2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN) | 978-1-6654-3572-7/21/$31.00 ©2021 IEEE | DOI: 10.1109/DSN48987.2021.00035

Horus: Non-Intrusive Causal Analysis of Distributed Systems Logs

Francisco Neves
INESC TEC and U. Minho
Braga, Portugal
francisco.t.neves@inesctec.pt

Nuno Machado
Amazon and INESC TEC
Madrid, Spain
nuno.a.machado@inesctec.pt

Ricardo Vilaça and José Pereira
INESC TEC and U. Minho
Braga, Portugal
{rmvilaca,jop}@di.uminho.pt

***Abstract*—Logs are still the primary resource for debugging distributed systems executions. Complexity and heterogeneity of modern distributed systems, however, make log analysis extremely challenging. First, due to the sheer amount of messages, in which the execution paths of distinct system components appear interleaved. Second, due to unsynchronized physical clocks, simply ordering the log messages by timestamp does not suffice to obtain a causal trace of the execution.**

To address these issues, we present Horus, a system that enables the refinement of distributed system logs in a causally-consistent and scalable fashion. Horus leverages kernel-level probing to capture events for tracking causality between application-level logs from multiple sources. The events are then encoded as a directed acyclic graph and stored in a graph database, thus allowing the use of rich query languages to reason about runtime behavior.

Our case study with TrainTicket, a ticket booking application with 40+ microservices, shows that Horus surpasses current widely-adopted log analysis systems in pinpointing the root cause of anomalies in distributed executions. Also, we show that Horus builds a causally-consistent log of a distributed execution with much higher performance (up to 3 orders of magnitude) and scalability than prior state-of-the-art solutions. Finally, we show that Horus' approach to query causality is up to 30 times faster than graph database built-in traversal algorithms.

I. Introduction

Anomalies in systems can potentially impact users' lives and undermine their trust in a blink of an eye. Identifying and troubleshooting unexpected behavior in a quick and effective way is thus essential to maintain dependable services. Developers still use logs as the primary resource for debugging anomalies in distributed systems [1], as they are usually available out-of-the-box on every component via standard logging libraries [2], [3]. While the content of a log file may vary depending on the application and vendor, the common procedure is to have a sequence of unstructured textual messages with miscellaneous information regarding the system's state (e.g. timestamps, object identifiers, variable values, etc).

However, as modern distributed architectures increase in size, complexity and heterogeneity, troubleshooting issues via log inspection can be a daunting task for two main reasons:

- **Large number of intertwined events.** The log of a distributed execution often comprises a huge number of messages that belong to different requests and appear interleaved with each other. Since only a few log messages are actually relevant to diagnose an anomaly [4], [5], blindly analyzing the log for the whole execution is a metaphorical search for a *needle in a haystack*, which will only cause the problem to linger on.
- **Lack of causality.** As observed by Lamport, causality is fundamental to consistent reasoning about distributed executions [6]. Unfortunately, physical clocks on different machines drift apart, which prevents nodes of a distributed system from relying on real time to derive causality [7]. As the processing of a single request can be split across different nodes, one cannot reconstruct a request's causal history simply by collecting the nodes' log files and sort their messages by timestamp.

A popular approach to address the first challenge is to use toolsets such as the Elastic stack [8]. The Elastic stack provides support for aggregating, storing, analyzing, and visualizing logs from multiple components of a distributed system. The analysis of those logs is driven by a query language that enables the filtering of timestamped messages using conditional and regular expressions. Although widely used by developers to debug issues in production environments, the Elastic stack is no panacea. Since log messages are ordered by physical timestamp instead of causal dependencies, this toolset falls short for reasoning about bugs stemming from concurrent interactions across different nodes. In Section II-C, we show a concrete example of this limitation.

To tackle the aforementioned challenges, we propose Horus, a system that enables causally-consistent refinement of distributed system logs in a non-intrusive and scalable fashion. Horus traces lightweight kernel-level events with a clear *happens-before* relationship [6] (e.g. socket send and receive events), which are then used to encode a partial order of log messages from different sources. Since the inter-node causal dependencies stemming from the kernel-level events are independent of the message timestamp, Horus is not prone to physical clock drifting.

This idea of tracking causality in a distributed execution using kernel-level events was pioneered by Falcon [9]. However, Falcon relies on a *Satisfiability Modulo Theories* constraint solver to perform the causal ordering, which, using current state-of-the-art solvers [10], does not scale to executions with more than a few thousands of events, thus being unsuitable for production environments [11]. Moreover, Falcon does not provide support for filtering nor querying the resulting execution trace, which further hinders its ability to troubleshoot issues in real-world distributed systems.

Horus overcomes Falcon's limitations by explicitly encoding causality as a *directed acyclic graph (DAG)*, where nodes represent events (i.e. log messages and low-level operations) and edges indicate the causal dependencies between them. This execution graph is then stored in a graph database, allowing for both better scalability and the use of rich query languages to analyze the content of the logs.

To speed up graph traversals for query computation, Horus assigns both logical [6] and vector [12] clocks to each event. Given the DAG properties of the execution graph, the logical timestamps can be used to bound the subset of events relevant to a given query. This way, portions of the graph that comprise events outside the time span given by the logical clocks can be safely and efficiently excluded from the traversal, as they will not contribute to the result set.

Our case study with TrainTicket [1], a ticket booking application with 40+ distributed microservices, shows that Horus surpasses the Elastic stack in pinpointing the root cause of an anomaly in a distributed execution via log analysis.

We also conducted an experimental evaluation of Horus that demonstrated that our system is capable of causal ordering the log messages of a distributed execution with much higher performance (up to 3 orders of magnitude) and scalability than prior state-of-the-art solutions such as Falcon. Moreover, we show that, by leveraging logical time, Horus is able to dramatically reduce the query computation time (up to 30$\times$) with respect to traditional traversals in graph databases.

In summary, this paper makes the following contributions:

- A system, named Horus, that combines log messages and kernel-level operations to produce a graph of a distributed execution that can be inspected with rich queries.
- A technique that leverages logical clocks to dramatically reduce the time to run high-level queries that search of anomalies over a stored execution graph.
- An evaluation, using multiple benchmarks, that demonstrates the efficiency and effectiveness of Horus with respect to other state-of-the-art solutions to analyze logs of distributed systems.

The rest of this paper is organized as follows. §II presents a motivating example. §III provides an overview of Horus, while §IV and §V detail how it achieves causality and query refinement, respectively. §VI shows how Horus can be used in practice using a case study. §VII discusses the experimental evaluation. §VIII overviews the related work. Finally, §IX summarizes the main findings of this paper.

II. Background and Motivation

Causality is key to understand the behavior of a distributed system and, therefore, should be a core feature of any log analysis tool. In this section, we start by reviewing the concept of causality from the literature and, then, present a motivating example with TrainTicket to demonstrate that current log analysis solutions without causality guarantees fall short for debugging distributed system bugs.

A. The Need for Causality

Lamport introduced the *happens-before* relation $\rightarrow$ to enable a correct reasoning about causality in distributed executions [6]. Formally, there is a happens-before relationship between two events a and b, denoted $a \rightarrow b$, if: a and b belong to the same process and a precedes b in the execution, and a and b belong to different processes and a is the sending of a message m and b is the reception of m.

The happens-before relation is transitive, irreflexive and antisymmetric. When $a \nrightarrow b$ and $b \nrightarrow a$, then a and b are considered to be *concurrent*.

The ability to order log messages respecting causality is thus paramount to effectively troubleshoot a distributed execution, as illustrated by the motivating example in the next section.

