A MapReduce Approach for Traffic Matrix Estimation in SDN

WANDER J. QUEIROZ1, MIRIAM A. M. CAPRETZ1, (Senior Member, IEEE), AND MARIO A. R. DANTAS2

1Department of Electrical and Computer Engineering, Western University, London, ON N6A 5B9, Canada
2Department of Computer Science, Federal University of Juiz de Fora, Juiz de Fora 36036-330, Brazil

Corresponding author: Miriam A. M. Capretz (mcapretz@uwo.ca)

This work was supported in part by the Brazilian Ministry of Science and Technology, Brazilian Council for Scientific and Technological Development (CNPq), under Grant 203590/2014-7, and in part by an Natural Sciences and Engineering Research Council of Canada - Strategic Partnership Grants for Projects (NSERC STPGP) at Western University under Grant STPGP 506840-17.

ABSTRACT A traffic matrix (TM) is a source of critical traffic throughput information for traffic engineering activities and network management tasks such as traffic prediction, capacity planning, network provisioning, and anomaly detection. However, estimating TM poses several challenges for network engineers. One of the challenges is that traffic data statistics are constantly changing, and their aggregation for real-time monitoring becomes a difficult task. This paper presents a near real-time TM estimation approach for OpenFlow (OF) networks. It makes use of Big Data techniques based on MapReduce operations to tackle the aggregation problem. The proposed method uses traffic data statistics collected from OF switches through an SDN controller as input and aggregates these data in a Big Data streaming processing environment. This paper explores the benefits of the distributed MapReduce computing model to provide an estimate of the TM for all origin-destination (OD) pairs of hosts in the network in two ways: 1) the accumulated throughput and 2) the throughput between two sequential TM estimates. This procedure enables network engineers to monitor the behavior and evolution of the throughput on each OD pair in the network and on each link in the path between each OD pair. The generated TM is persisted in a NoSQL database and can be made available for a variety of network traffic monitoring applications. The results of the simulations show the potential of the proposed MapReduce approach for TM estimation.

INDEX TERMS Traffic matrix estimation, big data, MapReduce, SDN.

I. INTRODUCTION

A traffic matrix (TM) provides the traffic throughput between any OD pairs in a network over a specific time interval [1], [2]. This information is essential for several activities in traffic engineering [42] (TE) such as switch load balancing, traffic analysis, and fault tolerance analysis. For example, if a link fails, a new route between the OD pair affected by this failure must be selected among the available routes that connect these nodes. To choose the best route, the volume of traffic in each possible route must be available for an optimal decision.

Due to a large number of combinations of OD pairs present in large networks, accurate measurement of the TM becomes a hard task. One of the significant challenges is the number of links between any OD pairs. As the number of links increases, providing an accurate measurement of the TM becomes even harder. The traffic on each link in the path must be retrieved to aggregate the traffic data statistics between nodes A and B at a specific time. Collecting traffic information on all links at the same time is extremely difficult because of the lack of measurement infrastructure [1], [4]. For IP networks, some previous works have inferred the TM indirectly by observing SNMP link loads [5], [6]; another approach estimated TM using sample flow statistics with Cisco NetFlow [7], [8], and Zhao et al. used a combination of both [1].

One of the features introduced by the OF protocol is the use of counters [9], [10]. Counters are maintained for a variety of objects in an OF switch. The introduction of counters enables a more direct approach to collect traffic data statistics. There are basically two approaches to collecting counter values: (1) indirectly using the SDN controller and (2) directly sending specific messages to the switches. The more common approach is to use an SDN controller, which usually, in addition to collecting counter values, provides a wide range of configuration and service request APIs. Using the appropriate APIs, an application can retrieve counter values of any switch connected to the SDN controller and estimate the TM.
However, even with the introduction of counters by the OF protocol, two challenges for TM estimation still remain: i) which approach to use to collect the traffic data statistics, namely, direct measurement or inference techniques, and ii) how to aggregate the traffic data statistics to produce the estimated TM.

Direct measurement can lead to small errors in the estimated TM [7] because this technique aims to collect traffic data statistics for all devices and all links between OD pairs in the network; however, as previously stated, this technique requires a measurement infrastructure. Inference techniques collect traffic samples and use specific methods to estimate the TM based on these samples. Both techniques provide a set of values that can be used to estimate the TM.

A common problem faced by previous TM estimation methods is related to aggregating the collected traffic data statistics [4], [28], [29], [37]. For real-time measurement, depending on the amount of collected data, calculating aggregated link, switch, and route traffic throughput may be time-consuming, making methods unable to meet real-time constraints. The aggregation process starts by storing the collected data, which arrive in a streaming fashion while previous data are still being processed. The stored data are then grouped by a pre-determined key as they arrive in random order. The desired outputs, such as link traffic throughput, are computed according to the grouped keys. Each one of the previous aggregation process steps must be addressed by any TM estimation approach.

To solve the issue of aggregating the collected traffic data statistics in near real-time, this paper proposes an approach to TM estimation that uses a Big Data programming model called MapReduce [11]–[14], [19]. MapReduce is a highly scalable programming paradigm capable of processing massive volumes of data through parallel execution. Even though this approach is considered generic enough to work with inference techniques, the proposed solution is focussed on direct measurements. The counter values for the switch ports collected from the SDN [16], [17] controller are processed by the MapReduce approach to generate the estimated TM for all the OD pairs in the network every t seconds. The challenge posed by direct measurement is the number of counter values to be processed. The counter values are initially mapped to key-value pairs, according to the OD pairs of hosts, to be later reduced to generate the TM. MapReduce not only provides straightforward primitives to generate the key-value pairs but also offers a highly scalable processing environment capable of reducing the grouped keys in near real-time. These MapReduce features are crucial for the proposed approach.

The contribution of this paper is as follows: (1) The proposed solution for the TM estimation problem uses Big Data processing techniques and tools to address the issue of aggregating traffic data statistics, which to the best of the authors’ knowledge, is the first attempt to use this approach for SDN. In addition, the current research work defines the persistence of the estimated TM, which lays the foundation for further traffic analysis using processes similar to Big Data analytics. (2) The proposed approach can produce the estimated (a) accumulated TM between all OD pairs in the network in near real-time. It also provides and persists the traffic throughput in each active link in the network. (3) In addition to the traditional TM, the proposed approach also produces what is called here the (b) actual TM between all OD pairs. The actual TM consists of the difference between the previous and the current calculated throughput for each OD pair [15], revealing the momentaneous traffic change. The power of Big Data processing tools permitted our approach to generate both TMs: (a) accumulated, and (b) actual, in the same processing window. The generated TMs enable network managers to keep track of traffic evolution: i) between any OD pair, and ii) on each link that composes the path between any OD pair, in near real-time. The recurrent estimation of both TMs under a restricted time requirement is simply achievable by using a powerful programming model like MapReduce. This becomes more evident if a more detailed view of network resources usage is needed.

