

Total traffic statistics. First, we compute the total traffic statistics such as byte/packet/flow counts for IPv4/v6/nonIP. As shown in Fig. 4, the first MapReduce job calculates the total traffic information. In order to count the number of unique IP addresses and ports, we need another MapReduce job. For this purpose, the first job emits a new key-value pair consisting of a key combined with the text name and an IP address, and a value of 1. This key will be used for identifying unique IP address or port records. The second job summarizes the number of unique IP addresses and ports.

Periodic flow statistics. Periodically, we often assess each flow information consisting of 5-tuples of IP addresses and ports from packet trace files. For flow analysis, we have implemented two MapReduce jobs for periodic flow statistics and aggregated flow information, respectively. As shown in Fig. 5, the first job computes the basic statistics for each flow during the time interval. Then, the second job will aggregate the same flows lasting longer than the small time interval into a single flow. Thus, the first job emits a new key-value pair for the aggregated flows. The key consists of the 5-tuple text concatenated by the masked timestamp.

Periodic simple statistics. A simple statistics is to tally the total byte/packet and bit/packet rate per each interval, which could be implemented with a single MapReduce job. During the map phase, a simple statistics job classifies packets as non-ip, IPv4, and IPv6, and creates a new key with timestamp. The timestamp for the new key is masked by the interval value to produce periodic statistics.

Top N statistics. Given traffic traces, we are interested in finding the most popular statistics such as top 10 flow information. The `Top N` module makes full use of MapReduce features to solve this purpose. We create a new key of the record for identifying or grouping records within the map phase. Thus, in the reduce phase, all the records hashed by keys become sorted by the build-in MapReduce sorting function. Therefore, the reduce task just emits specified number of record data. Figure 6 shows the process for computing top N information. The input could be the results of periodic simple statistics job or periodic

