SDN based performance monitoring for cloud-scale applications

Mona Elsaadawy
McGill University

Abstract

The recent rise of cloud applications, representing large complex modern distributed services running across hundreds to thousands of nodes, has made performance monitoring one of the major issues and a critical process for both cloud providers and cloud customers. Many different monitoring techniques are used for such applications. In many cases, software instrumentation is required for taking measurements, making it a tedious task to implement. However, using new trends like Software Defined Networking (SDN) and Network Function Virtualization (NFV) show promise to provide some of the required measurements without the need of customization. In this report, we review some of the most common monitoring approaches. A major focus is on SDN and/or NFV based performance monitoring techniques that were developed to provide the scalability and agility needed in cloud-scale applications.

1 INTRODUCTION

Cloud applications currently cover many domains (e-commerce, health, sciences, gaming, and many more). The performance of such applications has a direct impact on business metrics such as revenue and customer satisfaction. Examples of this impact are well documented. For example, Google loses 20% traffic if their web sites respond 500 ms slower, Amazon loses 1% of revenue for every 100 ms in latency and Mozilla's study showed that users usually leave the web site if the page is not loaded within one to five seconds [4]. Therefore, the state of the application-relevant performance measures needs to be continuously monitored. With the help of these monitored data, performance-related problems can be detected, diagnosed and resolved.

Most cloud applications are deployed across multiple tiers where client requests flow from one tier to the next, each performing a different task to complete the request. This makes it difficult to identify which tier of the application is the bottleneck and how to resolve performance problems. Understanding the behavior of these applications is further complicated by the nature of modern data centers that rely on sharing resources for cost and energy efficiency.

Many performance monitoring tools offered by cloud providers use the unit of a virtual machine on a hypervisor to observe the behavior of individual application components [3]. Performance bottlenecks are identified by monitoring hardware resource utilizations or application-specific measures such as request service times, obtained by instrumenting the application itself. However, this still typically provides only coarse grained resource consumption information, and it must be aggregated across hosts to understand overall application performance [13]. In addition, instrumenting the application code is a less practical technique and even impossible when access to end-user devices is required.

Instead, application-specific measures can be obtained by intercepting network packets and infer application behavior from this information. This approach make use of network components (such as routers and switches) to obtain such measures. The recent emergence of SDN and NFV increases these computational power of the network components. For instance, SDN enabled switches and routers can provide per-flow statistics to determine end-to-end network performance. In addition, any kind of user defined monitoring function can be deployed into the network as a software based network function using NFV.

Consequently, we believe that pushing the application monitoring functions down into the network layer, and utilizing SDN and/or NFV has the potential to provide both the cloud providers and consumers with a real-time, flexible, scalable and unified application performance monitoring tool. Therefore, in this report we first provide some basic background on monitoring application performance in cloud environments, and on SDN and NFV. Then we have a closer look at some specific logging architectures used for monitoring application performance that are based on instrumenting the application. Finally we review several performance monitoring techniques that adapt SDN and/or VNF to monitor either network or application dependent parameters at the network layer.

2 BACKGROUND

2.1 CLOUD APPLICATION MONITORING

The applications running inside public clouds and private data centers are to be composed of multiple interacting components. An example of such multi-tier architecture is illustrated by gray-shaded boxes in Figure 2.1: a front-end server, a load balancer, an application server and as a data storage a cache and a database. Each component normally runs on a dedicated machine or cluster of machines and may be replicated to support higher demand. The front-end web server receives client requests and forwards them to the application server where the processing logic is stored. Processing client requests can involve data retrieval from different data sources such as databases and/or caches. Caches store information temporarily needed to fulfill a client request in order to reduce the latency. Moreover, load balancers may be used in cloud architectures to distribute the client requests received by the front-end servers across a cluster of the application servers to maximize application throughput and minimize response time.

In such application architecture, there are many sources that can lead to a degradation in the application performance. Figure 2.2 shows how application components are installed on virtual machines distributed over physical resources. Accordingly, the source of performance problems can be quite varied, and thus, a variety of measurements are needed.

In particular, the source of performance problem can be an individual component or the communication between the components. Within a component, we can distinguish between hardware and software resources:

a) *Hardware resources*: the most common cause of performance issues in a cloud based application is incoming load exceeding capacity of one or more hardware resources like CPU, memory, I/O and network used by an application component.

b) *Software resources*: a performance issue could also be the result of a software resource bottleneck such as lock contention among multiple threads for shared data.

In regards to communication between components, a possible problem could be a connection misconfiguration between application components.
For example, consider again the gray-shaded multitier architecture depicted in Figure 2.1. Assume we have a high request delay at the client side for a subset of the requests and we would like to troubleshoot this problem. There are many possible causes. First of all, the network layer may have inconsistent bandwidth, high latency or packet loss. Secondly, maybe we have an increase in the application usage – a number of users simultaneously connects submitting the

same kind of transactions - which leads to the saturation in one of the application hardware resources or contention on one data lock. Lastly, we may have a parameter misconfiguration for one of the application replicas so that its link to the cache node is not properly used resulting in high delay on processing the subset of client requests that is handled using this misconfigured application replica.

