Cost Effective Troubleshooting of NFV Infrastructure

Ran Ben Basat*, Gil Einziger†, Maayan Goldstein‡, Liat Pele§, Itai Segall‡
*Harvard University
ran@seas.harvard.edu
†Ben Gurion University of the Negev
gilein@bgu.ac.il
‡Nokia Bell Labs
{itai.segal,maayan.goldstein}@nokia-bell-labs.com
§Nokia
{liat.pele@nokia.com }@nokia.com

***Abstract*—Fine-grained telemetry data enables troubleshooting teams to pinpoint the root cause of many faults in NFVI products. However, the volume of telemetry data rapidly accumulates, which makes it expensive to retain indefinitely. This work performs automatic analysis on the telemetry data, and only retains telemetry data if we suspect that it might contain a fault. Our evaluation uses 313 real application traces and shows that we can retain all faulty traces, and reduce the volume of stored data by over 99 %. We also show three case studies of real errors detected (and explained) by our system. Finally, we also deployed our system in a controlled testing environment and demonstrated its ability to detect microburst faults on a variety of network devices.**

I. Introduction

Network Function Virtualization (NFV) is a promising paradigm that brings the advantages of cloud management to modern networks, such as scale-up and on-demand deployment of network components, as well as popular software engineering trends such as continuous delivery. That is, the transition to NFV allows for rapidly addressing bugs and security vulnerabilities, as well as for the agile deployment of new features. However, network services are critical infrastructure whose clients expect a high degree of reliability, and thus each change in such networks requires extensive testing. Further, we need to pinpoint the root cause of emerging faults promptly.

Telemetry information such as per-application log files, virtual machines memory consumption, CPU consumption, network consumption, and other application-specific metrics is fundamental in pinpointing the root cause of faults. However, we may need to retain such information for extensive periods until faults become apparent. We face multiple problems in retaining telemetry information. First, the combined volume of such information accumulates quickly into tens of Gigabytes per day. Thus, we cannot retain telemetry for extended periods. For example, in our commercial cloud management infrastructure, telemetry information used to be kept only for a few days due to space complexity. Thus, we had no telemetry data to pinpoint their root cause by the time the faults became

Annex to ISBN 978-3-903176-28-7 ©2020 IFIP

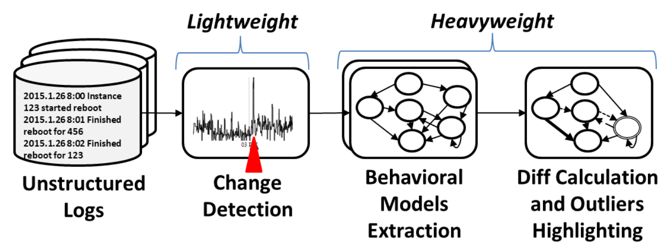


Fig. 1: An illustration of a single instance of our change detection architecture. A lightweight change detection algorithm monitors the generation rate of log files and triggers the heavyweight method if it suspects a change. The heavyweight method performs a deeper inspection, only retains the log files if it detects concrete behavioral change.

evident. In principle, we can never know precisely how long we should retain telemetry data.

Handling all the available telemetry data is also tricky since there are diverse applications and diverse data streams. It is unclear exactly how a fault is going to manifest in this richness of data. While we have expert systems such as [15], which are capable of analyzing some of the data streams, the performance of such systems is insufficient for large scale deployment. Thus, we require a system that can perform preliminary analysis on telemetry data to reduce the volume of stored data while missing as few faults as possible. Such a system should be agnostic of the functionality of the monitored application and the nature of the telemetry used.

Contribution: Our work is about reducing the volume of storage required to retain telemetry information for extended periods. That is, instead of storing all the telemetry data, we automatically perform a preliminary analysis and only retain a small portion (less than 1%) of the telemetry data. Our system analyzes multiple streams in parallel, and with acceptable CPU, memory, and network requirements.

In Figure 1, we explain how our method performs statistical analysis on log-file streams. First, we perform a *lightweight*

analysis on the log generation rate (without reading the actual log entries). Such an analysis requires very little memory and processing. For log-files, the lightweight method only analyzes the log generation rate without reading the traces. If the method detects a change in the log generation rate, it triggers the *heavyweight* method that performs an extended analysis on the logs. By focusing the heavyweight method only on short time intervals where we suspect a change, we fully automate the heavyweight method and drastically improve its performance. Finally, we only retain telemetry data if the heavyweight method detects a behavioral change.

Finally, we also use our lightweight method to detect "microbursts" of above second intervals (as opposed to microbursts in the literate that take sub-second intervals [6], [16]). In principle, our method can also work in sub-second intervals, but the overheads are currently considered impractical by our product teams. Interestingly, we still detect some traffic anomalies even with the above second resolution. Our method can be applied to a large variety of network devices from different vendors as it only relies on standard interfaces. In contrast, the existing micro-burst detection algorithms are suitable for specific network devices. We deployed our system as part of an extensive scale commercial NFVI system. Our evaluation includes three case studies detailing three real faults that were successfully detected by our approach. Finally, we show additional examples of patterns that appear in the telemetry data of network products and are successfully detected by our lightweight method.

II. System Description

This section describes our system, which is composed of multiple local change detection algorithm instances, each monitoring a single stream in a resource-efficient manner. We call these instances the *lightweight* method. Once such an algorithm suspects a change in the monitored stream (as we further explain in Section II-A), we escalate the process to a *heavyweight* method. Here, we offer two implementations. The *basic* method records telemetry data and then stores it for long-term storage. The *advanced* method also analyzes this data by automating the tool of [15] to detect behavioral changes. The advanced method retains the data only if it discovers a change. Thus, we further reduce the stored telemetry data. Section II-C describes the basic method, while Section II-E describes the advanced method.

