Decentralized Monitoring for Large-Scale Software-Defined Networks

Gioacchino Tangari, Daphne Tuncer, Marinos Charalambides, George Pavlou
Department of Electronic and Electrical Engineering, University College London, UK

***Abstract*—The Software-Defined Networking (SDN) paradigm can allow network management solutions to automatically and frequently reconfigure network resources. When developing SDN-based management architectures, it is of paramount importance to design a monitoring system that can provide frequent and consistent updates to heterogeneous management applications. For the monitoring functionality to scale according to the requirements of large-scale networks a distributed monitoring approach is required. In this paper we present a decentralized approach for resource monitoring in SDN, which is designed to support a wide range of measurement tasks and requirements in terms of monitoring rates and information granularity levels. Our solution leverages effective processing of the monitoring requests to reduce the consumption of limited resources, such as the control plane bandwidth of OpenFlow switches. To demonstrate the benefits of the proposed approach, our evaluation is based on a realistic and demanding use case, where a distributed management application coordinates a content distribution service in an ISP network.**

I. Introduction

Efficient resource monitoring is a fundamental requirement for any network management system. Accurate and timely updates are needed to support resource reconfigurations and to warrant precision when troubleshooting failures or detecting anomalies. Over the last few years current practices in network management have been challenged by the advent of Software Defined Networks (SDNs). SDN technologies have emerged as promising solutions to improve and simplify the operator's tasks [1] as they enable the development of applications that reconfigure the network automatically [3][4]. These advances pose new requirements on the monitoring functionality especially in terms of measurement frequency and information granularity.

Recent research on SDN monitoring investigated the implementation of task-specific measurements [12][13] and their adaptation with respect to traffic workloads and resource conditions [14][15]. However, these solutions rely on the assumption of a centrally-managed network, which is an important limiting factor for the case of large-scale network topologies, as identified in [6][10]. Centralized approaches have considerable scalability limitations, due to the amount of traffic and processing burden converging to a single controller/manager. In addition, they cannot support applications for which the time between each execution is comparable with the time to collect, compute and disseminate the results [6]. While several decentralized solutions have been proposed in the literature, e.g. [5][9][8], their main focus was on the control plane (i.e. routing functionality), devoting less attention to monitoring, which was reduced to periodically synchronizing topology databases. Distributing the monitoring functionality introduces a fundamental challenge: how monitoring entities can efficiently extract *local views* and share this information, while striking the right balance between accuracy and overhead for a wide range of applications.

In this paper we propose a decentralized monitoring system for large scale SDNs that achieves the goal of high reconfiguration reactivity with acceptable accuracy and small overhead. Our approach involves multiple monitoring entities, each able to perform monitoring tasks autonomously without relying on a central manager and without maintaining a global view of the network run-time state. Although decentralized monitoring is not a new topic in network management, previous solutions like [7][11] are not directly applicable to the new domain of software-defined networks due to: *i)* the technological novelty of SDN, *ii)* the shift towards new measurement enablers and *iii)* the heterogenous requirements of management applications that can reconfigure the network at a wide range of timescales and granularity levels (*e.g.* up to a single TCP flow).

The proposed approach is designed to operate within a distributed management environment such as the one in [6], where local managers, hosting the application logic, can adaptively reconfigure the network resources under their scope of responsibility at short timescales. Our solution can support the monitoring requirements of a wide range of management applications by relying on a flexible high-level interface and effectively aggregating new measurement tasks to limit the amount of resources consumed at the switches. To deal with the diversity of controller implementations and improve configuration flexibility, we abstract most of the monitoring functionality from the control plane and rely on measurement primitives as well as minimal interface(s), which could be extended to support new control software without requiring significant changes.

In this work, we investigate the benefits of the proposed decentralized monitoring system based on a realistic and demanding use case, where a distributed management application coordinates a content distribution service in an ISP network. We compare its performance in terms of monitoring latencies, as well as traffic overhead, to the one obtained with a centralized solution and show the impact of distributed monitoring on the application performance. In addition, by following an approach similar to the one presented in [10], we investigate possible tradeoff(s) between application reactivity/accuracy and monitoring scalability/overhead. The results show that our monitoring solution can reduce the monitoring delays by up to 69% compared to a centralized approach, which translates into more reactive control loops. We also show that while additional overhead is incurred by a distributed solution, this can be reduced by configuring the monitoring parameters without

978-3-901882-89-0 @2017 IFIP

significantly affecting the application's performance.

The remainder of this paper is organized as follows. Section II provides background information on the distributed network management framework considered in the paper and presents the main monitoring techniques used in SDN environments. In Section III, we describe in detail the design of the proposed architecture. Section IV describes the use case application considered for the evaluation of our solution. Experiment setup and evaluation results are presented in Section V. Section VI describes related work and Section VII concludes the paper.

II. Background

In this section, we provide background information about the SDN-based resource management framework considered for the design of our monitoring solution, as well as an overview of the techniques used to measure SDN networks.

A. Distributed Resource Management Framework

In [6], co-authors of this paper present a novel SDN-based network management and control framework that supports dynamic resource management applications in fixed backbone infrastructures. In this paper we adopt the design principles of the relevant architecture, which separates management and control functionality, allowing the two to evolve independently. A set of *local managers* (LMs), distributed over the network, hosts various management applications (MAs) that implement the necessary logic to decide on network (re)configurations. MAs are instantiated on the local managers as modules embedding information data structures and running on a common execution environment offered by the LMs. Each MA can execute in all LMs or in a subset of them (e.g. ones operating at the edge of the network). Configuration decisions taken by LMs are provided to *local controllers* (LCs), which define and plan the sequence of actions to be enforced for updating the network parameters.

