Online Internet traffic monitoring system using spark streaming

Article *in* Big Data Mining and Analytics · March 2018

DOI: 10.26599/BDMA.2018.9020005

CITATIONS
24

READS
1,659

7 authors, including:

Jie Li
University of Tsukuba
760 PUBLICATIONS **9,817** CITATIONS
SEE PROFILE

Xiaoyan Wang
Ibaraki University
83 PUBLICATIONS **488** CITATIONS
SEE PROFILE

li xu
Fujian Normal University
119 PUBLICATIONS **978** CITATIONS
SEE PROFILE

BIG DATA MINING AND ANALYTICS
ISSN 2096-0654 0?/?? pp???–???
Volume 1, Number 1, January 2018

Online Internet Traffic Monitoring System Using Spark Streaming

Baojun Zhou, Jie Li*, Xiaoyan Wang, Yu Gu, Li Xu, Yongqiang Hu, and Lihua Zhu

Abstract: Owing to the explosive growth of Internet traffic, network operators must be able to monitor the entire network situation and efficiently manage their network resources. Traditional network analysis methods that usually work on a single machine are no longer suitable for huge traffic data owing to their poor processing ability. Big data frameworks, such as Hadoop and Spark, can handle such analysis jobs even for a large amount of network traffic. However, Hadoop and Spark are inherently designed for offline data analysis. To cope with streaming data, various stream-processing-based frameworks have been proposed, such as Storm, Flink, and Spark Streaming. In this study, we propose an online Internet traffic monitoring system based on Spark Streaming. The system comprises three parts, namely, the collector, messaging system, and stream processor. We considered the TCP performance monitoring as a special use-case of showing how network monitoring can be performed with our proposed system. We conducted typical experiments with a cluster in standalone mode, which showed that our system performs well for large Internet traffic measurement and monitoring.

Key words: Spark Streaming, network monitoring, big data, TCP performance monitoring

1 Introduction

To provide a secure and well-performing network for the continually transforming cyberspace, Internet operators need to monitor and analyze the network status in real time. However, this is difficult nowadays due to the huge scalability of networks and the huge amount of traffic to be analyzed. According to the latest statistics by Cisco[1], the annual global IP traffic was 1.2 zettabyte (ZB) in 2016 and expected to reach 3.3 ZB by 2021. The rapid growth of traffic volume has imposed great challenges for traditional Internet monitoring platforms.

Traditionally, Internet traffic measurement and analysis have been executed on a high-performance central server[2]. However, due to computing ability limitations, central servers cannot cope with large volumes of data in a short period of time. Nowadays, this renders them unsuitable given the huge amount of Internet traffic today. For example, when a DDoS attack occurs, a monitoring system is required in order to deal with the huge amount of Internet traffic, which is a tough task for a single server. Various monitoring methods use packet sampling in order to reduce the amount of input data. However, this produces an inaccurate result[3]. Moreover, a single server makes the system vulnerable to failures. If the server crashed, we would not be able to recover it quickly without affecting the ongoing task[4].

• Baojun Zhou and Jie Li are with the Department of Computer Science, University of Tsukuba, Tsukuba 305-8577, Japan. zhoubaojun@osdp.cs.tsukuba.ac.jp and lijie@cs.tsukuba.ac.jp
• Xiaoyan Wang is with the College of Engineering, Ibaraki University, Hitachi 316-8511, Japan. xiaoyan.wang.shawn@vc.ibaraki.ac.jp
• Yu Gu is with the School of Computer and Information, Hefei University, Hefei 230601, China. yugu.bruce@ieee.org
• Li Xu is with the College of Mathematics and Computer Science, Fujian Normal University, Fuzhou 350007, China. xuli@fjnu.edu.cn
• Yongqiang Hu and Lihua Zhu are with the Institute of Scientific and Technical Information of Qinghai, Xining 810008, China. yqhuu@163.com and zlh197330@163.com
* To whom correspondence should be addressed.
Manuscript received: xxxx-xx-xx; accepted: xxxx-xx-xx

In this study, we propose an online Internet traffic monitoring system based on Spark Streaming, which is a big data platform that can efficiently process a huge amount of traffic data so that we can monitor the network status in real time and is robust enough so as to suffer a failure without aborting the entire monitoring process.

Big data platforms such as Hadoop and Spark provide an efficient way of processing a huge amount of data. For example, the MapReduce model and its open-source version, Hadoop[5], have been widely adopted by the big data analytics community due to their simplicity and ease of programming[6]. However, the intermediate data of Hadoop are stored on disk (which usually has poor I/O performance); therefore, there will be dramatic performance degradation for algorithms requiring plenty of iterations. To improve the performance of Hadoop, in-memory computing methods, such as Apache Spark[7], have been proposed. The intermediate data in Spark are stored in resilient distributed datasets (RDD), which are cached in memory; therefore, data can be processed much faster in comparison with Hadoop[8].

Some offline Internet monitoring systems [2, 9–11] have used big data platforms to improve their processing efficiency. However, only a few studies [12, 13] have focused on online network monitoring.