The motivation of the proposed approach is that direct measurement can be feasible if appropriate infrastructure is provided, not only for collecting the counters but also for aggregating the collected traffic data statistics.

The remainder of this paper is structured as follows: Section II presents related work concerning TM estimation. Section III describes the proposed MapReduce approach. Section IV shows the results of the simulations performed in this study. Finally, section V concludes this paper.

II. RELATED WORK

The TM plays an essential role in the field of network management and traffic engineering and has been extensively investigated over the years.

For traditional IP networks, several different approaches were used to estimate TM. Soule et al. [20] created a state space model to capture temporal and spatial correlations between pair of hosts and the applied Kalman Filters to estimate and predict TMs. Xie et al. [21] proposed a Sequential Tensor Completion algorithm (STC) to recover missing Internet traffic data and also proposed a Reshape-Align scheme [22] to reshape inconsistent traffic matrices and align these matrices to form a tensor. Polverini et al. [23] proposed \( mrT \), which is a MapReduce-based framework for estimating the TM of an Internet Service Provider (ISP) network.

The OF specification defines a number of flow counters that can be used to increase the accuracy of the TM estimate. Tootoonchian et al. [4] presented OpenTM, a traffic matrix estimator for OF networks. OpenTM was implemented as an application for the SDN controller to monitor active flows by polling the switches periodically. Tootoonchian et al. [4] also explored various algorithms for choosing which switch to query. The difference between OpenTM and the present work is that in this study, ports are monitored instead of flows. Moreover, the previous paper created flows between specific pairs of hosts and provided statistics for these flows.
The MapReduce approach proposed here can provide the statistics between any pair of hosts, even if no explicit flow has been created between them.

Gong et al. [28] proposed two strategies to design traffic measurement rules to be installed in the OF switches flow tables. They assume that rules for routing flows in each SDN are aggregated, whenever is possible. These aggregated routing rules can be used to estimate the TM. In their work the aggregated rules are disaggregated to improve the estimation accuracy. Li et al. [29] developed a method to determine which flows are most informative to construct a measurement flow set iteratively until an accuracy requirement is satisfied or a measurement resource constraint is reached. FlowMon [30] provided a sample and fetch based mechanism to detect large flows. The sample mechanism detects suspicious large flows and the fetch mechanism install measurements rules in the OF switches. OpenSample [32] leverages sFlow packet sampling to provide near real-time measurements of both network load and individual flows.

Luong et al. [34] proposed a solution for monitoring throughput of link traffic and a new forwarding algorithm for the control plane. It is based on two modules for the SDN controller. The Throughput Monitor module focuses on the number of packets and bytes that crossed the switch. The Packet Forwarding module implements the forwarding algorithm. The statistics provided by the Throughput Monitor module are available for use by any monitoring application. Volpato et al. [35] proposed OFQuality, a module for QoS configuration based on the OVSDB protocol, which benefits from TM generation to manage QoS policies on the OF switches.

Tian et al. [36] proposed a new framework for TM estimation in an SDN-based IP network. Their work shows that if a flow’s traffic can be derived from other flows in the network, adding a new entry for this flow for traffic measurement does not provide new information about the TM.

iSTAMP [37] used a well-compressed aggregated flow measurement and the K most informative flows to infer the TM. iStamp partitions the flow tables into two parts: the first for aggregate measurements, and the second for per-flow monitoring of selected flows. Using these two distinct parts, iStamp estimates the traffic matrix.

Jiang et al. [38] developed an algorithm to estimate and recover the network traffic matrix at fine time granularity from sampled traffic traces. Their algorithm is based on fractional interpolation, cubic spline interpolation, and the weighted geometric average algorithm.

Azzouni and Pujolle [39] presented NeuTM, a framework for TM prediction based on Long Short-Term Memory Recurrent Neural Networks. NeuTM trained a neural network using historical traffic data to predict the future TM.

Choudhury et al. [40] described an application of machine learning for TM prediction in SDN-Enabled IP/Optical networks. Their work used machine learning for short- and long-term prediction of all elements of the traffic matrix.

The previous studies mentioned above estimated the TM based on modelling and optimization procedures. For modelling, mathematical models were used to estimate the TM based on sample mechanisms for traffic data statistics. The proposed MapReduce approach uses direct measurement and, with the help of big data tools, tries to minimize the TM estimation error. The proposed approach monitors switches and ports traffic data volume. Using this method, all traffic that crosses a link between an OD pair is used to estimate the TM.

III. MapReduce APPROACH FOR TM ESTIMATION

The proposed MapReduce approach is part of a TE method based on Big Data techniques shown in Fig. 1 [41], [42]. This TE method delivers fine-grained statistical analysis of network resources, such as the ratio of each port in the switch payload, throughput capacity used on ports, and the TM estimation. Two modules compose the TE method: The Big Data Streaming module and the Traffic Engineering module. The traffic statistics data coming from an OF switch can be collected in a streaming fashion because of the fast changes in the counters due to dynamic traffic crossing the switches. The Big Data Streaming module collects network data and forwards them to be processed according to the data purpose (Flow Data, Topology Data, Controller Data, and Traffic Data Statistics).

The Traffic Engineering module performs four TE activities as shown in Fig. 1. 1) The Traffic Analysis/Characterization activity processes the data coming from the Big Data Streaming module and performs the Traffic Monitoring and Analysis tasks. 2) The Flow Management activity carries out load balance on switches and controllers. 3) The Fault Tolerance activity provides fast recovery in case of component failures, and 4) the Topology Update activity works on network topology update for planned changes. The Traffic Analysis/Characterization activity provides the data required by the other three TE activities (flow management, fault tolerance, and topology update) to make decisions that are applied by the Policy Dispatcher to the network.

Fig. 1 presents two methods: (a) Big Data Monitoring Method (red dashed line) to provide online measurements of traffic throughput and (b) Big Data Traffic Analysis Method (black dashed line) to process historical data to reveal traffic trends and behavior.

The proposed MapReduce approach can produce several throughput measurements including the TM.
estimation method. The focus of this paper is on TM estimation, using the online traffic data statistics provided by the Big Data Streaming module, for the monitoring activity.

The proposed approach for TM estimation is described in detail in the next subsections. Subsection A provides details of how to process the traffic data statistics related to switch ports that are needed to calculate the traffic throughput in every link formed in the network. It is then followed by the definition of the data structure required to store the network elements and topology data in Subsection B. Finally, the algorithms to estimate the TM are described in Subsection C.