Monitoring is an essential component of LMs. First, it is concerned with extracting raw statistics from the physical resources and generating useful information for applications. In this context, each LM needs to implement the necessary capabilities to collect the status of variables (e.g. links, traffic flows) within its local scope and make this information available to local MA instances. Second, since MA instances operating at different locations may need monitoring data gathered from outside their local scope, the monitoring functionality is therefore also concerned with disseminating frequent network state updates to remote LMs. In a SDN environment such a synchronization phase is essential for reconfiguring the network parameters based on a global, unified network view. This information can be exchanged between instances of a distributed MA through the signaling framework proposed in [33].

B. Monitoring Software-Defined Networks

Compared to traditional computer networks, where monitoring solutions require ad-hoc software installation / configuration and low-level tools, SDN has introduced a set of simple and reusable primitives for the collection of network variables at different granularity levels, which make them suitable to a wide range of management tasks. SDN flow-based switches (e.g. OpenFlow) allow network operators to flexibly specify the flows to monitor based on different packet fields (e.g., source and/or destination IP addresses), and to count the number of bytes or packets for these flows. Counters are fetched by polling a switch using ad-hoc *Read State* messages. The switch flow rules can be adapted or replaced depending on the analysis performed on the corresponding counters or according to predefined expiration timeouts.

This measurement approach is affected by several hardware technology issues. First, flow-based counters are maintained in expensive and power-hungry TCAMs and, as such, only a limited number of entries can be used for measurements. Another issue is the limited bandwidth between the switch and the SDN controller, which limits flow fetching to no more than a few thousand per second [27]. Finally, OpenFlow switches may also exhibit inaccuracies when updating the flow counters. For example, as discussed in [30], some devices do not update the counters every time a new packet matches a rule, but perform the updates periodically instead. Furthermore, devices from different vendors introduce different biases in measurements and may even present some limitations in terms of protocol support. Despite these open issues, our proposal relies on the *counting* approach due to its implementation simplicity, the wide support by different vendors, and configuration flexibility in terms of information granularity and measurement frequency. Alternative methods, such as hashing techniques (e.g. *sketches*) and mechanisms leveraging increased programmability at the switch [20][21][19], have still none or very limited support on devices, which makes their applicability uncertain.

III. System Architecture

This section presents the proposed monitoring system and motivates the design principles of the associated architecture. Our solution leverages a decentralized approach where each of the local managers described in Section II hosts a monitoring entity, called the *monitoring module* (MM), which is responsible for gathering information within the scope of the LM. Scalability for coping with a large number of network devices and their geographical span was the main driver for selecting a distributed approach.

A. Design Requirements

Effective design of distributed monitoring functionality has to take into account a number of key issues. If very intrusive, monitoring operations can adversely affect the network performance. At the same time, these operations need to be frequent and fast to enable management applications to operate at short timescales. In addition, they should provide accurate and high-granularity information to support configuration decisions. The impact of these issues is amplified in the case of large-scale SDNs, since configuration decisions might be taken far away from the locations where monitoring is performed. We identify below the three main requirements that have been taken into account for the design of the proposed monitoring approach.

Scalability: The monitoring system should be able to cope with a large number of information sources. As the number of physical resources under the scope of a single MM increases, the monitoring traffic converging to it, as well as the associated computational load, could drastically impact the system reactivity, as was shown in [8][27]. While in dense networks

with small diameter (e.g. data centers) this drawback can be mitigated through replication or by investing more CPU cycles and memory, in wide area networks (WANs) the monitoring responsiveness is significantly affected by network latencies. Based on the same motivation for distributing the SDN control plane [5], i.e. switch-to-controller latency reduction, we consider a decentralized monitoring solution for reducing monitoring delays and avoiding processing bottlenecks.

Programmability: The frequency and granularity of measurements have to be highly configurable based on the requirements of heterogeneous management applications. While some applications, such as elephant-flow detection, need fine-grained flow-based measurements, others only require aggregate statistics. In such a case, low-granularity measurements, which can be retrieved at a lower cost would be preferable (e.g. switch port measurements as opposed to individual flow measurements).

Responsiveness: MAs can change their monitoring requirements based, for example, on the analysis of measured metrics. The MM should be responsive in adapting measurement parameters, such as the polling frequency or the flow-level granularity, according to new requirements. Fast adaptations, as argued in [23][15], are essential for warranting acceptable information accuracy and can additionally reduce the monitoring overhead.

B. Monitoring Module

Figure 1 presents the architecture of the MM, which sits between MA instances and controller software. Applications use a common interface, e.g. a RESTful interface, offered by the MM for both injecting new monitoring requirements and receiving the corresponding measurement results. The positioning of the MM allows to abstract the application monitoring requirements from specific controller implementations. In addition, applications can communicate their needs in a high-level form, agnostic of the measurement techniques used or their implementation.

Each MM relies on a modular composition to maximize the system extensibility and improve the overall flexibility of the solution. The modular structure allows to decouple the logic involved in the processing of the application requirements from the one operating on the raw measurement primitives. This reduces the deployment effort when new types of requirements need to be supported, or new measurement mechanisms become available. The various components of the MM are described below.

Persistent Data Repository This component maintains network information which is not updated frequently, such as the topology graph representation (*e.g.* switches and links) and the current setup of paths between pairs of edge nodes. Such information can be represented through transactional databases and can be flexibly accessed/modified by a SQL-like querying mechanism.

Requirements Processor The first task of this component is to parse new monitoring requirements received from applications. These are registered in a local data store (*Requirements Table*, c.f. Figure 1), with each requirement represented as a tuple:

⟨ `Req_id, MA_id, Task, HL_targets, Mon_times` ⟩

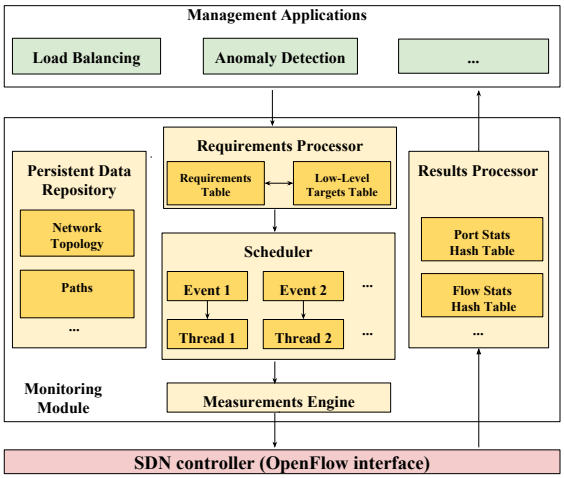

Fig. 1. Monitoring module architecture.

