International Journal of Core Engineering & Management
Volume-7, Issue-04, 2022 ISSN No: 2348-9510

CAUSAL EVENT REPLAY FOR DEBUGGING DISTRIBUTED MICROSERVICE TRANSACTIONS IN KUBERNETES-BASED SYSTEMS

Srinivas Nune
Independent Researcher
Franklin, USA
srinivas.nune9792@gmail.com

Abstract

In a Kubernetes orchestrated microservice deployment, failures may happen across dynamically scheduled pods, asynchronous message channels, and independently versioned services, making it challenging to reproduce and localize using traditional logging. To build an integrated picture of how causal event replay can be used in transactions involving microservices running in Kubernetes, this paper compiles fifteen key and current references in distributed-systems causality theory, network-level and production tracing infrastructures, log-based anomaly detection, container orchestration, and causal-consistent replay debugging. The synthesis started from the logical-clock formulation of the happens-before relation proposed by Lamport, then followed the evolution of work in causal instrumentation from research prototypes to low-overhead, production-grade deployments in X-Trace, Dapper, and Pivot Tracing. The paper then reviews material on container orchestration in the literature that describes the Borg and Kubernetes scheduling semantics, which it demonstrates as why causal tracing was developed to aid with fault localization is made more challenging by the ephemerality and dynamic scheduling. On the basis of this, the paper discusses causal-consistent replay debugging for message-passing programs and the interactions of the causal-replay principle with other root-cause localization techniques like MicroRCA and Seer, as well as log-based anomaly detection techniques shown to be applicable in the field, such as DeepLog. All these contributions are proposed to be unified into a coherent debugging workflow for Kubernetes environments, via a five-stage conceptual pipeline-instrumentation, causal capture, event-log assembly, deterministic replay, and root-cause localization. The discussion then covers benchmark infrastructure like the DeathStarBench suite, which provides a representative microservices topologies to be used as a set of comparison points for pipelines, and a container scheduling survey that catalogues the behaviors of the containers that any replay-based debugger would need to support. The authors conclude that causal event replay, combined with orchestration-aware instrumentation, provides a principled approach to reproducible debugging of failures in non-deterministic microservices and that there are open challenges in overhead management, non-determinism in scheduling, and incompleteness in causal graph representation, all of which were outstanding issues in the literature.

Index Terms—causal event replay, distributed debugging, Kubernetes, microservices, logical clocks, distributed tracing, root-cause localization, container orchestration, deterministic replay, anomaly detection.

International Journal of Core Engineering & Management
Volume-7, Issue-04, 2022 ISSN No: 2348-9510

I. INTRODUCTION

Microservice architectures break down the application into a collection of individually deployable services that use a network to communicate, and Kubernetes is the service scheduling, scaling, and healing platform for these services across clusters of machines [1, 2]. The modularity and independent scalability of the architecture also change the type of software failures: the first transaction that a user sees could now cross several dozens of services, asynchronous queues and dynamically re-scheduled pods to complete. A failure of such a transaction or a degradation in its latency is often difficult to detect in a single service, as typically the chain of services that caused the failure is spread across independently maintained logs and execution contexts [1, 6]. Traditional debugging methods such as manual log analysis and single-process breakpoint debugging rely on a common sequential debugging history which microservices don't exist [1].

Even before the invention of microservices, the theory of reasoning about ordering in distributed systems was well developed and Lamport formalized the happens-before relation and introduced logical clocks as a tool for imposing the same partial order on events that do not share a common global clock [3]. This insight is the foundation for nearly all subsequent distributed tracing and causal monitoring systems: any method that attempts to answer the question "what happened, and in what order" in a collection of independent processes must first be able to determine the causal ordering of events that may have been recorded on machines with unsynchronized clocks [4, 5]. Causal event replay takes the idea of passive observation a step further, and is the extension of it to active reconstruction: Given that event A occurred before event B, the replay-capable debugger would be able to capture enough causal information to recreate the same relative ordering of the two events in a controlled environment so that a developer could step through the failure deterministically [5].