A. Change Detection

We now introduce a new lightweight change detection algorithm that monitors a continuous stream of numbers and detects when this stream deviates from its 'normal' behavior. Such streams may represent telemetry data and such as CPU utilization, memory usage, network usage, log file generation rate, and messages sent. For generality, we treat all the data sources as streams of numbers that arrive at fixed intervals. We assume no prior knowledge of the functionality of the network components. For example, we do not know how many log entries are 'normally' generated, or what is the 'normal' packet rate distribution.

Since we do not know what is the normal behavior, we use a large *Lag* window as a baseline behavior. Similarly, we use a small *Lead* window to identify the current trends. For example, the Lag window can be about 8 hours, while the Lead can be 15 minutes.

Specifically we calculate the standard error on the Lead and Lag (approximate windows). To do so, we monitor the total sum (of all numbers) within windows (σ), and the sum of squared numbers (σ^2) . Whenever a new number arrives, we add it to both the Lead and Lag windows. We then estimate the standard deviation for both windows and denote it by S_{Lag} and S_{Lead}. Our lightweight method looks at the ratios between standard deviations on the Lead, and Lag windows. Specifically, we define:

$$E_1 = \frac{S_{Lag}}{S_{Lead}}, E_2 = \frac{S_{Lead}}{S_{Lag}}.$$

E_1, and E_2 provide us a notion of how much the distribution of numbers varies between the Lead and Lag windows. This notion is not enough to decide that something changed, as we do not know the 'normal' values of E_1, and E_2. However, E_1, E_2 provide us with informative values to monitor.

We monitor the maximum vales of E_1 and E_2 over the Lead window (excluding the Lag window), and use these values to determine reporting values for E_1 and E_2 that adjust to their current behavior. We denote the maximum of E_1, and of E_2 to be M_1, M_2. We then set the *reporting thresholds* as:

$$T_1 = \gamma \cdot M_1, \quad T_2 = \gamma \cdot M_2.$$

Notice that while E_2 is the inverse of E_1, we need both of them as E_1 is compared to M_1, and E_2 is compared to M_2.

We trigger the heavyweight method if the current $E_1 \geq T_1$ or if the current $E_2 \geq T_2$. Parameter $\gamma \geq 1$ is used to avoid multiple triggers if the maximum (M_1, and M_2) is insignificantly increased. We trigger the heavyweight method less frequently as we increase γ, but if we set it too high, we may fail to detect changes.

B. Efficient Implementation

Our work borrows ideas from the work of [7], to efficiently maintain its various estimators (E_1, E_2, M_1, and M_2) on sliding windows.

Instead of working with exact sliding windows, we work with variable-sized slack windows. For example, instead of maintaining the sum of all counters within a fixed 8 hours window, we can maintain it on a window, which is at least 7.5 hours, and at most 8 hours. Doing so allows us to sum together all the numbers within 30 minutes intervals, which drastically reduces the required space. For example, if we get a new number each second, then we read a total of 28,800 within an 8 hours window, and each of these numbers requires a single counter since we update the window for each second. However, by grouping numbers within 30 minutes intervals, we reduce the number of counters within an 8 hours window to

16. One per 30 minutes, we erase the oldest counter and begin filling it again with new numbers. Thus the result is a window size of 7.5-8 hours. This drastic reduction is at the key of our change detection method as it allows us to scale to a vast number of instances within adequate space. We further reduce the computational overheads by implementing the lightweight method with SIMD instructions to update multiple streams with a single vectored instruction.

Given the definition above, let us consider the following example that would give a feeling about the resources required to run our lightweight process. Say that we maintain the Lag window to be 7.5-8 hours and the Lead window to be 15-16 minutes. We thus require 32 counters to record standard deviation in the Lead window 16 for maintaining the sum on the window (σ) and 16 for maintaining the sum of squares (σ^2). Similarly, the Lag window requires 32 counters for the same reasons. We need an additional 32 counters to maintain E_1, and E_2 on the Lead window, and yet another 32 counters to record T_1, and T_2 totaling in 128 counters for the lightweight method. The lightweight method requires no additional memory. In contrast, an exact window would require at least 28,800 counters. That is, the slack window saves 99.5% of the memory required to maintain an accurate window of all the numbers. For ease of reference, Table I summarizes the notations used by the lightweight method.

Symbol	Meaning
σ_{Lead}, (σ_{Lag})	Sum of numbers in Lead (Lag) window.
σ^2_{Lead} (σ^2_{Lag})	Sum of numbers' squares in Lead (Lag) window.
S_{Lead}, (S_{Lag})	Standard deviation in Lead (Lag) window.
E_1, (E_2)	$E_1 = \frac{S_{Lead}}{S_{Lag}}$, $\left(E_2 = \frac{S_{Lag}}{S_{Lead}}\right)$.
M_1, (M_2)	Maximum value of E_1 (E_2) in the Lead minus Lag window.
γ	Sensitivity parameter for lightweight method.
T_1, (T_2)	Detection threshold $T_1 = \gamma \cdot M_1$,($T_2 = \gamma \cdot M_2$).
Detection Rule	Change is triggered when: $E1 > T_1$ or $E_2 > T2$.

TABLE I: A list of symbols and notations for the lightweight method.

C. Basic Heavyweight Method

When the lightweight detects a change, the basic response is to move telemetry data from the Lag window to long-term storage. Otherwise, telemetry data is only retained for a short while.

D. Micro-burst Detection Method

The NFV products that host our system use the Zabbix [5] open source project to maintain telemetry data. Zabbix periodically polls the virtual machines for telemetry and stores the collected data. However, collecting telemetry data at short intervals is expensive, and in practice, telemetry data is only collected in relatively large intervals that are too sparse to detect microbursts. While increasing the collection resolution is a feasible solution, the overheads are unacceptable. Therefore, we *selectively* increase the collection resolution for VMs, where the lightweight method detects a change. Such a compromise allows for detecting microbursts with minimal overheads.