Req_id and *MA_id* are the unique identifiers of the monitoring requirement and the requesting management application. *Task* represents the overall goal of the measurements, for example the utilization of one or a set of links. *HL_targets* is the list of targets (high-level identifiers in the application's abstract view of the network) of the monitoring task, for instance, in case of a path utilization request, the corresponding list of paths. *Mon_times* can be a single parameter, i.e. the polling period, or an explicit sequence of measurement intervals. This allows to support both fixed polling periods and variable-rate measurements.

Recent proposals have enriched monitoring with the logic for performing sampling rate adaptations, which can be used, for example, to reduce the measurement overhead or mitigate route flapping [12]. However, most of these adaptive techniques are tailored to a specific measurement task, e.g. algorithms based on linear regression have been specifically developed for flow counting [15]. For this reason, we find more convenient for such logic to be implemented at each MA and not in the MM. This feature incurs additional communication cost between the MAs and the underlying MM, for example to enable reactive reconfigurations of the measurement times. On the other hand, it considerably improves flexibility, since different MAs can implement and tune ad-hoc adaptation algorithms, based on their monitoring needs.

The next procedure performed by this component is a translation routine, based on the *Task* specification, that maps each new table entry into one or more *Low-level targets*. Any extension of the MM-MA interface to accommodate new monitoring tasks needs to be reflected into one or more additional translation routines. Each low-level target (*LL_target*) can identify a specific physical resource, e.g. a switch port, or map a set of flow rules, i.e. a specific subset of the switch flow table. These entities are stored in the *Low-level targets* table with the following format:

⟨ `LL_target, Op_type, [Req_id], Sched_state` ⟩

Op_type indicates what type of measurement operation should be performed, for example collecting the average traffic rate of a specific switch interface. [*Req_id*] is the list of pointers to the corresponding application requirements, used for the

reverse translation. *Sched_state* is a flag indicating whether the low-level target refers to a new monitoring requirement (i.e. measurement operations have to be scheduled from scratch), or to a previous task for which some adaptation is required (i.e. operations have to be re-scheduled).

Before the insertion of a new low-level target, the table is looked up for similar entries. An existing entry is considered *similar* if the target is equivalent or included, e.g. two targets with the same *Task* and *Mon_times* attributes are similar if one refers to the flows matching source IP address 128.40.200.1 and the other corresponds to the flows for any source IP in the subnet 128.40.200.0/24. In such a case, the MM will merge the two low-level targets and the corresponding measurement times will be accordingly updated to satisfy both requests. Effective aggregation of the application requirements can achieve considerable reduction of the measurement traffic and alleviate the burden on the switch CPU. This is very important when monitoring processes compete with other control plane operations, such as flow setup, for accessing the switch resources.

Scheduler This component is in charge of generating and managing the individual measurement procedures (*e.g.* the ones requiring a single message exchange with a switch), which are executed as threads. It is called on every insertion in the *Low-level targets* table, and on any modification of existing tuples involving the measurement times. Scheduling new measurements indiscriminately can lead to none of them getting enough switch resources. As such, once invoked, the scheduler executes an admission control routine, in which it verifies, depending on the current measurement load, whether the measurement procedures for the new low-level target can be performed. In case there are not enough resources available to accommodate the new measurement procedures, these are rejected and the corresponding MAs are notified so that monitoring requirements can be renegotiated. The measurement load for a specific switch is defined by the expected monitoring bandwidth, which is estimated on a time-window basis given the list of the low-level targets already scheduled. This metric depends on the current measurement rates (i.e., the polling frequency) and on the number of flow (or switch port) records returned for each measurement procedure in the corresponding OpenFlow *Statistics Reply* messages. In this respect, our approach differs from recent proposals, which focus on the limited flow table TCAM space [14] rather than the control bandwidth. Once accepted, the new low-level target is mapped onto a set of events, each one associated with a timer to trigger the new measurement thread. Each thread finally generates a call to the measurement engine.

Measurements Engine This operates as an interface between the measurement thread under execution and the measurement mechanisms implemented at the controller. It assigns individual measurement procedures to one of the available primitives offered by the controller and supported by the underlying device. Such an interface is essential to allow most of the monitoring operations to remain independent of the specific controller implementation.

Results Processor This component receives the raw measurement results, for example messages of type OpenFlow statistic reply, from the controller. These are parsed (e.g. into JSON format) and the *Low-level targets* table is looked up for the corresponding target(s). Based on the operation type specified in matching table entries, the measurements are filtered to select the required counters. These are stored in corresponding data structures (hash-tables) and used for computing the metrics of interest. The number of samples kept in memory for each counter depends on the metric required by the applications, e.g. a single sample is enough for computing the average bandwidth. Finally, the processed results are associated to the relevant high-level targets and delivered to MAs through update messages.

C. Workflow Examples

As concrete examples of the MM workflow, we consider two simple monitoring procedures:

1. Average link utilization The requirements processor registers a requirement of type *link utilization*, with a fixed measurement period, for a set of links $l_1, l_2, ..l_n$. Each of these is translated into a low-level target s_x:p_y, where s_x identifies the switch and p_y the port on which the bitrate should be measured. According to the specified measurement period, for each target the scheduler periodically generates measurement threads, each calling the controller to create an OpenFlow *Port Statistics Request* and send it to s_x. The results processor receives the corresponding *Port Statistics Reply* messages from the controller, together with the measurement timestamps, and extracts the current byte counters (*tx_bytes*, *rx_bytes*). It then uses the new sample and the previous one, which is stored in a hash-table, to compute the average link utilization, and finally returns the value to the application.