To be able to identify the root causes of the detected performance problem in such multi-tiered and highly distributed architecture, application performance-relevant measures need to be collected at mainly three different layers as shown in Figure 2.2:

1) *Application layer measures include*:

a) Application component performance measures such as per component response time and throughput are typically offered by instrumenting the platform software (front-end server, application server, etc.). NEVPerf [10] is an example research proposal that collects these application component performance measures at the end hosts as will be discussed in Section 3. In contrast, NetAlytics [13] exploits SDN and NFV to get these measures as will be presented in Section 4.

b) Application dependent measures are measures about the application-internal behavior including executions of methods, occurrences of exceptions and calls to remote services or databases. Application instrumentations [11], [17] are typically required to generate log messages to be used in performing this kind of application dependent measures. A more detailed description of logging mechanisms will be presented in Section 3.

2) *System layer measures* collects the state of hardware resources such as CPU load, memory consumption, I/O statistics and network related measures like throughput, bandwidth, latency and packet loss. These kind of measures should be collected for both the shared hardware resource, which may host more than one application as mentioned before, represent by solid arrow in Figure 2.2 and for each individual application to estimate the application consumption ratio of the shared resources- the dashed arrows in Figure 2.2. There are plenty of cloud resource monitoring tools presented in the literature. A good survey on these cloud recourses monitoring tools can be found in [3]. Recent research proposals [7], [8], and [9] utilize the SDN technology deployed into today's cloud networks to collect network related measures. More details about these research proposals will be presented in Section 4.

2.2 SOFTWARE DEFINED NETWORKING

2.2.1 Architecture

Software Defined Networking (SDN) is a relatively new computer networking paradigm providing a fundamental shift in the way network configuration and real-time traffic management is performed. In classic networking devices, the forwarding plane (data plane) and routing plane (control plane) are on the same device. The SDN framework decouples the control plane from the data plane to make the data plane completely programmable, and thus, eliminates the manually intensive regime of fine tuning individual hardware components. Overall, the SDN architecture consists mainly of three layers namely data plane, control plane and application plane:

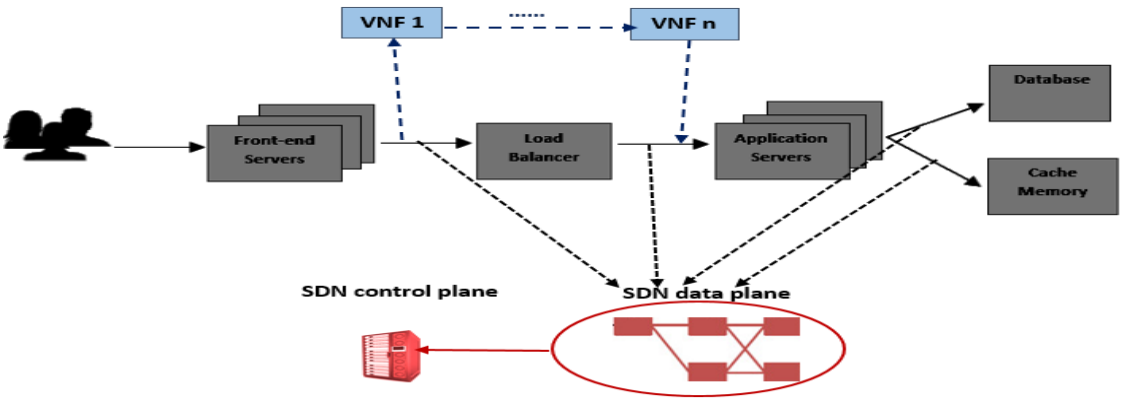

Figure 2.1. Multi-tier application architecture with and without SDN and NFV.

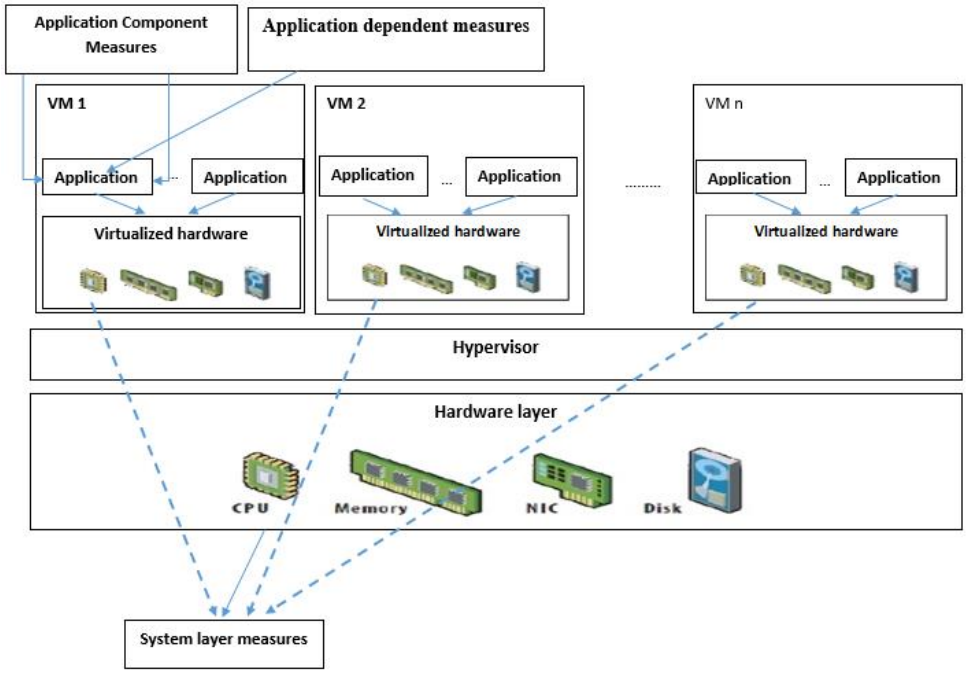

Figure 2.2. Application performance-relevant measures.