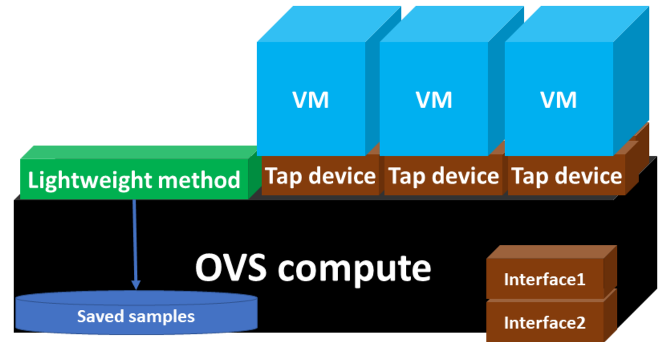


Fig. 2: An illustration of our deployment, the lightweight method (in green), is deployed inside an OpenStack compute, which it uses to store fine-grained telemetry at selected times. The lightweight method probes the OVS (or any other network device) and examines the log generation rate of the infrastructure's components.

1) Minimizing Overheads: We place the lightweight method in the same compute node as the monitored VM, which reduces the overheads of collecting and retaining fine-grained telemetry.

When a change in one of the metrics is detected, we locally retain the telemetry data on the compute node, and we only read the stored data when trying to pinpoint the root cause of problems. We chose to leave the fine-grained telemetry data on the compute nodes to prevent additional overheads in times when the system may be unstable (e.g., the bandwidth required for sending telemetry data to a centralized location).

That is, our telemetry data includes central sparse measurements, along with select periods of high-resolution measurements. Figure 2 illustrates our deployment. We perform microburst detection on multiple entities such as OVS [2] virtual switches and physical switches.

E. Advanced Heavyweight Method

In addition to the basic method, when the metric in question is a log file, we monitor the generation rate as it requires considerably fewer resources than reading the log. Log files are special as they have tools to structure and analyze them. However, such tools are semi-manual in the sense that we need to provide them with a suspected part of the log to compare to a 'normal' part of the log. Such separation is typically manually done when searching for faults. Our goal is to automate such capabilities to allow for quicker detection of faults, and additional compression.

Our prototype utilizes the technique of [15] that performs behavioral analysis on the log and identifies a deviation in the behavior of the normal and suspected logs. Most notably, it outputs graphical descriptions of the behavioral changes

it detects which are human-readable, and allow an expert to decide if there is a fault quickly.

While the technique is known, we use the lightweight method to automate it. We define the Lag window (minus the Lead window) as the normal behavior and the Lead window as the suspect behavior. This decision reduces complexity as we only run the method of [15] when we suspect a change. Even when we suspect a change, we run the tool on short traces.

For completeness, let us briefly summarize the workflow of the technique of [15]. The technique has four steps:

- Log normalization, where we normalize the log data by filtering out less important information, normalizing formats (e.g., field separators), extraction of identifier fields.
- Behavioral model extraction for normal and abnormal logs, collected during the Lead and Lag windows, respectively. In this step, we build a model for each window. Such a model is a finite state automaton that captures the behavior witnessed in the log. We use existing spec-mining techniques in this step, e.g., kTails [9].
- Computation of differences between the two models that identify behavioral changes between the logs. Examples of extracted differences are states encountered in one model but not in the other, differences in frequencies at which each behavior occurs, as well as significant changes in the transition time between states.
- Visualization of the extracted differences and highlighting of substantial artifacts.

In our system, once invoking the method of [15], it becomes responsible for deciding if the log files should be stored long term. That is, if the method detects a behavioral change, it saves the files; otherwise, it discards them to prevent excessive storage usage.

III. Evaluation

This section evaluates the lightweight method and the advanced heavyweight technique using 313 real commercial systems traces. We ran the experiments on an HP EliteBook laptop, with 16GB of RAM, i7-6600U 2.6GHz CPU, and a 64 bit Windows 10 OS. All of the experiments in this section use existing logs collected in the past. Therefore, our system expert already knows which errors appear in the logs and their timing. We sent the results of our analysis to the experts without knowing in advance where the errors are.

We set $\gamma = 1.4$ in all evaluations; the window sizes are determined by the granularity of the data. Specifically, in log analysis, we need to make sure that there are enough log lines to generate a behavioral model for the Lead window. The Lag window is set to be 8-48 times larger than the Lead (the system seems indifferent to the exact sizing of the Lag window) e.g., 30 minutes and 24 hours or 5 minutes and 200 minutes. The exact details are provided for each experiment.

We start this section by evaluating the compression rate and the false positive rate of our system (Section III-A), then we continue with two case studies of real faults that were detected by our system(Section III-B and Section III-C. Following, we show interesting examples of patterns (and possibly faults) that were detected by our system III-D but were unknown to our system experts.

A. Data Reduction and False Positive Evaluation

We start by evaluating the impact our system has on the storage requirements of telemetry data. We run our system on 313 application traces containing a total of 30,392,300 log lines. Our lightweight method detected 3674 changes, about one change per 80k log lines. Out of these changes, two changes were real faults known to our system experts, and the rest were either false positives or errors that went unnoticed. That is, the false positive ratio is 99.95%. However, our method is very successful as a compression tool as we store less than 1% of the original logs and retain the telemetry from the real faults.

We run the advanced heavyweight method on a subset of these traces for two weeks of the MANO system's log. In this subset, there were 114 detected changes, and the heavyweight method narrowed the number to 97 without configurations, and to 41 when eliminating changes that appear more than once. In essence, once the system expert reports that a change is not a fault, we can dismiss that change if we re-encounter it. Unfortunately, even with this optimization, we require the system expert three times a day in a properly working system, which is still unacceptable. That is, we conclude that our system is still not efficient enough to detect faults in the infrastructure proactively.