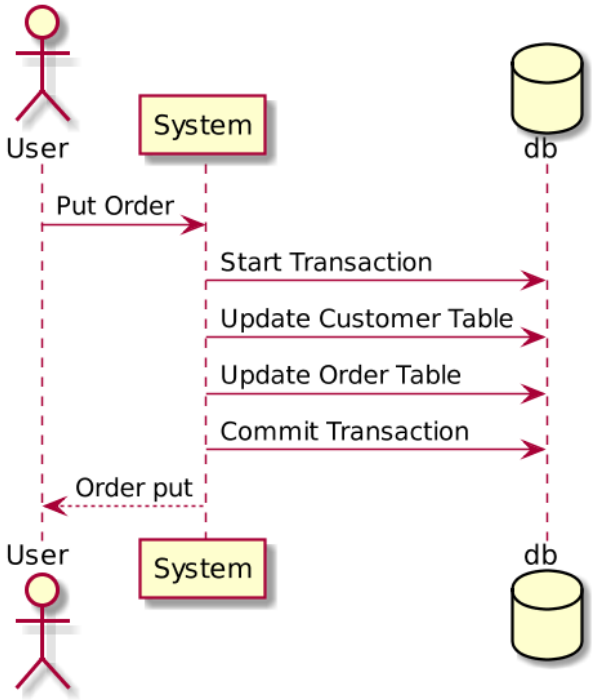

Figure 1: Evolutionary Timeline of Causality, Tracing, and Orchestration Research Underlying Causal Event Replay (RedHat Developer,2018)

International Journal of Core Engineering & Management
Volume-7, Issue-04, 2022 ISSN No: 2348-9510

II. BACKGROUND: DEBUGGING CHALLENGES IN DISTRIBUTED AND CONTAINERIZED SYSTEMS

The problem of debugging a distributed system is in general more difficult than its single-process counterpart, as failures are often the result of failures that occur in other systems, but involve only two or more independently correct components that interact with each other [6]. The diagnosis of distributed computing systems is complicated by three properties: the partial failure of some components, the asynchrony of message delivery order and timing and the concurrency of the interleaving of independent processes that are hard to reproduce as needed [7]. These properties make it possible for a bug to only appear with a rare interleaving of messages, and the same request can be run repeatedly without causing the same bug in the code, defeating the iterative development process most developers use when debugging serial programs [7] [8].

On top of this difficult base are added the "containerized orchestration platforms". In [9] Verma et al. explain how heterogeneous workloads are scheduled by Google's Borg system across shared clusters, which are continuously reallocated to machines due to resource pressure, failures and priorities changes. Kubernetes adopts and builds on this approach to scheduling, and Turin et al. present a formal operational model that documents the interactions between pods, replica sets, and controllers that ensure a desired cluster-state, in the face of continuous churn [10]. Also, as revealed by Rodriguez and Buyya's taxonomy of container-based cluster orchestration systems, there are many different scheduling policies, placement constraints, and networking overlays between orchestration systems, so things like an IP address used to correlate events are only temporary and could be assigned to a different unrelated workload in minutes [10]. This instability is confirmed by the survey of container scheduling techniques conducted by Ahmad et al [11], which includes the load balancing, energy efficiency, and affinity-aware placement scheduling objectives that lead to the frequent change of the physical/ logical location of a microservice instance in the lifetime of a long running application.

Debugging Challenge	Source of the Challenge	Referenced Contribution Addressing It
Ephemeral, dynamically scheduled pods	The scheduler continuously creates, migrates, and terminates pods across nodes, breaking static identifiers used for correlation.	Container scheduling survey; formal Kubernetes model
Non-deterministic message interleaving	Concurrent requests across services can arrive in different orders on different executions, making bugs hard to reproduce.	Causal-consistent replay debugging; Lamport clocks
Fragmented, siloed logs	Each service and sidecar emits independent logs with no shared context, obscuring the end-to-end transaction path.	X-Trace; Dapper; Pivot Tracing
Cascading, cross-service failures	A fault in one microservice propagates latency or errors to dependent services, obscuring the true origin.	Root cause detection in SOA; MicroRCA; Seer
High cardinality of interacting components	Benchmark microservice topologies involve tens of loosely coupled services communicating over varied protocols.	DeathStarBench benchmark suite; container orchestration taxonomy