A. MapReduce DESIGN

The TM estimation consists of aggregating the traffic throughput of all links in the path between an OD pair of hosts. This aggregation becomes harder when the number of possible combinations of OD pairs increases due to the number of hosts in the network. This scenario poses two challenges: 1) identify the links that connect all OD pairs, and 2) aggregate the values of all links to provide individual OD throughput estimation. For a static solution where the network topology is previously known, and the links are easily identified, the aggregation process is easier to implement. The MapReduce approach proposed in this paper, on the other hand, considers that the network topology is provided by the SDN controller and, from the textual description of the network topology, derives the links connecting all OD pairs. Once the links are identified, the aggregation process is easier to implement. The MapReduce approach proposed in this paper, on the other hand, considers that the network topology is provided by the SDN controller and, from the textual description of the network topology, derives the links connecting all OD pairs. Once the links are identified, a Map operation generates all the \((K, V)\) pairs, where the \(K\) contains the identification of the link and the OD pair that this link connects to be aggregated for each OD pair of hosts every \(t\) seconds, and \(V\) is the link calculated throughput. Hence, all values of \(V\), with the same \(K\), will be added to generate the estimated throughput for each OD pair of hosts.

The design described in this subsection provides the functions needed to estimate the TM. The initial requirement to estimate the TM is to collect traffic data statistics related to switch ports. As shown in Fig. 1, the collection process is carried out by the Big Data Streaming module, and this process is detailed by Queiroz et al. [41]. For the context of this paper, it is considered that the Big Data Streaming module provides the traffic data statistics as messages, which are used as input for the proposed MapReduce Approach. These messages are referred to as Raw Data because they deliver the number of bytes that crossed a switch port at a specific moment. These data are used to calculate the traffic throughput in the switch port and need to be organized to reflect links throughput (two ports of distinct switches compose a link).

The next requirement for the TM estimation is to identify the links that connect an OD pair and aggregate the traffic throughput of these links. The MapReduce approach initially calculates the throughput for each port and identifies the links formed by each pair of ports. Next, the links for each OD pair are aggregated using a unique key that represents that OD pair. The last requirement to estimate the TM is to sum all the calculated throughput with the same key to generate the estimated TM; the Reduce function performs this summation. The MapReduce approach for the TM estimation consists of four map functions, and one reduce function as shown in Fig. 2.

The Map Raw Data function receives raw data (traffic data statistics) as messages and generates a key/value map where the timestamp of each message is the key; and the data needed to calculate the port traffic throughput are the value.

Next, the Sort By Key function sorts the keys (timestamp) generated by the Map Raw Data function in ascending order. This step is necessary to calculate the current link throughput. The current link throughput is used to calculate the current throughput between OD pairs.

The maps generated by the map functions are shuffled before they are processed by the next function. Shuffling is a default operation performed by MapReduce tools to enable distributed processing. However, the map generated by the Sort By Key function needs to be processed in the same order that it was generated, i.e., in ascending order by timestamp because the current link throughput is the difference between the previous and current throughput. To avoid the shuffle in the map produced by the Sort By Key function, the function Coalesce is executed.

The Generate Final Map function generates a map that provides the current and accumulated link throughput that is aggregated to estimate the TM. This map is the last requirement to estimate the TM.

The Reduce by Key function processes the final map to estimate and persist the TM.

The MapReduce paradigm enables the approach for the TM estimation to be focused on the solution and not on the implementation details. MapReduce abstracts the data structure to store the keys to be aggregated. Also, the ability to process the aggregation of the keys in parallel saves great effort in the design of a parallel environment to reduce the keys. These features enabled our approach to deliver not only one TM, but two TMs (accumulated and actual), which is an advance when compared with previous solutions. In addition, the approach proposed here did not select one specific OD pair to monitor, but all OD pairs in the network topology.
B. DATA STRUCTURES TO ESTIMATE TM

This subsection provides the second requirement to estimate the TM, i.e., the data structures to store hosts, switches, switch port, and links data that will be used to generate the keys. All these data need to be identified and structured to reflect the network topology. This is necessary to identify all the links that are in the path between two OD pairs. This section introduces the basic definitions of all these data along with their structures.

The definitions are as follows:

- **Set** \( N = \{x_1, x_2, \ldots, x_n\} \). This set represents the network nodes.
- **Set** \( H = \{h_1, h_2, \ldots, h_m | h_i \in N\} \). This set represents all hosts in the network.
- **Set** \( S = \{s_1, s_2, \ldots, s_k | s_j \in N\} \). This set represents all switches in the network.
- **Set** \( U = \{ (h_i, s_j) | s_j \in S \} \). This set represents all switch/switch connections.
- **Set** \( U_1 = \{ (h_1, s_j) | s_j \in S \} \). This set represents host/switch connections.
- **Set** \( U_2 = \{ (s_i, s_j) | s_j \in S \} \). This set represents switch/switch connections.

The network is represented by a graph \( G \) given by \( (N, U) \).

\[ L_1 = \{ (h_i, s_j, s_k) | s_k \in S \} \]

This set represents the links between switch and ports.

\[ L_2 = \{ (s_i, s_j, s_k) | i \not\in j \} \]

This set represents the links between two switch ports.

\[ L = L_1 \cup L_2 \]

\[ HC = \{ (h_i, h_j) | h_i, h_j \in H \} \]

This set represents the combinations of OD pairs.

A Path \( P \) from \( h_i \) to \( h_j \) is a sequence \( (h_i, s_1, \ldots, s_k, s_j) \) such that each switch in the sequence \( s_1, s_2, \ldots, s_k \) is distinct.

\[ LP = \{ (s_i, (h_1, h_2), \ldots, (h_{n-1}, h_n)) | i = 1 \ldots n, j = 1 \ldots l \} \]

This set represents all paths that include the switch port \( s_j \).

To generate the TM, some basic prerequisites are required:

1. The set \( H \) (hosts), \( S \) (switches), and the graph \( G \) representing the network are given.
2. The set \( HC \) (OD pairs) is built using a function \( f : H \to HC \). The cardinality of \( HC \) is given by the combination of all elements of \( H \) (n) taken from \( r \), as shown in Equation 1:

\[
\text{Cand_HC} = \binom{n}{r} = \frac{n!}{(n-r)! \cdot r!}
\] (1)

Because each element in \( HC \) is a binary relation between members of \( H \), the value for \( r \) is 2.

3. The set \( L \) (links between host and switch ports) and links between two switch ports) is given.

The \( LP \) set provides, for each switch port, the list of OD pairs for which this port is part of the path. To build this set, the first step is to build the \( HC \) set, which contains all the possible combinations of OD pairs. For each pair in \( HC \), the path \( P \) is retrieved. Each path between the members of an OD pair consists of three types of links: (1) a link between the origin host and the first switch, expressed as \( (h_i, s_1) \), (2) links between switches, expressed as \( (s_i, s_j) \), and (3) a link between the last switch and the destination host, expressed as \( (s_k, h_j) \). For each type of pair, the set \( L \) is searched to find the ports used in the link. For link types (1) and (3), a match is sought for elements in \( L_1 \) (links between hosts and switch ports). Elements in \( L_1 \) are described as \( (h_i, s_j, s_k) \) or \( (s_j, h_i, s_k) \). When a match is found, the switch port \( (s_j) \) is returned. For link type (2), the same procedure is followed, with elements in \( L_2 \) (links between two switch ports) searched for matching. In this case, when a match \( (s_i, s_j) \) is found, both ports \( (s_i, s_k) \) are returned. For each returned port, a pair \( (\langle port\rangle, \langle O, D\rangle) \) is added or updated in \( LP \).