1) *Data (forwarding) plane* is a set of forwarding elements which can be switches, routers, virtual networking equipment, firewalls, and so forth. The purpose of the data plane is to forward network traffic based on certain forwarding rules instructed by the control plane.

2) *Control plane*: the basic component of a control plane is the SDN controller. The SDN Controller has mainly two purposes. First of all, the SDN controller is responsible for making traffic routing decisions based on end user application policies and, then communicating resulting routing rules to the data plane. Secondly, it provides an abstraction and centralized view of the underlying network and, thus provides a powerful network management tool to fine tune performance. A multi-controller-based architecture may be

needed based on the network size to prevent controller bottleneck. In that case, an interface protocol is used to manage interactions between the various controllers.

3) *Application plane* consists of network-specific and business applications. An abstract view of the underlying network is provided to applications via the SDN controller. The abstract view may contain network parameters like delay, throughput, and availability descriptors giving the applications a wide view of the network. Based on the provided network abstraction, applications or network services communicate end-node connectivity requests to the SDN controller, which in turn configures individual network elements in the data plane for efficient traffic forwarding.

Adapting SDN technology o the cloud application architecture in Figure 2.1 means that the application flows between application components will be now manipulated and deployed to network data plane components through the SDN controller as depicted by the bottom red boxes in Figure 2.1.

2.2.2 Openflow

OpenFlow - the de facto standard of SDN – is an Application Program Interface (API) for exchanging control messages between control and data planes. Since Openflow is concerned about network flow management, we have first to define network flow. A network flow is a unidirectional series of Internet Protocol (IP) packets protocol traveling between a source and a destination IP/port pair within a certain period of time. An OpenFlow compatible switch has one or more flow tables -configured by the SDN controller – which contain matching fields to match incoming flows and/or packets with certain actions such as prioritization, queueing, packet forwarding and dropping.

Each flow table rule includes three basic fields: *matching fields*, *actions* and *counters*. Matching fields are layer 2-4 packet parameters such as packet header and ingress ports. Actions are processing instruction to be executed for the matched packets such as “sent to destination (output port)”, “forward to controller”, “drop packet” or “send to other flow tables”. In addition, the action field can refer to a group of actions - which is called action bucket - to be executed on matched packet(s) and/or flow(s). Finally, the counters field can measure flow statistics for real-time traffic monitoring and are updated based on the matched packets. Moreover, flow table rules have a priority of execution in case one packet matches two different flow table rules. Once a packet arrives at the switch, the packet parameters are matched against the entries of one single flow table, and the corresponding matched flow table rule action is executed and counters are updated accordingly.

The current Openflow API has mainly two limitations regarding the capability of its data plane elements. Firstly, the matching fields for network flows are predefined and restricted only to layer 2-4 (data link, network and transport layers) packet information. However, many applications could benefit from inspecting packets based on higher layer information. For example, an Intrusion Detection System (IDS) needs to look at the whole packet and even buffer a couple of packets to detect an intrusion. In Section 4, we will review some of the research work that extends the Openflow API to support inspecting the packets based on application layer information.

Secondly, all OpenFlow actions are predefined and deeply hardcoded in the hardware. An external middleboxes can be used to perform customized actions such as firewalls. This leads to message overhead in the network. Instead, both [5] and [16] prove the feasibility of extending the current Openflow switches to support customized actions. The authors in [5] developed a software SDN switch using network function virtualization and deployed it on a commodity hardware. Obviously, using a software SDN switch induces a degradation in the performance compared to a hardware SDN switch. Therefore, the authors compensate this performance degradation by deploying their software SDN switch on a powerful commodity hardware that supports parallelism. On the other hand, [16] extends the Openflow switch architecture to have an application table – very similar in spirit to flow tables – where a specific application function could be called to be the performed action in this augmented application table. More details about this proposal will be given in Section 3 as this can be used for performance monitoring purposes.

2.3 NETWORK FUNCTION VIRTUALIZATION

Network Function is a functional block within a network infrastructure that provides a well-defined functional behavior [2]. Examples of network functions are intrusion detection, load balancers and firewalls. These network functions are traditionally implemented as custom hardware appliances, where software is tightly coupled with specific proprietary hardware. These physical network functions need to be manually installed into the network, creating operational challenges and preventing rapid deployment of new network functions. Therefore, network function virtualization has been proposed to provide more agile networks, with significant operation and capital expenses savings, by leveraging the virtualization technology to design, deploy and manage network functions and services. This means that network functions – such as firewall - can be implemented as an instance of plain software that is decoupled from the underlying hardware, and running on standardized compute nodes. A set of these Virtual Network Functions (VNFs) can be then composed to provide a network service.