Table 1: Debugging Challenges in Kubernetes-Orchestrated Microservice Systems

Five problems from the literature that typically occur are summarized in Table 1. Each challenge will be a reminder of the need to not simply copy and paste a method of debugging a monolithic

or even simple client-server application to a microservice deployment orchestrated by Kubernetes. The rest of this paper explores the development of causal tracing and replay techniques, with most of them developed independently of the orchestration literature to address the aspects of ordering and correlation in this problem, and how such two streams of work can be combined and used to the advantage of both.

III. FOUNDATIONAL CAUSALITY AND TRACING TECHNIQUES

A. Logical Clocks and the Happens-Before Relation

Lamport's original formulation stipulates a partial order for events of a distributed system: an event A is before an event B if either A is in the same process as B and A occurs before B, or A is a send operation in one process and B is a receive operation in another process, and the send operation in the former process is the one that was sent and received by the latter process [12]. Logical clocks are clocks that assign monotonically increasing counters to events, ensuring that this happens-before relation is maintained, without requiring any physical synchronization between the two observers who observe them [12]. This is because any causal replay mechanism should at least be able to reconstruct an ordering that is consistent with the happens-before relation [12, 13] before it can claim to be a faithful reproduction of a failure.

B. From Network-Level Tracing to Production-Scale Infrastructure

Fonseca et al. proposed an X-Trace pervasive tracing framework that carries metadata from network level to application level, thus enabling a single logical task (e.g., client request) to be traced as a task tree across multiple administrative domains [13]. X-Trace showed that causal metadata can be added unobtrusively to existing network protocols, but it was designed for an environment in which the researcher, not a live production system's operator, would control the instrumentation [13]. To handle the transition to continuous production use, Sigelman et al. added sampling, in which a subset of requests is fully traced, and a design that only added additional common middleware and threading libraries already deployed at scale to the applications/processes [14]. Originating from Dapper, a representation of spans where each unit of work is paired with a trace identifier and a parent span identifier, enables an entire distributed call tree to be reconstructed after the fact from independently emitted span records, and that's how many contemporary tracing systems are modelled [14].

This was extended by Mace et al. with the addition of an ad hoc causal query, where an operator could specify a query without having to redeploy any instrumented code, and a relational "happened-before" join operator that enables metrics from one component to be related to events in a causally related but physically remote component [14]. This is especially important for microservices when the developer is trying to understand what event from upstream services caused a downstream service to become slow, for example; this is a causal question, for which the developer needs not have thought in advance when instrumenting the services. In addition to these tracing systems, Kim et al. formulated root-cause detection techniques for service-oriented architectures (SOAs) that statistically correlate anomalies among service call graphs, which is an early precursor to the graph-based root-cause localization technique discussed in Section V [14].

International Journal of Core Engineering & Management
Volume-7, Issue-04, 2022 ISSN No: 2348-9510