C. ALGORITHMS TO ESTIMATE TM

The \( LP \) set is the primary data structure in the proposed MapReduce approach because it provides, for each link, all OD pairs that the link connects.

The data provided by the Big Data Streaming module (Fig. 1) represent the raw data on the switch ports. This section presents the generation of the \( LP \) set and the procedure that combines the collected raw data with the \( LP \) set elements to generate the final map to estimate the TM.

The Map Raw Data function (Fig. 2) receives traffic data statistics as messages; these messages have the layout provided in Fig. 3. The timestamp field is in the Unix format, where the last three digits refer to the millisecond of the data collection time. The approach used here discards the last three digits because it provides the throughput in seconds. This operation changes the timestamp 1522598210107 to 1522598210 and makes it possible to group statistics collected in the same second. This grouping operation enables the TM estimation to take place at every \( t \) seconds.

The map generated by the Map Raw Data is sorted by the Sort By Key function. The sorted key is composed of the timestamp and switch port fields. This sort function enables the estimation of the current throughput, which is the difference between the previous and current calculated throughput.

The Generate Final Map function provides two algorithms that are needed to generate the final map. The goal of these algorithms is to generate a map (final map) holding the throughput information in every link based on the topology and the raw data collected from the network. The Generate LP set algorithm (Algorithm 1) generates the \( LP \) set, and the Generate (Key, Value) pairs algorithm (Algorithm 2), for every port statistics message, calculates the throughput of the link that include this port and generates the (Key, Value) for all the OD pairs that include the port. The following sections describe these algorithms.
1) GENERATE LP SET ALGORITHM

Algorithm 1 shows the steps to generate the LP set. The LP set includes, for each switch port, the list of OD pairs for which this port is part of the path.

Topology information is collected from the OpenDaylight SDN controller to build the sets H (hosts), S (switches), U (host/switch connections and switch/switch connections), and L (links between hosts and switch ports and links between two switch ports). OpenDaylight’s response comes in a JSON format and is converted to an internal CSV format. Fig. 4 shows the topology used in the experiments.

The elements of L are directly related to the elements of U because they provide a critical piece of information, the port number in each switch used in a connection. Therefore, for each element of U1 (hi, sj), the switch port (sji) is added to form the elements of L1. The host port is not used in L1 because it is assumed that only one network interface is available in the host.

The elements of L2 are generated in the same way; for each element of U2 (sj, hi), the ports in the connection (sji, sji) are then identified.

The graph G is built using GraphStream [43], which is a dynamic graph library in Java. The approach used here extracts the nodes (hosts (H), switches (S), and links (U)) and feeds the graph with these values.

Lines 8–13 from Algorithm 1 generate the HC (OD pairs) set. For each OD pair (hi, hj) in HC, the path connecting the two hosts is returned by the function getPath(hi, hj) (Lines 15–16). In the topology presented in Fig. 4, if an attempt is made to find the paths between any two hosts, loops can be found in the network graph. To overcome this issue, the topology was divided into two parts: the first with S1 as the root, and the second with S2 as the root. With this approach, the solution presented here provides the TM between all possible combinations of OD pairs traversing S1 or S2.

The getPath((β0, β1)): linkports[2] function (lines 41–52) returns the (β0, β1) link ports. If the (β0, β1) link is an element of the L1, the function returns only the switch port (lines 43–45). If the (β0, β1) link is an element of the L2, the function returns both switch ports (lines 46–49). This function searches the set L to provide these data. Using them, the elements of LP can be generated (lines 17–34). For each value returned from the getPath() function, an entry in LP is added. At the end of this procedure, each switch port LP will have the OD pairs that include the switch port.

As previously mentioned, the approach used here divides the network topology into two parts to avoid loops in the network graph. Loops in the network are not the only problem faced when traversing a network graph. Another problem is to find the shortest path between an OD pair. Even though the Dijkstra algorithm solves this problem, it is still possible to find more than one shortest path between an OD pair. In this case, one of the shortest paths can be selected for monitoring. The MapReduce approach proposed here enables the monitoring of more than one shortest path. The solution is to add a different identifier to the generated OD pair definition.
Algorithm 2 Generate \((key, value)\) pairs

Input: Ports Statistics Messages, \(LP, L\)
Output: \((key, value)\)

1. global: \(path\_text{\_list} \// \text{List of all paths}
2. locals: \(previous\_Stat \// \text{Previous Statistics}
3. \(throughput \// \text{Accumulated throughput}
4. \(curr\_Throughput \// \text{Current throughput}
5. \(pair \// \text{pair of the current port}
6. interval = 3 \// \text{interval between readings}
7. foreach message do
   8. \(pair = getPairStat\_List(message, switch\_port)
   9. previous\_Stat = getLinkStat(message, switch\_port, pair)
   10. if ((message.BT + message.BR) > (pair.BT + pair.BR)) then
      11. \(throughput = (message.BT + message.BR) / previous\_Stat / interval
      12. \(update\_linkStat(message, switch\_port, pair, (message.BT + message.BR))
      13. \(path\_list = LP, get\_List\_Paths(message, switch\_port)
      14. foreach element pl in path\_list do
         15. \(key = concatenate(pl, message, timestamp)
         16. \(value = (throughput, curr\_Throughput)
         17. generate (key, value)
      18. end
   19. end
20. end

in \(LP\). For instance, in a network with two shortest paths from host \(Ha\) to host \(Hb\), where the first shortest path includes switch port \(SAp1\) and the second shortest path includes switch port \(SBp2\), the entries in \(LP\) for \(SAp1\) and \(SBp2\) can differentiate the two shortest paths by creating OD pair definitions. The presence of the strings \PATH1\ and \PATH2\ differentiates the OD pair definitions for the two shortest paths.

2) \text{GENERATE} \((key, value)\) \text{PAIRS ALGORITHM}

The goal of this algorithm is to generate the final map. This map contains the traffic throughput for every link in the network. The link is not part of the \text{key}; only the OD pairs that are connected through this link compose the \text{key}. The map consists of OD pairs as the \text{key} and the traffic throughput of the link as the \text{value}.

The pseudo-code provided in Algorithm 2 describes the \text{Generate} \((key, value)\) \text{pairs algorithm}. The \text{generated} \((key, value)\) \text{pairs compose the final map}, which is used by the \text{Reduce By Key} \text{function (Fig. 2)} to estimate the TM.

