The Journal of Systems & Software 181 (2021) 111015

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

In Practice

Improving observability in Event Sourcing systems☆

Stanley Lima [a,*], Jaime Correia [a], Filipe Araujo [a], Jorge Cardoso [a,b]

[a] University of Coimbra, Centre for Informatics and Systems of the University of Coimbra, Department of Informatics Engineering, 3030-290, Coimbra, Portugal
[b] Huawei German Research Center (GRC), 80992 Munich, Germany

ARTICLE INFO

Article history:
Received 9 March 2020
Received in revised form 1 March 2021
Accepted 20 May 2021
Available online 9 June 2021

Keywords:
Event Sourcing
Logging
Tracing
Distributed systems
Microservices

ABSTRACT

Event Sourcing (ES) systems use an event log with the double purpose of keeping application state and providing decoupled communication. While ES systems keep track of all business events, other untracked events, either from internal components or from external systems may still cause failures. Determining the root cause of such failures usually involves complex procedures based on replaying the event log. Unlike this, in distributed systems, developers often instrument the source code, for the sake of improving observability and perform *tracing* on workflows and data.

Adding tracing to ES thus seems like an unexplored and powerful approach to improve the observability of the system. In this paper, we suggest possible implementations of the idea and discuss their merits. These include the adoption of well-known tracing-related tools and standards in ES systems, with the respective advantages for root-cause analysis, anomaly detection, profiling and others.

1. Introduction

Event Sourcing is a design approach where distributed applications keep their state as a sequence of state-changing operations. Instead of storing mutable objects, applications keep an immutable sequence of changes to such objects. Changing state and writing an event to the log is one single, therefore atomic, operation. The log becomes the authoritative source of truth, offering eventual consistency, as events propagate to different parts of the distributed application. This design entails a strongly decoupled architecture, typical of publish–subscribe systems (Clayman et al., 2010), while providing reliable auditing and logging functionality. For example, developers may bring the application back to a previous execution state, by replaying the events in the log. This feature is particularly powerful, not only for the sake of failure recovery and state management (most services can be stateless), but also for debugging and to experiment alternative *what-if* scenarios. Furthermore, it makes the system's data schema more flexible, as it becomes possible to recover field values or even calculate additional ones from the log.

The literature (Fowler, 2005a) mentions ease of debugging as one of the advantages of having the event log and the ability to do partial and branched replays. In this context, ES is often presented as a way to deal with some of the complications created by distribution. However, this approach preserves no metadata about the system, which is necessary to observe failures and assert the correctness of the ES mechanisms themselves. To investigate and correct these situations, developers normally have to resort to unstructured logging. It is worth noting that the cost of branching or replaying the log is very high, both in source code complexity and resources, as a portion of the events will essentially have to be reprocessed.

At the same time, the rising trend of fine-grained distributed systems using microservices and the Function-as-a-Service (FaaS) paradigm made applications more fragmented than ever. Essentially, while breaking the monolith simplifies programming and provides horizontal scalability, complexity goes into the distributed system, which become difficult to observe and comprehend (Pautasso et al., 2017). Getting a consistent picture of state involves causality relationships. End-to-end tracing (Xiang et al., 2016) is one such method, preserving a portion of causality relationships at the cost of requiring code instrumentation, for the sake of system analysis and debugging.

As we discuss in Section 2, tracing and ES are usually taken as separate paradigms. To some degree, ES is presented as a complete solution for debugging any issue, as it can theoretically recreate state at any point in time for analysis. State-changing events, however, lack information of the system's internal variables and code execution path, not to mention interaction with external components. Log events lack structured metadata, thus failing to provide end-to-end tracing information. Operators thus

☆ Editor: Earl Barr.