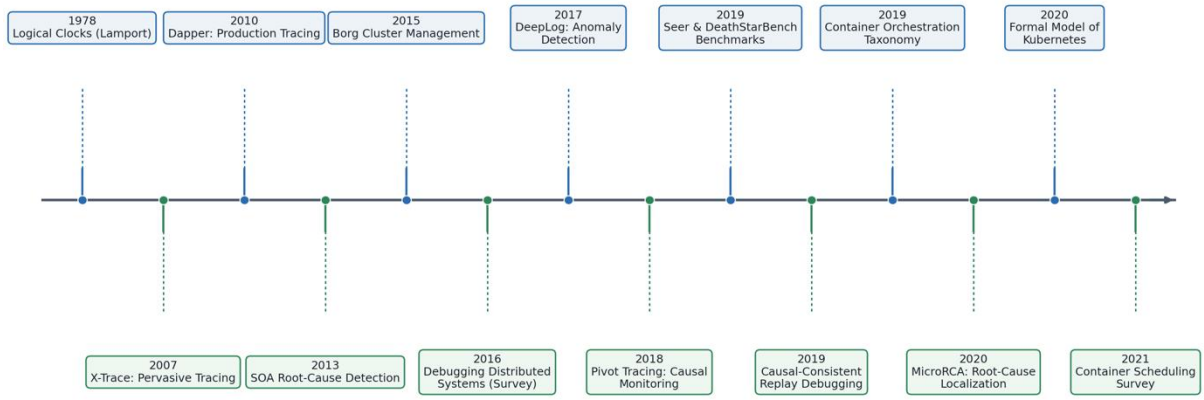

Figure 2: Evolutionary Timeline of Causality, Tracing, and Orchestration Research Underlying Causal Event Replay

Between 1978 and 2021, the field of causal ordering theory, the field of production tracing infrastructure, and the field of container orchestration research grew independently but at the same time in parallel and converging paths as illustrated in Figure 2. It is important to note in Figure 2 that the horizontal distance does not correspond to time, but it does correspond to the ordered nature of the points; the close grouping of points between 2016 and 2021 makes it easier to read the graph.

IV. CONTAINER ORCHESTRATION AND KUBERNETES-BASED MICROSERVICES

The formal model of Kubernetes offered by Turin et al. describes the pods, deployments, and reconciliation loop which continuously checks the observed cluster with the desired cluster as described in the controller's declarative configuration. This reconciliation loop is critical to the self-healing nature of Kubernetes and is also one of the root causes of non-determinism that can help make debugging difficult: When a pod fails, it is simply recreated, possibly on a different node, with a new name, so techniques that depend on stable process or network identifiers must instead focus on the logical service or transaction.

To address this need for representative experimental infrastructure, Gan et al. have developed an open-source collection of microservice benchmark applications, called DeathStarBench, which covers social networking, media services, and e-commerce workloads, and which is specially crafted to highlight the hardware and software implications of realistic microservice topologies for cloud and edge deployments [15]. DeathStarBench applications are made up of tens of loosely coupled services communicating via a variety of protocols, making them a natural choice for evaluating causal tracing and replay systems; they feature topologies that are complex enough to expose cross-service causal dependencies that simpler test applications would not. In an accompanying paper, Gan and colleagues presented Seer, a system built from a collection of fine-grained tracing data from a microservice application that can be used to learn a predictive model of latency degradation; the learned causal traces were used not just to explain a failure, but to

predict potential performance problems before they became user-facing faults [15].
Ahmad et al.'s survey placed this performance and debugging concerns in a broader context of evaluating container scheduling techniques, which are divided into various categories based on the goals they achieve such as resource utilization, load balancing, energy efficiency and quality of service guarantees . The survey notes that scheduling decisions are often not 'visible' to application-level observability tools, so a purely application-layer-based causal tracing system may not provide insight into performance anomalies caused by placement decisions made by the scheduler, e.g., placing two resource-contending services on the same node. The observation encourages the incorporation of orchestration-aware metadata, such as node identifier, scheduling-event identifier etc., in the causal graphs generated when tracing a system, which is further discussed in Section VII.