B. Authentication Failure in an NFVI system

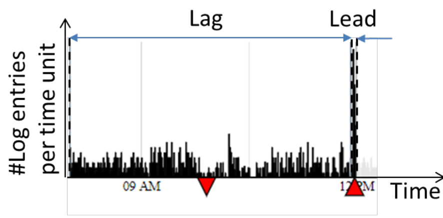
The first system is an OpenStack aligned, NFV infrastructure that implements NFVI and VIM support. The infrastructure is composed of multiple OpenStack and proprietary projects [1]. One such project is Gnocchi [1], which performs storage and indexing of time series data and resources at a large scale. The Gnocchi subsystem maintains an extensive log that we collected during the operation of the NFVI.

Figure 3 shows an example of analyzing Gnocchi logs. Here, the Lead window is 5 minutes, and the Lag is 4 hours.

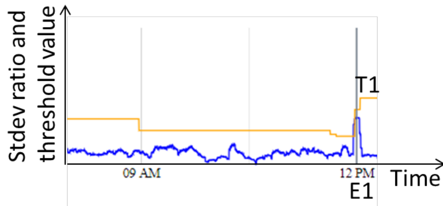
In this example, the lightweight method triggers the advanced heavyweight method at around 10 AM and 12 PM. In this case, the heavyweight method dismisses the 10 AM change, and only stores the logs of the 12 PM incident. We used a system expert to verify that there is indeed no fault at 10 AM and that there is a real fault at 12 PM. At that time, there were many connection failures to the authentication server. The root cause of such failures was that the server was down, and Gnocchi repeatedly tried to authenticate a tenant.

The next two graphs in Figure 3, demonstrate the internal state of our lightweight method. Recall that the method uses the ratio of standard deviations on the Lead and Lag window, where E_1 is the standard deviation of the Lead window divided by that of the Lag, and E_2 is the other way around. Notice that the 10AM change is due to the pair E_1, T_1, and that of 12PM is due to the pair E_2, T_2. At

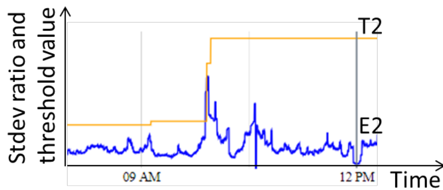
[1]The system's name is omitted to preserve authors' anonymity.



(a) Number of log entries per second in Gnocchi.



(b) Ration between standard deviations E_1 and the reporting threshold T_1.



(c) Ration between standard deviations E_2 and the reporting threshold T_2.

Fig. 3: Detected changes for Gnocchi logs.

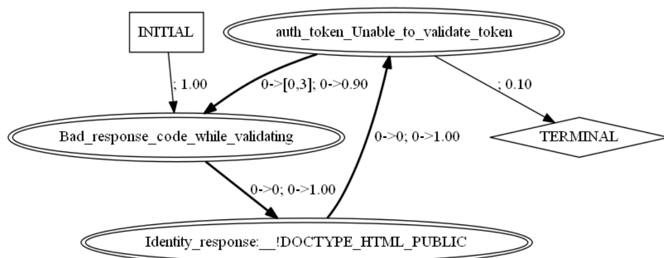
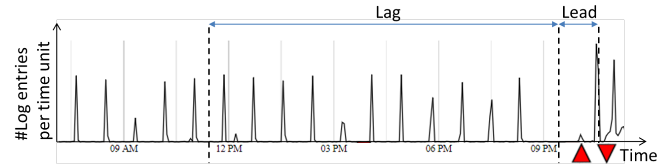
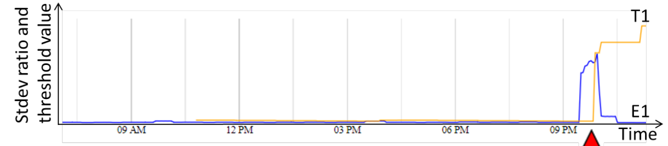


Fig. 4: Diff model with emphasized outliers for Gnocchi logs.

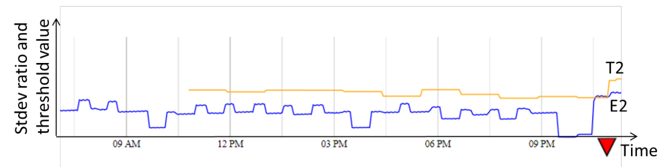
around 10 AM, the log generation rate drops, which triggers the advanced heavyweight method as $E_1 > T_1$. That is, we run the method of [15] and automatically configure it according to the Lead and Lag windows. Since we detect no behavioral change, we dismiss the change and continue as usual. At about 12 PM, the log generation rate spikes, which causes a second triggering since $E_2 > T_2$. Again, we trigger the (automatically configured) heavyweight method to read the logs and search for behavioral changes. This time, we discover a behavioral change, so we store the logs, and (potentially) alert the user with a graphical explanation of the change. The visualization is found in Figure 4. In this



(a) Number of log entries per minute in management system logs.



(b) Ration between standard deviations E_1 and the reporting threshold T_1.



(c) Ration between standard deviations E_2 and the reporting threshold T_2.

Fig. 5: Detected changes for management system logs.

visualization, the INITIAL and TERMINAL nodes are nodes representing entrance and exit from the generated model. All other nodes represent states with edges acting as transitions. We automatically extracted the field names from the log messages. Observe that there are three states (in double-lined ovals) *auth_token_Unable_to_validate_token*, *Bad_response_code_while_validating*, and *Identity_response:_!DOCTYPE_HTML_PUBLIC* that occur during the Lead window, but not during the Lag window. Thus, the fault manifests by a large number of these error messages appearing in the log (which initially only affects the Lead window). It is not hard to deduce the real problem from this graph, which is a failure of the authentication server. Specifically, our system expert considers this visualization informative.