* Corresponding author.
E-mail addresses: stanleylima@dei.uc.pt (S. Lima), jaimec@dei.uc.pt (J. Correia), filipius@dei.uc.pt (F. Araujo), jcardoso@dei.uc.pt (J. Cardoso).

https://doi.org/10.1016/j.jss.2021.111015
0164-1212/

lack good means to recover the workflows taking place inside the distributed system. On the other hand, tracing is an observability tool that uses source code instrumentation to pass specific data, known as baggage, between different parts of the application at run-time. Baggage relates and stitches the different services of the application together, for the sake of *a posteriori* debugging of their interactions. Tracing has a meta-functionality; it is not meant to be the source of truth of business data. Tools like zipkin.io (2021) or Jaeger (2021) demonstrate the acceptance of tracing in the industry.

In Section 3, we review other work that explores or enriches ES in ways that have similarities to our own, *e.g.*, to recover the state of the distributed system or for simulation purposes. Interestingly, our work also displays similarities to the topic of process mining, where the goal is to retrieve operational processes from system logs. We, therefore, briefly review the topic in this section.

In Section 4 we formally reason about ES and tracing, their differences and complementarities. We propose to use them together in Section 5. We do this by keeping internal events that, while not changing the business state of the application, are either in the causation or consequence chain of such changes. Processing and storage restrictions aside, this would enable a complete recovery of the execution state of the entire system.

Our goal is to increase observability, by recovering causality inside the entire system, across separate components. Our contribution is an approach to leverage distributed tracing, to preserve relationships between events, as well as connecting these to software metadata, such as versions, ultimate event publishers and subscribers, and other runtime data, such as exceptions and logging. This will eventually make the identification of failure root causes much cheaper, by resorting to proven standards, tools and approaches.

To better illustrate our idea, in Section 6, we explore a Guitar Store as an ES use case and analyze scenarios where an event may be missing from the log, duplicated, or incorrect. We then address these scenarios with and without tracing and evaluate the differences. We extend this analysis to a case where system developers and operators can take advantage of tracing to eliminate unused code.

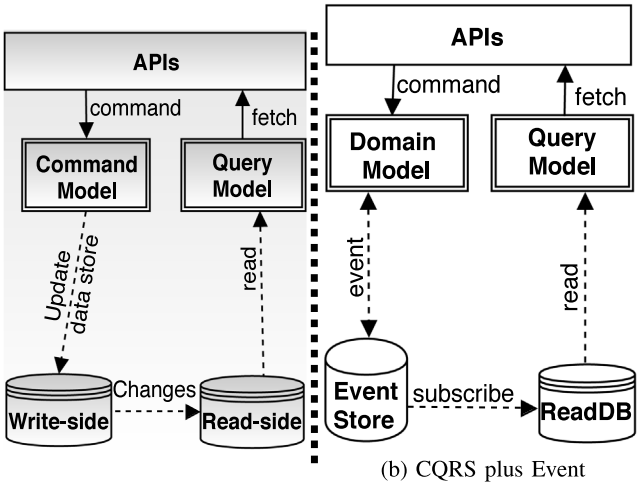
We discuss possible implementations for adding tracing to ES and elaborate on the advantages of the idea in Section 7. In Section 8, we conclude the paper and enumerate some consequences of our proposal for developers and operators.

2. Background

According to Vernon (2013), Domain-Driven Design (DDD) is an approach to complex software development, in which software developers: (i) focus on a domain; (ii) work in close collaboration with domain experts; (iii) and use a ubiquitous language within the *bounded* context of a domain.

DDD divides the problem of creating software into *bounded contexts* aligned with real-world domains (*e.g.*, an ontology). Reducing the scope of the problem enables *domain experts* to have a deep understanding of the language and business rules within their own context. Bounded contexts promote division of concerns in the implementation and communication between different contexts through public Application Programming Interfaces (APIs), thus bearing some similarities with microservices (Newman, 2015). Both push complexity from within components to their interactions. Patterns like CQRS and ESdo an attempt on regulating such interactions.

Refer to Fig. 1(a). The CQRS is a design pattern that separates the data reading and writing models (Debski et al., 2017). The notion of separating read and write concerns, originality published by Meyer (1988) came as a way to deal with the observed



(a) Simplified CQRS setup. (b) CQRS plus Event Sourcing setup.

