Near-Realtime Server Reboot Monitoring and Root Cause Analysis in a Large-Scale System

Fred Lin, Bhargav Bolla, Eric Pinkham, Neil Kodner, Daniel Moore, Amol Desai, Sriram Sankar
Facebook Inc.

***Abstract*—Large-scale Internet services run on a fleet of distributed servers, and the continuous availability of the hardware is key to the robustness of the services. Unplanned reboots disrupt the services running on the hardware and lower the fleet availability. Server reboots are also important signals that could indicate underlying issues such as memory leaks from the services, catastrophic hardware failures, and network or power disruptions at the datacenters.**

In this paper, we present an at-scale, near-realtime reboot monitoring framework built with multiple state-of-the-art data infrastructures, as well as machine learning-based anomaly detection and automated root cause analysis across hundreds of server attribute combinations. We observed that 1% of the reboots in our hardware fleet were associated with kernel panics and out-of-memory events, and these reboots exhibit strong locality temporally and across services

I. Introduction

Server reboot is a critical signal when monitoring the health of a distributed infrastructure. Unplanned server reboots are disruptive to the services and could indicate underlying service or hardware issues. For example, memory leaks from a service may lead to out-of-memory (OOM) or kernel panic incidents, which could cause the server to crash and require a reboot to recover [1]. Catastrophic hardware failures such as a CPU or network interface card (NIC) failure could cause the server to reboot or lose connection to the rest of the network. Additionally, a group of servers going offline could be a signal of an unexpected power or connectivity issue that impacts the Internet service. From the service's perspective, server reboot and offline events imply server downtime, which lowers the fleet availability for running the service [2].

While some reboots are associated with tooling or server log events, reboots triggered by power or network disruptions may be completely unexpected and not correlate to any of the tooling or server logs. In order to detect all types of reboots, we use server *uptime* as the ground truth for the detection. As some reboots are needed for the normal maintenance of the servers such as a firmware or kernel upgrade, a provisioning job, and a hardware repair work, we further categorize the reboots as *planned* or *unplanned*.

In this paper, we present a production framework for monitoring all possible server reboot and offline events by the cause, with the integration of automated anomaly detection and root cause analysis at scale and in near-realtime. From the production data we found that 89% of the reboots in our hardware fleet were planned, while 10% of the reboots were due to hardware repairs. The other 1% of the reboots were associated with OOMs and panics, and these reboots are highly correlated temporally and to the services that are running.

Following the reboot detection and categorization, a machine learning-based anomaly detection [3] is run for monitoring the spikes and slow drifts in the reboot rates across hundreds of dimensions. A fast dimensional analysis presented in [4] then automatically finds the most relevant server attribute combinations for explaining the reboots at scale. For example, the framework has been able to detect a spike of server reboots and correlate it to a specific main switchboard in a datacenter.

The rest of the paper is organized as the following: Server reboot categorization and datasets are discussed in Section II. The reboot detection and categorization framework and the field data are presented in Section III. Section IV and Section V explain the setup of the anomaly detection and root cause analysis, respectively. Section VI concludes the paper.

II. Server Reboots in a Large-Scale System

A. Reboot Categorization

In production we categorize reboots as either *planned* or *unplanned*. In addition to the unplanned reboots initiated by the server itself, the hardware auto-remediation tools [5] can also trigger reboots through out-of-band (OOB) management if the server is detected to be unreachable through in-band connection such as Secure Shell (SSH). In this scenario, we also consider the reboot to be unplanned.

Planned reboots can usually be executed *gracefully*, in which case all services are shut down properly as designed. More often observed from unplanned reboots, ungraceful shutdowns may corrupt the data on the servers. With fault tolerance mechanisms, the corrupted data might be recovered but the recovery process is typically time-consuming.

B. Data Sources for Server Reboot and Offline Events

The logs that record the reboot-related events are stored either on- or off-server:

- On-Server logs: Reboots triggered by the kernel or the hardware would typically correlate with events recorded in the logs on server such as the dmesg log or the system event log (SEL). Tooling-triggered reboots would also be logged on server when executed through an application on server. We push these logs to a centralized database either in realtime, *e.g.* netconsole log, or at hourly or daily schedule, *e.g.* SEL.