In summary, our system processed 53.4K Gnocchi log lines. The lightweight method detected a change twice, and the advanced heavyweight method dismissed one of these changes (while the other contains a real fault). That is, out of all the 53.4K logs, we only store the logs of a single point in time. All in all, we retain less than 1% of the Gnocchi logs.

C. A slowdown in a telecommunication operator's system

The next system is a large telecommunication system that manages networked services at scale. The system collects log files from numerous services and devices. When a fault becomes apparent, the troubleshooting team searches these traces to pinpoint the error, and determine its cause. This manual process may require a long time (measured in days or even weeks) to resolve, and includes many work hours of system experts to sift through numerous large volume logs.

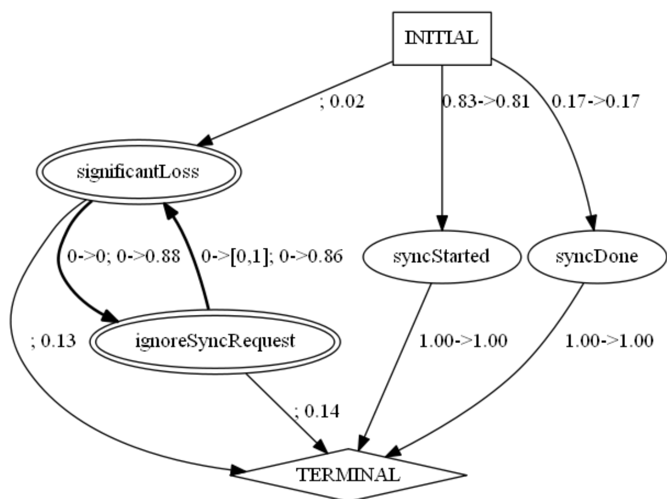
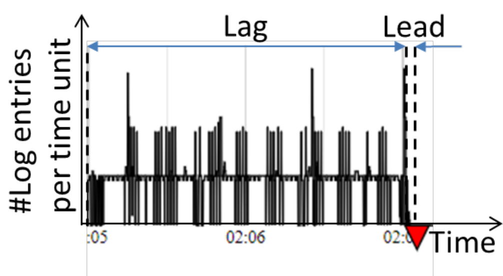
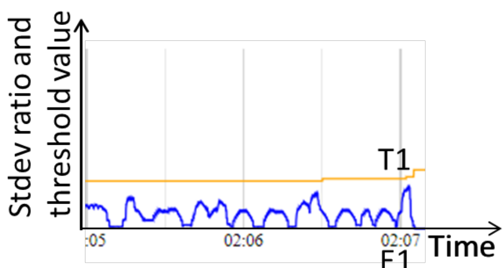


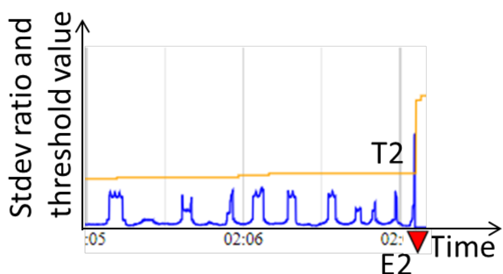
Fig. 6: Diff model with emphasized outliers for management system logs.



(a) Log entries per second in a Web Server Gateway Interface.



(b) Ration between standard deviations E_1 and the reporting threshold T_1.



(c) Ration between E_2 T_2.

Fig. 7: Results for Web Server Gateway Interface (WSGI).

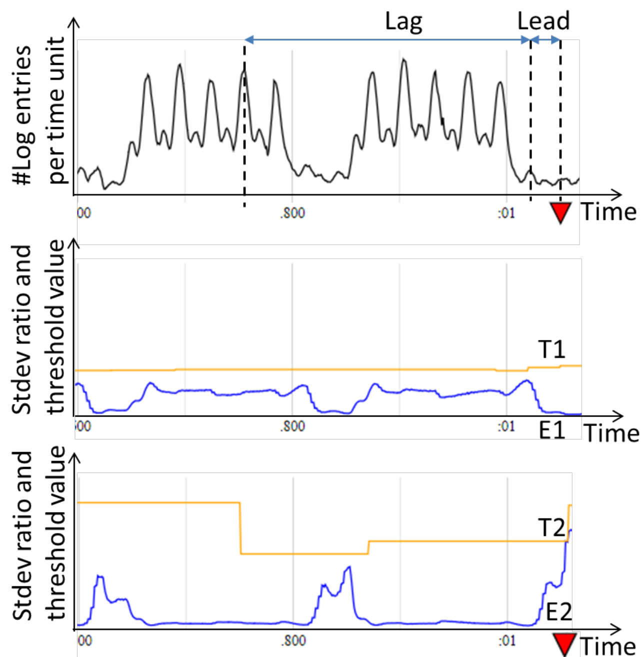


Fig. 8: Alerts generated for logs of a Security Application. The upper most graph shows a periodic pattern that does not trigger an alert. The two graphs below show the E_1 and E_2 ratios, and their corresponding thresholds.

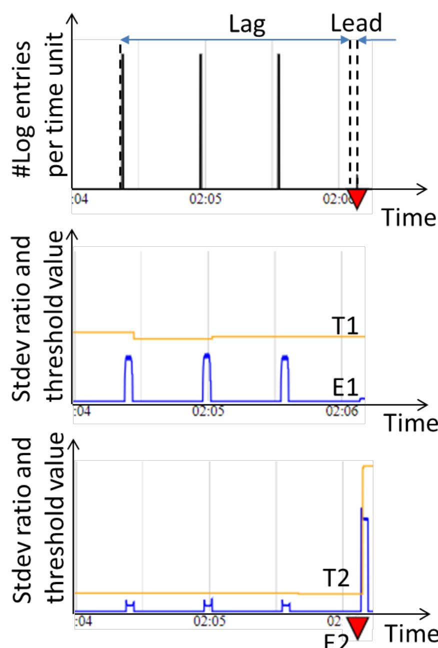


Fig. 9: Alerts generated for logs of a Mail application. The uppermost graph shows a periodic pattern that does not trigger an alert. The two graphs below show the E_1 and E_2 ratios, and their corresponding thresholds.