Fig. 1. CQRS architecture with and without ES.

mismatch between requirements for reading and writing a data model. A schema is normally optimized for one of those use cases, and other strategies like caching used to compensate the other. A good example would be normalizing the data schema on a relational database; typically one can choose to denormalize to improve reads, but this penalizes writes.

2.1. Event Sourcing (ES)

ES arises as a system design style to manage state change in distributed systems, while maintaining a set of desirable functional and non-functional properties, such as auditability, scalability, and decoupling. ES draws on concepts from other styles and approaches, such as publish–subscribe systems (Eugster et al., 2003) and CQRS.

In an ES system, the state of the application is determined by a sequence of events (Overeem et al., 2017). All changes to the application business state are stored as a sequence of events in an immutable log (Gousios et al., 2012), usually in the form of a source code artifact; i.e., each business state change produces an event that goes to the log.

In an ES setup, preserving raw events provides a complete audit trail, to simulate execution of the system from the start, thus allowing to go back in time, create alternative realities and understand the behavior of a system, at a given point in the past. For the sake of illustrating the idea, we can provide a simplistic comparison between ES and the ledger of a banking account, as the latter registers deposits and withdrawals in sequence, to become the single source of truth. Running a query to determine the total balance may imply going through the ledger to add all movements. Keeping a log of all changes is common in applications such as word processors, configuration files, version control systems, like Git or SVN, as these typically use techniques to track and provide control of changes, to resolve bugs, perform test cases among others.

A natural way to understand and implement the domain model in ES is to use DDD through commands and events, wherein commands are ordered from a user request "imperative sentence" sent to aggregates (aggregates are a cluster of domain objects treated as a single unit (Evans, 2004), for example a post and its comments). Commands may trigger events that represent an action that changed the state of the system, i.e., something that happened in the past, like "ticket purchased" or "room reserved".

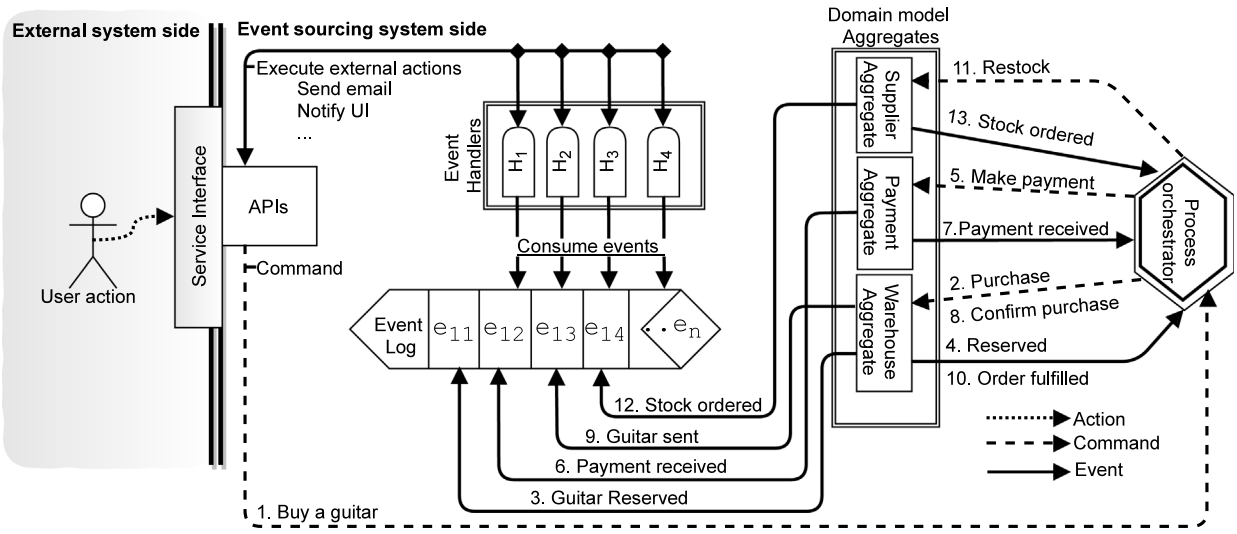


Fig. 2. Overview of ES pattern architecture.