B. Motivating Example – TrainTicket

TrainTicket [1] is an open-source train ticket application developed to foster research on fault analysis and debugging of distributed systems, namely those based on microservices. It consists of a ticket booking application with multiple functionalities: ticket inquiry, reservation, payment, order updates, and user notifications. TrainTicket's architecture comprises 40+ microservices written in different programming languages (e.g., Java, Nodejs, Python, Go), along with a user interface and third-party components, namely MongoDB for data storage and RabbitMQ for message queueing.

We use TrainTicket in this work because it mimics a real-world, complex distributed system and already contains 22 representative faults collected from a recent industrial survey on microservice applications [1]. Each fault was injected into an independent source code snapshot, available at a public repository.[1] For this motivating example, we use the one labeled as F13, as it represents an order violation caused by a message race, which is a common type of distributed concurrency bug [13].

a) The F13 fault: The F13 fault results from the interleaved processing of two messages arriving concurrently at the system. In TrainTicket, each order has a property representing its current state. When an order is created, it has state `UNPAID`. The order state can then migrate to either `PAID` or `CANCELED`, according to whether the client issues a *Payment Order* or a *Cancel Order*, respectively.

The *Payment Order* succeeds when both the following conditions hold: *i)* the client has enough funds, and *ii)* the order state transition from `UNPAID` to `PAID` is valid. Otherwise, the request fails, and the order state remains unchanged. In turn, the *Cancel Order* succeeds if the state can be set to `CANCELED`.

Since both requests change the state of an order and the TrainTicket application processes them without synchronization guarantees, their concurrent execution may non-deterministically lead to a payment failure.

The test driver for replicating this error in TrainTicket consists in a client application, *Launcher*, that first books a

[1]https://fudanselab.github.io/research/MSFaultEmpiricalStudy

```
 1  [Launcher-1.1] - [Reservation Result] Success
 2  [Payment-1.1] - [URI:/pay][Request: {"orderId":"652aaf9b"}]
 3  [Order-1.1] - [URI:/getById][Request: {"orderId":"652aaf9b"}]
 4  [Order-1.2] - Response: {"status":true, order":{"id":"652aaf9b", "status":"UNPAID"}}
 5  [Payment-1.2] - Response: "false"
 6  [Cancel-1.1] - [URI:/cancelOrder][Request: {"orderId":"652aaf9b"}]
 7  [Order-2.1] - [URI:/getById][Request: {"orderId":"652aaf9b"}]
 8  [Order-2.2] - Response: {"status":true, order":{"id":"652aaf9b", "status":"CANCELED"}}
 9  [Payment-2.1] - [URI:/drawBack][Request: {"userId":"c01d7008"}]
10  [Payment-2.2] - Response: "true"
11  [Cancel-1.2] - Response: {"status":true, "message":"Success."}
12  [Launcher-1.2] - java.lang.RuntimeException: [Error Queue]
```

Fig. 1: The set of log records aggregated by Elastic Stack for the failed payment request. Records are ordered by the timestamp assigned at the moment they were collected. To ease readability, we tag each record with the service in which it occurred and a local event counter (e.g., `Launcher-1.2` is the second event generated by the first thread in the Launcher service).

train trip with a new client account and, then, issues concurrent *Payment Order* and *Cancel Order* requests. When the payment fails, TrainTicket renders a page with the error message.

C. Debugging F13 *with Elastic Stack*

The increasing size and complexity of architectures motivated the industry to adopt toolsets tailored for distributed log monitoring. Among those toolsets, the *Elastic Stack* [8] is arguably one of the most popular and widely used. It provides support for collecting large sets of log data from multiple sources (via *Logstash*), analyze them using queries and filters (via *Elasticsearch*), and creating user-friendly visualization dashboards (via *Kibana*).

However, the Elastic Stack does not guarantee that log data is causally ordered, which hinders its effectiveness to diagnose unexpected behavior in distributed systems. In this section, we demonstrate such limitation during the F13 fault's diagnosis.

For this experiment, we deployed TrainTicket on a cluster of three *n1-standard-8* Google Cloud Engine instances managed by Docker Swarm. In addition, we set up a Docker container running the Elastic Stack toolset to collect and aggregate the logs produced by the TrainTicket microservices. More concretely, we placed a Filebeat daemon on each instance to continuously send container log messages (application logs augmented with container monitoring data) to Logstash, which is the event processing component of the Elastic Stack. Finally, we ran the F13 test driver until observing a failing execution and collected the corresponding logs captured by the toolset.

In the following, we describe how a developer would typically use Elastic Stack for debugging the error observed in the experiment, using an fictional character named Steve.

Steve is a software engineer at the company that owns TrainTicket and has been assigned to fix an issue ticket reporting an intermittent error related to payment requests.

Steve starts by inspecting the logs produced by the *Launcher* service, as it was the service in which the unexpected behavior surfaced. Using the container information present in the logs, Steve is able to isolate the services that processed the request and find the events that delimit the payment request. Having identified the portion of the logs relevant to the analysis, he finally gathers the log messages that will hopefully reveal the root cause of the failure.

Figure 1 unveils the subset of log messages, ordered by timestamp, that Steve relies on to reason about the execution path that led to the error at line 12. Each log statement begins with the corresponding service's name where it was collected. The events logged by the *Launcher* service (lines 1 and 12) delimit the portion of the execution that comprises the failure. The log messages in-between (lines 2-11) were recorded by three other services, specifically *Payment*, *Order*, and *Cancel*, and provide applicational context such as the identifier and the status of the order at any given moment.

By inspecting the logs, Steve realizes that, after completing the reservation step, the client sent a payment request (line 2), followed by a cancellation request for the same order (line 6).

Throughout the processing of the payment request, the logs show that the *Payment* service fetched the details of the order from the *Order* service (lines 3 and 4). At that moment, the order's state was `UNPAID` – a valid state for transitioning to `PAID` (line 4). Yet, the request later returned `false` (line 5), thus indicating that the payment had failed to complete.

In turn, the cancel request completed successfully (line 11), as it managed to set the order's state to `CANCELED` (lines 7 and 8) and issue a refund through the *Payment* service (lines 9 and 10).

The fact that the order's state was valid for the payment request (i.e., the response in line 4 had status `UNPAID`) leads Steve to believe that the reason for the payment failure was insufficient money. However, after carefully reviewing the account balance, he verifies that there were enough funds to pay for the reservation order.

Steve thus concludes that there is surely some other factor causing the error, although from the logs it is not clear what that factor might be. Moreover, as the failure is non-deterministic, Steve is unable to obtain additional details about the problem simply by re-executing the application. Consequently, he marks the ticket as "cannot reproduce" and the issue remains unresolved.

This cautionary tale aims at showing that the lack of causality in toolsets like Elastic Stack often renders the analysis of distributed system logs an inconclusive task, which further contributes to the lingering of harmful bugs in production.

In the next sections, we describe Horus in detail. Later, in Section VI, we show how it can be used to debug the failure presented in this example.

III. Horus Overview

We propose Horus, a system that addresses the limitations of prior log analysis solutions to further ease the burden of debugging distributed executions. To this end, Horus provides the following key features:

- **Lightweight and non-intrusive causality tracking.** Horus leverages kernel probes to efficiently capture low-level operations (e.g. socket sends and receives) that are then used to establish a causal order of log messages. These kernel probes rely on eBPF, a low-overhead technology widely used in performance analysis tools [14]–[18].
- **Causally-consistent aggregation of distributed events.** Horus combines both kernel events and logging events from different sources into a single directed, acyclic graph of the production run. In this graph, nodes represent the execution events and edges represent the happens-before (HB) relations between them. Intra-node HB relations are encoded using the original event timestamps, whereas inter-node HB relations are encoded using the kernel-level event causality rules.
- **Execution analysis and refinement via rich querying.** Casting the causal aggregation of multiple logs of a distributed execution as a graph generation problem gives Horus the ability to leverage off-the-shelf third-party graph databases [19], [20], which not only scale to executions with millions of events, but also provide support for rich query languages.