- Off-Server logs: Tooling-triggered reboots are recorded in the tooling logs in our centralized, off-server databases. In production, the tooling data are typically stored in MySQL databases to support realtime production queries, and pushed to Scuba [6] for more expensive realtime analytical queries. Some data are further backed up in Hadoop Distributed File System (HDFS) with longer retention and can be queried through Presto [7].

Additionally, there are unplanned server reboot and offline events that are triggered physically, *e.g.* voltage and network outages, without any preceding signal. For detecting these unplanned reboot and offline events, we need to look beyond the tooling and server logs and use system uptime as the ground truth for detecting reboots, as described in Section III-A.

III. Server Reboot Detection at Scale

In summary, the requirements for this reboot monitoring framework are:

1) **Detecting all possible reboot and offline events**
2) **Categorizing reboots by cause**
3) **Near-realtime** - record every event within 10 minutes
4) **At scale** - achieve items 1 to 3 for any and all servers across all geographically distributed datacenters, and store the result to allow for realtime query at per-server and fleet-wide levels
5) **Detecting anomalous reboot trends** - detect spikes and slow drifts in the reboot rates across hundreds of server attribute dimensions
6) **Automating root cause analysis** - automatically identify the combinations of hardware and software attributes with which the reboots can be best isolated

In the output of the framework, each reboot or offline event should be recorded with three pieces of information: *hostname*, *event*, and *event_time*. We will discuss how we achieve the first four requirements in this section.

A. Server Uptime Monitoring

We use system *uptime* as the ground truth for deciding whether a server is rebooted or offline. The system uptime of a server is the duration of time that the server has been operating since it is booted. In our system, we collect the uptime signals from each server every few minutes and store the data in a centralized database, Operational Data Store (ODS), powered by Gorilla [8]. ODS stores key-value pairs with timestamps, and the workload is write-intensive.

Unfortunately, while ODS handles server-level queries well, querying the system uptime of the entire fleet of servers every 10 minutes would significantly overload ODS. To meet the *near-realtime* requirement, we need to build a more scalable framework for detecting and categorizing reboots at scale.

B. Reboot and Offline Event Definition

In order to detect reboot and offline events correctly, we need to first design robust logic to address the noise in the collected signals. Fig. 1 shows the actual system uptime of a server that was rebooted between 10:32 and 10:38.

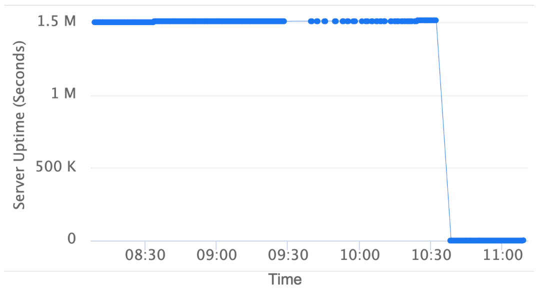


Fig. 1: Server uptime signals around a reboot.

From this example, it is clear that system uptime would not be populated until the data collection daemon starts running after the reboot. It took 191 seconds for the first uptime signal to populate in Fig. 1, and this delay is nondeterministic. In most cases, this delay would be shorter than 10 minutes, *i.e.* our detection frequency, but some reboots can take longer.

Another observation is the missing uptime signals even when the server is not being rebooted. This could be because the server is hanging or having connection issues so the daemon could not send out the uptime signal. In rare cases, there could also be dips and bumps in the uptime time series.

Fig. 2 shows two scenarios where there is a time window of missing uptime data points, and the first data point after the window has an uptime value lower than that of the last data point before the window. Let t_{last} and u_{last} be the timestamp and uptime value of the last data point before the window, and t_{new} and u_{new} be the timestamp and uptime value of the first data point after the window. Since uptime should be monotonically increasing at a fixed rate with respect to time, we can derive the latest boot time as $t_{new} - u_{new}$.