Figure 5(a) shows the log generation rate, window sizes, and detected changes. Here, the Lead window is 20 minutes, and the Lag is 200 minutes. We determined these numbers by the log generation rate as we need enough log lines in the Lead window to generate a reliable behavioral model. As can be observed, at around 10 PM, we detect two changes. Figure 5(b), and Figure 5(c) shows the internal state of our lightweight algorithm including the thresholds T_1 and T_2 and the estimators E_1 and E_2. The first detected change is caused by the pair E_1, T_1 as the log generation rate in the Lead window is especially low, which reduces the standard deviation of the Lead window. This in turn causes E_1 to drastically raise and cross T_1 (See Subfigure (b)). Next, the log generation rate steeply increases, and the standard deviation of the Lead window is higher than that of the Lag window.

We verified with the system's experts that there indeed is a real error around the detection time (our system pinpointed the beginning of the error). The error is a slowdown of the system, which explains the first change. However, the heavyweight method does not yet find a behavioral change since the log lines that describe the error were not yet written. Luckily, a change is also detected shortly after due to a spike in the log generation rate. This spike is due to numerous timeout messages, which allows our heavyweight method to discover a behavioral change. Figure 6 shows the visual explanation of the behavioral change. As illustrated, two states occur only in the Lead window, *significantLoss* and *ignoreSyncRequest*. This alarm captures the root cause of a real error, which was verified by the system's expert. The failure is due to the system trying to get a response from some devices that are not responding correctly. The failure triggers a synchronization message. When the synchronization fails, it records a failure message and repeats this process. This excessive repetition creates a loop between these two states, and that loop enabled our lightweight method to trigger a change in the first place.

The visualization emphasizes significant changes, whereas the threshold to determine if a change is significant is: $\frac{abs(freq_{Lag} - freq_{Lead})}{freq_{Lag} + freq_{Lead}} > 30\%$. Here, $freq_{Lag}$ and $freq_{Lead}$ are frequencies of a certain transition in the logs of the Lead and Lag windows.

The time it took us to execute the behavioral analysis was less than 2 seconds. In contrast, the authors of [15] report running the analysis tool for 3 hours on the entire trace (2.5M loglines). That is, we optimize the behavioral analysis tool by running it on short traces.

D. Exploring the lightweight system

Next, we evaluate our lightweight method on other applications where we have no access to system experts. We use this evaluation to see what kind of periodic behaviors are captured by the lightweight method. Figure 7 shows the result of running the lightweight method for analyzing the log generation rate of a Web Server Gateway Interface (WSGI) application, which is a part of OpenStack [4]. We select the Lead window's size to half an hour and the Lag's window size to one day. We chose these window sizes due to the expected day/night cycle of such services. Note that despite the noisy behavior of the log generation rate, the lightweight method does not trigger the heavyweight method. The triggering at the end is because the log generation rate drops to 0, likely due to a crash.

In Figure 8, we monitor the entry logs of a security application. We cannot provide many details on this application due to confidentiality requirements specified by the customer. We configured the Lead window to be half an hour, and the Lag window to be 5 hours. The reason for such configuration is the generation rate, which requires half an hour to accumulate enough log entries for the log analysis tool. We set the Lag window to be ten times the Lead as a rule of thumb.

Notice that the generation rate shows a periodic behavior with a non-trivial pattern that resembles fingers. The detected change is because the pattern changes at the end of the measurement. We do not know if there is a fault in this example but note the ability to cope with such patterns gracefully.

The last example in Figure 9 shows the log generation rate of a mailing service that has a clear and straightforward periodic pattern. Here, the Lead window is half an hour, and the Lag window is one day. These values are not optimized for this trace, and we repeated the values we selected for Figure 7. As can be observed, the mail log does not write the log at the expected interval (likely due to a crush), and our lightweight method identifies this case.

IV. Live Deployment

In this section, we show experiments with the basic heavyweight method that is running alongside a cloud Management And Orchestration system (MANO). This system utilizes OpenStack and other open-source projects.

Here, there is a desire to save fine-grained telemetry data from a large number of virtual machines, but doing so slows down network services, which is unacceptable by our users. The compromise is to use Zabbix for telemetry in one-minute intervals. Such intervals are too coarse to detect microbursts, which lead us to implement a single instance of our lightweight method for each OpenStack compute node. Such a location allows for fine-grained monitoring without external communication, which reduces the overheads to acceptable levels.

Our lightweight method monitors numerous metrics, but the most useful which we present here is the *Packets Per Second (PPS)*. Specifically, faults in our products are typically due to microbursts, which are invisible to sparse telemetry but may cause unexpected behavior. For example, losing some packets due to a burst may imply that we lose heartbeat packets, which may cause our clients to restart their services automatically. In such cases, it is essential to understand if the problem is part of the service (which means that the client should solve it), or if it is part of the infrastructure (which means that our troubleshooting team should explain it). That is, the detection of microbursts allows us to take responsibility for some problems and benefit our clients.