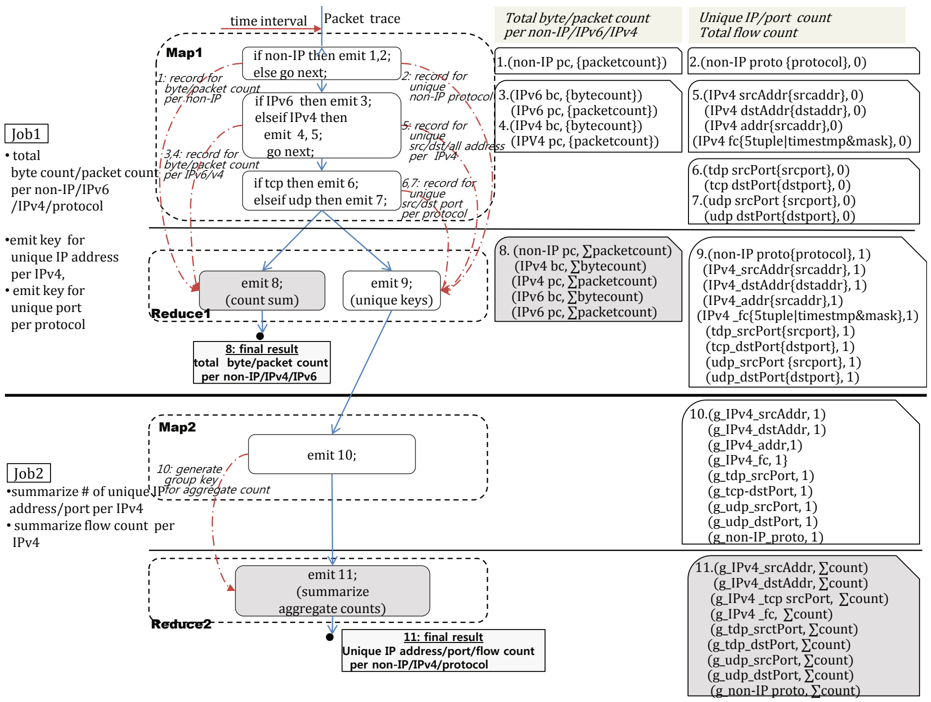
time interval
Packet trace
Map1
if non-IP then emit 1,2; else go next;
if IPv6 then emit 3; elseif IPv4 then emit 4, 5; go next;
if tcp then emit 6; elseif udp then emit 7;
1: record for byte/packet count per non-IP
2: record for unique non-IP protocol
3,4: record for byte/packet count per IPv6/v4
5: record for unique src/dst/all address per IPv4
6,7: record for unique src/dst port per protocol
Job1
• total byte count/packet count per non-IP/IPv6 /IPv4/protocol
•emit key for unique IP address per IPv4,
• emit key for unique port per protocol
Reduce1
emit 8; (count sum)
emit 9; (unique keys)
8: final result total byte/packet count per non-IP/IPv4/IPv6
Total byte/packet count per non-IP/IPv6/IPv4
Unique IP/port count Total flow count
1.(non-IP pc, {packetcount})
2.(non-IP proto {protocol}, 0)
3.(IPv6 bc, {bytecount}) (IPv6 pc, {packetcount}) 4.(IPv4 bc, {bytecount}) (IPV4 pc, {packetcount})
5.(IPv4 srcAddr{srcaddr}, 0) (IPv4 dstAddr{dstaddr}, 0) (IPv4 addr{srcaddr},0) (IPv4 fc{5tuple|timestmp&mask}, 0)
6.(tdp srcPort{srcport}, 0) (tcp dstPort{dstport}, 0) 7.(udp srcPort {srcport}, 0) (udp dstPort{dstport}, 0)
8. (non-IP pc, ∑packetcount) (IPv4 bc, ∑bytecount) (IPv4 pc, ∑packetcount) (IPv6 bc, ∑bytecount) (IPv6 pc, ∑packetcount)
9.(non-IP proto{protocol}, 1) (IPv4_srcAddr{srcaddr}, 1) (IPv4_dstAddr{dstaddr}, 1) (IPv4_addr{srcaddr},1) (IPv4 _fc{5tuple|timestmp&mask},1) (tdp_srcPort{srcport}, 1) (tcp_dstPort{dstport}, 1) (udp_srcPort {srcport}, 1) (udp_dstPort{dstport}, 1)
Map2
emit 10;
10: generate group key for aggregate count
Job2
•summarize # of unique IP address/port per IPv4
• summarize flow count per IPv4
10.(g_IPv4_srcAddr, 1) (g_IPv4_dstAddr, 1) (g_IPv4_addr,1) (g_IPv4_fc, 1} (g_tdp_srcPort, 1) (g_tcp-dstPort, 1) (g_udp_srcPort, 1) (g_udp_dstPort, 1) (g_non-IP_proto, 1)
Reduce2
emit 11; (summarize aggregate counts)
11: final result Unique IP address/port/flow count per non-IP/IPv4/protocol
11.(g_IPv4_srcAddr, ∑count) (g_IPv4_dstAddr, ∑count) (g_IPv4 _tcp srcPort, ∑count) (g_IPv4 _fc, ∑count) (g_tdp_srctPort, ∑count) (g_tdp_dstPort, ∑count) (g_udp_srcPort, ∑count) (g_udp_dstPort, ∑count) (g_non-IP proto, ∑count)

Fig. 4. Total traffic statistics (total byte/packet/flow count per IPv4/v6/non-IP and the number of unique IP addresses/ports): two MapReduce jobs are necessary to count the number of hosts and ports

flow statistics job. Thus, the `Top N` module will add one more MapReduce job. The `Top N` module requires two parameters: one is the column name to sort and the other is the number, N, to output. The column name can be byte count, packet count, and flow count. At the map task, the column is used for creating a new key which will be used for sorting in the running shuffle and sort phase by reduce task. The reduce task just emits records from top to N$^{\text{th}}$.

4 Experiments

For experiments, we have setup standard and high-performance Hadoop testbeds in our laboratory (Table 1). A standard Hadoop testbed consists of a master node and four slave nodes. Each node has quad-core 2.83 GHz CPU, 4 GB memory, and 1.5 TB hard disk. All Hadoop nodes are connected with 1 Gbps Ethernet cards. For the comprehensive test with large files, we configured a high-performance testbed of 10 slave nodes that have octo-core 2.93 GHz Intel i7 CPU, 16 GB memory, 1 TB hard disk, and 1 Gbps Ethernet card. We used Hadoop 0.20.3 version with the HDFS block size of 64 MB. The replication factor of standard/high-performance testbeds is two/three. For comparing our