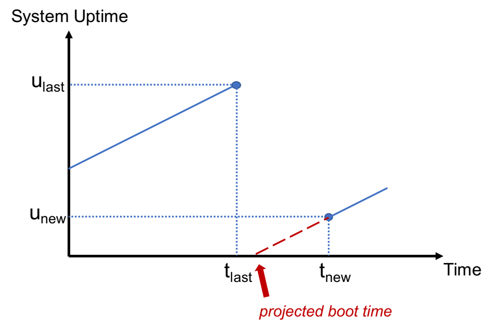
In Fig. 2a, $t_{new} - u_{new}$ is marked as the *projected boot time*. In Fig. 2b, however, $t_{new} - u_{new}$ is smaller than t_{last}, which implies that the server could not have been rebooted at $t_{new} - u_{new}$ because we know at t_{last} we have already logged a non-zero u_{last}. The uptime after the window, u_{new}, may be lower than the uptime value right before the window, u_{last}, due to transmission issues or delays in our system, but it is impossible that the server was rebooted in this window. From this analysis, we define our uptime *reboot check* as:

$$t_{new} - u_{new} > t_{last} \tag{1}$$

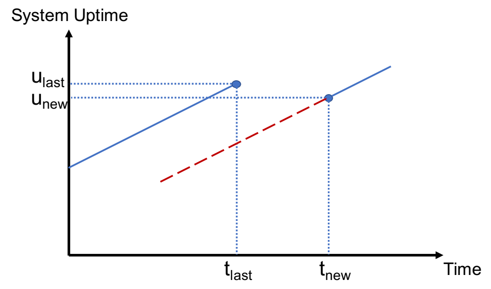
This check can be applied to any two consecutive uptime data points, regardless of whether there is a time window of missing uptime data points in between. In practice, we add a small margin to this comparison to account for potential delays in data delivery. In the output of the framework, we report t_{last} as the event_time of the reboot or offline event.

The logic for detecting an offline event is more straightforward. The *offline check* is simply:

$$t_{new} = NULL \tag{2}$$



(a) A reboot happened between the consecutive data points.



(b) A reboot could not have happened in the window.

Fig. 2: Two scenarios of system uptime signals.

There are three possibilities following an offline event the next time we check the uptime data: 1) The uptime is still missing, in which case the offline event continues. 2) We start seeing uptime signals that pass the reboot check, in which case we update the offline event to a reboot event. 3) If the new uptime signal fails the reboot check, we update the offline event as a *missed_signal* event. Given the space limit of this paper, we will skip the details about how we handle some other edge cases in the system, *e.g.* when new servers are deployed.

C. Near-Realtime Reboot Detection at Scale

In our production framework, server uptime signals are first streamed to Scribe, a persistent, distributed queueing system [9]. The Scribe data is then consumed by a stream processing system called PUMA [10], for filtering the uptime data and pushing it to a Scuba [6] dataset for batch processing.

Note that although the uptime data is streamed from the servers in realtime, we choose to process the data in micro batches. It is hence a *near-realtime* framework. Given the size of the data, the complexity of the logic, and the number of tooling and server logs to query, this micro-batch approach is more efficient than a realtime framework. The number of queries issued for the tooling and server logs can be reduced by multiple orders of magnitude, because we do not need to query all the logs when detecting each reboot. The data flow of the monitoring framework is illustrated in Fig. 3.

Table I shows the distribution of the cause of reboot from all the servers rebooted in our production environment over a 30-day period. 89% of the server reboots were triggered by planned operations. 10.0% of the reboots were initiated for

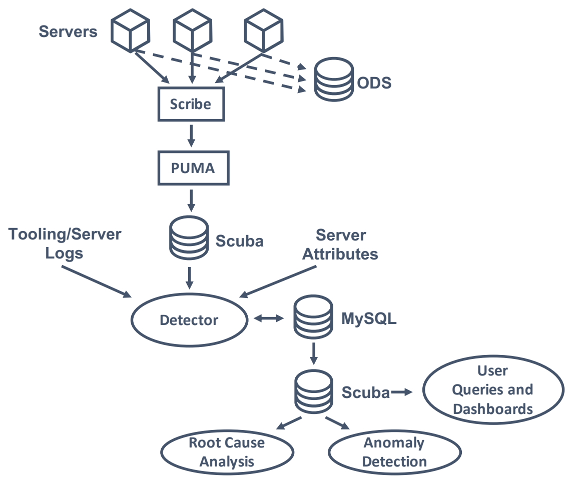


Fig. 3: The data flow in the server reboot detection, categorization, anomaly detection, and root cause analysis framework.

hardware repairs, and 1% of the reboots were associated with OOM or kernel panic events.