2. Average flow throughput We consider an application requiring the throughput of the flow identified by source IP y and destination IP z. The requirements processor translates this into a low-level target s_x:src_y:dst_z, where s_x identifies the switch from which the flow counters should be fetched. By default, the ingress switch for that flow is selected as the measurement target. Then, the scheduler generates periodic calls to the SDN controller to build a *Flow Statistics Request* and sends it to the target switch. As the corresponding *Flow Statistics Reply* is received by the controller, the result processor extracts the current flow *byte count* and *duration*. By comparing two consecutive samples, it computes the average flow throughput, and reports this to the application.

IV. Use Case: Content Distribution

To demonstrate the capabilities of the proposed approach, we consider a distributed SDN network environment where an ISP operates a content distribution service. In this scenario, a set of content items is cached within the ISP. The network management system periodically updates (*e.g.* in the order of hours) the content placement and the paths between user locations and content servers. Following an approach similar to the one proposed in [31], it reconfigures in real-time the routing of user requests by selecting an appropriate *path* between the user location and one of the available content servers, based on the current path utilization. The objective is to avoid congestion in the network and, as such, prevent potential Quality of Experience (QoE) degradation. To enforce the redirection decisions, the management system uses a method similar to the one proposed in [26]. The OpenFlow-enabled

forwarding hardware at the edge of the network is programmed in real-time to rewrite fields of the IP packet header (*e.g.* the destination address) in order to redirect the content requests to the selected server transparently to the client. The enforcement of the path selection decisions is part of the header rewriting operations. For example, the path selection can be encoded in the packet ToS field.

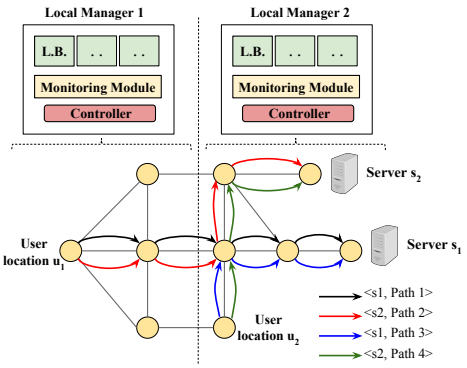

Fig. 2. Example representation of the use case (two network partitions).

Figure 2 exemplifies the use case. The network is divided into partitions, each under the control of a local manager. In addition, a controller is assigned to each partition and interacts with the network devices in the corresponding area. Each manager hosts an instance of a distributed Load Balancing (**LB**) application, which implements the necessary logic for reconfiguring, for each user location, the content server from which a requested content is retrieved and the path through which it is delivered. For simplicity, we assume the request and delivery paths to be symmetric. For each network edge switch mapped to a user location, the application keeps a list of available setups ⟨*Content_Server*, *Path*⟩ indicating how the incoming requests should be routed. Each LB instance operates periodically on a short timescale, i.e. every few seconds. At each execution, it obtains two statistics from the monitoring system.

The first statistic is the *average link utilization* for the links included in the local paths, *i.e.* the paths emanating from a client location within the scope of the relevant LM. Statistics for the links located in the local scope are directly collected and exposed by the underlying MM at each execution of the application. To this end, the MM periodically executes Procedure 1 as described in Section III-C. The monitoring of remote links is delegated to the LB instance operating on the relevant partition that registers the corresponding monitoring requirements on its MM and periodically synchronizes the results with the other LB instances.

The second statistic is the *average rate* of the content traffic originated by users in the local network partition. More specifically, each LB instance obtains the average throughput of all the flows matching the source IP address of one of the content servers and the destination IP address of one of the users. This measurement is used to determine the volume of traffic by which congested paths can be offloaded. The required statistics are generated by the local MM based on Procedure 2 as described in Section III-C.

In case of link congestion (*e.g.* average utilization exceeding a predefined threshold), the LB application is responsible for offloading part of the traffic from the congested link in order to bring its utilization below the threshold. Some flows are removed from the congested paths (paths including the congested link) and are (equally) assigned to alternative, non-congested, options represented by the 2-tuple ⟨*Content_Server*, *Path*⟩. The new configurations are enforced on the ingress OpenFlow switches. If a congested path spans multiple network partitions, the corresponding LB instances operate iteratively, as in the solution presented in [32]. The first decision is taken by the LB instance directly associated with the congested link based on the bandwidth availability on the alternative paths. The result is then communicated to the next LB instance until the process terminates.

V. Evaluation

We evaluate our monitoring system based on the use case described in Section IV by focusing on the performance in terms of latency and traffic overhead, as well as on the impact on the LB application. In addition, we investigate the gain that can be achieved through the processing of monitoring requirements, as described in Section III-B, at each MM. In the experiments we have used *Mininet* to emulate the network topology, including hosts, i.e. clients and content servers, and OpenFlow switches. The LM, including the MM and the LB application logic, has been implemented as a set of Python modules. Finally, we have reused a small set of APIs from the SDN controller POX [35] to implement the controller functionality.

A. Experiment Setup

We use the topology depicted in Figure 2 with 10 OpenFlow-enabled switches. All links have 10 Mbps bandwidth and 10 ms latency. Clients are distributed over two user locations u_1, u_2. Each client can reach two servers (s_1, s_2), each being accessed using one path only. Each experiment has a duration of 5 minutes and is preceded by a short startup phase in which paths are installed and the MM and LB application are initialized. The placement of LMs is provided as an input and used to compute the relevant hop-count and corresponding latencies between pairs of LMs. After the startup, each client starts generating content requests following a pattern derived from the one used in [34]. The content size has been scaled down in accordance to the reduced link bandwidth. The total bandwidth required for content distribution is on average 2 Mbps (20% of link capacity). In addition to video traffic, we use fixed-rate UDP flows generated with *Iperf* from the server locations to emulate background traffic.