V. CAUSAL-CONSISTENT REPLAY DEBUGGING

In a causal-consistent replay debugging, Lanes, Palacios and Vidal propose to enable a developer to step forward and backward through the execution of a program, and they show that any replayed execution will maintain the causal order of the program's messages. Causal-consistent replay only needs to preserve the relative order of causally dependent events, so it is tolerant of scheduling jitter and network latency variation within a Kubernetes cluster that can never be eliminated pre-meditated by the devs. This is important for microservice debugging since the exact timing of an original transaction cannot be reproduced on a shared, multi-tenant cluster for diagnostics purposes.
The causal-consistent replay technique assumes they have a causal event log like Dapper or Pivot Tracing that includes enough metadata about the messages to explain the order of events, as well as the message's impact on the program state. With the existence of this type of log, replay debugging can be paired with automatic root cause localization. In MicroRCA, Wu et al. build an attributed graph that connects services and the physical or virtual machines that provide them and propagate the scores of anomalies through this graph to find the service most likely to cause an observed degradation in performance [15] without requiring any prior knowledge of the type of the fault. The graph construction algorithm developed by MicroRCA makes use of the same causal service-call relationships that are already captured by tracing systems, therefore can be considered as a downstream analysis step, that also requires the causal event log that a pipeline supporting replay would require [15].
Log-based anomaly detection can give a complementary signal for more replay-based investigation of a specific part of the causal graph. Du, et al. propose DeepLog models for system logs, which are encoded as sequences of log-key patterns, and a recurrent neural network to learn a normal execution pattern for an application and detect deviations as candidate anomalies. For microservice transactions, DeepLog-style anomaly detection can be applied to determine which subgraphs of the large transaction log are not as they would normally be, and thus limit the scope of a follow-on causal-consistent replay session to the transactions most likely to include the fault.

International Journal of Core Engineering & Management
Volume-7, Issue-04, 2022 ISSN No: 2348-9510