Since performance is crucial when we leverage Single Instruction Multiple Data (SIMID) [3] instructions when im-

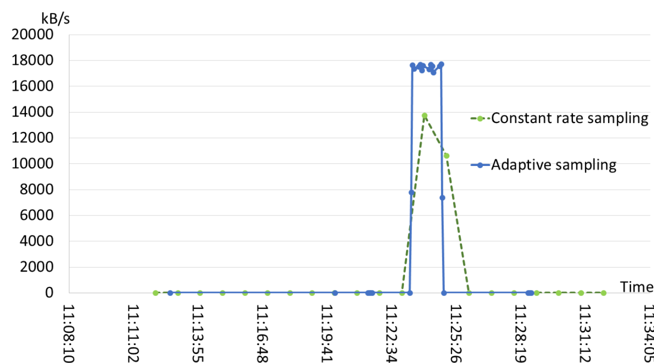


Fig. 10: *Our lightweight method (blue), in comparison to the existing coarse grain telemetry through Zabbix, when we artificially increase the packet rate for a short amount of time.*

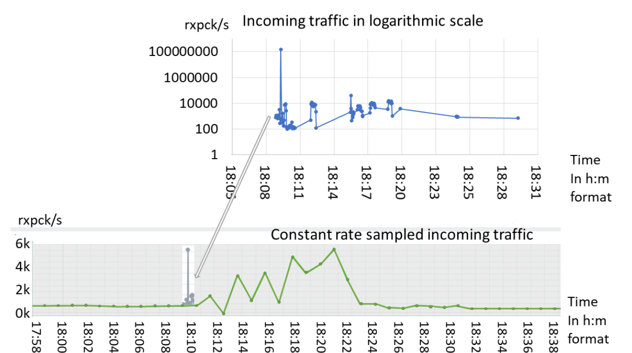


Fig. 11: *Microbursts missed by Zabbix's coarse grained monitoring (in green), but are detected and stored by our system (in blue). The microbursts were detected at the point marked grey in the bottom graph.*

plementing the lightweight method, such instructions allow us to update multiple instances at once. Here, the lightweight method saves fine-grained telemetry data when it detects a change in PPS, compared to the baseline. Doing so allows us to figure out what caused the microbursts, especially when they are the root cause of problems.

A. Synthetic example

Our first experiment is a demonstration of the capability to detect a rise in traffic accurately. Figure 10 shows results when we artificially increase the traffic. The green line is the default Zabbix coarse-grained sampling, while the blue line shows that our method accurately identifies the increases in packet rate. The reported starting and ending times of the increased rate are almost identical to the real times.

1) Overheads: Our lightweight methods maintains a window of 500 seconds and records a single measurement per second (60 times faster than Zabbix). The overheads for all the instances of the lightweight method are an increase of 0.5% in one of the CPUs, which we consider acceptable. In contrast, sampling at fine granularity all the time increases these overheads by a factor of x10. The memory overheads of our solutions are fixed, and negligible compared to the amount of RAM in the servers. Specifically, the entire overheads including containers are measured in tens of megabytes.

B. Detecting microbursts at an End-to-End 5G lab

The 5G lab is the final and most realistic testing that the MANO system undergoes before being released into production. Thus, experimenting in this lab allows us to test our system within a very realistic deployment. In Figure 11, we show results for a three days period where the QA teams tested the product while our system was active. The bottom part of the figure shows the one-minute interval of Zabbix data, which is collected regardless of our system. The upper part of the figure is the fine-grained measurement data, which we collected at the point marked with a line. As can be observed, the default one-minute measurement fails, while our system detects a series of microbursts where the traffic momentarily increases by x10 to x1000 of the average.

V. Related work

Detecting faults in network components is a well-studied issue. Specifically, the work of [11] attempts to identify SDN switches that were compromised by an attacker. Alternatively, the work of [14] attempts to detect attack patterns in the traffic. In comparison, our work may identify the behavioral changes of a compromised switch, but it is more general and more modest. It monitors a variety of network components, and its primary goal is to retain enough information to detect faults.

The work of [22] processes a metric over a time series and indicates how normal it currently is. This approach compares its past patterns on two identically sized time frames called Lead and Lag windows. Their work lacks a mechanism to determine when to trigger an alarm, and they offer no action to do once a change is detected. Further, they require some meta-knowledge of the problem by determining the length of the monitored words.

Alternatively, in [19], machine learning models are trained to try and predict the next metrics value, and the anomaly score is determined by how accurately these models predict future value. The assumption is that in normal operation, the predictions would be relatively accurate, but once an anomaly occurs, they'll be inaccurate. However, this approach behaves poorly if the behavior gradually changes over time.

Analyzing logs to understand service behavior and to find anomalies in that behavior is also a well-researched field [13], [23], [20], [18], [12], [21], [17]. Some [21], [12], [20] focused on developing statistical approaches to visualize logs and detect outliers. These approaches monitor frequencies of fixed length patterns, and the most infrequent patterns are considered anomalies. Others [18], [23] adapted machine learning algorithms to achieve a similar goal.

Finally, microbursts are a known pain-point for networked systems. They often cause increased delays and packet loss in the link level [24], and one can remedy the impact of

microbursts by using larger buffers, or by dropping the packets of the largest flows to limit the impact to as few flows as possible [10]. It is challenging to measure microbursts in traditional methods due to their short duration [6], [8], which is often less than a second (e.g., 100 microseconds) [10], [24].

The existing approaches for handling microbursts assume unrestricted access to the internal data of network devices [10], [24], [16]. Such access is sadly impractical when you need to support a large variety of network devices. Our method only requires very basic access to the network device but can only detect ”microbursts” that are measured in seconds. While less than ideal, we still catch some of the problems.

VI. Discussion

Our system reduces the volume of telemetry data, which is retained indefinitely and is required to pinpoint the root cause of faults in NFVI products. Our system discards most of the data and retains the data from incidents where some automated analysis process detects a change. We presented a new lightweight algorithm designed to perform an initial analysis within minimal space and processing overheads. Then, we reduce the amount of stored data by automating existing log-analysis tools to inspect the triggers of the lightweight method.

Our evaluation includes multiple case studies and 313 real application traces. In these traces, we demonstrate a reduction of over 99% in log file storage, without missing out on any of the (few) real faults. Such a reduction extends the storage time of telemetry information from one day to over three months without allocating additional storage for telemetry data. Such an increase makes it easier to pinpoint the root cause of problems when they become apparent.