The LB application reconfigures the flow routing in case of link congestion, as described in Section IV, based on local knowledge, including the utilization of local links and the throughput of local video-traffic flows (if any), as well as information about remote links, which is accessed through periodic synchronization. For simplicity, all LB instances run simultaneously (same frequency and clock reference). For each experiment, we configure two parameters: p_l is the period of local measurements performed by each MM, and p_s is the synchronization period of link status between the LB instances, with $p_s \geq p_l$. Link congestion is generated by spikes of content demand. To adjust the number of congestion episodes, as well as their duration, we act on the volume of UDP traffic which we set to 70% of the link capacity (7 Mbps). Another key parameter is the congestion threshold since it has a direct

impact on the flow (re)scheduling operated by LB. We set it to 85% of the link capacity in accordance to [22]. Such settings allow to avoid excessive route flapping and keep the average period of route reconfigurations (at least) one order of magnitude higher than the content download times, which fall between 0.5 and 1 second. These values are in line with traditional end-user redirection practices [29].

B. Performance of Decentralized Monitoring

In this subsection, we compare our decentralized monitoring approach with a centralized solution where the full state of the network is collected by a single management entity. We first focus on the **monitoring latency**, a measure of reactivity, defined as the delay between the time the measurement starts (*e.g.* the corresponding procedure is selected by the scheduler) and the time the requested information is made available to the LB instance performing the flow routing reconfigurations. We evaluate the monitoring latency for the link utilization measurements in 4 different setups: *Centralized* (single manager), *2 LMs*, *3 LMs* and *Fully distributed*, in which one LM is assigned to every single switch. For the centralized, 2 LMs and 3 LMs cases, we perform 5 experiments, each with a different manager allocation, and average the results. We fix $p_s = p_l = 5sec$, *i.e.* the link status is synchronized between LB instances with every new measurement, and we configure each LM to synchronize with every other LM in the topology.

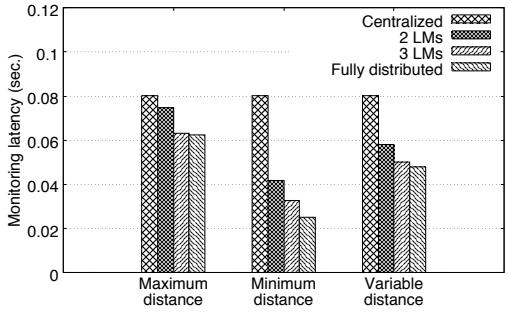

Fig. 3. Monitoring delay for different LMs setups.

Figure 3 depicts the average monitoring latency for 3 cases: *i) Minimum distance*: reconfigurations are computed close to where raw statistics are extracted, i.e. the closest LM; *ii) Maximum distance*: reconfigurations are computed by the farthest LM from where the statistics are collected and *iii) Variable distance*: reconfigurations are computed with the same probability by any of the available LMs. The performance obtained with the *centralized* setup (baseline scenario) is almost constant as the monitoring information is always processed at the central manager independently from where the statistics are gathered. For the decentralized setups, we observe a significant delay reduction by up to the 69% for *minimum distance* in comparison to the centralized scenario, while we notice a smaller reduction (up to 22%) for *maximum distance*. As expected, the higher the percentage of reconfigurations computed close to where the relevant knowledge is collected, the higher the reduction in terms of control-loop delays achieved with the decentralized approach.

In addition, we evaluate the cost of a decentralized solution in terms of monitoring overhead, which we define for each experiment as the generated monitoring traffic, *e.g.* sum of the size of each packet multiplied by the path length (number of hops). The results are shown in Figure 4. As can be

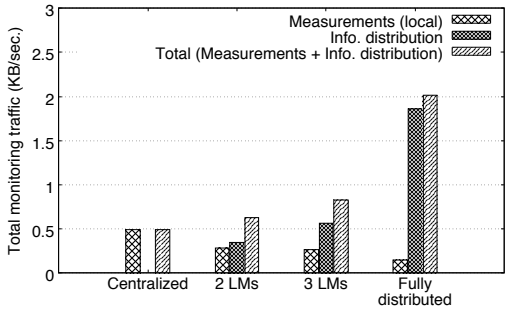

Fig. 4. Monitoring overhead for different LMs setups.

observed, the average overhead significantly increases (by more than a factor 4) when moving from the *centralized* to the *fully distributed* setup due to the increasing amount of traffic required for distributing the monitoring information between LMs. However, given the small size of the link utilization update messages, the resulting overhead is at most 2 KB/sec.

C. Monitoring Information Distribution

In the considered decentralized management framework [6], instances of a distributed application can take decisions based on information extracted at a remote location. In this subsection we investigate the effects of the dissemination of monitoring information between LMs on the performance of the LB application for the *Fully distributed* setup. Episodes of congestion are taken into account on a specific link l, located under the scope of a LB instance called *local LB*. The offloading decisions are made outside the local partition by another application instance called *remote LB*. We fix $p_l = 1$ second (minimum value in the current MM implementation) and let p_s vary in the range $(1, 10)$ seconds.

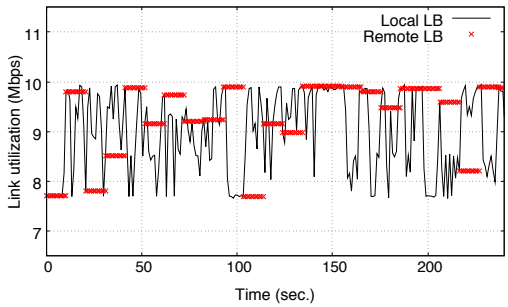

Fig. 5. Link utilization timeseries at Local LB and Remote LB for $p_s = 10$.

Figure 5 illustrates the link utilization exposed to the *local* and *remote* LB instances for the worst-case scenario, *i.e.* $p_s = 10$ seconds. We observe that, with such a high synchronization period, the *remote LB* fails to catch utilization spikes of short duration. In this specific case, no action is performed for 53% of the congestion events. Table I reports the error between the local and remote LB views of link utilization.