To better understand how Horus operates, we depict its architecture and execution flow in Figure 2. As shown in the figure, Horus comprises four main components: *Event Sources*, *Event Queue*, *Event Processor*, and *Graph Database*. Each component is described as follows.

a) Event Sources: This component represents the set of heterogeneous and independent sources that produce events at runtime. To handle different types of event sources, Horus requires the existence of adapters, which are responsible for collecting the data, normalizing it into a Horus-compatible format, and shipping it to the Horus backend for further processing (see ① in Figure 2). Our current prototype provides adapters to automatically collect events from:

- *Log4j* [2]. The adapter for Log4j consists of a simple formatter which outputs log messages as JSON objects indicating the timestamp, the name of the process/thread, and the textual message written in the source code. Each log message is thus considered an independent event.
- *I/O and Process System Calls*. The adapter for I/O and process system calls reuses eBPF probes from Falcon [9] to trace events regarding: the *start*, *end*, *fork*, *join* of a process or thread; the *request* and *accept* of a network connection between two processes; and the *sending* and *receiving* of a message. eBPF allows implementing efficient and safe programs that run inside the kernel for multiple purposes [18], namely I/O analysis, performance, monitoring, security and tracing.

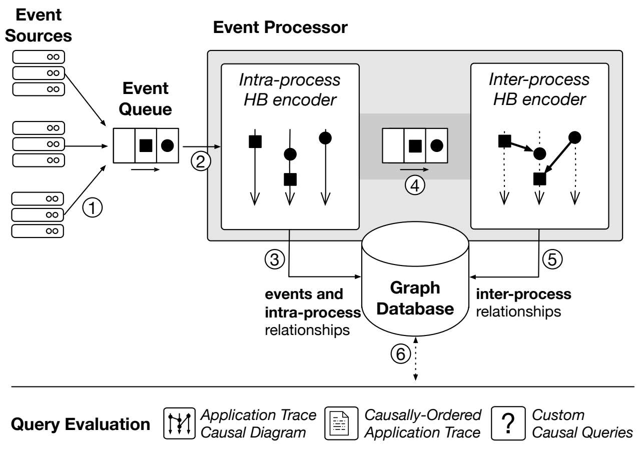


Fig. 2: Horus architecture and event flow.

We plan to extend Horus with adapters for additional logging libraries in the future, such as Logrus [21].

b) Event Queue: This component abstracts a set of persistent, distributed, and replicated queues within Horus that keep events waiting for being processed. In detail, Horus maintains two types of event queues: one responsible for storing the stream of events coming from the sources (see ②), and another responsible for linking the different stages for building the causal graph (see ④). The current prototype uses Apache Kafka [22] to manage the event queues.

c) Event Processor: This component is the processing core unit of Horus and aims at generating the execution causal graph. To this end, the Event Processor comprises two sub-components that operate as a two-stage pipeline. In the first stage, the *Intra-Process HB Encoder* establishes the HB relations between events of the same process, which are then periodically persisted in the graph database (see ③) and forwarded to the next processing stage (see ④).

In the second stage, the *Inter-Process HB Encoder* encodes the HB dependencies between events of distinct processes, which are then flushed to the database in periodic batches like in the previous step (see ⑤). Section IV describes the causal graph generation procedure in more detail.

d) Graph Database: The current prototype of Horus uses Neo4j [19] to store the execution causal graph and computing the developer's queries at debugging time (see ⑥). Neo4j is a graph database that provides a rich query language named Cypher for filtering, refining, and visualizing graphs.

Although Neo4j supports graphs with millions of events, our experiments have shown that the execution of causality queries does not scale well with the size of the graph. In Section V, we show how Horus dramatically improves query processing in Neo4j with a novel technique that leverages logical time to prune irrelevant portions of the graph.

Our prototype of Horus is publicly available at `https://github.com/DistributedSystemsAnalysis/horus`.

IV. Causal Graph Generation

This section details how Horus builds the causal graph of a distributed execution. As mentioned before, the nodes of this graph denote execution events (i.e., log messages and kernel operations) and the edges indicate the happens-before relationships between them. More concretely, there are two types of edges: *intra-process* and *inter-process*.

A. Intra-Process HB Relationships

In the first stage of Horus' event processing pipeline, the *Intra-Process HB Encoder* computes the happens-before relationships between the subsets of events belonging to the same process. As stated by the first property of the definition of causality (see Section II-A), these relationships can be derived directly from the program order.

In practice, logging libraries already assign timestamps to the log messages with the purpose of later easing their analysis. This means that, for the same process, it suffices to order the messages by timestamp in order to obtain the causal ordering of events at runtime.

However, in scenarios with multiple independent loggers and tracers, the same process may trigger those tools without any kind of synchronization, thus permitting the resulting events to arrive out of order at the Event Queue component. For that reason, Horus requires the timestamp source adopted by those tools (e.g., physical clock) to be the same and accurate enough to define a total order of events across them. Otherwise, the program order property ceases to hold.

To handle events coming from several sources, the *Intra-Process HB Encoder* maintains *process timelines*. A timeline corresponds to a sequence of events, ordered by timestamp, that were executed by the same process. For each incoming event, the *Intra-Process HB Encoder* is responsible for inserting it into the corresponding process timeline in the correct position based on its timestamp. This procedure guarantees that, for a given process, events enqueued out of order will still produce a causally-consistent timeline.

The *Intra-Process HB Encoder* then periodically flushes the process timelines to the graph database. The flush interval is a tunable parameter: longer flush intervals provide lower runtime overhead (due to fewer connections to the database) at the cost of more memory consumption to maintain pending established causal relationships. Shorter intervals, in turn, reduce the memory footprint and make data more quickly available for querying (which is useful for online monitoring), but incur a performance slowdown.

In practice, storing a process timeline consists of creating a new node per event and encoding intra-process edges between pairs of nodes that correspond to contiguous events in the timeline. As an example, let us consider a timeline T with four events $\{A, B, C, D\}$ sorted in ascending order of their timestamp. Upon persisting T into the database, the *Intra-Process HB Encoder* creates four nodes – A, B, C, D – and three edges – $(A, B), (B, C), (C, D)$.

Finally, the *Intra-Process HB Encoder* sends the timeline events to the queue of the next processing stage.

B. Inter-Process HB Relationships

In the second processing stage, the *Inter-Process HB Encoder* computes the happens-before relationships between events of two different processes.

Inter-process causality stems from the second property of the happens-before definition and, in contrast to intra-process causality, does not rely on timestamps. Instead, it relies on a message m that unequivocally identifies that its departure from a process causally-precedes its arrival at another process. In practice, m is usually either a unique message identifier or a piece of context data that indicates the causal relation.

In the current version of Horus, the *Inter-Process HB Encoder* determines inter-process causal dependencies based on the event attributes captured by eBPF kernel probes. For instance, in a TCP connection scenario, events of sending (SND) and receiving (RCV) bytes include attributes concerning the source and destination IP addresses and ports. Hence, considering the TCP delivery and ordering guarantees, one can establish $SND \rightarrow RCV$ causal pairs for each connection channel. Alongside, one can define the following causal pairs of events between a parent process p and a child process c: $FORK_p \rightarrow START_c$ and $END_c \rightarrow JOIN_p$.

The *Inter-Process HB Encoder* operates in a stream fashion, building inter-process causal dependencies according to the nature of each event. To improve performance, incomplete causal pairs are kept in memory until the corresponding pairing event is consumed from the queue.