For each port traffic data statistics message (Fig. 3), the message of the port pair with the same timestamp is retrieved (Line 8). A port and its pair are an element of the set \(L\). If the port is an element of \(L\), the port and its pair have the same value because in \(L\) the link is formed between a host port and a switch port, but the host port is not monitored. For \(L2\) elements, \((s_i, s_{ia}, s_j, s_{jb})\), if the fields \text{second} and \text{nanosecond} are exactly the same for \((s_{ia}, s_{jb})\), the value of bytes received (BR) for \(s_{ia}\) should be the same as the value of bytes transmitted (BT) for \(s_{jb}\), and the other way around. However, collecting statistics for \((s_{ia}, s_{jb})\) at the same \text{second} and \text{nanosecond} is an unrealistic scenario. Hence, one of the messages has the most up-to-date statistics for the link.

To use the most up-to-date statistics, the message with the larger value of \((BR + BT)\) represents the throughput of the link (Line 10). Fig. 5 provides an example. The “S1:eth1” and “S3:eth3” ports form a link, and the statistics of the “S3:eth3” port are used as the throughput of the link in that timestamp.

For each message (Fig. 3), two calculations are performed:

1) \text{Current throughput in every link}. The current throughput is calculated according to Equation 2 [15]:

\[
T_i(t) = \frac{c_i(t) - c_i(t - T)}{T}
\]

where \(T_i(t)\) is the current throughput of link \(i\) at instant \(t\), \(c_i(t)\) is the statistics value \((BR + BT)\) at instant \(t\), and \(T\) is the polling interval (Line 11). Lines 9 and 13 respectively retrieve and update \(c_i(t - T)\) for each link.

2) \text{Accumulated throughput in every link}. The accumulated throughput is calculated according to Equation 3 (Line 12):

\[
T_i(t) = \frac{(BR + BT)}{S}
\]

where \(T_i(t)\) is the accumulated throughput of link \(i\) at instant \(t\) and \(S\) is the number of seconds for which the link is active.

The final step in Algorithm 2 is map generation. The \(LP\) set is queried to provide the paths that include the selected port (Line 14). For every returned path, a \((key, value)\) pair is generated (Lines 15–19). Table 1 provides an example of the generated \((key, value)\) pairs for the S3-eth3 port. The \text{key} is composed of the path and timestamp, and the \text{value} contains the calculated current \(C_v\), Equation 2) and accumulated \(A_v\), Equation 3) throughputs.

For all port traffic data statistics in the same timestamp, the same type of \((key, value)\) pairs as those generated for the S3-eth3 port and shown in Table 1 are generated.

The \text{Reduce By Key} \text{function aggregates \text{values} with the same \text{key}}. Table 2 shows the reduce operation applied to

\[\text{TABLE 1. Generated } (key, value) \text{ pairs.}\]

\begin{tabular}{|c|c|c|}
\hline
\text{Key} & \text{Current} & \text{Accumulated} \\
\hline
S1:/host:01/host:03 1522598234 & \(C_v\) & \(A_v\) \\
S2:/host:02/host:05 1522598234 & \(C_v\) & \(A_v\) \\
S2:/host:02/host:06 1522598234 & \(C_v\) & \(A_v\) \\
S1:/host:01/host:03 1522598234 & \(C_v\) & \(A_v\) \\
S2:/host:02/host:05 1522598234 & \(C_v\) & \(A_v\) \\
S2:/host:02/host:06 1522598234 & \(C_v\) & \(A_v\) \\
S2:/host:02/host:03 1522598234 & \(C_v\) & \(A_v\) \\
\hline
\end{tabular}
some of the \((key, value)\) pairs presented in Table 1. The TM consists of records composed of the fields provided by Table 2 \((Key, Current, Accumulated)\).

Figure 6 shows the TM computation procedure. Messages are processed in message sets, and each message set is composed of all the messages that arrive in the Queue Messages module during one MapReduce execution. During the processing of Batch 1, the messages that compose Batch 2 arrive in the Queue Messages module. At the end of Batch 1 processing, the computed TM is sent to storage. During the processing of Batch 2, the messages that compose Batch 3 are still arriving. This is a continuous process. The MapReduce module is scheduled to execute every \(t\) seconds.

### IV. EXPERIMENTS AND RESULTS

In this section, we present the real-time measurements provided by the MapReduce approach. The topology used in our experiments is the one presented in Fig. 4.

The experiment is robust enough to challenge the MapReduce paradigm in two different ways: i) the number of keys created for each OD pair of hosts, because of the number of links presented in the topology. This procedure also demands the Map operation to sort all the created keys to set up the TM with the current throughput. ii) network basic traffic statistics arrive in a streaming fashion, so, at every \(t\) seconds the two TMs (accumulated and actual) are generated by Reducing the created keys.

Four machines were used to run the experiments. They are described in Table 3.

Servers 1, 2, and 3 are connected to an HP MSA 2040 SAN 12 TB storage system. The storage is divided into four logical partitions of 2.5 TB each. The SDN monitoring application presented in Fig. 1 uses the following big data tools: Apache Kafka, Apache Spark Streaming, and Elasticsearch.

Each big data tool was configured in a cluster formed by four Linux containers running Linux Ubuntu Server 16.04. A feature called Multipath was configured in the servers enabling the containers to have multiple I/O paths to the storage, allowing parallel access to the four configured partitions.

Mininet was used to create the network topology and run the experiments. Using Iperf3, we created TCP flows according to the scheduling diagram shown in Fig. 7. This diagram shows the generated traffic between pairs of hosts and the respectively configured bandwidth. We also added some flow table entries to change the default route configuration provided by OpenDaylight. These flow entries define layer 3 routing, as shown in Table 4.

The collected traffic data statistics processed by Spark streaming were persisted in Elasticsearch every 3 seconds. Even though all calculated throughput were provided in real-time, the graphics were generated off-line. The graphics provided in this section show the throughput for selected links and OD pairs, but the MapReduce approach provides the throughput for all links and OD pairs in our topology. The scheduling diagram shown in Fig. 7 displays the bandwidth

#### TABLE 2. Reduce by key applied to \((key, value)\) pairs.

| Key          | Value          |
|--------------|----------------|
| S1/host:01   | Current \(C_u\) | Accumulated \(A_u\) |
| S1/host:02   | Current \(C_u\) | Accumulated \(A_u\) |
| S2/host:02   | Current \(C_u\) | Accumulated \(A_u\) |
| S2/host:03   | Current \(C_u\) | Accumulated \(A_u\) |

#### TABLE 3. Hardware environment.

| Id | CPU       | RAM  |
|----|-----------|------|
| Server 1 | 2 x Intel(R) Xeon(R) CPU B5-2630 v2 @ 2.6GHz | 96 GB |
| Server 2 | 2 x Intel(R) Xeon(R) CPU B5-2630 v2 @ 2.6GHz | 96 GB |
| Server 3 | 2 x Intel(R) Xeon(R) CPU B5-2630 v2 @ 2.6GHz | 96 GB |
| Server 4 | Intel(R) Core(TM) i7-4710MQ CPU @ 2.5GHz | 16 GB |