ES systems usually implement a strict separation of the read and write sides, thus following a CQRS pattern, as we show in Fig. 1(b).

Fig. 2 presents the core of an ES system. This is an e-commerce example, where an outside agent buys a guitar, by sending a command to the *Supplier Aggregate*. This operation involves multiple validation steps in the inventory and payment details. Once the aggregate accepts the command, it sends a message to the process orchestrator, as aggregates do not communicate directly (Evans, 2004). For this purpose, the SAGA (Garcia-Molina and Salem, 1987) pattern is commonly used in DDD, to orchestrate interactions between aggregates (Ritchie, 2016). The process orchestrator receives the event and issues a validate payment command to the *Payment Aggregate*, then back to the *Supplier Aggregate* to confirm the purchase and finally to the *Warehouse Aggregate*, to wrap and send the guitar. In the process, as the aggregates change state, they persist domain events to the log in steps 4, 7, and 8. Note that, in ES, queries are often not performed directly on the event log. Aggregates have associated command handlers that are responsible for retrieving a stream of events from the event log, building a snapshot of the state from the stream and performing queries from the snapshot.

So far, we have seen how aggregates create and persist events from the commands they receive. Event-driven publish–subscribe architectures also include event handlers that subscribe to events, thus playing the complementary role of publishers. Event handlers can also provide third-party integration to applications through the forwarding of events about something that happened in the domain.

ES systems keep track of state changing events, offering a number of benefits concerning accountability, loose coupling, or temporal queries over application state. However, different types of faults might corrupt the event log. There are typically three cases: (*i*) an event might have incorrect data, (*ii*) it might be out of order, (*iii*) it should have not existed or it might me missing. For example, some software might have mixed orders from different costumers, thus writing incorrect data in the event log. In another situation, the delivery semantics of a message might not have been properly implemented and the same event ends up duplicated, *e.g.*, the user gave a single order to purchase some item, but a software glitch repeated the order. It could also happen that a purchase order fails due to insufficient funds, but the respective event fails to reach the event log. As a final example, improperly defined transaction isolation on the event log, could result in out-of-order events: switching a purchase and a payment event may result in some part of the system charging the client twice. We may also consider an additional case, pointed out by Hohpe Hohpe (2005), (*iv*) where an action should respond to some event, but fails, *e.g.*, because some human operator did not package and send the item.

ES makes auditing the state possible, thus enabling compensatory actions, like writing an event to the log to send the missing package. In some cases, one might need to replace or reorder the events. For example, an invoice might be wrong and need replacement. One might have to rewind and rerun the log, doing the correction where appropriate. In *Retroactive Event* (Fowler, 2005c) (which derives from the *Parallel Model* (Fowler, 2005b')), Martin Fowler proposes an automatic mechanism for doing this. This essentially works by either allowing the system to run copies of itself, or by using a single copy that is aware of multiple *embedded parallel models* (i.e., a single copy that keeps track of multiple states). The Parallel Model allows exploration and manipulation of different alternatives of the past or future state of an application, while keeping the main system online. The idea is similar to version control systems, and involves *branches* and *merges*. Essentially, the Parallel Model enhances the ability of re-executing events even, by letting the user experiment with multiple alternative realities. While the current reality might include some erroneous event, recovery may come from a "correct branch", where we applied some corrective action to compensate a wrong event or skipped the faulty event altogether. In the end, the system should converge to the "corrected reality".

2.2. The relation between domain-driven design, command and query responsibility segregation, event sourcing and microservices

Even though DDD and CQRS are not necessarily related, the notion that aggregate roots implementing a domain separate queries and commands naturally fits CQRS. Indeed, they do not expose internal state to other aggregates, providing access only through public APIs. Since ES separates the reading from the writing sides, it naturally implements the CQRS pattern. Nonetheless, the reverse is not necessarily true, as one might implement the write side of the latter resorting to a database, *e.g.*, via object-relational mapping.