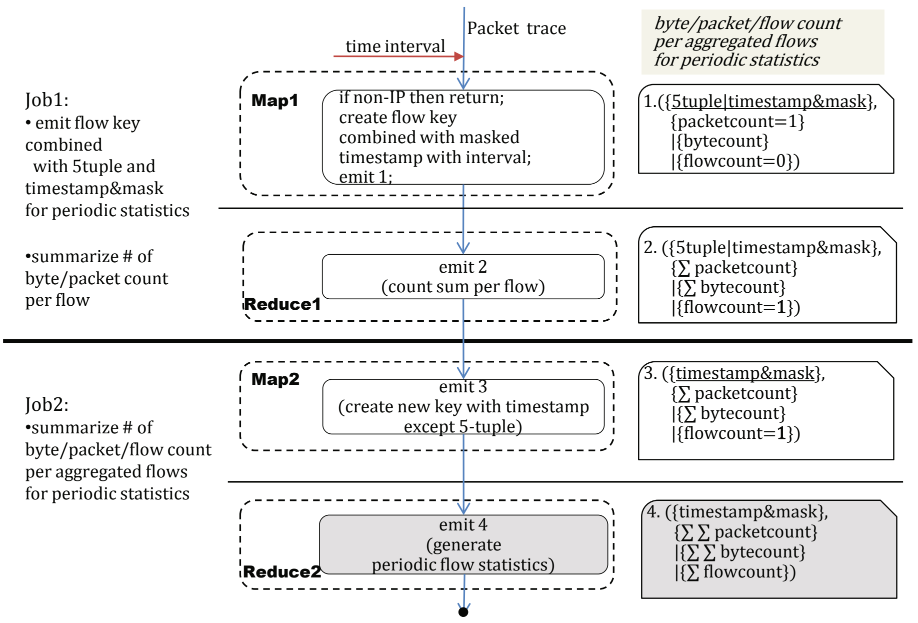
Packet trace
time interval
byte/packet/flow count per aggregated flows for periodic statistics
Job1:
• emit flow key combined with 5tuple and timestamp&mask for periodic statistics
•summarize # of byte/packet count per flow
Map1
if non-IP then return; create flow key combined with masked timestamp with interval; emit 1;
1.({5tuple|timestamp&mask}, {packetcount=1} |{bytecount} |{flowcount=0})
Reduce1
emit 2 (count sum per flow)
2. ({5tuple|timestamp&mask}, {∑ packetcount} |{∑ bytecount} |{flowcount=1})
Job2:
•summarize # of byte/packet/flow count per aggregated flows for periodic statistics
Map2
emit 3 (create new key with timestamp except 5-tuple)
3. ({timestamp&mask}, {∑ packetcount} |{∑ bytecount} |{flowcount=1})
Reduce2
emit 4 (generate periodic flow statistics)
4. ({timestamp&mask}, {∑ ∑ packetcount} |{∑ ∑ bytecount} |{∑ flowcount})

Fig. 5. Periodic flow statistics (flow statistics regarding byte/packet/flow counts per time window): two MapReduce jobs are necessary to perform the aggregated flow statistics

tool with CoralReef, we also have configured a single node that has the same hardware specification with a node of the standard Hadoop testbed. Table 2 shows datasets we used for our experiments. We performed experiments with various amount of datasets from 10, 100, 200, and 400 GB.

Table 1. P^3 testbed

Type	Nodes	CPU	Memory	Hard Disk
Single CoralReef node	1	2.83 GHz (Quad-core)	4 GB	1.5 TB
Standard Hadoop Testbed	5	2.83 GHz (Quad-core)	4 GB	1.5 TB
High-performance Hadoop Testbed	10	2.93 GHz (Octo-core)	16 GB	1 TB

4.1 Scalability

For the scalability test, we ran total traffic statistics and periodic simple statistics jobs under various file sizes. In order to know the impacts of Hadoop resources on performance, we executed P^3 on two Hadoop testbeds of standard 4 nodes and high-performance 10 nodes. For the comparison, we measured the computation time of CoralReef on the standard node. Figure 7 depicts the average computing time for the total traffic statistics (total byte/packet/flow count per IPv4/IPv6/non-IP and the number of unique IP addresses/ports) when the file size varies from 10 to 400 GB. We observed that job completion time is directly proportional to the data volume to be processed by tools. Overall, P^3 on 10