Periodically, *Inter-Process HB Encoder* flushes the complete causal pairs by inserting a new edge per pair into the graph database. Once the causal pairs are persisted, the events exit the processing pipeline and become available for analysis.

We note that the *Inter-Process HB Encoder* can be easily extended with new causality rules based on event attributes. This feature renders Horus with the ability to accommodate arbitrary types of events and happens-before dependencies into the execution causal graph.

V. Efficient Causal Graph Querying

Once the causal graph is stored in the graph database, the developer can start zeroing in on the error's root cause. In Horus, this is done via refinement queries written in Cypher (which is an expressive query language provided by Neo4j).

In general, the analysis of the causal execution graph comprises two main types of queries:

Q1. May event $\underline{a}$ causally affect event $\underline{b}$?

Q2. What are the causal paths between $\underline{a}$ and $\underline{b}$?

Q1 is the fundamental query for evaluating the happens-before relation between any two events a and b. In turn, *Q2* aims at extracting the sub-graph of causal events occurring

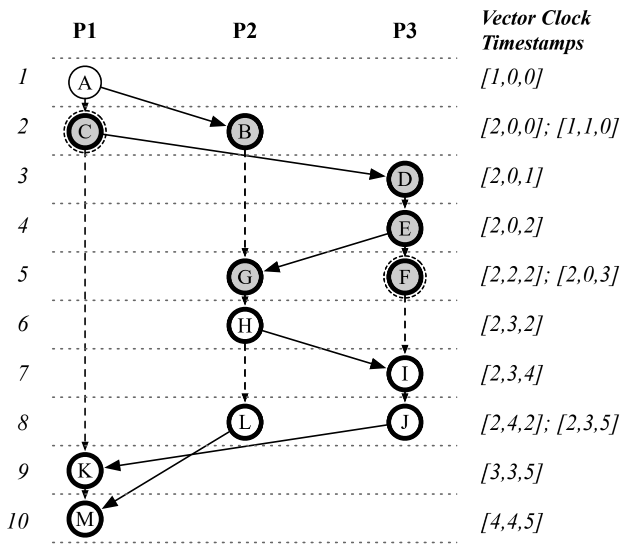


Fig. 3: Causal graph with three process timelines. Nodes denote events along their execution, and edges their causal dependencies. Logical clocks and vector clocks are depicted on the left and on the right of the events, respectively. Thick borders (respectively, shades) indicate the nodes explored by a default graph database traversal (respectively, Horus) to compute the causal paths between C and F.

in-between them, which is particularly useful to reconstruct the causal trace of a request. For instance, computing *Q2* for events `Launcher-1.1` and `Launcher-1.2` in Figure 1 would produce a graph with all events in the figure causally ordered.

These queries can be expressed with path discovery and traversal operations that are provided by the graph database. More concretely, one can answer *Q1* with a *shortest path* algorithm, and answer *Q2* by finding all paths between the two events. Unfortunately, as these built-in traversals are oblivious to the semantics of distributed systems, they are inefficient for causal queries and cannot scale for large graphs. To understand why this is the case, let us consider the causal graph depicted in Figure 3 with three process timelines ($P1, P2$, and $P3$). Recall that nodes denote events along their execution and edges the happens-before relations between them.

Assume that we want to obtain the events that happened between C and F, which is a query of type *Q2*. Answering this with a graph database's default traversal would correctly yield the result set $\{C, D, E, F\}$, although at the expense of visiting the nodes $\{C, D, E, G, F, H, I, L, M, K\}$ (represented in Figure 3 with a thick border). Observing the figure, one can notice that this approach is far from optimal, as, for example, nodes I and J are visited despite being clearly irrelevant to the query because they occur after F. The reason behind this inefficiency is that the traversal performed by the database does not take into account the notion of time nor the directed acyclic properties of the causal graph.

To address this limitation and speed up query processing, Horus augments the causal graph with logical timestamps, namely *logical clocks (LC)* [6] and *vector clocks (VC)* [23].

Logical clocks are scalar values assigned to each event such that, for every two events a and b, the following condition holds: $a \rightarrow b \implies LC(a) < LC(b)$. In Figure 3, logical clocks are depicted as the timestamps on the left. For example, $B \rightarrow G \implies LC(B) = 2 < LC(G) = 5$. Note, however, that the reverse implication is not guaranteed: $LC(C) = 2 < LC(G) = 5$, but C does not happen before G. In fact, logical clocks alone are not enough to reason about event causality.

In contrast, *vector clocks* suffice to reason about the causal ordering of events in a distributed execution by providing the property: $a \rightarrow b \iff VC(a) < VC(b)$. We represent vector clocks in Figure 3 next to each event: for example, $C \rightarrow D \iff VC(C) = [2, 0, 0] < VC(D) = [2, 0, 1]$. In turn, for events C and G, since neither $VC(C) < VC(G)$ nor $VC(G) < VC(C)$ holds, we can conclude that they are concurrent.

Horus performs a graph traversal similar to topological sorting to assign both logical and vector clocks to each event in the graph. These logical timestamps are then used to speed up querying in a twofold fashion, as described below.

Let $G = (V, E)$ be the causal graph under analysis and $a, b \in V$ the start and end event of a query, respectively.

First, Horus leverages the logical clocks of a and b to quickly bound the portion of the graph that is relevant to the query. In practice, this corresponds to using the logical clocks as database indexes and compute the following over-approximation V' of the nodes in the result set: $V' = \{v \in V \mid LC(a) <= LC(v) <= LC(b)\}$. For the example of computing the causal paths between C and F considered earlier in this section, this step produces the following subset of nodes $\{C, B, D, E, G, F\}$ (shaded nodes in Figure 3). Note that VCs would also allow computing V', however they are inappropriate for graph database indexing due to their non-scalar nature.

Second, Horus uses vector clocks to prune out the events in V' that are *concurrent* to a and b. This operation yields a sub-set V'' containing only the events in the causal paths between a and b: $V'' = \{v \in V' \mid VC(a) < VC(v) < VC(b)\}$. For the previous example, this step will discard events B and G and leave only the sub-set $\{C, D, E, F\}$.

Finally, Horus computes the result set of the query by collecting the edge set E' with the connections between the nodes in V'': $E' = \{e_{x \rightarrow y} \in E \mid x \in V'' \wedge y \in V''\}$.

In summary, Horus leverages logical time to answer the two types of refinement queries as follows.

Q1. $VC(a) < VC(b)$;

Q2. $CausalPath_{a \rightarrow b} = (V'', E')$, where:

$$V' = \{v \in V \mid LC(a) <= LC(v) <= LC(b)\}$$
$$V'' = \{v \in V' \mid VC(a) < VC(v) < VC(b)\}$$
$$E' = \{e_{x \rightarrow y} \in E \mid x \in V'' \wedge y \in V''\}$$

We implemented the logical time optimizations in Neo4j as two new procedures denoted `happensBefore()` and `getCausalGraph()`. This way, it becomes possible to write and execute efficient causal queries using Cypher.

Event Type	**Occurrences (approx. %)**
LOG	4,531 (22.52%)
RCV	4,339 (21.57%)
CREATE	3,618 (17.99%)
START	3,340 (16.60%)
SND	2,689 (13.37%)
END	660 (3.28%)
JOIN	357 (1.77%)
CONNECT	260 (1.11%)
FSYNC	173 (0.86%)
ACCEPT	149 (0.74%)
Total Events	**20,116**

TABLE I: Number of events per type present in the causal graph under analysis. *LOG* events are produced by the *log4j* logging library, while the rest is captured by Horus' kernel-level tracer.

VI. Case Study – Debugging *F13* with Horus

Recall the TrainTicket's *F13* fault introduced in Section II, which causes an error due to the interleaved execution of two requests. In this section, we demonstrate how Horus' causal-consistent approach overcomes the limitations of toolsets like Elastic Stack and allows troubleshooting the *F13* error.