Both Hadoop and Spark are based on batch processing, which is suitable for offline data analysis. Batch processing is applied to process large datasets, where operations on multiple data items can be batched for efficiency[14]. This requires input data to be readily accessible when the calculation begins so that all of the data can be simultaneously processed. Online Internet traffic monitoring resembles a stream analytics problem, where the input is an unbounded sequence of data. Although MapReduce does not support stream processing, it can partially handle streams using a technique known as micro-batching. Here the stream is treated as a sequence of small batch data chunks. At short intervals, the incoming stream is packed to a chunk of data and is delivered to the batch system for processing[15]. Spark, for example, has provided the Spark Streaming library to support this technique. In addition, other platforms exist and are inherently designed for big data streams, such as Apache Storm and S4, where data is processed through several computing nodes. Each node can process one or more input stream(s) and generate a set of output streams. Data will be processed as soon as they arrive.

Network traffic analysis and monitoring can be regarded as a statistics problem for a set of packets in a period of time. Therefore, micro-batching may be a better choice for a network monitoring system. In this study, we propose an online Internet traffic monitoring system based on Spark Streaming. This could be used for data traffic performance analysis, such as TCP performance.

Our contributions in this study are as follows:

- We propose a distributed architecture as an online Internet traffic measurement and monitoring system.
- We implement a parallel algorithm for monitoring TCP performance parameters, such as delay and retransmission ratio with a very short delay.
- We conduct typical experiments showing that the proposed system is feasible and efficient.

The rest of this study is organized as follows. In section II, we introduce various related works on real-time Internet monitoring. In section III, we describe the architecture of our proposed monitoring system and introduce the tools we used. In section IV, we consider TCP performance analysis as an example of how the network monitoring problem can be solved by the stream-processing method in our system. In section V, we evaluate the performance of the proposed system and present the experimental results. Finally, we conclude this study in section VI.

2 Related Work

Cyberspace is dynamical and vulnerable to attacks. Therefore, it requires network providers to monitor the status of their network in real time. Online Internet traffic monitoring technologies have been extensively studied. In 1999, Paxson et al.[16] proposed the Bro system to detect network intruders in real time. Bro first captured a packet stream using `libpcap` and then reduced the incoming stream into a series of higher-level events using an event engine. They also proposed a custom scripting language called Bro scripts, which can be executed by the policy script interpreter to deal with events. Although Bro is single threaded, it can be set up in a high throughput cluster environment. Similar studies include Snort[17] and Suricata[18], which are inherently based on single-machine computing.

Various related studies have been conducted on online Internet traffic measurement and monitoring using Spark. Gupta et al.[12] used Spark Streaming to analyze the network in real time. They presented three network monitoring applications that can be expressed as streaming analytics problems; namely, reflection attack monitoring, application performance analysis, and port scan detection. They made use of programmable switches, such as OpenFlow switches, to extract only the traffic that was of interest, which reduced the data that needed to be processed. However, their system was not feasible for networks using non programmable switches. In this study, we propose an Internet traffic measurement and monitoring system that works on both programmable and non programmable switches. Karimi et al.[13] proposed a distributed network traffic feature extraction method with Spark for a real time intrusion detection system. They used a collector component to capture packets from the switch and extract the required information from packet headers. These headers are written in CSV files and separated by the time window. Then, Spark periodically reads data from the CSV files within a small-time window to make it nearly real-time data. However, the periodic writing and reading of files degrades the performance of Spark as an online Internet traffic monitoring system. Our system uses Spark Streaming to directly cope with the stream in order to achieve a higher speed.

3 Architecture of The Online Internet Traffic Monitoring System

As illustrated by Figure 1, the proposed online monitoring system comprises three components, namely the *collector*, *messaging system*, and *stream processor*. A *collector* is a device that is used to capture packets from the network. It captures all the inbound and outbound packets from a switch using port mirroring. To capture packets from multiple switches, multiple *collector*s may be present in the system. The *stream processor* is the core component of our system and processes the input data transmitted from the *collectors*. First, the *collector* preprocesses the captured packets and only sends necessary protocol header data to the *stream processor* to reduce the amount of input data. Moreover, we use a *messaging system* as a bridge to help the data transmission from the *collectors* to the *stream processor*.

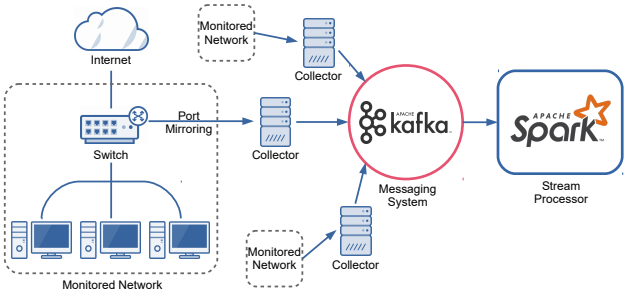


Fig. 1 The architecture of the proposed online monitoring system

3.1 Collector

To implement the *collector* component, we used `Jpcap`, which is an open-source Java library, to capture packets and efficiently extract the required packet header data such as time stamp and TCP segment number. To reduce the number of captured packets, we set a filter (protocol, destination IP address, etc.) for `Jpcap` so that it captures only the packet that we are interested in.