TABLE I: Distribution of server reboots by cause.

Cause of Reboot	Percentage of Servers Rebooted
firmware upgrade	34.9%
kernel upgrade	42.5%
hardware repair	10.0%
OOM	0.8%
kernel panic	0.2%
other operations	11.6%

The OOM- and panic-related reboots exhibit strong temporal locality. Over this 30-day period, servers on average had 2.8 and 1.6 unplanned reboots per kernel panic and OOM issue respectively, until the issue was mitigated. Note that we count a series of consecutive panic or OOM events as one issue. The distribution of the number of unplanned reboots on the servers has a long tail, and some servers could be rebooted more than 10 times a day. Fig. 4 shows the uptime of a server that was constantly rebooted due to kernel panics that were triggered by uncorrectable memory errors. The server was not able to run for more than 40 minutes without being rebooted. We fixed the server by replacing the bad memory card.

The OOM- and panic-related reboots are also highly correlated with the services that are running on the machines. 93.4% of the servers with OOM reboots were associated with three specific services, while 89.3% of the servers with kernel panic reboots were associated with one of these services and two other different services. These services are generally for supporting engineering development and testing.

IV. Server Reboot Monitoring at Scale

Many production issues can be detected as either a spike or a slow drift in the reboot rate of some cohorts of the servers. There are many dimensions along which we can define the server cohorts, *e.g.* by server or component model, firmware or kernel version, geographical location, position in a datacenter

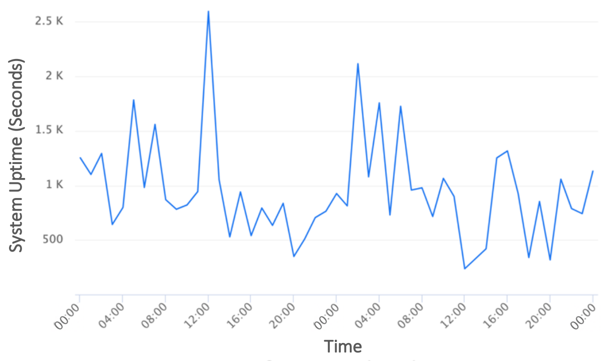


Fig. 4: System uptime of a server that had a series of kernel panics due to uncorrectable memory errors.

or a rack, and so on. Each of these dimensions could have tens to hundreds of distinct values, and the reboot rate in each of these dimensions could exhibit different norm and seasonality. It is therefore challenging to detect emerging issues from each of these time series with fixed thresholds.

In our framework, we first join the reboot and offline detection result with a collection of server attributes, as shown in Fig. 3. We then utilize the predictive thresholds generated by machine learning algorithms [3] to automatically set up a forecasted reboot rate range, which scales our anomaly detection significantly for detecting emerging issues.

In addition to the spikes and slow drifts in the reboot rates, we also detect anomalous servers that are rebooted many times in a short period. These *repeat offenders* typically suffer from individual server problems, and the services cannot run on these servers without frequent disruptions.

V. Root Cause Analysis for Server Reboots

When an anomalous reboot rate is detected, we need to quickly direct our investigation by identifying the differences between the servers that were rebooted or offline and the servers that were not. As mentioned above, in a production environment, a server can be described by tens of attributes, each with tens to hundreds of distinct values. The computational complexity for finding the best combination of these attribute values to explain the reboots is therefore very high. To automate the root cause analysis (RCA) process, we apply the scalable fast dimensional analysis proposed in [4].