In a typical NFV deployment, a network operator sets up an NFV infrastructure (NFVI). NFVI consists of servers, hypervisors, operating systems, virtual machines, virtual switches and network resources that host the VNFs. In addition, the network operator obtains VNFs from vendors that build the network function software, and installs the VNFs on the NFVI using the NFVI infrastructure manager (VIM), which controls the allocation of resources for the VNFs. OpenStack is an example of an open source VIM, controlling the physical and virtual resources for VNFs. Finally, the network is configured to correctly forward the packets along the VNF forwarding chain (i.e., the sequence of VNF components through which a request flows to implement a specific network service). In fact, these network functions can be interleaved with the original application components as they are now also at the VM layer, as depicted with the blue top boxes in Figure 2.1.

NFV and SDN have a lot in common since they both advocate for a passage towards open software and standard network hardware. However, SDN and NFV are different concepts, aimed at addressing different aspects of a software-driven networking solution. NFV aims at decoupling NFs from specialized hardware elements while SDN focuses on separating the handling of packets and connections from overall network control.

In fact, NFV and SDN may be highly complementary, and hence combining them in one networking solution may lead to greater value. For example, if it is able to run on a VM, an SDN controller may be implemented as part of a service chain. This means that the centralized control and management applications (such as load balancing, monitoring and traffic analysis) used in SDN can be realized, in network, as VNFs, and hence benefit from NFV's flexibility. Some examples of research work that combines SDN and NFV will be presented in Section 4.

3 LOGGING ARCHITECTURES

3.1 LOG MESSAGE

As mentioned before, cloud applications need to be monitored to detect performance bottlenecks. This monitoring could be done by instrumenting the end hosts where the application components are running, to generate log messages. These generated log messages are then used in processing application performance metrics and analyzing any detected performance issues. A log message is a text string with an abundance of contextual information about events that occur during run time, or messages traces between different communicating components of application. Some examples are as follows:

- System logs: produced by instrumenting the operating systems to monitor the health of an individual system's resources, and to detect system anomalies. System logs monitor CPU usage, disk I/O, network connectivity status and memory usage, for both the whole system and for each running process. In addition, system logs record events such as system errors, warnings, startup messages, system changes, abnormal shutdowns, etc.

- Application logs: produced by instrumenting the application component to provide insight into the operational status of an application and give a primary indicator of service failure. Logs of this type record informational, warning, error, and critical log entries. Informational entries are often related to general execution requests and times, or configuration settings. Warning log entries may result from improper or missing configuration. Error entries indicate that something failed to complete, however, the application will continue to operate. Critical failures indicate that the application encountered an unrecoverable error and will shut down or run in degraded mode. The following are instances of application logging for different application components:

1) *Web Access Logging*: logs about the web server requests which are very useful for reporting on website traffic. Web access log entries contain the IP address of the client, URL requested and resulting request response time.

2*) Database Logging*: typically generates three type of log files for performance monitoring purposes: A general query log that includes queries issued and the overall query time, slow query log for recording queries where execution time exceeds a specific threshold, and error log for logging errors like failed to start SQL.

3.2 ARCHITECTURE

Log messages need to be collected, aggregated and processed to calculate application performance metrics and provide insights into the application performance issues. In non-automated approaches, this is often performed manually or with some simple tools that check

the log files on a small number of servers. Given the growth and complexity of current cloud applications, this becomes more difficult. Hence, scalable automated log collection and processing is increasingly important. There are two main phases that need to be supported by a comprehensive logging architecture:

1) *The collection phase:* this phase is for the automatic collection of different log messages generated by the different hosts/ components of a cloud application. This phase is concerned about what, where and how to collect log messages and in which format according to the type of performance measures needed to be performed.

2) *Storage and analysis phase:* as the log data accumulates on multiple hosts, a more centralized powerful storage system is needed to be able to handle the growth in data over time. After that, the log data on the centralized storage system is processed and analyzed. Many of the application performance measurements mentioned before are calculated in this phase and performance issues are reported such as exceeding set thresholds or detecting specific patterns.

3.3 EXAMPLES

Most of the logging mechanisms available today are quite limited to collect unstructured data logs from application execution and hence still require significant manual effort to analyze them which makes this process hard and inefficient. Twitter [11] tackles this issue by proposing a *unified application logging format* for client events to simplify and enhance the efficiency of further data analytics. A Twitter client event is represented by hierarchical six-level naming which records the client, page, section, component, element and action of the event. For example, the following event, "web:home:mentions:stream:avatar:profile_click" is logged whenever there is an image profile click on the avatar of a tweet in the mentions timeline for a user on twitter.com. The authors showed that if all log messages created during the collection phase adhere to this unified client event log format, then this helps in the analysis phase by accelerating the execution of many statistics such as how often do users take advantage of twitter features and numbers of user sessions targeted to twitter per day.