For collecting TrainTicket's events at runtime, we set up Horus with two event sources, as follows. First, we configured Horus' kernel-level tracer to attach details about the Docker containers in which TrainTicket's services run. The goal was to later allow uniquely identifying each individual host in the system. Second, we enabled a new *log4j* appender for attaching useful process information to TrainTicket's log messages before forwarding them to Horus. We note that, on each host, both event tracers rely on the same monotonically increasing physical clock to later ensure the correct encoding of the *intra-process* causality.

Table I reports the amount of events, grouped by type, captured by Horus when building the causal graph for TrainTicket's *F13* failing execution. This causal graph results from six minutes of execution, which generated 20,116 events, spread across 96 process timelines, and 27,859 causal relationships, from which 4,593 (16.49%) encode *inter-process* causality. Despite *LOG* being the most common event type, *SND*s and *RCV*s together, which are essential for encoding inter-process causality, account for around 35% of the graph. The discrepancy between the percentage of *SND* and *RCV* events is explained by different buffer sizes on the hosts, which can cause a single message to be read by multiple partial *RCV*s.

In the following, we revisit the cautionary tale from Section II-C and describe how Steve, our fictional TrainTicket's software engineer, would use Horus to debug the payment failure report in the issue tracker.

Steve starts the day with an issue ticket on his hands, reporting that an error popped up when a user performed the payment request for order *652aaf9b*. Fortunately, the user has also provided the error message in the ticket description: `java.lang.RuntimeException: [Error Queue]`.

Steve decides to inspect the logs recorded from the moment the payment request started until the moment in which the error message appeared. To this end, he has to first identify the events in the causal graph that delimit that relevant portion of the execution.

The beginning of the payment is defined by a message sent from the *Launcher* service to the *Payment* service. Steve knows that, in Horus, this message is represented by the first $SND_{Launcher} \rightarrow RCV_{Payment}$ causal pair. As such, he is able to obtain the logs concerning the failing request by executing a query to compute the causal graph from that $SND_{Launcher}$ event to the LOG_{Launcher} event containing the message `java.lang.RuntimeException: [Error Queue]`. Also, to ensure that he is focusing on the right request, Steve augments the query with an additional clause stating that the *START* event of the payment request must happen before (in terms of logical time) the error event.

Figure 4a details the aforementioned Cypher query to refine the causal graph for the failing request under analysis. The query is composed of three parts, identified by the three comment blocks in Figure 4a. The first part filters the events that either belong to the beginning of a payment request or represent an error message. In other words, this part aims at finding the potential boundaries of the failing request.

The second part computes the causal paths that connect the events returned in the previous step and collects the log messages for each one of them. Note how Horus extends Neo4j with the procedure `getCausalGraph()` for efficiently extracting the causal graph between two events in the graph.

Finally, the third part of the query aims at filtering the events belonging solely to the order *652aaf9b*. The final output of the query is depicted in Figure 4b. Note that the log statements in the figure are the same as those in Figure 1, but this time the events are causally ordered.

After executing the query, Steve proceeds with the analysis by inspecting the log messages. At this point, he notices that the order status changed from `UNPAID` (line 4) to `CANCELED` (line 6). The only valid state transition after a payment request is from `UNPAID` to `PAID`. However, the logs show that the order state transits unexpectedly from `UNPAID` to `CANCELED`, which is an invalid final state for a payment request. On the other hand, Steve realizes that, in addition to the payment request (line 1), the logs also report a cancellation request (line 2) for the same order. Consequently, he suspects that the failure might be related to the concurrent execution of both requests.

To clarify the suspicion that the cancellation request was the culprit of the invalid state change, Steve decides to render the execution causal graph in ShiViz [24], a popular space-time diagram visualizer compatible with Horus.

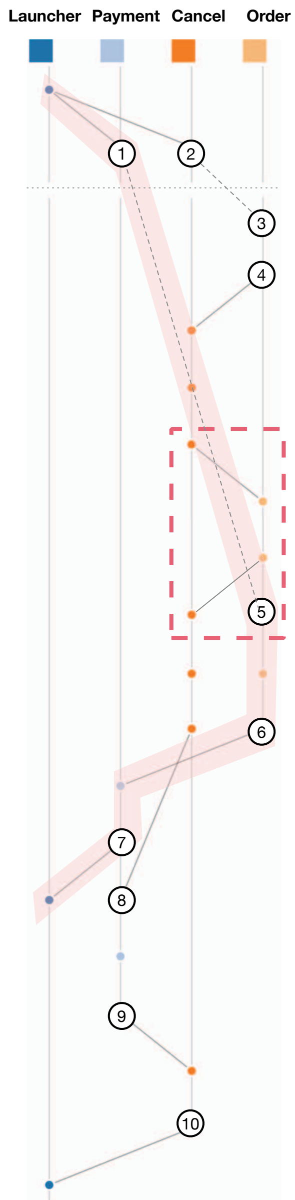
Figure 4c illustrates the diagram rendered by ShiViz. It comprises four process timelines regarding the *Launcher*, *Payment*, *Cancel* and *Order* services, respectively. Each thick-bordered number identifies the corresponding *LOG* event in

```
// Find events that denote the beginning of the payment request and the noticed error.
MATCH
  (reqSnd:SND {host: 'Launcher'})-->(:RCV {host: 'Payment'}),
  (reqError:LOG {host: 'Launcher'})
WHERE
  reqError.message CONTAINS 'java.lang.RuntimeException: [Error Queue]'
  reqError.lamportLogicalTime > reqSnd.lamportLogicalTime
WITH
  reqSnd.lamportLogicalTime as reqSndTime,
  min(reqError.lamportLogicalTime) as reqErrorTime

MATCH
  (reqSnd:EVENT {host: 'Launcher', lamportLogicalTime: reqSndTime}),
  (reqError:EVENT {host: 'Launcher', lamportLogicalTime: reqErrorTime})

// getCausalGraph(startNode, endNode, onlyLogs)
CALL horus.getCausalGraph(reqSnd, reqError, true) yield node
WITH reqSnd, reqError, node ORDER BY node.lamportLogicalTime ASC
WITH
  reqSnd.eventId as startEventId,
  reqError.eventId as endEventId,
  collect(node) as logs

// Return:
// 1. startEventId: the event that denotes the start of the payment request
// 2. endEventId: the event that denotes the noticed error
// 3. logs: log messages containing the order identifier.
UNWIND logs as log
WITH startEventId, endEventId, log
WHERE log.message CONTAINS '652aaf9b'
RETURN startEventId, endEventId, collect(log.message) as logs
```

a) Query to retrieve the events that denote the beginning of the payment request concerning the order '652aaf9b' and the noticed error. The returned result contains the events' identifiers and the logs containing the order identifier.

```
   // startEventId: hipster-1.europe-west2-c.c.horus-262311.internal64523
   // endEventId: ts-launcher-LOG-584277764938
①  [Payment-1.1] - [URI:/pay][Request: {"orderId":"652aaf9b"}]
②  [Cancel-1.1] - [URI:/cancelOrder][Request: {"orderId":"652aaf9b"}]
③  [Order-1.1] - [URI:/getById][Request: {"orderId":"652aaf9b"}]
④  [Order-1.2] - Response: {"status":true, order":{"id":"652aaf9b", "status":"UNPAID"}}
⑤  [Order-2.1] - [URI:/getById][Request: {"orderId":"652aaf9b"}]
⑥  [Order-2.2] - Response: {"status":true, order":{"id":"652aaf9b", "status":"CANCELED"}}
⑦  [Payment-1.2] - Response: "false"
⑧  [Payment-2.1] - [URI:/drawBack][Request: {"userId":"c01d7008"}]
⑨  [Payment-2.2] - Response: "true"
⑩  [Cancel-1.2] - Response: {"status":true, "message":"Success."}
```

b) Causally-ordered log messages from the moment the request starts to the moment the error is noticed for order 652aaf9b. Each log statement is prefixed with its originating service's name and its thread's local counter.



c) Space-time diagram rendered by ShiViz for the failed payment request.