Our approach detects all of the real faults within the telemetry data that we currently have, but the number of such faults is not big enough to reason about false negatives. We will be able to evaluate this ratio more accurately in the future as our system would encounter additional real faults.

Many of the common faults in our products involve the formation of microbursts, and we show that our lightweight method can detect such microbursts without requiring non-standard interfaces. The usage of such interfaces is important as it allows our solution to monitor a large variety of network devices, whereas existing works target specific devices with specific programming models [6], [24], [16]. Due to performance limitations, our resolution is limited to above second microbursts (which differ from sub-second microbursts in the literature). We performed a controlled experiment in realistic conditions on our 5G lab and showed that detecting such microbursts improves our current capabilities. In the future, we will seek ways to reach subsecond intervals.

We are currently evaluating applications of evaluating metrics such as process level memory and CPU usage. We also consider comparing our estimators with the method of of [22], since their approach does not determine the threshold automatically, we will use it to replace E_1 and E_2 while retaining the rest of our architecture unchanged.

Acknoledgements The authors thank the anonymous reviewers and our shepherd, Stefano Secci. This research was partially funded by the Ben-Gurion University Cyber-Security Research Center and the Zuckerman institute.

References

[1] Gnocchi – metric as a service. https://wiki.openstack.org/wiki/Gnocchi, 2018.
[2] Open vSwitch. https://www.openvswitch.org/, 2019.
[3] Single instruction, multiple data (SIMD). https://en.wikipedia.org/wiki/SIMD, 2019.
[4] Web Server Gateway Interface (WSGI). https://docs.openstack.org/nova/pike/user/wsgi.html, 2019.
[5] Zabbix. https://www.zabbix.com/, 2019.
[6] Mina Tahmasbi Arashloo, Yaron Koral, Michael Greenberg, Jennifer Rexford, and David Walker. Snap: Stateful network-wide abstractions for packet processing. In *ACM SIGCOMM*, pages 29–43, 2016.
[7] Ran Ben Basat, Gil Einziger, and Roy Friedman. Give Me Some Slack: Efficient Network Measurements. In *MFCS*, 2018.
[8] Ran Ben-Basat, Gil Einziger, Roy Friedman, Marcelo Caggiani Luizelli, and Erez Waisbard. Constant time updates in hierarchical heavy hitters. In *ACM SIGCOMM*, 2017.
[9] Alan W. Biermann and Jerome A. Feldman. On the synthesis of finite-state machines from samples of their behavior. *Computers, IEEE Transactions on*, C-21(6):592–597, June 1972.
[10] Xiaoqi Chen, Shir Landau Feibish, Yaron Koral, Jennifer Rexford, and Ori Rottenstreich. Catching the microburst culprits with snappy. In *Proceedings of the Afternoon Workshop on Self-Driving Networks*, SelfDN, 2018.
[11] Po-Wen Chi, Chien-Ting Kuo, Jing-Wei Guo, and Chin-Laung Lei. How to detect a compromised sdn switch. In *IEEE NetSoft*, April 2015.
[12] Adrian Frei and Marc Rennhard. Histogram matrix: Log file visualization for anomaly detection. In *2008 Third International Conference on Availability, Reliability and Security*, 2008.
[13] Qiang Fu, Jian-Guang Lou, Yi Wang, and Jiang Li. Execution anomaly detection in distributed systems through unstructured log analysis. In *IEEE International conference on Data Mining (full paper)*, 2009.
[14] Kostas Giotis, Georgios Androulidakis, and Vasilis Maglaris. Leveraging sdn for efficient anomaly detection and mitigation on legacy networks. In *2014 Third European Workshop on Software Defined Networks*, 2014.
[15] Maayan Goldstein, Danny Raz, and Itai Segall. Experience report: Log-based behavioral differencing. In *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*, 2017.
[16] Raj Joshi, Ting Qu, Mun Choon Chan, Ben Leong, and Boon Thau Loo. Burstradar: Practical real-time microburst monitoring for datacenter networks. In *APSys*, 2018.
[17] Karthik Nagaraj, Charles Killian, and Jennifer Neville. Structured comparative analysis of systems logs to diagnose performance problems. In *USENIX NSDI*, 2012.
[18] Thomas Reidemeister, Miao Jiang, and Paul AS Ward. Mining unstructured log files for recurrent fault diagnosis. In *12th IFIP/IEEE IM*, 2011.
[19] Dominique Shipmon, Jason Gurevitch, Paolo M Piselli, and Steve Edwards. Time series anomaly detection: Detection of anomalous drops with limited features and sparse examples in noisy periodic data. Technical report, Google Inc., 2017.
[20] Tuomo Sipola, Antti Juvonen, and Joel Lehtonen. Anomaly detection from network logs using diffusion maps. In Lazaros Iliadis and Chrisina Jayne, editors, *Engineering Applications of Neural Networks*, 2011.
[21] Risto Vaarandi. A data clustering algorithm for mining patterns from event logs. In *Proceedings of the 3rd IEEE Workshop on IP Operations Management (IPOM 2003) (IEEE Cat. No.03EX764)*, 2003.
[22] Li Wei, Nitin Kumar, Venkata Lolla, Eamonn J. Keogh, Stefano Lonardi, and Chotirat Ratanamahatana. Assumption-free anomaly detection in time series. In *Proceedings of the 17th International Conference on Scientific and Statistical Database Management*, SSDBM'2005, 2005.
[23] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael Jordan. Mining console logs for large-scale system problem detection. In *ACM SysML*, 2008.
[24] Qiao Zhang, Vincent Liu, Hongyi Zeng, and Arvind Krishnamurthy. High-resolution measurement of data center microbursts. In *ACM IMC*, 2017.