SAAD [17] is another example of a logging framework that collects and analyzes logging messages. It's targeted at stage-based execution platforms for real-time anomaly detection. Stage-based architectures are commonly found in high performance servers where server code is divided into different smaller modules called stages. Each server stage can be executed simultaneously by many tasks that are queued in a task queue to be executed by different threads. During the collection phase, each task execution generates a task signature that consists of the ordered logging calls made by that task. In addition, SAAD leverages the staged code structure to track the start and end of each task and uses log statements as trace points to track the execution flow of the tasks during the run-time. During the analysis phase, SAAD exploits the statistical similarity of tasks to detect two types of anomalies: flow anomalies which are rare execution paths, and performance anomalies which represent unusually high execution times. For each stage, the tasks are grouped and counted based on their signatures, a percentile rank is determined in descending order for each signature and signatures with rank higher than a threshold are considered flow outliers. For example, if the threshold is set at the 99th percentile, signatures that account for less than 1% of tasks are considered outliers. Then, the 99%

percentile is calculated for task duration per each task signature group as well. The tasks with durations exceeding that threshold are considered performance outliers.

Both Twitter [11] and SAAD [17] are logging and monitoring architectures that rely on application-dependent logs for analysis. For Twitter, this is really the analysis of the log content, while SAAD uses similarity of logs for cluster purposes.

NFVPerf [10] is a logging platform that provides application component performance measures in NFV environments. NFVPerf sniffs packets on all communication paths between NFV components of the running application to compute per-hop throughputs and delays, and uses these measurements to identify both hardware and software performance bottlenecks. NFVPerf takes as input a configuration file specifying the VNF forwarding graph as well as logic to parse and identify different types of packet requests and responses between the VNF components. Using this information, NFVPerf computes the per-hop (and end-to-end) application-layer throughputs and delays along the VNF forwarding chain. By analyzing both application-specific message information and hardware resource utilization, NFVPerf can estimate the capacity of each component of the VNF, and identify performance bottlenecks both hardware and software related such as multiple threads contending for a lock. NFVPerf basically consist of two components: a local capture module that runs on every physical machine in the NFV Infrastructure (for example a private cloud), and a central analysis module (that can run anywhere in the operator's network). The local capture module collects the measurements and computes application-layer performance metrics. It passes them on to the central analysis module periodically where the bottleneck detection algorithm is executed.

4 SDN BASED LOGGING

SDNs are currently being deployed in large data center networks to provide a quick response to changing application requirements. Thus, SDN is considered a key element to implement network optimization algorithms such as traffic prioritizing, access control, and bandwidth management that are used to achieve the required Quality of Service (QoS) defined for the applications. Additionally, SDN has shown to have promising features to also provide support for performance monitoring. In recent research work, SDN switches and controller are instrumented to monitor either network parameters and/or application layer metrics. In the following two subsections, we are presenting some of this research. SDN based logging mechanisms can be seen to have the same two phases described in the previous section, where the collection of application performance metrics is done by instrumenting the SDN network (with or without end hosts' instrumentation), followed by a storage and analysis phase of the application performance metrics. Additionally, SDN-based mechanisms often exploit the SDN programmability features to reconfigure the network according to the analysis phase results.

4.1 NETWORK PARAMETER MEASURES

There are mainly two ways to measure network parameters, either passive or active [9]. With passive measurements, in-network monitors are installed to monitor application network traffic. In contrast with active measurements, additional packets in terms of probe packets are injected into the network to monitor their behavior.

OpenNetMon [9] uses both active and passive methods to measure network parameters. OpenNetMon instruments the SDN controller to continuously measure throughput, packet loss and delay for all flows between predefined link-destination pairs .The per-flow throughput and packet loss is derived from the queried flow counters (passive measurements). Delay, on the contrary, is measured by injecting probe packets directly into switch data planes, traveling the same flow paths (active measurement). To calculate the per-flow throughput, OpenNetMon receives the amount of bytes sent and the duration of each flow by polling the flow path's last switch on regular intervals. In addition, OpenNetMon estimates packet loss by polling flow statistics from the first and last switch of each flow path and then subtracting the increase of the source switch packet counter with the increase of the packet counter of the destination switch. Finally, OpenNetMon regularly injects packets at the first switch of each monitored flow path, such that that probe packet travels exactly the same flow path, and have the last switch send it back to the controller. The complete path delay is estimated as the difference between the packet's departure and arrival times, after subtracting the estimated latency from the switch-to-controller delays.

As another example of the SDN switch measurements, UMON [8] also uses the statistics to monitor network flows. However, UMON decouples monitoring from forwarding by defining a monitoring flow table in the openflow switch to separate monitoring rules from forwarding rules. Thus, users can freely install monitoring rules without worrying about the possible interactions with forwarding rules. In addition, UMON supports sub-flow monitoring. Subflows are defined as the fine-graind flows that belong to a mega-flow. For instance, individual flows from different source IPs to a destination host A are subflows of the mega-flow defined by the rule "Destination IP is A" with other routing fields being wildcarded. If packet counts of individual subflows need monitoring, then the aggregated packet count the SDN switch keeps track of for the mega-flow is not sufficient. Instead, UMON forces the switch to send the first packet from every individual subflow to the central controller which in turn installs the corresponding subflow rule into the SDN switch's monitoring table. The monitoring table will now keep track of the subflow packet count.