With the server attribute data shown in Fig. 3, the RCA framework would identify combinations of server attribute values that could explain the rebooted and offline servers in a format similar to this example:

firmware_version = A and server_model = B and service = C correspond to 70% *of the rebooted servers, which is 5X more likely than expected.*

More details on the algorithm for identifying, sorting, and deduplicating the correlations and calculating the metrics can be found in [4]. In practice, this RCA method has been able to pinpoint reboots to root causes such as a specific main switchboard in a datacenter.

VI. Conclusion

In this paper we present a near-realtime, at-scale framework for server reboot detection, categorization, anomaly detection, and root cause analysis. We discuss the reboot population observed in a production system, and demonstrated the temporal and service locality of OOM- and panic-related reboots.

One of the biggest challenges in this framework is that we need to maintain a comprehensive collection of relevant messages from the logs, for correctly categorizing the reboot and offline events. In practice, we monitor the reboots categorized as having unknown causes closely to see if new logs or messages should be added to the correlation.

This framework has a strong dependency on the platform that executes the micro batches regularly. If this execution platform fails or delays, we would lose the visibility of the reboot and offline events, which could potentially be the cause of the issue with the execution platform in the first place. To handle this scenario, we monitor the execution platform separately, and when failures or delays are observed, the RCA framework in [4] can help us identify the root cause quickly.

Acknowledgement

The authors would like to thank Ravi Singamsetty, Anuj Mittal, and Andrew Hoogeveen for their extensive contribution to this framework. We also greatly appreciate the insightful inputs from Justin Akers, Mihai-Valentin Curelea, Chris Down, Angelo Failla, Sumedha Gholba, Tyler Graf, Alex Levchuk, Keyur Muzumdar, Gene Oden, and Olivier Raginel.

References

[1] W. Gu, Z. Kalbarczyk, R. K. Iyer, and Z. Yang, "Characterization of linux kernel behavior under errors," in *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, 2003.

[2] L. A. Barroso and U. Hölzle, *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*, 3rd ed. Morgan & Claypool, 2018.

[3] S. J. Taylor and B. Letham, "Forecasting at scale," *The American Statistician*, vol. 72, 2018.

[4] F. Lin, K. Muzumdar, N. P. Laptev, M.-V. Curelea, S. Lee, and S. Sankar, "Fast dimensional analysis for root cause investigation in a large-scale service environment," in *Proceedings of the ACM on Measurement and Analysis of Computing Systems (POMACS)*, 2020.

[5] F. Lin, M. Beadon, H. D. Dixit, G. Vunnam, A. Desai, and S. Sankar, "Hardware remediation at scale," in *IEEE/IFIP International Conference on Dependable Systems and Networks Workshops*, Jun. 2018.

[6] L. Abraham, J. Allen, O. Barykin, V. Borkar, B. Chopra, C. Gerea, D. Merl, J. Metzler, D. Reiss, S. Subramanian, J. Wiener, and O. Zed, "Scuba: Diving into data at facebook," in *International Conference on Very Large Data Bases (VLDB)*, Aug. 2013.

[7] M. Traverso. (2013) Presto: Interacting with petabytes of data at facebook. [Online]. Available: https://www.facebook.com/notes/facebook-engineering/presto-interacting-with-petabytes-of-data-at-facebook/10151786197628920/

[8] T. Pelkonen, S. Franklin, J. Teller, P. Cavallaro, Q. Huang, J. Meza, and K. Veeraraghavan, "Gorilla: A fast, scalable, in-memory time series database," in *Proceedings of the VLDB Endowment*, 2015.

[9] M. Karpathiotakis, D. Wernli, and M. Stojanovic. (2019) Scribe: Transporting petabytes per hour via a distributed, buffered queueing system. [Online]. Available: https://engineering.fb.com/data-infrastructure/scribe/

[10] G. J. Chen, J. L. Wiener, S. Iyer, A. Jaiswal, R. Lei, N. Simha, W. Wang, K. Wilfong, T. Williamson, and S. Yilmaz, "Realtime data processing at facebook," in *Proceedings of the International Conference on Management of Data (SIGMOD)*, 2016.