3.2 Messaging System

We use Kafka to develop our *messaging system*. Kafka is a high-performance distributed messaging system for record streams[19]. The *collector*s publish the required packet header data as messages on Kafka using a common topic. Then, the *stream processor* subscribes to the topic and obtains the message stream from Kafka.

3.3 Stream Processor

The *stream processor* component is a cluster running Spark Streaming. It processes the packet information collected by the *collector*s and shows the monitoring results to the operators. This is similar to the Spark program for batch processing, whose logic is expressed by RDD transformations. The logic in Spark Streaming is expressed by discretized stream (DStream), which is an internal sequence of RDDs, transformations[20]. Spark Streaming has provided multiple transformation APIs, such as `flatMap()`, which maps each input item in the source DStream to 0, or more output items, and `groupByKey()`, which can group key-value pairs together according to their keys. Some useful transformation APIs are listed in Table 1[20].

4 TCP Performance Monitoring

TCP is a widely used Internet protocol. In this study, we focus on TCP performance monitoring. The performance monitoring problems of other protocols

Table 1 Various useful transformations Spark Streaming APIs

Transformation	*Meaning*
map()	Map each element in the source stream to a new value.
flatMap()	Similar to map(), but each element can be mapped to 0 or more output items.
mapValues()	Map the value of each key-value pair without change the key.
reduce()	Aggregate each 2 elements in the source stream to 1 new element.
reduceByKey()	Aggregate 2 key-value pairs with the same key to a new key-value pair.
groupByKey()	Group all key-value pairs with the same key together.
countByValue()	Count the frequency of each element, and return a key-value pair stream whose key is the element, value is the count.
join()	Join two key-value pair streams (K, V) and (K, W) together, return a new stream of (K, ⟨V, W⟩) pairs with all pairs of elements for each key.

can be dealt with in a similar way. We used our system to calculate the TCP segment throughput, average round-trip time (RTT), retransmission ratio, and out-of-order ratio from the Internet traffic stream in real time.

4.1 Metrics

4.1.1 Throughput

Our system can measure the number and total length of TCP segments per second.

4.1.2 Retransmission and Out-of-Order

TCP provides an ordered delivery of data. Each segment's protocol header contains a sequence number (SEQ), which specifies the position of this segment. When a TCP connection is established, both the sender and receiver in the connection are assigned an initial sequence number. The offset between the actual sequence number in a segment and the initial one is called the relative sequence number and is regarded as the number of bytes sent by the sender before this segment.

From the sequence number and payload (data length) of a segment, we can calculate the next expected sequence number, which indicates the sequence number that the next segment from the host should have. For a normal TCP data segment that is not SYN/FIN, the next expected sequence number should equal the sum of the current sequence number and payload. However,

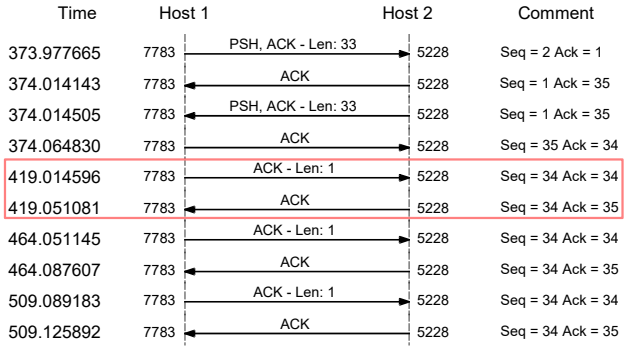


Fig. 2 Typical TCP keep-alive

for the SYN/FIN segment, the next expected sequence number should equal the previous one plus one because the SYN/FIN information is also considered as data that is sent to the receiver.

In normal TCP conversation, the sequence number of SYN/FIN or a segment containing data should not be smaller than the maximum next expected sequence number in previous segments. Otherwise, it must be a segment that is either retransmitted or out-of-order. Retransmission means that a segment is considered lost and is therefore resent by the sender; out-of-order means that the segments are not received by the receiver in the order they are sent. In fact, it is quite difficult to identify whether, in some cases, a segment is retransmitted or just out-of-order. Here we use a simple assumption to determine the out of order segment. In other words, if the time stamp of the segment is very close (within three milliseconds) to the segment with the highest next expected sequence number, we regard it as an out-of-order segment. A high retransmission and out-of-order ratio will cause poor network performance and indicate problems in the transmission paths; therefore, they are very important metrics in TCP performance monitoring.

In addition, we shall consider an exception in our retransmission and out-of-order detection; namely, keep-alive. A TCP keep-alive segment is an acknowledgment segment (ACK) with its sequence number set to one less than the next expected sequence number and has a payload of 0 or 1 byte. Keep-alive is often used to verify whether the TCP connection with the remote host is still available. Although it may carry 1 byte of data and does not advance the sequence number, it is not a retransmitted or out-of-order segment. A typical TCP keep-alive is shown in Figure 2.