In an effort to protect SDN network traffic from security attacks, the authors in [7] propose Detection as a Service (DaaS). DaaS is an SDN application architecture that combines intrusion detection system (IDS) with SDN programmability features to passively detect anomalies in the traffic that passes through the SDN network and prevents malicious traffic from flowing into the SDN network. IDS is a standard defense solution for networks where network traffic is monitored in real time and compares the received packet patterns with known patterns to detect anomalies. In the collection phase, DaaS instruments SDN switches to mirror any packet that arrives to the first flow path switch and forwards the mirrored packet for analysis to a DaaS node where an IDS instance is running. During the analysis phase, the DaaS node in turn will decide whether the flow traffic is malicious or normal. A network reconfiguration will be needed if the DaaS node tags the traffic flow as malicious in order to block it. The malicious flow will be blocked by inserting a flow blocking entry into the switch flow table with the help of the SDN controller. Considering the amount of network traffic and the computation requirements, various DaaS units can be utilized for load balancing purposes and any clustering algorithm can be used to forward certain categories of traffic to respective DaaS nodes for analysis.

4.2 APPLICATION LAYER MEASURES

As mentioned in the previous subsection, SDN is mainly designed to monitor network traffic. However, enabling application-specific monitoring in the SDN data plane has many advantages. First of all, it eliminates the need of instrumenting the application to perform application performance measures as done in some of logging mechanisms mentioned in Section 3. Secondly, real-time application performance measures can be obtained at the network layer. Therefore, in this subsection, we will review some research work that extends the capability of the SDN data plane to perform such application performance measurements.

The authors in [12] address the challenge of associating network flows to the application components they connect or even the users who make the original requests. Typically, Openflow rules are used to define application policies. For instance, if an application policy needs to block the access of a certain user connected to a specific host, an Openflow rule can use the IP address of the user host and the port number of the application to implement this policy. However, if the targeted user moves to a different host, this rule doesn't longer satisfy the application policy. The authors address this problem by retrieving application–level information in real-time from the network.

The proposed framework is composed of two components: the augmented controller (AC) and the system probe (SP). The SP is instantiated at all hosts hosting users and applications (end host instrumentation). Its main role is to collect application-level information about traffic flows at these hosts. This information is then forwarded and stored in the database of the AC. The AC, the analysis node, takes as input application-level policies composed of rules. These rules are very similar to OpenFlow rules with additional fields to match application level information. The rule engine of the AC translates these rules to OpenFlow ones (IP addresses, protocols and port numbers) and forwards them to the OpenFlow controller and then to the switches. In fact, the overall architecture is more like the logging architectures discussed in Section 3, as end-host instrumentation for log collection is used while analysis uses again a dedicated node. The difference is that analysis is followed by a reconfiguration phase that affects the SDN layer.

The authors in [16] extend the SDN architecture by enabling customized packet handling in the SDN data plane switch. SDN switches in the proposed system are augmented to have application processing logic defined in a table called application table. Initially, the SDN controller installs application specific packet-processing actions in the augmented switches' application table. This augmented switches' application table is similar in spirit to the OpenFlow flow table. However, the application table actions are customized to be either ordinary Openflow API functions or specific application functions. Therefore, more sophisticated and application specific functions such as firewall and load balancer and/or new flow forwarding rules for incoming packets can be generated locally within the switch by executing application table actions for captured packets installed by the controller. Moreover, application chaining to implement network services (service chaining) can be achieved using the proposed system by ordering application actions for targeted flows. For example, a network service could require any web traffic to server X go through the firewall and then the load balancer. This service is implemented by inserting a rule to the switch application table with application actions that call firewall and then load balancer application functions. This proposed SDN modification decreases the communication and processing overhead between controller and switch as the

switch works autonomously. However, the performance impact on the switch is significant. The switch now needs to scan every packet. As a result, it can handle significant less packets per time unit than without application-specific functionality.

As mentioned before, the Openflow standard inspects network traffic based on layer 2-4 packet information (which are corresponding to network layer parameters). To monitor higher-level application parameters, the current Openflow structure needs to be modified. The following three proposals address this issue.

The authors in [15] extend the OpenFlow architecture to inspect not only the packet header but also the payload information in the packets. A set of predefined string patterns is inserted into the switch at the time of the switch initialization. If a packet matches any flow entry and its flag bit for Deep Packet Inspection (DPI) is set, then that packet is compared with the set of predefined string patterns before being forwarded. If the packet matches any of those patterns, a DPI module generates a log and sends it to the log queue in the switch. The log includes the information of both the matched pattern and flow. The collected logs are then sent to a log server for further analysis. The work has mainly two contributions: First, a DPI module is implemented in a virtual OpenFlow switch rather than a VM that runs on a hypervisor. This new approach shortens the distance from the source to the destination because there is no additional hop count to perform DPI. Analysis is handled by a dedicated log server instead of the controller. This means that there is no interference to the basic operation of the controller, the basic format of OpenFlow and the path of a secure channel between the switches and controller.

However, the proposed architecture in [15] has three limitations: First, if a predefined string pattern is fragmented and placed in two or more packets, a switch cannot detect this pattern because DPI inspects each packet individually. Second, if a switch detects two or more predefined string patterns in one packet, the switch cannot determine which policy of that packet or flow should be set. Third, if many packets in one flow match their payloads with a predefined string pattern, a switch has to individually send the log information related to those packets to the log server while the packets are still represented as the same packet flow. As an improvement, the switch could periodically send a log that represents all the matched packet information, but this method involves a trade-off between fast analysis and communication overhead. Similar trade-off can generally found for all monitoring information that is sent to a central service for analysis.