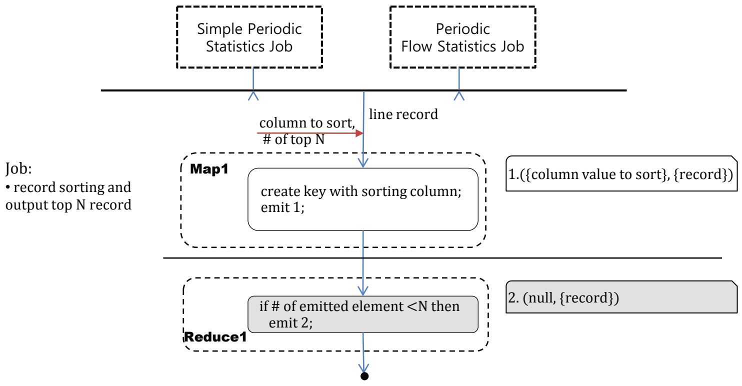
Simple Periodic Statistics Job
Periodic Flow Statistics Job
column to sort, # of top N
line record
Job:
• record sorting and output top N record
Map1
create key with sorting column; emit 1;
1.({column value to sort}, {record})
if # of emitted element <N then emit 2;
Reduce1
2. (null, {record})

Fig. 6. Top N statistics: the top N number of records of periodic flow statistics or periodic simple statistics

Table 2. Test packet trace files

Type	# of packet files	# of packets
10 GB	1	9.4 M
100 GB	1	92.7 M
200 GB	2	185.4 M
400 GB	7	441.1 M

high-performance Hadoop nodes ($P^3(H, 10)$) shows the better performance than CoralReef as well as P^3 on four standard Hadoop workers, $P^3(S, 4)$. It is shown that $P^3(S, 4)$ does not much improve the performance than CoralReef, because the packet processing MapReduce job spends its time in reading/writing data with disk I/O's more than in performing the computation task. However, in case of high-performance 10 worker nodes, P^3 outperforms CoralReef. In 200/400 GB data, $P^3(H, 10)$ completes the job 6.2 (5.8) $\times$ faster than CoralReef.

Next, we conducted an experiment with the periodic simple statistics command. In Fig. 8, it is seen that P^3 on 10 Hadoop nodes ($P^3(H, 10)$) finishes the computation job faster than CoralReef and $P^3(S, 4)$. In 200/400 GB, compared with CoralReef, $P^3(H, 10)$ achieved 7.5 (7.3) $\times$ performance improvement. Though the speed-up ratio is the maximum at 200 GB, we could still reduce the computation time at 400 GB with $P^3(H, 10)$. From the experiments, it is seen that resource-proportional computing could be possible for a large data input with P^3.

4.2 Observation

From the experiments, we could find an interesting observation that Hadoop is especially useful for computing-intensive jobs. We compared the total traffic statistics and periodic flow statistics commands. The total traffic statistics command is more complex than the periodic flow statistics command, because it calculates 14 different analysis items. Therefore, we first expected that the

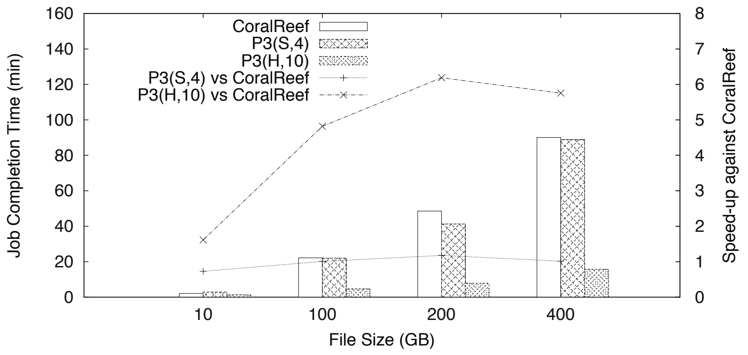

Fig. 7. Completion time of a total traffic statistics job (total byte/packet/flow count per IPv4/v6/non-IP and the number of unique IP addresses/ports): CoralReef vs. P^3 regarding various file sizes and the speed-up