4.1.3 RTT

RTT is the time required for a network communication to travel from the source to the destination and back. In TCP, when the receiver successfully receives a segment, it sends back a response segment that contains an ACK number. The latency between the former TCP segment and the corresponding ACK is measured as RTT. This is an important network parameter that reflects not only network quality but also server response time. The ACK number tells the sender how many bytes have been successfully received by the receiver, which should be equal to the next expected sequence number of the received TCP segment. In other word, besides the reverse IP addresses/ports, a TCP segment and its corresponding ACK segment should also satisfy:

$$\begin{aligned}&\text{SEQ number + payload (in TCP segment)}\\&\quad = \text{ACK number (in ACK segment)}\end{aligned} \tag{1}$$

Especially, for SYN/FIN segment:

$$\begin{aligned}&\text{SEQ number + payload + 1 (in TCP segment)}\\&\quad = \text{ACK number (in ACK segment)}\end{aligned} \tag{2}$$

We use relationships (1) and (2) to find the corresponding TCP and ACK segments in the packet list. The RTT is calculated as the difference between their time stamps. However, if there are retransmissions in the TCP conversation, the problem becomes complicated, such that, when an ACK is received for a retransmitted segment, we cannot determine whether the ACK corresponds to the retransmission or the original segment. Even if we decide that the segment was lost and retransmitted, it would still be possible for the segment to eventually arrive after a long time or for the segment to arrive quickly while the ACK takes a long time to arrive. This is called *acknowledgment ambiguity*[21]. In our algorithm, we select the original segment to calculate the RTT. Although it may not be a real RTT, we think it is more meaningful than the real one because it shows the delay from the first time of the sender sending the segment to the time of knowing that the receiver has received the segment.

4.2 TCP Performance Monitoring with The Proposed System

TCP performance analysis is divided into five steps, namely preprocessing, throughput calculation, retransmission and out-of-order statistics, RTT calculation, and sum up.

4.2.1 Preprocessing

We set the *collector*s to only capture TCP segments from the Internet. Spark Streaming receives header information from them through sockets. The information for each segment is a formatted string containing the time stamp (divided into a high and low part), source IP, source port, destination IP, destination port, SYN/FIN flag, ACK flag, SEQ number, ACK number, payload, and packet length. We denote this input stream `DStream0`.

Input stream data preprocessing is required to extract the information required for the calculation to be performed later. We map each input data as a tuple ⟨time stamp, source IP, source port, destination IP, destination port, `boolean`(SYN/FIN, or carry data), `boolean`(ACK?), sequence number, acknowledgment number, next expected sequence number, payload, and packet length⟩. For a segment that is not SYN/FIN and carries no data, the next expected sequence number is simply its sequence number. These tuples are stored in `DStream1`.

4.2.2 Throughput Calculation

Throughput is very easy to calculate by Spark Streaming. First, we transform each tuple in `DStream1` to a key-value pair, whose key is ⟨source IP, destination IP⟩ and value is ⟨packet length, 1⟩. Next, we use `reduceByKey` to sum up the values according to the key. Finally, we obtain the total packet length and total number of packets for each source and destination IP pair and store them in `DStream2`.

4.2.3 Retransmission and Out-of-Order Statistics

We calculate the retransmission and out-of-order number. First, we map the tuples in `DStream1` to a key-value pair, whose key is ⟨source IP, source port, destination IP, destination port⟩ and value is ⟨`boolean`(SYN/FIN or contains data?), sequence number, payload, next expected sequence number, time stamp⟩. We group the key-value pairs according to the key and obtain the lists of segments that share the same source and destination IP addresses/ports. Then we can calculate the number of retransmission and out-of-order segments for each list using algorithm 1. We only count retransmission and out-of-order for SYN/FIN segments or for segments carrying data.

We generate a new key-value pair for each list. The key is a tuple of ⟨source IP, destination IP⟩, and the value is a tuple of ⟨number of retransmission, number of out-of-order⟩. Because each list only contains segments

Algorithm 1: Get statistics for each segments list.

Data: A list of segments that have the same source and destination IP/port
Result: Get the number of retransmission and out-of-order.
sort(); `/* sort the list in time order */`
foreach *segment in the segments list* **do**
  if *is an SYN/FIN or carries data and SEQ* $<$ *maxNextSEQ* **then** `/* SEQ number< max next expected SEQ, then it must be either a keep-alive, retransmission or out-of-order */`
    if *not keep-alive* **then**
      if $\Delta(maxTimeStamp, timeStamp) < 3$ **then** `/* within 3 milliseconds since the segment with the highest next expected sequence number */`
        $outOfOrder \leftarrow outOfOrder + 1$;
        `/* we assume it is an out-of-order */`
      else
        $retransmission \leftarrow retransmission + 1$; `/* we assume it is a retransmission */`
  if *next expected SEQ is the max one ever seen* **then**
    `/* record the max next expected SEQ and its time stamp */`
    $maxNextSEQ \leftarrow nextSEQ$;
    $maxTimeStamp \leftarrow timeStamp$;
return new key-value pair (⟨Source IP, Destination IP⟩, $\langle retransmission, outOfOrder\rangle$);