NetAlytics [13] integrates NVF with SDN to implement a general cloud application monitoring tool. Given the topology of each application that must be monitored within the cloud (that includes both the application components and their physical location in the network), NetAlytics can determine which network links need to be monitored and utilize NFV to deploy software-based network monitoring agents at hosts connected to these links accordingly. A control framework uses the SDN controller to instrument the switches attached to the hosts of deployed monitoring agents to duplicate and forward their captured flows to the monitoring agents. As they reside on the same link as the original host, this duplication does not cause a lot of overhead. Once data has been captured, it is aggregated and sent to a processing engine where the desired analysis on the data is performed. The monitoring agents can perform some minimal processing of captured data such as calculating the response time for HTTP web requests. The authors present two use cases of the NetAlytics framework: In the first use case NetAlytics is

used to diagnose high response times in a multi-tier web application by measuring per-tier response times calculated by the monitoring agents. In the second use case, NetAlytics is utilized to keep track of top-k popular YouTube video clips within a measurement interval. The role of the monitoring agents in this use case is to perform a deep inspection of client request packets to reveal the requested content and then forward this information to a powerful processing engine to execute "top-k" content popularity analysis in real-time.

Similar to NetAlytics, the authors in [14] use both NFV and SDN to deploy monitoring functions into a cloud infrastructure. In contract to NetAlytics, the main purpose of the deployed monitoring functions is to classify and tag packets along their path. For example, to monitor packet delays along its flow path, first the SDN controller receives application layer information from a DPI function deployed in the traffic path and maps it to layer 2-4 information. Using this information, the SDN controller configures one or more of the deployed monitoring functions to classify packets and tag them with timestamps and relative position (offset). More preciously, the timestamp and offset are inserted into a metadata field in one of the optional packet headers. Along the path, intermediate monitoring functions capture the tagged packets and record their system clock time in a respective timestamp tag position. The packet metadata timestamps are forwarded to the SDN controller when arriving its egress monitoring function. The SDN controller is the one responsible for further analysis.

SDN-PANDA [6] is a SDN-based experimental platform where anomaly detection algorithms can be plugged and deployed for SDN-enabled networks in real time. SDN-PANDA achieves the aforementioned goal through main three modules NMM, NID and NIRM. The network monitoring module (NMM) is responsible for collecting and preprocessing switch aggregated flow statistics sent to the SDN controller. The network intrusion detection module (NID) is a pluggable interface to deploy available flow-level anomaly detection methods. And the network intrusion response module (NIRM) is where the automated execution of the remediation polices for detected anomalies by NID is taken place with the help of the SDN controller. NIRM is completely independent on the anomaly detection technique used by the NID module to provide more flexibility.

5 OTHER APPLICATION LAYER FUNCTIONALITIES

There are many research efforts to take advantage of SDN and NVF technology to application layer functionalities. For example, NETKV [20] leverages NFV technology to provide an application-level load balancer for memcached servers. NETKV deploys a NVF-based middlebox in the network to automatically redirect packets to appropriate servers. NETKV bypasses the kernel and responses from servers are directly sent to clients. This enhances throughput and latency compared to a load balancer that is completely implemented as an application component.

The NETKV architecture consists of five main components: 1) the dispatcher interfaces with NFV platform to read packets and forward to the appropriate memchached server. 2) The key detector gathers statistics about requested content such as the frequency of each request and the estimated number of total requests. 3) The replication engine uses the statistics gathered by the key detector to determine which contents are hot. 4) The write coordinator serializes the write requests to the replicated data to ensure consistency between replicas. 5) The local cache

maintains the most popular and frequently updated contents with NETKV for even better performance. NETKV adjusts the replication factor for popular contents based on the predicted imbalance factor and changing workloads on predefined intervals. One can see that this is traditional application semantics that is now implemented as NFV for better performance.

Moreover, SDN can be used in other domains that need low latency and line-rate performance such as publisher/subscriber (pub/sub) middlewares. For example, PLEROMA [19] is a scalable SDN-based pub/sub middleware that uses the SDN data plan to achieve line-rate performance for content-based pub/sub applications.

6 CONCLUSION

In this report, we presented an overview of cloud-based application performance monitoring. Application logging mechanisms and resource monitoring tools are commonly used to troubleshoot application performance issues. However, the main logging mechanisms are limited by its need for application instrumentation at end hosts. Instead, by using new emerging trends such as SDN and NFV to support logging and collection, the network components can be instrumented as well to provide real-time performance measurements for the application flows. In this report, some of the state of the art SDN and/ or NFV-based cloud monitoring approaches are reviewed. Both application and SDN-based logging mechanisms have the same architecture, where performance related information is gathered during the collection phase at end hosts or in the network components. The gathered information is then transferred to a centralized processing and analysis node. Based on the results of the analysis phase, application reconfiguration may be executed in some of the discussed approaches such as [8] and [12].

REFERENCES

[1] Yosr Jarraya, Taous Madi, and Mourad Debbabi. “A Survey and a Layered Taxonomy of Software Defined Networking”. IEEE Communications Surveys & Tutorials, Volume: 16, Issue: 4, 2014.