To measure the effect of relaxed synchronization, we use the RMSE (root-mean-square error), where a RMSE of 0 corresponds to perfect synchronization. As expected, the RMSE

TABLE I. AVERAGE UTILIZATION ERROR AND RMSE

Info. Distr. Period p_s(sec.)	Avg Util. Error	RMSE
2	0.44501	0.78
4	0.65137	1.0096
6	0.73237	1.0298
8	0.90245	1.1952
10	1.30	1.5448

increases as the information dissemination period increases, which indicates an increase in terms of inconsistency between the views of the two LB instances. To quantify how the LB application is affected by the inconsistency, we sample the utilization of link l at rate $1/p_l$ (*i.e.*, every 1 second) throughout the duration of the experiment. In addition, we collect the *time to first byte*, *i.e.* the response time for all the requested contents, as a measure of the user's QoE, from the set of users in the scope of the *remote LB*. In order for the results not to be affected by network latencies, the selected users are served through paths of equal lengths. Due to space limitations, we do not report the performance obtained with other user metrics, such as the average download speed or the total download time.

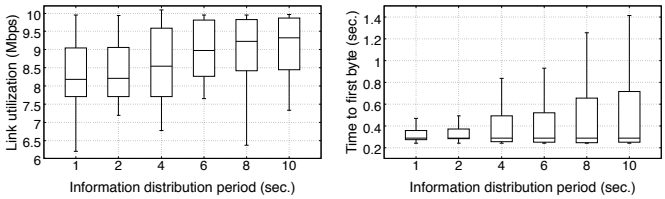

Fig. 6. Load Balancing application performance

Results are shown in Figure 6 in the form of boxplots, with the whiskers extending from the box (first and third quantile boundaries) to the 95 percentiles. It can be observed that the utilization of l is significantly affected by the synchronization period. For low values, such as $p_s = 1, 2$, the utilization is kept below the congestion threshold for more than half of the total duration of the experiment. Starting from $p_s = 4$, larger values of RMSE lead to a noticeable increase of the utilization median. This is reflected on the user perceived quality. For $p_s = 1$, almost the totality of the requests start being served within $0.5sec$. For $p_s \geq 4$, the increasing degree of inconsistency has an impact on up to 50% of content requests. For example, in the case $p_s = 8$, more than 25% of the requests get a first response in more than $0.6sec$.

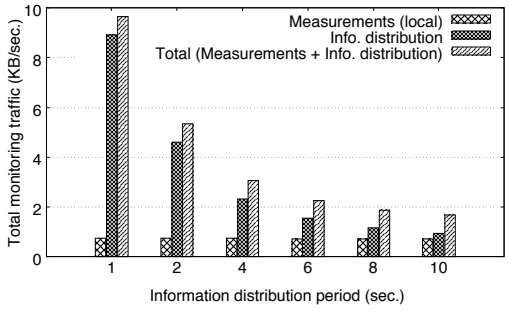

Fig. 7. Monitoring overhead vs. Information distribution period

Finally we also evaluate the monitoring overhead in terms of monitoring traffic as defined in Section V-B. As shown in Figure 7, the overhead is split in two components: i) the *measurements* overhead, *i.e.* the traffic incurred by the collection of the raw statistics from the physical devices, and ii) the traffic incurred by the distribution of the link status between the LB instances that linearly increases with the frequency of the information distribution. We can first observe that the generated traffic is dominated by the dissemination of monitoring information, even for limited synchronization frequencies, which is in accordance to the results reported in Section V-B. We also observe that significant overhead reductions can be obtained by trading off a bit the LB performance. A saving of 42% can be achieved with $p_s = 2$ compared to with $p_s = 1$, while incurring limited disruption on the application performance, as shown in Figure 6.

D. Measurements Aggregation

The acquisition of statistics via a pull-based mechanism consumes a significant amount of switch control bandwidth, which is a scarce resource in OpenFlow-enabled switches. To mitigate this issue, our solution leverages the aggregation of different monitoring tasks. This is performed by the *Requirements Processor* (c.f. Section III-B). Such a feature provides advantages for MAs requiring flow measurements at similar times, and/or for similar portions of the switch flow space, in particular. To evaluate the gain that can be achieved through aggregation, we implement a scenario where 3 monitoring requirements m_1, m_2, m_3 are registered at the same time and on the same MM by three different MAs. The execution of the measurements associated with each m_i results in fetching a fixed number of flow table entries k at a period $p_i \in [1, 2, ..7, 8]$ from the same switch. Two parameters α and β are associated with each set. Parameter α represents the level of temporal concurrency of the required measurements, which ranges from 0 (lowest level of concurrency, e.g. $[p_1, p_2, p_3] = [6, 7, 8]$) to 4 (maximum level of concurrency, i.e. $p_1 = p_2 = p_3$). Parameter β represents the level of overlap in terms of similar flow entries required from the switch. $\beta = 0$ represents no overlap. For $\beta = 1$, there exists an overlap of 50% between the requirements of two MAs. For $\beta = 2$, the three MAs share 50% of requests. For $\beta = 3$, two MAs have completely overlapping requirements and share 50% with the third one. $\beta = 4$ represents an overlap of 100% between the three MAs. Figure 8 shows the resulting savings in terms of switch control bandwidth. The case $\beta = 0$ is not represented as it does not result to any savings.

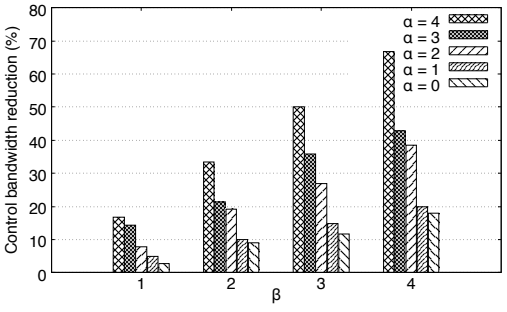

Fig. 8. Reduction of switch control bandwidth by requirement aggregation.