#### TABLE 4. Layer 3 routes.

| Switch | Source IP | Destination IP | Action     |
|--------|-----------|----------------|------------|
| S1     | 10.0.0.1  | 10.0.0.5       | output:eth3|
|        | 10.0.0.2  | 10.0.0.8       | output:eth4|
|        | 10.0.0.3  | 10.0.0.7       | output:eth4|
| S2     | 10.0.0.1  | 10.0.0.7       | output:eth3|
|        | 10.0.0.2  | 10.0.0.6       | output:eth3|
|        | 10.0.0.4  | 10.0.0.8       | output:eth4|
| S3     | 10.0.0.1  | 10.0.0.7       | output:eth4|
|        | 10.0.0.2  | 10.0.0.6       | output:eth4|
|        | 10.0.0.2  | 10.0.0.8       | output:eth3|
| S4     | 10.0.0.3  | 10.0.0.7       | output:eth3|
|        | 10.0.0.4  | 10.0.0.8       | output:eth4|
| S5     | *         | 10.0.0.5       | output:eth1|
| S6     | *         | 10.0.0.6       | output:eth2|

#### FIGURE 7. Scheduling diagram.
selected to every generated traffic. The bandwidth value is expressed in Mbits per second.

To validate the TM, packets statistics were captured on the switch ports using tcpreplay. Using tshark and a python script, the throughput at every 3 seconds for each link was calculated, and then plotted along with the calculated throughput provided by the MapReduce. This plot is intended to show that the MapReduce calculated throughputs follow the same pattern as the one calculated using tshark. With tshark we calculated the statistics at every 3 seconds and the counters values provided by OF standard includes seconds and nanoseconds, which did not allow us to match the values exactly at the nanosecond level. Thus, it is not expected the values calculated using tshark and the ones provided by MapReduce to be the same.

The MapReduce approach generates the current and accumulated throughput between all OD pairs in the topology. As these values are calculated using the individual values on each link, we provide the results not only for OD pairs but also for individual links.

A. CURRENT LINK THROUGHPUT
The current throughput of a link is calculated according to Equation 2. Fig. 8 shows the current estimated throughput at every \( t - T \) time interval in the links formed by S2:eth4 and S6:eth4, S1:eth1 and S3:eth3, and S2:eth3 and S5:eth4. The link S2:eth4/S6:eth4 is used by the traffic from h1 to h7 and from h4 to h8. S1:eth1/S3:eth3 carries the traffic from h1 to h5 (at 0 and 25s) and from h2 to h8. The traffic from h2 to h6 at instant 10s drives the throughput in S2:eth3/S5:eth4.

B. LINKS ACCUMULATED THROUGHPUT
The accumulated throughput in a link is calculated according to equation 3. Fig. 9 shows the estimated throughput in the links formed by S2:eth4 and S6:eth4, S1:eth1 and S3:eth3, S2:eth3 and S5:eth4, and S1:eth2 and S4:eth3. The throughput in link S2:eth4/S6:eth4 increases during the traffic from h1 to h7, experiences a small decrease, and then stops decreasing because of the traffic between h4 to h8. The throughput behaviour observed in link S1:eth1/S3:eth3 is driven by the traffic between h1 and h5 and between h2 and h8. The traffic from h2 to h6 is the only instance that uses the link S2:eth3/S5:eth4. Link S1:eth2/S4:eth3 is used by the traffic from h3 to h7.

C. HOST-TO-HOST CURRENT THROUGHPUT
As shown in Table 2, the current throughput of an OD pair is an aggregation of the values of individual links between hosts. Fig. 10 shows the estimated current throughput between h3 and h8 (root S2). The first four top graphics show the throughput on each link that connects the hosts, and the graphic on the bottom shows the aggregation of the individual links. The links between h3 and h8 are: S4:eth1, S2:eth2/S4:eth4, S2:eth4/S6:eth4, and S6:eth2. The only traffic flows in S4:eth1 is that from h3 to h7, which starts at 25s. The S2:eth2/S4:eth4 link is used by traffic from h4 to h8, which starts at instant 50s. The traffic from h1 to h7 at instant 10s and from h4 to h8 at instant 50s drives the S2:eth4/S6:eth4 link throughput. The S6:eth2 link carries traffic from h2 to h8 at instant 45s and from h4 to h8 at instant 50s.

D. HOST-TO-HOST ACCUMULATED THROUGHPUT
The OD pair accumulated throughput is also an aggregation of the values of individual links between the hosts. The MapReduce approach provides an estimate of the throughput.
between all possible combinations of OD pairs traversing S1 or S2 according to Table 5.

Fig. 11 shows the estimated throughput between the following pairs of hosts: h1–h5 (root S1), h3–h8 (root S2), and h2–h4 (root S2). Table 5 provides the rules for verifying which ports are used by each OD pair. For instance, the following links connect h3 to h8 using S2 as a root: S4:eth1, S4:eth4/S2:eth2, s2:eth4/S6:eth4, and S6:eth2. The OD pair h1–h5 has two specific traffic states defined at instants 0 and 25s. No traffic occurred between h3 and h8, meaning that the links that connect them were used to estimate throughput. In this case, the following traffic flows were used in throughput estimation: h1–h7, h3–h7, and h4–h8. The same situation applied to the OD pair h2–h4. In this case, the traffic between h1–h7, h2–h6, and h4–h8 generated the throughput for this OD pair.

E. PERFORMANCE EVALUATION

The proposed MapReduce approach was executed inside an Apache Spark Streaming environment. A performance evaluation of the execution of the proposed approach based on several metrics produced by the Spark Streaming monitoring system is now presented.

Fig. 12 shows the record input rate statistics. The average value of 7.62 records per second is shown on the left side of the figure. The graph in the centre shows the evolution of the input rate over the simulation period. The first observed spike from left to right is related to the time that Spark takes to set up the environment, which causes an accumulation of records. Subsequent spikes are produced by the records arriving every 3 seconds. Spark streaming processes the input records in batches, and the histogram on the right side shows the distribution of batches/records per second. Scheduling delay is the time between the instant that a collection of jobs for a batch was submitted and the instant that the first job was started. The scheduling delay for the present simulation is shown in Fig. 13. According to the graph in the centre, the first value for the scheduling delay was over 2 seconds because of the environment set-up, but the following batches had values near zero, making the overall average 173 ms. The histogram graph on the right shows that most of the batches had nearly zero scheduling delay time.