Fig. 4: Refinement queries, written in Cypher, used to debug the *F13* failure, along with a snippet of the ShiViz diagram generated for the causal graph produced by Horus after processing the queries.

Figure 4b. The execution path of the payment request under analysis is highlighted in red.

In the diagram, Steve observes that log messages ① and ② appear at the same level, which confirms that both payment and cancellation requests started concurrently. Since both requests depend on the order's status, the diagram shows each one issuing a getById request to the *Order* service (③ and ⑤). However, while the response sent to the *Cancel* service indicated the status UNPAID (④), the one sent to the *Payment* service revealed that the order status was CANCELED (⑥). Steve thus confirms his suspicion: somehow the order state changed between events ④ and ⑤.

From a closer look at the diagram, he identifies that, after receiving the order details, the *Cancel* service issued another request back to the *Order* service, this time to update the order state to CANCELED (see the red dashed box). Since this request executed right before the arrival of the *Payment*'s request, the order was already canceled at the time of the fetching. However, the application requirements state that users cannot pay orders that are already canceled.

Steve was finally able to explain the root cause of the failure: as the order status was changed after the payment had started, the payment request received the response *false* (⑦), which then raised an exception in the *Launcher* service. The fact that the TrainTicket permits a payment request to be performed after an order cancellation represents an *order violation*, a common type of distributed concurrency bugs.

To fix this incorrect behavior once for all, Steve implements a synchronization mechanism to enforce the correct execution between payment and cancellation requests. It took a full day of work, but he successfully managed to mark the issue ticket as "resolved" and accomplish his goal.

VII. Performance Evaluation

In this section, we conduct an experimental evaluation of Horus aimed at assessing the benefits and limitations of our system with respect to: *i)* the scalability of the event processing pipeline; *ii)* the performance of the logical time assignment algorithm, and *iii)* the performance impact of leveraging logical time during query processing.

To enable experiments with causal graphs of different sizes, we implemented a synthetic event generator that mimicks an arbitrary number of rounds of a synchronous client-server scenario. In detail, it generates *request-reply* interactions between two processes *P1* and *P2* by creating the causal pairs $SND_{P1} \rightarrow RCV_{P2}$ and $SND_{P2} \rightarrow RCV_{P1}$. The output of this micro-benchmark is thus an execution causal graph with N events and $3N/2 - 2$ edges in total (comprising both intra- and inter-process edges).

In the rest of this section, we describe the environment setup and discuss the results for each experimental scenario.

A. Event Processing

Event processing consists of determining intra- and inter-process causality for each event and storing the event and its corresponding causal relationships in the graph database.

In this experiment, we evaluate how Horus scale in terms of the number of processed events per time unit.

We configured two instances in Google Cloud Platform. The first instance is of type *n1-standard-16* (16 vCPUs and 60GB RAM) and hosts the Horus' pipeline. The other instance, *n1-standard-8* (8 vCPUs and 30GB RAM), hosts event generator clients. Each experiment ran for 15 minutes, with stress clients performing an intensive workload by submitting as many events as possible to Horus. The flush interval is set to 100ms and 200ms for events and causal relationships, respectively.

Figure 5 illustrates the evolution of the event processing throughput as the amount of clients increases. The dashed line indicates the incoming event rate measured in Apache Kafka. Horus follows the incoming rate until the 18 clients setting, which produces close to 6,000 events/second. Note that this is a stress scenario, as the 18 clients generate in less than 4 seconds the same amount of events generated by TrainTicket in 6 minutes. Thus, Horus would scale to a setup with 1500+ microservices performing the same workload as TrainTicket.

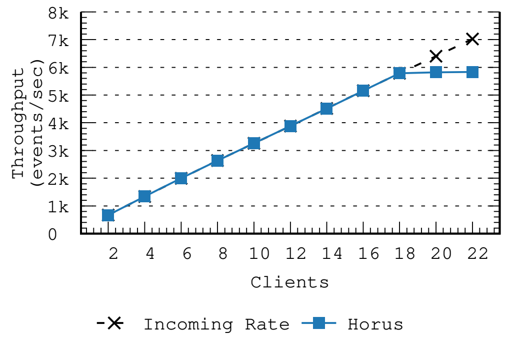


Fig. 5: Horus throughput as the number of clients increases. The incoming event rate reveals the amount of events produced by running clients in a stress scenario.

When Horus reaches its maximum throughput, pending events still remain in the event queues. Therefore, Horus will still be able to process all events generated during workload peaks and make them available for analysis, even with delay.

Horus architecture, however, allows for scale-out of causality encoders within the pipeline. To achieve this, one must configure Horus to ensure the following: *i)* all events in a process are processed by the same intra-process causality encoder, thus preserving the program order; *ii)* the events belonging to a causal pair are delivered to the same inter-process causality encoder; and *iii)* the program order is preserved from the intra-process encoder to the inter-process encoder. This allows distributing event processing among several encoders without requiring synchronization and still guarantees correctness in constructing the causal execution graph.

In short, this experiment shows how many events can be handled in real-time with a single event-processing server, while knowing that, even when this rate is exceeded momentarily, no events are lost.

B. Logical Time Assignment

Horus introduces an algorithm based on graph traversal to assign logical time to events in the stored causal graph.

In this experiment, we compare the performance of the Horus' algorithm with the Falcon's approach that, in turn, resorts to a state-of-the-art SMT constraint solver to causally order logs. We first populated the Horus database using the synthetic event generator. Then, we exported the unordered events in the format compatible with the Falcon's solver.

Figure 6 illustrates the evolution of the execution time of the Falcon's solver and the Horus' algorithm for execution graphs with different sizes. The Falcon's solver depicts exponential behavior as graph size increases whereas Horus evolutes slightly linearly.

Falcon is thus unable to assign logical time to more than few thousands of events in a timely manner. It spends more than

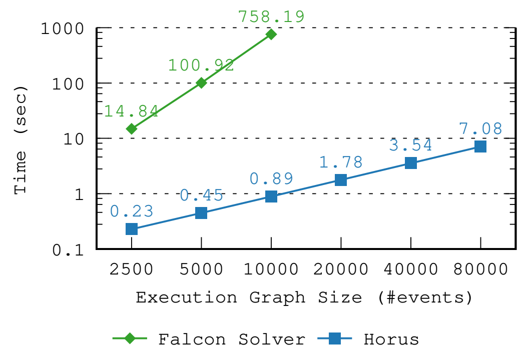


Fig. 6: Comparison between the execution time of Falcon and Horus to assign logical clocks on different graph sizes.

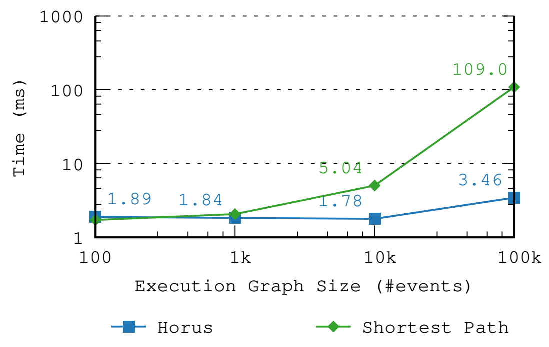


Fig. 7: Comparison between the execution time of the *shortest path* algorithm and Horus to answer query Q1, for different graph sizes.