Algorithm 2: Calculate the RTTs from the list of paired TCP/ACK segments

Data: A list of paired TCP/ACK segments
Result: Calculate the RTT
sort(); `/* sort the list in time order */`
$f \leftarrow false$; `/* whether there is a new TCP segment from sender to receiver after last ACK segment */`
foreach *segment in the list* **do**
  if *is TCP segment that from sender to receiver and* $f = false$ **then**
    $time1 \leftarrow$ TCP segment's time stamp;
    $f \leftarrow true$;
  else if *is corresponding ACK segment and* $f = true$ **then**
    $time2 \leftarrow$ ACK segment's time stamp;
    $f \leftarrow false$;
    $RTT \leftarrow (time2 - time1)$;
    generate new key-value pair (⟨Sender's IP, Receiver's IP⟩, $\langle RTT, 1\rangle$);

from a source IP/port to a destination IP/port, the generated key-value pair only contains the statistics of segments between the specific IP/ports. Therefore we use `reduceByKey()` to group all the pairs that have the same source and destination IP and calculate their total number. The group was stored in `DStream3`.

4.2.4 RTT Calculation

According to the relationship of the TCP and ACK pair, if we use the tuple of ⟨source IP address, source port, destination IP address, destination port, next expected sequence number⟩ as a key for the TCP segment and ⟨destination IP address, destination port, source IP address, source port, acknowledgment number⟩ as a key for the ACK segment, then a TCP segment and its corresponding ACK segment should share the same key. Therefore, we can group all TCP segments and their corresponding ACK segments in Spark Streaming using `groupByKey()` and perform the parallel calculation of RTT for each TCP segment.

First, we map each segment in `DStream1` to several key-value pairs. If it is an SYN/FIN segment or contains data we generate a key-value pair, whose key is ⟨source IP address, source port, destination IP address, destination port, next expected sequence number⟩ and value is ⟨time stamp, `true` (means it is a data segment)⟩. If it is a segment whose ACK flag is set to true, we generate a key-value pair, whose key is ⟨destination IP address, destination port, source IP address, source port, acknowledgment number⟩ and value is ⟨time stamp, `false` (this means it is an ACK segment)⟩. Note that an ACK segment can also carry data (piggyback), and a segment can be neither a data nor an ACK/FIN/SYN segment; therefore, each segment can finally generate 0, 1, or 2 pair(s) according to their type. Because the former TCP segment is actually a segment sent from the sender to the receiver and the ACK segment is sent from the receiver to the sender, the keys of the key-value pairs can be regarded as ⟨sender IP address, sender port, receiver IP address, receiver port, next expected sequence number⟩ and ⟨sender IP address, sender port, receiver IP address, receiver port, acknowledgment number⟩, respectively.

We use `groupByKey()` so that the paired TCP/ACK segment can be grouped into the same list. Then, we can calculate RTTs from the list using algorithm 2.

Table 2 Configurations for each component

Component	*Configuration*
Collector	2 machines. Model t2.micro, 1 GB memory.
Messaging System	1 machine. Model r4.large, High-frequency Intel Xeon E5-2686 v4 (Broadwell) Processors, 15.25 GB memory. Bandwidth up to 10 Gbps.
Stream Processor	5 machines. Model c4.large, High-frequency Intel Xeon E5-2666 v3 (Haswell) processors, 3.75 GB memory. Bandwidth of 500Mbps.

We generate a set of key-value pairs for each sender and receiver IP/port pair, whose key is ⟨sender IP, receiver IP⟩, value is ⟨RTT, 1⟩. Then, we use `reduceByKey()` to sum up the values by key, obtain ⟨total RTT, total count⟩ for each IP pair, and use `mapValues` to calculate the average RTT, stored in `DStream4`.

4.2.5 Sum Up

In steps 2, 3, and 4, we obtain the packet count, total length, retransmission, out-of-order statistics, and the RTTs for each source IP/port and destination IP/port pair and store them in `DStream2`, `DStream3`, and `DStream4`, respectively, so that we can use the `join()` method to merge these results into one stream.

To present the Spark Streaming calculation results, we develop a graphic user interface to show the monitored Internet performance metrics for each specified source and destination IP pair as line charts.

5 Experimental Result

We used eight servers from Amazon Web Services to implement the proposed online TCP performance measurement system. Their configurations are presented in Table 2. The structural details are shown in Figure 3.

Currently, in the an experimental environment, the *collector* only captures packets from a single computer. The batch interval is set to 1 s, which means that Spark Streaming will start a computation every 1 s and information from all segments within this time interval will be collected and processed as a batch task.