Fig. 14 shows the processing time for each batch of data. The average processing time in the present simulation was 2.5 seconds. According to the graph in the centre and the histogram in Fig. 14, most batches finished their execution in less than the defined time to process each batch (3.5 seconds, as shown by the dotted line labelled stable). The reason why this time is not the same for all batches is that the record input rate (Fig. 12) is not the same. The record input rate is related to the Apache Kafka record processing time and affects the time that all MapReduce functions (Fig. 2) take to run in a single batch. The time needed to persist each set of calculated statistics is already included in the batch processing time. According to Spark Streaming documentation, the total delay parameter shows whether or not the system is stable. The system is considered stable if it can keep up with its record input rate. If the delay is maintained to be comparable to the batch size, the system is considered stable, which is true in the approach proposed here as shown in Fig. 15.

The results provided by the metrics used to evaluate the Spark Streaming performance show a stable running environment capable of seamlessly estimating the TM. Any online traffic monitoring system requires this stability because of the
relevance of the TM for activities such as traffic prediction, capacity planning, and anomaly detection. Also, the results endorse the use of Big Data techniques to aggregate network traffic data statistics.

V. CONCLUSION

Estimating the TM in real time is a challenging task. Two common approaches to tackle this problem are direct measurement and inference techniques. This paper proposes a solution to this problem based on the MapReduce Big Data programming paradigm. The importance of the proposed approach is the use of direct measurement, which poses several challenges. One of the main challenges is to aggregate the large amount of collected data between all OD pairs and to produce the TM in real-time. The proposed approach uses the data distribution and parallel processing features of MapReduce to estimate not only one, but two TMs (accumulated and actual) in near real-time. This result shows the potential of Big Data processing techniques to estimate the TM using direct measurement of network traffic.

The MapReduce paradigm speeds up TM generation because it can aggregate keys in parallel. In the next steps of this research, the authors will examine the scalability of the proposed approach as the numbers of hosts and switches in the network topology are increased because increasing the number of hosts and switches will make collecting traffic data statistics even harder. In addition, in the context of real-time TM estimation by direct measurement, the use of different tools of Big Data streaming processing tools that implement the MapReduce paradigm will be evaluated.

REFERENCES

[1] Q. Zhao, Z. Ge, J. Wang, and J. Xu, “Robust traffic matrix estimation with imperfect information,” ACM SIGMETRICS Perform. Eval. Rev., vol. 34, no. 1, pp. 133–144, Jun. 2006.

[2] J. Chen, M. Chen, X. Wei, and B. Chen, “Matrix differential decomposition-based anomaly detection and localization in NFV networks,” IEEE Access, vol. 7, pp. 29320–29331, 2019.

[3] Z. Shu, J. Wan, J. Lin, S. Wang, D. Li, S. Rho, and C. Yang, “Traffic engineering in software-defined networking: Measurement and management,” IEEE Access, vol. 4, pp. 3246–3256, 2016.

[4] A. Tootoonchian, M. Gobhadi, and Y. Ganjali, OpenTM: Traffic Matrix Estimator for OpenFlow Networks. Cham, Switzerland: Springer, 2010, pp. 201–210.

[5] J. Cao, D. Davis, S. Vander Wiel, and B. Yu, “Time-varying network tomography: Router link data,” J. Amer. Stat. Assoc., vol. 95, no. 452, pp. 1063–1075, Dec. 2000.

[6] Y. Zhang, M. Roughan, C. Lund, and D. Donoho, “An information-theoretic approach to traffic matrix estimation,” in Proc. Conf. Appl., Technol., Archit., Protocols Comput. Commun., 2003, pp. 301–312.

[7] K. Papagiannaki, N. Taft, and A. Lakhina, “A distributed approach to measure IP traffic matrices,” in Proc. 4th ACM SIGCOMM Conf. Internet Meas., 2004, pp. 161–174.

[8] Q. Zhao, A. Kumar, J. Wang, and J. Xu, “Data streaming algorithms for accurate and efficient measurement of traffic and flow matrices,” in Proc. ACM SIGMETRICS Int. Conf. Meas. Model. Comput. Syst., 2005, pp. 1–14.

[9] (2014). OpenFlow Switch Specification 1.5.0. [Online]. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-switch-v1.5.0.noip.pdf

[10] S. Qazi, S. M. Atif, and M. B. Kadri, “A novel compressed sensing technique for traffic matrix estimation of software defined cloud networks,” IS&T Trans. Internet Inf. Syst., vol. 12, no. 10, pp. 4678–4702, 2018.

[11] K. Grof, M. Hayes, W. A. Higashino, A. L’Heureux, D. S. Allison, and M. A. M. Capretz, “Challenges for MapReduce in Big Data,” in Proc. IEEE World Conges Services, Jun. 2014, pp. 182–189.

[12] J. Lin, “In defense of map reduce,” IEEE Internet Comput., vol. 21, no. 4, pp. 985–994, May 2017.

[13] A. Shu, F. Liu, and D. Ye, “Rapid screening of big data against inadvertent leaks,” in Big Data Concepts, Theories, Application. Cham, Switzerland: Springer, 2016.

[14] A. Khaleel and H. Al-Raweshidy, “Optimization of computing and networking resources of a Hadoop cluster based on software defined network,” IEEE Access, vol. 6, pp. 61351–61365, 2018.

[15] P. Megyesi, A. Botta, G. Aceto, A. Pescapé, and S. Molnár, “Challenges and solution for measuring available bandwidth in software defined networks,” Computer Communication, vol. 99. Amsterdam, The Netherlands: Elsevier, Feb. 2017, pp. 48–61.

[16] Q. Zhang, X. Wang, M. Huang, K. Li, and S. K.Das, “Software defined networking meets information centric networking: A survey,” IEEE Access, vol. 6, pp. 39547–39563, 2018.

[17] J. H. Cox, J. Chung, S. Donovan, J. Ivey, R. J. Clark, G. Riley, and H. L. Owen, “Advancing software-defined networks: A survey,” IEEE Access, vol. 5, pp. 25487–25526, 2017.

[18] H. Farhady, H. Lee, and A. Nakao, “Software-defined networking: A survey,” Comput. Netw., vol. 81, pp. 79–95, Apr. 2015.

[19] A. Leureux, K. Grof, H. F. Elamyani, and M. A. M. Capretz, “Machine learning with big data: Challenges and approaches,” IEEE Access, vol. 5, pp. 7776–7797, 2017.

[20] A. Soule, K. Salamatian, A. Nucci, and N. Taft, “Traffic matrix tracking using Kalman filters,” ACM SIGMETRICS Perform. Eval. Rev., vol. 33, no. 3, pp. 24–31, Dec. 2005.

[21] K. Xie, L. Wang, X. Wang, G. Xie, J. Wen, and G. Zhang, “Accurate recovery of Internet traffic data: A tensor completion approach,” in Proc. 35th Annu. IEEE Int. Conf. Comput. Commun., Apr. 2016, pp. 1–9.