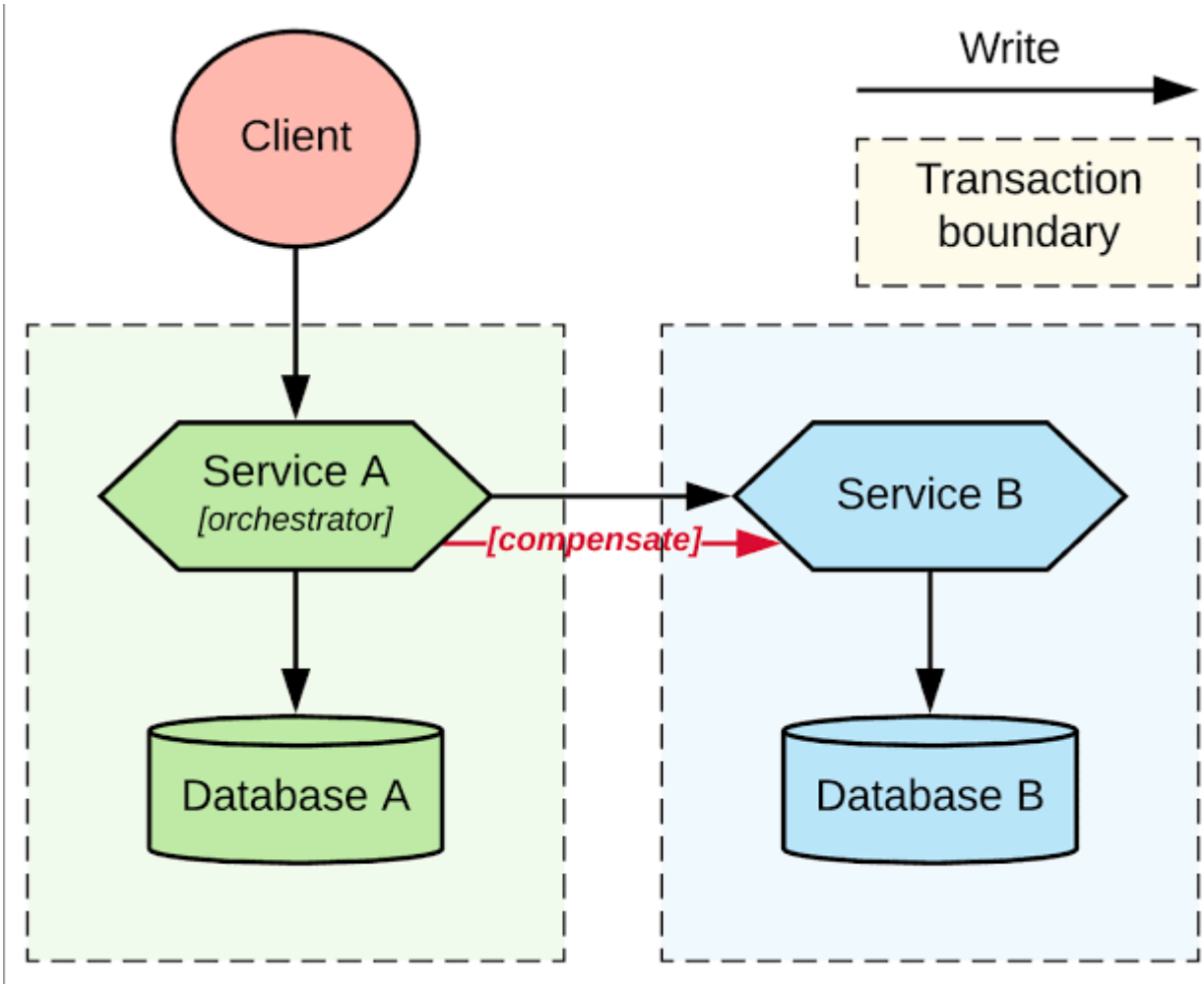

Figure 3: Conceptual Pipeline Integrating Causal Capture, Deterministic Replay, and Root-Cause Localization

A five-stage conceptual pipeline is shown in Figure 3 that summarizes the mechanisms described in Sections III-V. Dapper and Pivot Tracing provide the basis for instrumentation to add causal context to inter-service calls. Causal capture uses the metadata of the Lamport ordering technique to create happens-before relationships [2]. Event-log assembly is an aggregation of spans that are broken up into a causal graph per transaction, built on top of the data models introduced in X-Trace and Dapper. A deterministic replay is a technique that uses the guarantee of causal-consistency described by Lanese et al. to replay the transaction in an isolated environment. Once root cause localization has been done by applying graph-based methods, e.g. MicroRCA, optionally with the help of DeepLog-like anomaly scoring, the responsible service is discovered [15].

VI. COMPARATIVE ANALYSIS OF DEBUGGING APPROACHES

Table 2 presents a comparison of the six most relevant systems and techniques discussed in this synthesis along the factors of the core mechanism, the causality model and the prime use case,

with the aim of evaluating their complementary use in a single causal-replay pipeline.

System	Year	Core Mechanism	Causality Model	Primary Use Case
Lamport Logical Clocks	1978	Scalar counters attached to events establish a partial happens-before order.	Partial order (logical clocks)	Foundational ordering theory for distributed systems
X-Trace	2007	Metadata propagated across network layers reconstructs end-to-end causal task trees.	Explicit causal edges (path reconstruction)	Pervasive network-layer tracing
Dapper	2010	Sampled, low-overhead spans linked by trace and span identifiers form request trees.	Parent–child span causality	Production-scale latency tracing at Google
Pivot Tracing	2018	Dynamic instrumentation combined with a relational "happened-before" join operator.	Cross-component causal joins	Ad hoc causal monitoring without redeployment
Causal-Consistent Replay Debugging	2019	Message-passing traces are replayed while preserving the original causal message order.	Causal-consistent replay	Deterministic reproduction of concurrency bugs
MicroRCA	2020	Anomaly propagation is modeled over a service-call and hosting-machine attributed graph.	Graph-based causal propagation	Root-cause localization in production microservices

Table 2: Comparative Overview of Causality-Aware Debugging and Tracing Systems

This comparison reveals a number of patterns. First, there is a historical shift from theoretical ordering constructs, like Lamport's logical clocks, to increasingly production-oriented instrumentation, such as Dapper's low overhead sampling strategy and Pivot Tracing's dynamic query capability. Second, causal-consistent replay debugging and MicroRCA do not compete: the former supports manual inspection by reconstructing a faithful execution, the latter supports automatic localization by automating the latter step; and a well-developed debugging pipeline is likely to benefit from both using a common causal event log [8, 15]. Third, all the surveyed tracing/replay systems were not designed to be orchestration-aware, meaning orchestration-aware metadata, as described in the surveys by Ahmad et al. and Rodriguez and Buyya must be explicitly added.