Our proposed system monitors the packet rate (packet/s), throughput (byte/s), average RTT (ms), retransmission ratio (%), and out-of-order ratio (%) for each source and destination IP pair in real time. Figure 4 shows the network performance that was monitored by

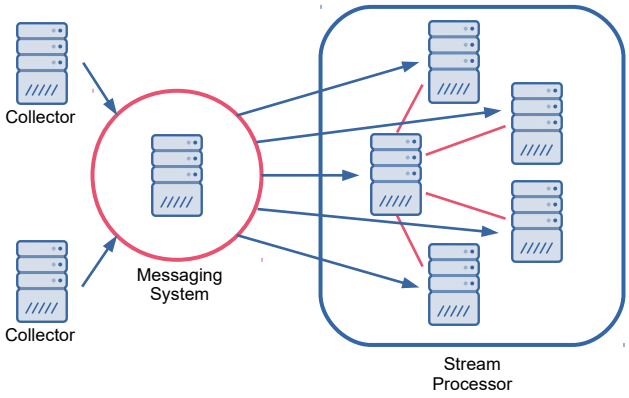


Fig. 3 System structure

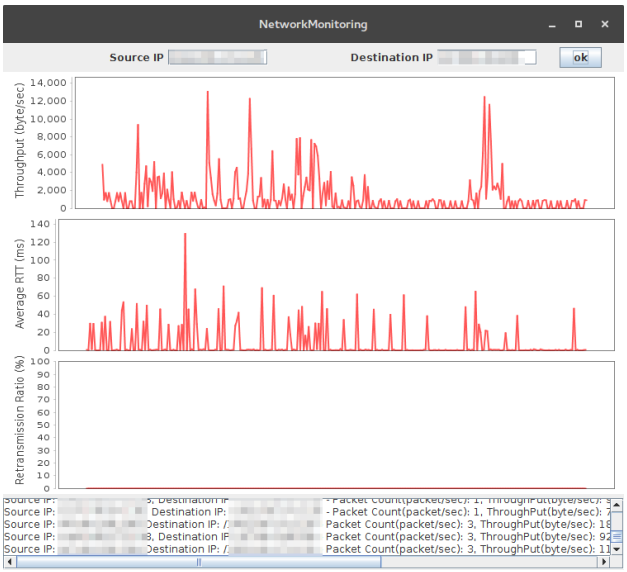


Fig. 4 Network performance measured by our system

our proposed system. The metrics are presented through line plots so that the network situation can be clearly monitored.

We investigated the TCP monitoring performance of the proposed system both in terms of time cost and robustness.

5.1 Time Cost

We developed a special *collector* to evaluate the performance of our system. Unlike conventional *collector*s that capture packets from Internet in real time, this special *collector* reads packet records from a large packet capture (`pcap`) file.

Figure 5 shows the performance statistics of our stream processor. In this experiment, the average speed of the input stream was approximately 75,000 records/s. We found that our proposed system processed the input stream very quickly. Because we set the batch interval to 1 s, if the batch task could be processed within the

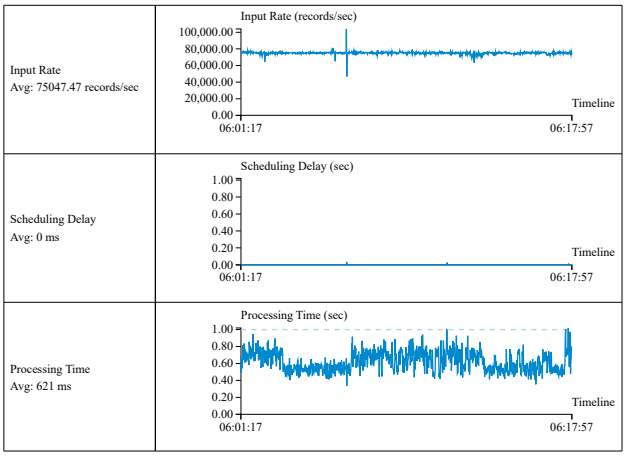


Fig. 5 Performance statistics of *stream processor* in Spark UI, where processing time is the time taken to process all jobs for a batch, and scheduling delay is the time to ship the jobs from scheduler to executor

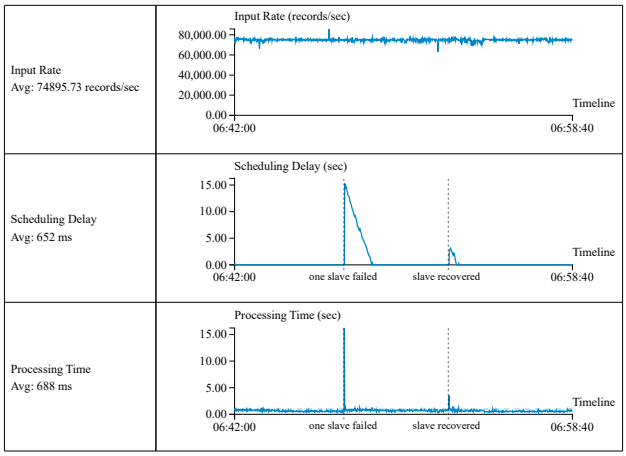


Fig. 6 System performance changes when a slave crashes

interval, the system would be stable. We found that the average total delay was 621 ms, which was within the 1 s interval. Therefore, it was demonstrated that our system can handle such a high speed input stream.