[22] K. Xie, C. Peng, X. Wang, G. Xie, and J. Wen, “Accurate recovery of Internet traffic data under dynamic measurements,” in Proc. IEEE Int. Conf. Comput., May 2017, pp. 1–9.

[23] P. Molverini, A. Cianfrani, A. Baoicchi, M. Listanti, and V. Salvatore, “From raw data packets to ingress egress traffic matrix: The distributed MapReduce-based solution,” in Proc. IEEE IETF IPFIP Oper. Manage. Smp., Apr. 2018, pp. 1–6.

[24] Y. Guo, Z. Wang, X. Yin, X. Shi, and J. Wu, “Traffic engineering in hybrid SDN networks with multiple traffic matrices,” Comput. Netw., vol. 126, pp. 187–199, Oct. 2017.

[25] Y. Guo, Z. Wang, X. Yin, X. Shi, and J. Wu, “Traffic engineering in SDN/OSPf hybrid network,” in Proc. IEEE 22nd Int. Conf. Netw. Protocols, Oct. 2014, pp. 563–568.

[26] J. He and W. Song, “Achieving near-optimal traffic engineering in hybrid software defined networks,” in Proc. IFIP Netw. Conf., May 2015, pp. 1–9.

[27] J. Zhang, K. Xi, M. Luo, and H. J. Chao, “Load balancing for multiple traffic matrices using SDN hybrid routing,” in Proc. IEEE 15th Int. Conf. High Perform. Switching Routing (HPSR), Jul. 2014, pp. 44–49.

[28] Y. Gong, X. Wang, M. Malboubi, S. Wang, S. Xu, and C.-N. Chua, “Towards accurate online traffic matrix estimation in software-defined networks,” in Proc. 1st ACM SIGCOMM Symp. Softw. Defined Netw. Res., 2015, pp. 1–7.

[29] D. Li, C. Xing, N. Dai, F. Dai, and G. Zhang, “Estimating SDN traffic matrix based on online adaptive information gain maximization method,” Peer Netw. Appl., pp. 1–16, Mar. 2018.

[30] C. Xing, K. Ding, C. Hu, and M. Chen, “Sample and fetch-based large flow detection mechanism in software defined networks,” IEEE Commun. Lett., vol. 20, no. 9, pp. 1764–1767, Sep. 2016.

[31] M. Malboubi, L. Wang, C.-N. Chua, and P. Sharma, “Intelligent SDN based traffic (de) aggregation and measurement paradigm (iSTAMP),” in Proc. IEEE Conf. Comput. Commun., Apr. 2014, pp. 934–942.

[32] J. Suh, T. T. Kwon, C. Dixon, W. Felter, and J. Carter, “Open sample: A low-latency, sampling-based measurement platform for commodity SDN,” in Proc. IEEE 34th Int. Conf. Distrib. Comput. Syst., Jun. 2014, pp. 228–237.

[33] M. Yu, L. Jose, and R. Miao, “Software defined traffic measurement with opensketch,” in Proc. 10th USENIX Symp. Networked Syst. Des. Implement., 2013, pp. 29–42.

[34] D.-H. Luong, A. Outtagarts, and A. Hebbä, “Traffic monitoring in soft-ware defined networks using openskydlight controller,” in Mobile, Secure, and Programmable Networking (Lecture Notes in Computer Science), vol. 10026. Cham, Switzerland: Springer, 2016, pp. 38–48.
[35] M. P. Da Silva, M. A. R. Dantas, and F. Volpato, “OFQuality: A quality of service management module for software-defined networking,” Int. J. Grid Utility Comput., vol. 10, no. 2, p. 187, 2019.

[36] Y. Tian, W. Chen, and C.-T. Lea, “An SDN-based traffic matrix estimation framework,” IEEE Trans. Netw. Service Manage., vol. 15, no. 4, pp. 1435–1445, Dec. 2018.

[37] M. Malboubi, S.-M. Peng, P. Sharma, and C.-N. Chuah, “A learning-based measurement framework for traffic matrix inference in software defined networks,” Comput. Electr. Eng., vol. 66, pp. 369–387, Feb. 2018.

[38] D. Jiang, L. Huo, and Y. Li, “Fine-granularity inference and estimations to network traffic for SDN,” PLoS ONE, vol. 13, no. 5, May 2018, Art. no. e0194302.

[39] A. Azzouni and G. Pujolle, “NeuTM: A neural network-based framework for traffic matrix prediction in SDN,” in Proc. IEEE/IFIP Netw. Oper. Manage. Symp., Apr. 2018, pp. 1–5.

[40] G. Choudhury, D. Lynch, G. Thakur, and S. Tse, “Two use cases of machine learning for SDN-enabled IP/Optical networks: Traffic matrix prediction and optical path performance prediction [Invited],” J. Opt. Commun. Netw., vol. 10, no. 10, p. DS2, 2018.

[41] W. Queiroz, M. A. M. Capretz, and M. Dantas, “An approach for SDN traffic monitoring based on big data techniques,” J. Netw. Comput. Appl., vol. 131, pp. 28–39, Apr. 2019.

[42] W. Queiroz, “Big Data for Traffic Engineering in Software-Defined Networks,” Ph.D. dissertation, Dept. Elect. Comput. Eng., Western Univ., London, U.K., 2019.

[43] GraphStream A Dynamic Graph Library. Accessed: Apr. 2019. [Online]. Available: http://graphstream-project.org

**WANDER J. QUEIROZ** is a Postdoctoral Associate with the Department of Electrical and Computer Engineering, Western University, London, ON, Canada. His research interests include software-defined networks, traffic engineering, and big data processing.

**MIRIAM A. M. CAPRETZ** (Senior Member, IEEE) received the B.Sc. and M.E.Sc. degrees from the UNICAMP, Brazil, and the Ph.D. degree from the University of Durham, U.K. She is a Professor with the Department of Electrical and Computer Engineering, Western University, Canada. Before joining Western University, she was with the University of Aizu, Japan. She has been working in the software engineering area for more than 35 years. She has been involved in the organization of workshops and symposia and has been serving on program committees of international conferences. Her current research interests include cloud computing, big data, machine learning, and service-oriented architecture privacy and security.

**MARIO A. R. DANTAS** received the master’s degree from the Technology Centre (CTC), Federal University of Santa Catarina (UFSC), through the Graduate Program in Computer Science (PPGCC), and the Ph.D. degree in computer science from the University of Southampton, U.K. He was a Visiting Professor with Western University, ON, Canada, and a Senior Visiting Researcher with the RIKEN, Japan. He is a Professor with the Department of Computer Science (DCC), Exact Sciences Institute (ICE), Federal University of Juiz de Fora (UFJF). He has authored over one hundred scientific articles, and dozens of book chapters and three books. He has advised numerous undergraduate, specialization, master’s, and doctoral research work. He has acted as a consultant on various projects with industry in the areas of the IoT, networking, distributed systems, and high-performance environments.

***