Significant reductions can be achieved when $\beta \geq 2$. For example, for $\beta = 2$, an average reduction close to 20% is

obtained. Such savings can allow to increase the overall measurement rate, thus enhancing the reconfiguration reactivity. For example, assuming that for $\beta = 0$ the bandwidth between the switch CPU and the controller is fully saturated, an increase of the monitoring rate by up to a factor of 3 can be achieved by using aggregation.

VI. Related Work

The de-facto monitoring standard of today's IP network is NetFlow, which is based on packet sampling. Netflow samples packets with the same probability and aggregates them into flows. However, as discussed in [28], several studies have shown the limitations of packet sampling to perform fine-grained monitoring (*e.g.* biases toward sampling larger flows) making it unsuitable for many management applications.

The advent of SDN has empowered network monitoring with new measurement enablers as OpenFlow switches can keep track of active flows in the network and update per flow counters. A number of proposals have recently exploited this feature to provide direct and precise flow measurements without resorting to packet sampling. In OpenTM [24] the SDN controller pulls, at fixed intervals, the switch counters collected by explicitly polling the switches in order to periodically generate traffic matrices. In [25] the authors propose FlowSense, an approach where the network utilization is measured using a different, push-based, approach. This uses the messages generated during the setup and eviction of flows from the switch flow table. Compared to the technique used in our paper that leverages explicit switch polling, the solution in [25] can reduce the measurement overhead but suffers from limited flexibility since it only works with short-lived flows.

While most of the recent proposals have focused on specific measurements or a very limited set of measurement tasks, the approaches presented in [18] and [23] provide a measurement API for supporting a wide range of tasks. OpenSketch [18] relies on a clean-slate approach where a novel processing pipeline is used on the switch to support many different measurement tasks. In addition, a library is developed for the control-plane to reconfigure the pipeline. Payless [23] resembles more the approach adopted in this paper as it provides an API to serve different monitoring requests, all executed through pull-based measurements. However, the monitoring system comes with a single algorithm for polling-rate adaptation. In our architecture the monitoring adaptation logic is implemented at each MA, thus increasing the flexibility, while the role of monitoring entities is to responsively absorb the new parameters to reconfigure the measurement scheduling.

In contrast to our approach, all the aforementioned proposals are mainly tailored to early SDN solutions that rely on a physically centralized control infrastructure. This assumption has been questioned in [8] and [5] where distributed control planes have been proposed to overcome scalability issues such as processing bottlenecks at the central controller and large control latencies. However, the main focus of these papers is on how distributed controllers can unify their local views of the network, paying little attention to measurement issues. In [5] Tootoonchian et al. present a controller-to-controller communication mechanism based on a pub/sub paradigm. In [8], Koponen et al. propose a distributed database for the dissemination of slowly changing network state and a distributed hash table for exchanging volatile information. Another important work is [10], which investigates the main issues posed by state distribution in a logically centralized, physically distributed SDN architecture. One of these issues, i.e. the trade-off between performance optimality and state distribution overhead, has been considered in Section V-C, where we have evaluated how the timeliness of synchronized monitoring information impacts the MA performance and the overhead in terms of additional monitoring traffic. In a less recent work [11], the authors introduce a model, called A-Gap, for adaptive reduction of the traffic overhead in distributed monitoring based on filtering. However, this technique addresses a different, hierarchical, monitoring architecture where the information is aggregated and transmitted on a spanning tree toward a central management station.

VII. Conclusion

In this paper we have presented a novel monitoring approach for SDNs that can provide heterogeneous management applications with frequent and consistent network state updates, thus enabling fast and effective reconfigurations. Our solution relies on a decentralized architecture satisfying the requirements of large-scale networks with a high number of geographically dispersed devices. To reduce the consumption of the switch control bandwidth, it performs effective processing of heterogeneous monitoring requirements and flexible measurement scheduling.

The evaluation, which is based on a realistic use case, has shown that a decentralized monitoring approach can improve the reconfiguration reactivity by significantly reducing the control-loop delays, in particular when a large percentage of reconfiguration decisions can be taken close to where the relevant statistics are collected. However, decentralizing the monitoring functionality incurs additional communication overhead, which increases proportionally with the frequency of the monitoring information distribution between local managers. Experiments conducted using a realistic distributed management application have highlighted that some overhead reductions can be achieved obtained through limited synchronization while maintaining acceptable application performance.

The results encourage us to explore in the future techniques to obtain further overhead reductions, while warranting acceptable consistency levels, for a large set of management tasks. Furthermore, we envision to empower our distributed framework with the necessary mechanisms to mitigate the TCAM utilization at individual OpenFlow switches and avoid additional bottlenecks. To investigate these issues, we plan to extend our virtual testbed to support larger topologies, different traffic characteristics and implement a wider set of management applications.

Acknowledgment

This research was funded by the EPSRC KCN project (EP/L026120/1) and by the Flamingo Network of Excellence project (318488) of the EU Seventh Framework Programme.

REFERENCES

[1] H. Kim and N. Feamster. Improving network management with software defined networking. In *IEEE Communication Magazine*, vol. 51, no. 2, pp. 114-119, Feb. 2013.

[2] Y. Yuan, R. Alur, and B. T. Loo. NetEgg: Programming network policies by examples. In *Proc. ACM Hotnets*, Los Angeles, CA, USA, Oct. 2014.

[3] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: dynamic flow scheduling for data center networks. In *Proc. NSDI*, San Jose, CA, USA, Apr. 2010.

[4] S. Agarwal, M. Kodialam, and T. Lakshman. Traffic engineering in software defined networks. In *Proc. IEEE INFOCOM*, Apr. 2013.

[5] A. Tootoonchian and Y. Ganjali. HyperFlow: a distributed control plane for OpenFlow. In *Proc. USENIX INM/WREN*, San Jose, CA, USA, pp. 3-9.

[6] D. Tuncer, M. Charalambides, S. Clayman, and G. Pavlou. Adaptive resource management and control in software defined networks. In *IEEE TNSM*, vol. 12, no. 1, pp. 18-33, Mar. 2015.