Another comparative factor is overhead. Low runtime overhead was a key design consideration for sampling since continuous full tracing was not seen as an option for production performance at scale by the designers of Dapper. By contrast, Lanese et al. describe causal-consistent replay debugging for message-passing programs, and the cited literature does not show that the causal-consistency guarantees offered by the method can be achieved without similar overhead reduction measures at the Dapper-like scale. A meaningful boundary condition for any practical causal replay system for Kubernetes is that it may be necessary to selectively apply the fidelity required for deterministic replay, such as by only applying it after an anomaly detector, like DeepLog, identifies a candidate transaction, instead of applying it to all traffic.

VII. INTEGRATION CHALLENGES AND FUTURE DIRECTIONS

We can derive several open challenges from the analysis of the causality and orchestration literature. The first is identity stability: since the pods are ephemeral and new pods are assigned with new identities when they are rescheduled or fail, the event log needs to be anchored by logical service and transaction identities, not transient pod or IP level identities, which is implicit in Turin et al.'s formal model but not explicitly addressed in existing tracing systems that were built before container orchestration became mainstream.

The second challenge is related to non-determinism caused by the scheduler. The scheduling goals in Ahmad et al.'s survey can lead to two logically identical transactions being scheduled over two different topologies on two separate executions, even though the order of messages is kept exactly the same, which results in a replayed execution having different performance characteristics from the original execution. This would require the causal event log to track scheduler level placement decisions in addition to the application-level placement decisions, which are not available in any of the systems mentioned in the references, but in line with what Rodriguez and Buyya were suggesting in their call for orchestration systems with richer observability interfaces.

Third challenge is the tools for the evaluation. Gan et al. provide a viable testbed for the integrated causal-replay pipeline to test against a realistic microservice topology in their DeathStarBench suite and Seer's work on predictive performance modelling using accumulated tracing data indicates that causal event logs gathered for debugging can be informatively used for proactive anomaly anticipation, not just for reactive root-cause analysis. A direction that is not shown directly in the literature read, but implied in the presented work and is to combine the Seer-style predictive modelling with the DeepLog-style anomaly detection over the same causal event log.

Lastly, scalability of causal-consistent replay is left unanswered in the case studied in this paper. Lanese et al. show how to implement message-passing programs much smaller than a production Kubernetes cluster running dozens of interacting services and how to reconcile causal-consistency guarantees with Dapper style sampling, where only a minority of traces are kept, and the rest discarded, only to be promoted to full fidelity when an anomaly detector or root-cause localizer deems them sufficiently diagnostic [15].

VIII. CONCLUSION

The authors have gathered the 15 papers that cover four decades of distributed systems research and combined them into a single source for an overview of the causal event replay for debugging transactions that are distributed across Kubernetes-based microservices. Starting from the original happens-before relation introduced by Lamport, the evolution of causal instrumentation into X-Trace, Dapper and Pivot Tracing was explored, as well as how container orchestration research, defined by the formal model of Borg, and the research into container scheduling, such as container scheduling surveys, introduces ephemerality and scheduling dynamism which make causal reconstruction more complicated. Further, how causal-consistent replay debugging can be combined with root-cause localization (RCL) techniques like MicroRCA with log-based anomaly detection techniques like DeepLog was examined. The proposed conceptual five-stage pipeline provides one possible way of structuring these individually developed contributions as a coherent debugging process, and the contributions from DeathStarBench and Seer represent a plausible approach for empirically testing such a pipeline with realistic microservices topologies. The synthesis also uncovered some open problems and challenges that are currently unsolved in the

literature surveyed until 2021, such as the identity stability under pod rescheduling, the non-determinism induced by schedulers, and scalability of causal-consistency guarantees at production scale. These challenges will likely require further integration between the tracing and orchestration research communities than the literature reviewed allows, in the sense of making the placement decisions made by the scheduler first-class causal events along with the application-level placement events.

ACKNOWLEDGMENT

The preferred spelling of the word "acknowledgment" in America is without an "e" after the "g". Avoid the stilted expression, "One of us (R. B. G.) thanks . . ." Instead, try "R. B. G. thanks". Put applicable sponsor acknowledgments here; DO NOT place them on the first page of your paper or as a footnote.