We increased the packet rate and found that our system could deal with nearly 150,000 packets per second.

5.2 Robustness

The traditional network monitoring systems process on a single server and abort when the server crashes. In a cluster that is set up in Spark standalone mode, the jobs will be distributed to and completed by several slave machines. When a slave breaks down, other slaves will resume its remaining jobs without aborting the entire process.

We conducted a typical experiment to investigate the robustness of our proposed system. We manually shut down one slave when the program was running and restarted it later. The system status is shown in Figure 6. When the slave crashed, its tasks were retransmitted to other slaves, and we could observe a jitter in the scheduling delay and processing time. The master had to retransmit the jobs to the slave after it was recovered. Therefore, a jitter was also observed in scheduling delay. Although the failure and recovery of a slave impacted the performance of the system, the system returned to normal operation very quickly and the entire job was not aborted. These experimental results show that the system is robust.

6 Conclusion and Future Work

With the growth of Internet traffic, traditional network analysis methods that work on single machines are no longer suitable. Existing approaches take advantage of big data frameworks to improve processing efficiency. However, these approaches mainly focus on offline data analysis. In this study, we proposed an online Internet traffic monitoring system that utilizes Spark Streaming. We demonstrated that Internet measurement and monitoring can be treated as a stream analysis problem and can be handled via a streaming processing platform. Extensive experimental results show that our system achieved good performance and robustness.

In future, we will implement *collector*s to capture packets from switches through port mirroring so that our system can analyze all the traffics passing through monitored networks. Finally, we will test its performance in practice and compare it with some traditional single server systems in terms of scalability and reliability.

Acknowledgment

This work has been partially supported by Grant-in-Aid for Scientific Research from Japan Society for Promotion of Science (JSPS), Qinghai Joint Research Grant (no. 2016-HZ-804), and Research Collaboration Grant from NII, Japan.

References

[1] N. I. Visual, *Forecast and methodology, 2016-2021, white paper*, San Jose, CA, USA: Cisco, 2016.

[2] Y. Lee, W. Kang, and H. Son, An internet traffic analysis method with mapreduce, in *Network Operations and Management Symposium Workshops (NOMS Wksps), 2010 IEEE/IFIP*, Osaka, Japan, 2010, pp. 357-361.

[3] D. Brauckhoff, B. Tellenbach, A. Wagner, M. May, and A. Lakhina, Impact of packet sampling on anomaly detection metrics, in *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, Rio de Janeiro, Brazil, 2006, pp. 159-164.
[4] Y. Qiao, Z. Lei, Y. Lun, and M. GUO, Offline traffic analysis system based on hadoop, *The Journal of China Universities of Posts and Telecommunications*, vol. 20, no. 5, pp. 97-103, Oct. 2013.
[5] Hadoop, http://hadoop.apache.org/, 2017
[6] K. Kambatla, G. Kollias, V. Kumar, and A. Grama, Trends in big data analytics, *Journal of Parallel and Distributed Computing*, vol. 74, no. 7, pp. 2561-2573, Jul. 2014.
[7] Apache Spark, http://spark.apache.org/, 2017
[8] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, Spark: Cluster computing with working sets, in *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, Boston, USA, 2010, p. 10-10.
[9] J. Liu, F. Liu, and N. Ansari, Monitoring and analyzing big traffic data of a large-scale cellular network with hadoop, *IEEE network*, vol. 28, no. 4, pp. 32-39, Jul. 2014.
[10] Y. Lee and Y. Lee, Toward scalable internet traffic measurement and analysis with hadoop, *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 1, pp. 5-13, Jan. 2013.
[11] Z. Chen, G. Xu, V. Mahalingam, L. Ge, J. Nguyen, W. Yu, and C. Lu, A cloud computing based network monitoring and threat detection system for critical infrastructures, *Big Data Research*, vol. 3, pp. 10-23, Apr. 2016.
[12] A. Gupta, R. Birkner, M. Canini, N. Feamster, C. Mac-Stoker, and W. Willinger, Network monitoring as a streaming analytics problem, in *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, Atlanta, USA, 2016, pp. 106-112.
[13] A. M. Karimi, Q. Niyaz, W. Sun, A. Y. Javaid, and V. K. Devabhaktuni, Distributed network traffic feature extraction for a real-time ids, in *Electro Information Technology (EIT), 2016 IEEE International Conference on*, Grand Forks, USA, 2016, pp. 0522-0526.
[14] C. P. Chen and C. Zhang, Data-intensive applications, challenges, techniques and technologies: A survey on big data, *Information Sciences*, vol. 275, pp. 314-347, Aug. 2014.
[15] S. Shahrivari, Beyond batch processing: towards real-time and streaming big data, *Computers*, vol. 3, no. 4, pp. 117-129, Oct. 2014.
[16] V. Paxson, Bro: a system for detecting network intruders in real-time, *Computer networks*, vol. 31, no. 23, pp. 2435-2463, Dec. 1999.
[17] M. Roesch, Snort: Lightweight intrusion detection for networks, in *Proceedings of LISA '99: 13th Systems Administration Conference*, Seattle, USA, 1999, pp. 229-238.
[18] Suricata, https://suricata-ids.org/, 2017
[19] Kafka performance, https://engineering.linkedin.com/kafka/benchmarking-apache-kafka-2-million-writes-second-three-cheap-machines, 2017
[20] Spark Streaming, http://spark.apache.org/docs/latest/streaming-programming-guide.html, 2017
[21] Acknowledgment ambiguity, http://www.tcpipguide.com/free/t_TCPAdaptiveRetransmissionandRetransmissionTimerCal-2.htm, 2017