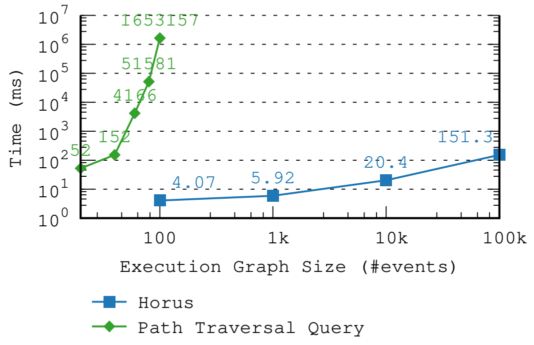


Fig. 8: Evolution of the execution time of obtaining the causal graph between two events using traversal-based query and the logical-time-based approach implemented in Horus.

12 minutes for 10,000+ events, while the Horus' algorithm spends 7 seconds to assign the logical time.

This algorithm can perform with low execution time, even when graph increases over time. In fact, running in a periodic basis, the algorithm is able to resume the procedure from the most recent event of each process timeline already with logical timestamps and proceeding to the recently added events. Thus, the execution time does not depend on the total amount of events in the graph but the amount of unprocessed events.

In summary, the execution time of the proposed logical time assignment algorithm scales with the amount of events and causal relationships in the execution graph and thus is suitable for real use cases.

C. Causal Graph Querying

Horus enables developers to refine the causal graph using the database's built-in query language, as it provides filtering and grouping operations for graph algorithms. As discussed in Section V, the built-in algorithms are not efficient to answer graph refinement queries such as *Q1* and *Q2*. Horus addresses this limitation by annotating events with logical time.

In this section, we conduct separate evaluations for Q1 and Q2 query types. For each type, we compose the corresponding Cypher queries on each approach, one resorting to built-in path traversal algorithms and the other leveraging logical time.

We setup a *n1-standard-16* instance that hosts the Horus pipeline, alongside the Neo4j database. For each experiment, we populated the database using the synthetic event generator.

a) Evaluation for Q1.: The Cypher query that follows a path-traversal-based approach makes use of the *shortest path* algorithm to determine whether a causal path between the two events exist. For the Horus' approach, in turn, the query just compares the logical timestamps of the two events.

We selected ten event pairs, each whose causal graph contains 10% of the graph's total events, and then executed the queries for each pair. For instance, for a graph with 100 events, each causal graph contains 10 events. Clearly, the dimension of each pair's causal graph also increases with the graph size.

Figure 7 unveils the execution time of each query for different graph sizes. Observe that both axes are represented using logarithmic scale. One important note, derived from the negligible *std. dev.*, is that both queries are insensitive to pair location. This is, each query shows similar performance either for a pair positioned at the top of the graph or for a pair positioned at the middle or at the bottom of it.

The performance of the *shortest path* algorithm decreases as the graph grows. For 100,000+ nodes, it becomes ≈30 times slower than comparing logical timestamps. This happens because comparing the logical time of any two events does not require any sort of path traversal. Instead, it compares logical timestamps according to the properties of *vector clocks*.

b) Evaluation for Q2.: For this experiment, the Cypher query that relies on built-in algorithms simply aims at finding all paths between the two events. In this case, the position of the pair and the direction of the internal traversal algorithm influence the query performance in the traversal-based approach. For instance, a pair positioned at the top would lead the query to be less performant if the traversal's direction from the top the the bottom of the graph and vice-versa.

To perform a rigorous and fair comparison, we evaluate the traversal-based query with the pair positioned in the middle of the graph, which causal graph contains 10 nodes. We selected this pair for two reasons: *1)* we assume that most analysis do not focus on the beginning nor on the bottom of the causal graph; and *2)* the performance of the path traversal is the same, independently of direction it takes.

In turn, for evaluating the Horus' approach, we adopt the same procedure as in for Q1, that is, we choose ten event pairs, each which causal graph contains 10% of the total events.

Figure 8 shows the performance degradation incurred when using the traversal-based query to extract causal paths on small graphs (from 10 to 100 nodes).

In contrast to the traversal-based query, our proposed logical-time-based approach is insensitive to pair location in the graph. Therefore, we can compare the performance of both approaches in the scenario of a graph with 100 events.

For the scenario that compares both approaches, i.e., graph with 100 events, using logical time decreases the execution time from $\approx$1653 *seconds* to $\approx$4 *milliseconds*.

In summary, the Horus' logical-time-based approach scales much better than traversal-based query for causal queries.

VIII. Related Work

Context propagation is a distributed tracing technique for causal debugging and performance analysis in distributed systems [25]. To capture causality, X-Trace [26], Dapper [27], Pivot Tracing [28], Canopy [29] and others [30]–[32] propagate identifiers in requests, jobs and tasks across software component and machine boundaries. Nonetheless, it requires instrumentation of the source code of software components, otherwise the context-propagation chain is incomplete. Moreover, all these are workflow-centric approaches, while Horus provides a comprehensive view of concurrent requests.

Watermelon [33] is distributed debugging framework capable of tracking comprehensive causality. Yet, it requires a training phase in which engineers must inspect execution traces of component's communications and write specifications that determine which request contributed to the values read by a later request. In modern distributed systems, analyzing execution traces of dozens of heterogeneous software components poses a hard challenge to adopt Watermelon.

XVector [34] is a vector clock logging library that augments log messages with vector timestamps to enable visualization of distributed executions using ShiViz [24]. However, it requires developers to adopt it as logging library, which may be unpractical for third-party components.

Log messages often carry valuable information about system's state and execution flow. As such, previous works resort to processing log entries for performance analysis [5], [35] and anomaly detection [36]–[39]. Briefly, lprof [5] resort to static analysis to detect possible identifiers that aid correlation of requests log entries. LRTrace [35] applies pattern matching to unstructured logs to correlate resource usage metrics with logs. Log3C [36] samples, clusters and matches log sequences for identifying problems by correlating clusters of log sequences with relevant key performance indicators. LogRobust [37], DeepLog [38] and CloudRaid [39] leverage machine learning techniques to aid log-based analysis, detect anomalies and automatically find concurrency bugs, respectively.

The operating system internals has been an exploration path to infer causality in end-to-end requests [9], [40], [41]. vPath [41] is an application-agnostic monitor that intercepts communication and process syscalls to precisely discover request processing paths in systems that follow well-established programming patterns. Falcon [9] is the most closely related work to Horus and the current state-of-the-art for tracking causality in a distributed execution via low-level tracing. Briefly, Falcon is a pipeline tool that generates a causally-coherent trace of logs from several logging sources by intercepting system calls and leveraging well-established causality between kernel events to construct a causal trace of application logs. Falcon relies on a *Satisfiability Modulo Theories* constraint solver, which does not scale to executions with more than a few thousands of events, as shown in Section VII-B. Moreover, Horus offers a powerful query language inherited from the underlying graph database and is able to output traces compatible with the causal diagram visualizer ShiViz [24].

IX. Conclusions

In this paper, we introduce Horus, a system for analyzing distributed system logs in a non-intrusive, causally consistent, and scalable fashion. Horus leverages kernel-level operations traced at runtime to generate a graph of the distributed execution in which log messages are causally ordered.

Given the large volume of data generated by kernel-level tracing in real applications, the key contributions of Horus are the storage and processing techniques proposed that deal with scale, in terms of the amount of data, but also conceptually, allowing debugging operations to be encoded in a high-level graph querying language. In particular, the combination of both scalar and vector clocks dramatically reduces the time to run queries over the execution graph.