[7] Shan-Hsiang Shen, Aditya Akella. DECOR: A distributed coordinated resource monitoring system. In *Proc. IEEE IWQoS*, Coimbra, Portugal, Jun. 2012.

[8] T. Koponen et al. Onix: a distributed control platform for large-scale production networks. In *Proc. USENIX OSDI*, Berkeley, CA, USA, 2010, pp. 351-364.

[9] P. Berde, et al. ONOS: towards an open, distributed SDN OS. In *Proc. ACM HotSDN*, Chicago, Illinois, USA, Aug. 2014, pp. 1-6.

[10] D. Levin, A. Wundsam, B. Heller, N. Handigol, and A. Feldmann. Logically centralized?: State distribution trade-offs in software defined networks. In *Proc. ACM HotSDN*, Helsinki, Finland, Aug. 2013, pp. 1-6.

[11] A. Gonzales, and R. Stadler. Adaptive distributed monitoring with accuracy objectives. In *Proc. ACM INM*, Pisa, Italy, Sep. 2006, pp. 65-70.

[12] N. Van Adrichem, C. Doerr, and F. Kuipers. OpenNetMon: network monitoring in openflow software-defined networks. In *Proc. IEEE/IFIP NOMS*, Krakow, Poland, May 2014.

[13] C. Yu et al. Software-defined latency monitoring in data center networks. In *Proc. PAM*, New York, NY, USA, Mar. 2015, pp. 360-372.

[14] M. Moshref, M. Yu, R. Govindan, and A. Vahdat. Dream: dynamic resource allocation for software-defined measurement. In *Proc. ACM SIGCOMM*, Chicago, IL, USA, Aug. 2014, pp. 419-430.

[15] T. Zhang. An adaptive flow counting method for anomaly detection in SDN. In *Proc. ACM CoNEXT*, Santa Barbara, CA, USA, Dec. 2013, pp. 25-30.

[16] Y. Yu, C. Quien, X. Li. Distributed collaborative monitoring in software defined networks. In *Proc. ACM HotSDN*, Chicago, IL, USA, Aug. 2014, pp. 85-90.

[17] N. McKeown et al. OpenFlow: enabling innovation in campus networks. In *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 3, pp. 69-74, Apr. 2008.

[18] M. Yu et al. software defined traffic seasurement with OpenSketch. In *Proc. USENIX NSDI*, Lombard, IL, USA, pp. 29-42, Apr. 2013.

[19] C. Kim et al. In-band network telemetry via programmable dataplanes. In *Proc. ACM SIGCOMM*, London, UK, Aug. 2015, Demo Session.

[20] P. Bosshart et al. P4: programming protocol-independent packet processors. In *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87-95, Jul. 2014.

[21] F. Uyeda, L. Foschini, F. Baker, S. Suri, and G. Varghese. Efficiently measuring bandwidth at all time scales. In *Proc. USENIX NSDI*, Boston, MA, USA, Mar. 2011, pp. 71-84.

[22] D. Tipper et al. An analysis of the congestion effects of link failures in wide area networks. In *IEEE JSAC*, vol. 12, pp. 179-191, Jan. 1994.

[23] S. Chowdhury, M. Bari, R. Ahmed, and R. Boutaba. PayLess: a low cost network monitoring framework for software defined networks. In *Proc. IEEE/IFIP NOMS*, Krakow, Poland, May 2014.

[24] A. Tootoonchian, M. Ghobadi, and Y. Ganjali. OpenTM: traffic matrix estimator for OpenFlow networks. In *Proc. PAM*, Zurich, Switzerland, Apr. 2010, pp. 201-210.

[25] C. Yu et al. FlowSense: monitoring network utilization with zero measurement cost. *Proc. PAM*, Hong Kong, China, Mar. 2013, pp. 31-41.

[26] M. Wichtlhuber, R. Reinecke, and D. Hausheer. An SDN-based CDN/ISP collaboration architecture for managing high-volume flows. In *IEEE TNSM*, vol. 12, no. 1, pp. 48-60, Mar. 2015.

[27] J. C. Mogul et al. Devoflow: cost-effective flow management for high performance enterprise networks. In *Proc. ACM Hotnets*, Monterey, CA, USA, Oct. 2010.

[28] V. Sekar, M. K Reiter, and Hui Zhang. Revisiting the case for a minimalist approach for network flow monitoring. In *Proc. ACM IMC*, Melbourne, Australia, Nov. 2010, pp 328-341.

[29] A. Su, D. Choffnes, A. Kuzmanovic, and F. Bustamante. Drafting behind Akamai. In *IEEE/ACM TON*, vol 17, no. 6, pp. 1752-1765, Dec. 2009.

[30] L. Hendriks, R. Schmidt, R. Sadre, J. Bezerra and A. Pras. Assessing the quality of flow measurements from OpenFlow devices. In *Proc. TMA*, Louvain La Neuve, Belgium, Apr. 2016.

[31] I. Poese et al. Enabling content-aware traffic engineering. In *ACM SIGCOMM Computer Communication Review*, vol. 42, no. 5, pp. 21-28, Oct. 2012.

[32] D. Tuncer, M. Charalambides, G. Pavlou, N. Wang. DACoRM: a coordinated, decentralized and adaptive network resource management scheme. In *Proc. IEEE/IFIP NOMS*, Westin Maui Maui, HI, USA, Apr. 2012, pp. 417-425.

[33] D. Valocchi et al. Extensible signaling framework for decentralized network management applications. In *Proc. IEEE/IFIP NOMS*, Istanbul, Turkey, Apr. 2016.

[34] M. Claeys, D. Tuncer, J. Famaey, M. Charalambides, S. Latre, G.Pavlou, F. De Turck. Hybrid multi-tenant cache management for virtualized ISP networks. In *Journal of Network and Computer Applications*, vol. 68, issue C, pp. 28-41, June 2016.

[35] *POX* OpenFlow controller. *http://www.noxrepo.org*