REFERENCES

1. Beschastnikh, I., Wang, P., Brun, Y., & Ernst, M. D. (2016). Debugging distributed systems. Communications of the ACM, 59(8), 32–37. https://doi.org/10.1145/2909480Lamport, L. (1978).
2. Time, clocks, and the ordering of events in a distributed system. Communications of the ACM, 21(7), 558–565. https://doi.org/10.1145/359545.359563.
3. Fonseca, R., Porter, G., Katz, R. H., Shenker, S., & Stoica, I. (2007). X-Trace: A pervasive network tracing framework. In Proceedings of the 4th USENIX Symposium on Networked Systems Design and Implementation (pp. 271–284). USENIX Association.
4. Sigelman, B. H., Barroso, L. A., Burrows, M., Stephenson, P., Plakal, M., Beaver, D., Jaspan, S., & Shanbhag, C. (2010). Dapper, a large-scale distributed systems tracing infrastructure (Technical Report). Google, Inc.
5. Mace, J., Roelke, R., & Fonseca, R. (2018). Pivot tracing: Dynamic causal monitoring for distributed systems. ACM Transactions on Computer Systems, 35(4), Article 11. https://doi.org/10.1145/3208104
6. Kim, M., Sumbaly, R., & Shah, S. (2013). Root cause detection in a service-oriented architecture. ACM SIGMETRICS Performance Evaluation Review, 41(1), 93–104. https://doi.org/10.1145/2465529.2465753
7. Du, M., Li, F., Zheng, G., & Srikumar, V. (2017). DeepLog: Anomaly detection and diagnosis from system logs through deep learning. In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (pp. 1285–1298). Association for Computing Machinery.
8. Lanese, I., Palacios, A., & Vidal, G. (2019). Causal-consistent replay debugging for message passing programs. In J. A. Pérez & N. Yoshida (Eds.), Formal Techniques for Distributed Objects, Components, and Systems (FORTE 2019, LNCS vol. 11535, pp. 167–184). Springer.
9. Verma, A., Pedrosa, L., Korupolu, M., Oppenheimer, D., Tune, E., & Wilkes, J. (2015).

Large-scale cluster management at Google with Borg. In Proceedings of the Tenth European Conference on Computer Systems (Article 18). Association for Computing Machinery.

10. Rodriguez, M. A., & Buyya, R. (2019). Container-based cluster orchestration systems: A taxonomy and future directions. Software: Practice and Experience, 49(5), 698–719. https://doi.org/10.1002/spe.2660

11. Ahmad, I., AlFailakawi, M. Gh., AlMutawa, A., & Alsalman, L. (2021). Container scheduling techniques: A survey and assessment. Journal of King Saud University – Computer and Information Sciences, 34(7), 3934–3947.

12. Turin, G., Borgarelli, A., Donetti, S., Johnsen, E. B., Tapia Tarifa, S. L., & Damiani, F. (2020). A formal model of the Kubernetes container framework. In Leveraging Applications of Formal Methods, Verification and Validation: Verification Principles (ISoLA 2020, pp. 558–577). Springer.

13. Gan, Y., Zhang, Y., Cheng, D., Shetty, A., Rathi, P., Katarki, N., Bruno, A., Hu, J., Ritchken, B., Jackson, B., Hu, K., Pancholi, M., He, Y., Clancy, B., Colen, C., Wen, F., Leung, C., Wang, S., Zaruvinsky, L., … Delimitrou, C. (2019). An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (pp. 3–18). Association for Computing Machinery.

14. Gan, Y., Zhang, Y., Hu, K., Cheng, D., He, Y., Pancholi, M., & Delimitrou, C. (2019). Seer: Leveraging big data to navigate the complexity of performance debugging in cloud microservices. In Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (pp. 19–33). Association for Computing Machinery.

15. Wu, L., Tordsson, J., Elmroth, E., & Kao, O. (2020). MicroRCA: Root cause localization of performance issues in microservices. In NOMS 2020 – 2020 IEEE/IFIP Network Operations and Management Symposium (pp. 1–9). IEEE.