Our case study with TrainTicket, a ticket booking application with 40+ microservices, demonstrates that Horus is effective in pinpointing the root cause of anomalous behavior. Moreover, the experimental evaluation of Horus against prior state-of-the-art solutions shows that it *i)* scales better with the number of events, *ii)* is faster to aggregate logs from multiple sources into a causally consistent execution trace, and *iii)* executes analysis queries over the trace more efficiently.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Cheng Li, for their helpful comments and suggestions. This work is funded by project AIDA (POCI-01-0247-FEDER-045907), co-financed by the European Regional Development Fund (ERDF) through the Operacional Program for Competitiveness and Internationalisation (COMPETE 2020) and by the Portuguese Foundation for Science and Technology (FCT) under CMU Portugal, and by National Funds through the Portuguese funding agency FCT - Fundação para a Ciência e a Tecnologia with grant SFRH/BD/129771/2017 and within project UIDB/50014/2020.

References

[1] X. Zhou, X. Peng, T. Xie, J. Sun, C. Ji, W. Li, and D. Ding, "Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study," *IEEE Transactions on Software Engineering*, pp. 1–1, 2018.

[2] "Apache log4j," https://logging.apache.org/log4j/2.x/.

[3] "Simple logging facade for java (slf4j)," http://www.slf4j.org/.

[4] D. Yuan, Y. Luo, X. Zhuang, G. R. Rodrigues, X. Zhao, Y. Zhang, P. U. Jain, and M. Stumm, "Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems," in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI?14. USA: USENIX Association, 2014, p. 249?265.

[5] X. Zhao, Y. Zhang, D. Lion, M. F. Ullah, Y. Luo, D. Yuan, and M. Stumm, "Lprof: A non-intrusive request flow profiler for distributed systems," in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI?14. USA: USENIX Association, 2014, p. 629?644.

[6] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, 1978.

[7] A. Sampath and C. Tripti, "Synchronization in distributed systems," in *Advances in Computing and Information Technology*, N. Meghanathan, D. Nagamalai, and N. Chaki, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012.

[8] "Elastic Stack: Elasticsearch, Logstash and Kibana," https://www.elastic.co/pt/products/.

[9] F. Neves, N. Machado *et al.*, "Falcon: A practical log-based analysis tool for distributed systems," in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2018, pp. 534–541.

[10] L. De Moura and N. Bjørner, "Z3: An efficient SMT solver," in *TACAS '08/ETAPS '08*. Springer-Verlag, 2008.

[11] X. Zhao, K. Rodrigues, Y. Luo, M. Stumm, D. Yuan, and Y. Zhou, "Log20: Fully automated optimal placement of log printing statements under specified overhead threshold," in *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017, pp. 565–581.

[12] F. Mattern, "Virtual time and global states of distributed systems," in *Parallel and Distributed Algorithms*. North-Holland, 1988, pp. 215–226.

[13] T. Leesatapornwongsa, J. F. Lukman, S. Lu, and H. S. Gunawi, "Taxdc: A taxonomy of non-deterministic concurrency bugs in datacenter distributed systems," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 517–530. [Online]. Available: https://doi.org/10.1145/2872362.2872374

[14] F. Neves, R. Vilaça, and J. Pereira, "Black-box inter-application traffic monitoring for adaptive container placement," in *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, ser. SAC '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 259–266. [Online]. Available: https://doi.org/10.1145/3341105.3374007

[15] "Recap: High-performance Linux Monitoring with eBPF," https://www.weave.works/blog/recap-high-performance-linux-monitoring-with-ebpf/.

[16] "eBPF - The Future of Networking and Security," https://cilium.io/blog/2020/11/10/ebpf-future-of-networking/.

[17] "Sysdig and Falco now powered by eBPF," https://sysdig.com/blog/sysdig-and-falco-now-powered-by-ebpf/.

[18] "iovisor/bcc: Bcc - tools for bpf-based linux io analysis, networking, monitoring, and more," https://github.com/iovisor/bcc.

[19] "Neo4j graph platform – the leader in graph databases," https://neo4j.com/.

[20] "Janusgraph - distributed, open source, massively scalable graph database," https://janusgraph.org/.

[21] "Logrus: Structured logger for Go," https://github.com/sirupsen/logrus.

[22] "Apache kafka: A distributed streaming platform." https://kafka.apache.org/.

[23] C. J. Fidge, "Timestamps in message-passing systems that preserve the partial ordering," 1987.

[24] I. Beschastnikh, P. Wang, Y. Brun, and M. D. Ernst, "Debugging distributed systems: Challenges and options for validation and debugging," *Communications of the ACM*, vol. 59, no. 8, pp. 32–37, Aug. 2016.

[25] J. Mace and R. Fonseca, "Universal context propagation for distributed system instrumentation," in *Proceedings of the Thirteenth EuroSys Conference*, 2018, pp. 1–18.

[26] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica, "X-trace: A pervasive network tracing framework," in *NSDI '07*. USENIX Association, 2007.

[27] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag, "Dapper, a large-scale distributed systems tracing infrastructure," Tech. Rep.

[28] J. Mace, R. Roelke, and R. Fonseca, "Pivot tracing: Dynamic causal monitoring for distributed systems," in *SOSP '15*. ACM, 2015.

[29] J. Kaldor, J. Mace, M. Bejda, E. Gao, W. Kuropatwa, J. O'Neill, K. W. Ong, B. Schaller, P. Shan, B. Viscomi *et al.*, "Canopy: An end-to-end performance tracing and analysis system," in *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017, pp. 34–50.

[30] "Opentelemetry," https://opentelemetry.io/.

[31] "Zipkin," https://zipkin.io.

[32] "Lightstep," https://lightstep.com.

[33] M. Whittaker, C. Teodoropol, P. Alvaro, and J. M. Hellerstein, "Debugging distributed systems with why-across-time provenance," in *Proceedings of the ACM Symposium on Cloud Computing*. ACM, 2018, pp. 333–346.

[34] I. Beschastnikh, P. Liu, A. Xing, P. Wang, Y. Brun, and M. D. Ernst, "Visualizing distributed system executions," *ACM Trans. Softw. Eng. Methodol.*, vol. 29, no. 2, Mar. 2020. [Online]. Available: https://doi.org/10.1145/3375633

[35] A. Pi, W. Chen, X. Zhou, and M. Ji, "Profiling distributed systems in lightweight virtualized environments with logs and resource metrics," in *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, 2018, pp. 168–179.

[36] S. He, Q. Lin, J.-G. Lou, H. Zhang, M. R. Lyu, and D. Zhang, "Identifying impactful service system problems via log analysis," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 60–70.

[37] X. Zhang, Y. Xu, Q. Lin, B. Qiao, H. Zhang, Y. Dang, C. Xie, X. Yang, Q. Cheng, Z. Li *et al.*, "Robust log-based anomaly detection on unstable log data," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 807–817.

[38] M. Du, F. Li, G. Zheng, and V. Srikumar, "Deeplog: Anomaly detection and diagnosis from system logs through deep learning," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 1285–1298. [Online]. Available: https://doi.org/10.1145/3133956.3134015

[39] J. Lu, F. Li, L. Li, and X. Feng, "Cloudraid: Hunting concurrency bugs in the cloud via log-mining," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 3–14. [Online]. Available: https://doi.org/10.1145/3236024.3236071

[40] P. Reynolds, J. L. Wiener, J. C. Mogul, M. K. Aguilera, and A. Vahdat, "Wap5: black-box performance debugging for wide-area systems," in *Proceedings of the 15th international conference on World Wide Web*, 2006, pp. 347–356.

[41] B.-C. Tak, C. Tang, C. Zhang, S. Govindan, B. Urgaonkar, and R. N. Chang, "vpath: Precise discovery of request processing paths from black-box observations of thread and network activities." in *USENIX Annual technical conference*, 2009.