[2] Rashid Mijumbi, Joan Serrat, Juan-Luis Gorricho, Niels Bouten, Filip De Turck, and Raouf Boutaba. “Network Function Virtualization: State-of-the-Art and Research Challenges”. IEEE Communications Surveys & Tutorials, Volume: 18, Issue: 1, 2016.

[3] Guilherme Da Cunha Rodrigues, Rodrigo N. Calheiros, Vinicius Tavares Guimaraes, Glederson Lessa dos Santos, Márcio Barbosa de Carvalho, Lisandro Zambenedetti Granville, Liane Margarida Rockenbach Tarouco, and Rajkumar Buyya. “Monitoring of cloud computing environments: concepts, solutions, trends, and future directions”. In Proceedings of the 31st Annual ACM Symposium on Applied Computing, April 04-08, 2016.

[4] Hiranya Jayathilaka, Chandra Krintz, and Rich Wolski. “Performance Monitoring and Root Cause Analysis for Cloud-hosted Web Applications”. In WWW '17 Proceedings of the 26th International Conference on World Wide Web, 2017.

[5] Hamid Farhadi, Ping Du, and Akihiro Nakao. “User-defined actions for SDN”. In CFI '14 Proceedings of the Ninth International Conference on Future Internet Technologies, 2014.

[6] Brian R. Granby, Bob Askwith, and Angelos K. Marnerides. “SDN-PANDA: SoftwareDefined Network Platform for ANomaly Detection Applications”. In IEEE 23rd International Conference on Ntwork Protocols (ICNP), 2015.

[7] Mehrnoosh Monshizadeh, Vikramajeet Khatri, and Raimo Kantola. “Detection as a service: An SDN application”. In 19th International Conference on advanced Communication Technology (ICACT), 2017.
[8] An Wang, Yang Guo, Fang Hao, and Songqing Chen. “UMON: Flexible and Fine Grained Traffic Monitoring in Open vSwitch”. In the 11th ACM Conference on Emerging Networking Experiments and Technologies, 2015.
[9] Niels L. M. van Adrichem, Christian Doerr, and Fernando A. Kuipers. “OpenNetMon: Network Monitoring in OpenFlow Software-Defined Networks”. In IEEE Network Operations and Management Symposium (NOMS), 2014.
[10] Priyanka Naik, Dilip Kumar Shaw, and Mythili Vutukuru. “NFVPerf: Online performance monitoring and bottleneck detection for NFV”. In IEEE conference on Network Function Virtualization and Software Defined Networks (NFV-SDN), 2016.
[11] R. R. Sambasivan, I. Shafer, J. Mace, B. H. Sigelman, R. Fonseca, and G. R. Ganger. “Principled workflow-centric tracing of distributed systems”. In ACM Symposium on Cloud Computing (SoCC), 2016.
[12] Lautaro Dolberg, Jérôme François, Shihabur Rahman Chowdhury, Reaz Ahmed, Raouf Boutaba, and Thomas Engel. “A generic framework to support application-level flow management in software-defined networks”. In IEEE NetSoft Conference and Workshops (NetSoft), 2016.
[13] Guyue Liu and Timothy Wood. “Cloud-Scale Application Performance Monitoring with SDN and NFV”. In IEEE International Conference on Cloud Engineering (IC2E), 2015.
[14] Meral Shirazipour, Heikki Mahkonen, Ming Xia, Ravi Manghirmalani, Attila Takacs, and Veronica Sanchez Vega. “A monitoring framework at layer4–7 granularity using network service headers”. In IEEE conference on Network Function Virtualization and Software Defined Network (NFV-SDN), 2015.
[15] ChoongHee Cho, JungBok Lee, Eun-Do Kim, and Jeong-dong Ryoo. “A Sophisticated Packet Forwarding Scheme with Deep Packet Inspection in an OpenFlow Switch”. In International Conference on Software Networking (ICSN), 2016.
[16] Hesham Mekky, Fang Hao, Sarit Mukherjee, Zhili Zhang, and T.V. Lakshman. “Application-aware Data Plane Processing in SDN”. In HotSDN '14 Proceedings of the third workshop on hot topics in software defined networking, 2014.
[17] S. Ghanbari, A. B. Hashemi, and C. Amza. “Stage-aware anomaly detection through tracking log points”. In ACM/IFIP/USENIX International Middleware Conference, pages 253–264, 2014.
[18] Omair Shafiq, Reda Alhajj, and Jon G. Rokne. “Handling incomplete data using semantic logging based Social Network Analysis Hexagon for effective application monitoring and management”. In IEEE/ACM International Conference on advances in Social Networks Analysis and Mining (ASONAM), 2014.
[19] Sukanya Bhowmik, Muhammad Adnan Tariq, Boris Koldehofe, Frank Dürr, Thomas Kohler, and Kurt Rothermel. “High Performance Publish/Subscribe Middleware in Software-Defined Networks”. IEEE/ACM Transactions on Networking, Volume: 25, Issue: 3, 2017.
[20] Wei Zhang, Timothy Wood, and Jinho Hwang. “NetKV: Scalable, Self-Managing, Load Balancing as a Network Function”. In IEEE International conference on Autonomic Computing (ICAC), 2016.