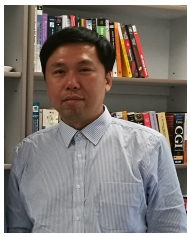
Baojun Zhou Photo. Baojun Zhou received the BE degree in Electronics and Information Engineering from the Huazhong University of Science and Technology, Wuhan, China, in 2014. He is currently working toward the ME degree at the University of Tsukuba, Tsukuba, Japan, in the Department of Computer Science. His research interests include big data, machine learning and network security.

Jie Li Photo. Jie Li received the B.E. degree in computer science from Zhejiang University, Hangzhou, China, the M.E. degree in electronic engineering and communication systems from China Academy of Posts and Telecommunications, Beijing, China. He received the Dr. Eng. degree from the University of Electro-Communications, Tokyo, Japan. He is with Faculty of Engineering, Information and Systems, University of Tsukuba, Japan, where he is a Professor. His current research interests are in mobile distributed computing and networking, big data and cloud computing, network security, OS, modeling and performance evaluation of information systems. He was a visiting Professor in Yale University, USA, Inria Sophia Antipolis, France and Inria Grenoble Rhone-Alpes, France. He is a senior member of IEEE and ACM and a member of IPSJ (Information Processing Society of Japan). He serves as an associated editor for many international journals and transactions such as IEEE IoT Journal, IEEE Access Journal, IEEE Trans on TVT, and IEEE Trans on Cloud Computing. He is the founding Chair of Technical Committee on Big Data (TCBD), IEEE ComSoc. He has also served on the program committees for several international conferences such as IEEE INFOCOM, IEEE GLOBECOM, and IEEE MASS.

Xiaoyan Wang Photo. Xiaoyan Wang (S'11-M'14) received the BE degree from Beihang University, China, and the ME and Ph. D. from the University of Tsukuba, Japan. He is currently working as an assistant professor with the College of Engineering at Ibaraki University, Japan. Before that, he worked as an assistant professor at National Institute of Informatics (NII), Japan, from

2013 to 2016. His research interests include wireless networks, network security, game theory and machine learning.

Yu Gu Photo. Yu Gu received his B.E. degree from the Special Classes for the Gifted Young (SCGY), University of Science and Technology of China (USTC) in 2004. He received his D.E. degree from the same university in 2010. From 2006.2 to 2006.8, he has been an intern in the Microsoft Research Asia, Beijing, China. From 2007.12 to 2008.12, he has been a visiting scholar in the University of Tsukuba, Japan. From 2010.11 to 2012.10, he has worked in the National Institute of Informatics (Japan) as a JSPS Research Fellow. Now He is a Professor in School of Computer and Information, Hefei University of Technology, China. His research interests include pervasive computing and affective computing. He is a senior member of IEEE. He received the Excellent Paper Award in the IEEE Scalcom 2009.



Li Xu Photo. Li Xu is a Professor and Doctoral Supervisor at the school of Mathematics and Computer Science at Fujian Normal University. He received his B.S and M.S degrees from Fujian Normal University in 1992 and 2001. He received the Ph.D. degree from the Nanjing University of Posts and Telecommunications in 2004. Now he is the director of office of informatization, in Fujian Normal University and Director of Key Lab of Network Security and cryptography in Fujian Province. His Interested include network optimization and network security, complex network and system, wireless communication network and communication, etc. Prof. Xu has been invited to act as PC chair or member at more than 30 international conferences. He is a member of IEEE and ACM, and a senior member of CCF and CIE in China. He has published over 150 papers in refereed journals and conferences.

Yongqiang Hu Photo. Yongqiang Hu is the director of Institute of Scientific and Technical Information of Qinghai Province, China. His current research interests are in science and technology information platform, big data analysis, rural information platform, information technology theory, science and technology strategy.

Lihua Zhu Photo. Lihua Zhu is with Institute of Scientific and Technical Information of Qinghai Province, China, where she is an associate research fellow of Library Science. Her current interests are in technical archives development, soft science, science and technology strategic development, science and technology information methodology.