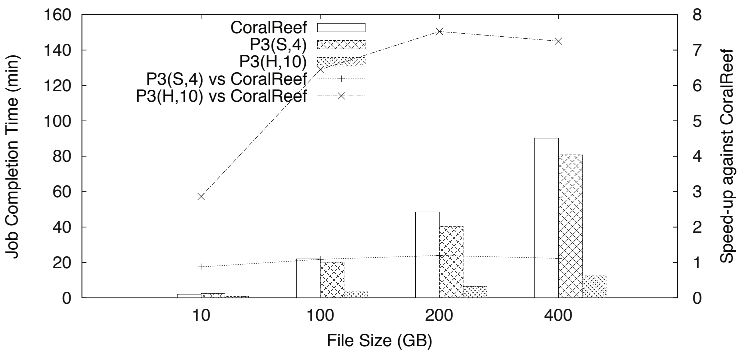

Fig. 8. Completion time of a periodic simple statistics job (byte/packet count per time window): CoralReef vs. P^3 regarding various file sizes and the speed-up

Table 3. Comparison of total traffic statistics and periodic flow statistics tools under 400 GB

	Total Traffic Statistics	Periodic Flow Statistics
# of MapReduce jobs	2	2
# of statistics items	14	2
# of intermediate records by map	3,961 M	441 M
# of intermediate bytes by map	122 GB	16 GB
Completion time (sec)	939	798

performance of the flow statistics command is much better than the total traffic statistic command. However, given the same input data of 400 GB, the completion time of the total traffic statistics command is longer than that of the flow statistics command only by 18%, even though that job emits 9 $\times$ more intermediate records from the map phase as shown in Table 3. This result implies that packet processing and analyzing jobs are I/O-intensive and I/O processing from/to HDFS is the bottleneck of the performance. Therefore, we have to integrate multiple computation jobs together while reducing I/O times in order to develop high-performance traffic analysis applications in MapReduce.

5 Conclusion

We have presented a scalable Hadoop-based parallel packet processor that could analyze large packet trace files. Compared to the conventional packet tools such as tcpdump or CoralReef, our proposal could easily manage large packet trace files of tera- or petabytes, because we have employed the MapReduce platform for parallel processing. In addition, due to the distributed filesystem, HDFS, we could provide the fault tolerance to our traffic analysis system. For packet analysis, we designed representative statistics computation modules with MapReduce. From the experiments, we have shown that our tool outperforms a typical traffic analysis method running on a single server. For the future work, we plan to optimize the performance of MapReduce jobs, to extend P^3 for real-time packet analysis under high-speed links and to enhance the pattern matching algorithm for parsing packet records.

References

1. Tcpdump, `http://www.tcpdump.org`
2. Wireshark, `http://www.wireshark.org`
3. CAIDA CoralReef Software Suite, `http://www.caida.org/tools/measurement/coralreef`
4. Roesch, M.: Snort - Lightweight Intrusion Detection for Networks. In: USENIX LISA (1999)
5. Cisco NetFlow, `http://www.cisco.com/web/go/netflow`
6. Dean, J., Ghemawat, S.: MapReduce: Simplified Data Processing on Large Cluster. In: OSDI (2004)
7. Ghemawat, S., Gobioff, H., Leung, S.: The Google file system. In: SOSP (October 2003)
8. Hadoop, `http://hadoop.apache.org/`
9. Chen, W., Wang, J.: Building a Cloud Computing Analysis System for Intrusion Detection System. In: CloudSlam (2009)
10. Lee, Y., Kang, W., Son, H.: An Internet Flow Analysis Method with MapReduce. In: 1st IFIP/IEEE Workshop on Cloud Management (April 2010)
11. Conner, J.: Customizing Input File Formats for Image Processing in Hadoop, Arizona State University Technical Report (2009)
12. Zaharia, M., Konwinski, A., Joseph, A.D., Katz, R., Stoica, I.: Improving MapReduce Performance in Heterogeneous Environments. In: OSDI (2008)
13. Kambatla, K., Pathak, A., Pucha, H.: Towards Optimizing Hadoop Provisioning in the Cloud. In: USENIX Hotcloud (2009)
14. Condie, T., Conway, N., Alvaro, P., Hellerstein, J.M., Elmeleegy, K., Sears, R.: MapReduce Online. ACM/USENIX NSDI (April 2010)
15. `https://sites.google.com/a/networks.cnu.ac.kr/yhlee/p3`