ES goes beyond the simple separation of the read and write sides, as it outputs and keeps track of the history of business state changes. ES thus enforces a physical separation between the write and the read sides. Events on the log serve to synchronize the

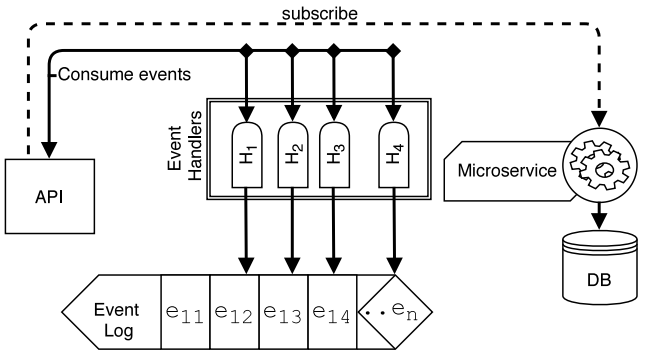


Fig. 3. Microservices plus ES setup.

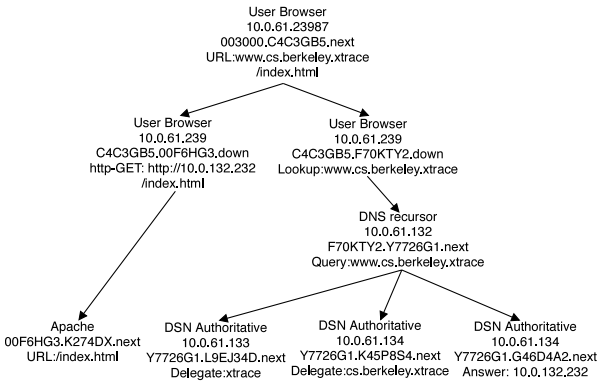


Fig. 4. Event DAG trace. *Source:* Adapted from Fonseca et al. (2007) X-Trace tool.

aggregates with their read side (see Fig. 1). While this follows the CQRS pattern, such separation is not truly mandatory in the latter. While the read side in ES can keep an updated snapshot of data, it is also able to completely rebuild such state from the event log, if it gets out of synchronization.

While a bounded context might be implemented as a monolith, a pattern that seems to be gaining momentum is that of microservices (Newman, 2015). These might be used to implement the DDD pattern, with the advantage they bring in terms of management, operation, resilience, and so on. Yet, dividing the monolith does not necessarily amounts to doing DDD. This will only happen if the division in microservices occurs according to non-conflicting models of bounded contexts. Microservices can also register for one or more events in the event log as in Fig. 3. Patterns for microservices, ES, CQRS and DDD were also studied in Debski et al. (2017), Nadareishvili et al. (2016), Erb et al. (2018, 2017a) and Molina et al. (2018).

2.3. End-to-end tracing

According to Xiang et al. (2016), "end-to-end tracing captures the workflow of causally-related activity (*e.g.*, work done to process a request) within and among the components of a distributed system". Tracing is a way to describe non-sequential behavior of a concurrent system via sequential observations and aims at improving system's observability. The concept was originally developed for individual programs, with the goal of observing the state changes during program execution. End-to-end tracing requires source code instrumentation, to let components pass metadata ("baggage" (Sigelman et al., 2010; Sun, 2016)) as they communicate. Given enough instrumentation, tracing can thus track information from the execution flow, data flows, state variations, or a mix of these and other system properties. This passing metadata connects interdependent points of the system, thus going beyond watching seemingly independent variables, as in monitoring. Its structure also makes tracing more powerful than logging, because one may use automated processing tools, instead of manual analysis.

Mazurkiewicz tries to formalize tracing, to define a mathematical framework describing the behavior of a concurrent system, using the well developed tools of formal language theory (Mazurkiewicz, 1977, 1987). As the definition of Mazurkiewicz makes explicit, this approach becomes even more useful for concurrent programs, giving operators the ability to observe state changes across different execution threads and processes. Compared to linear programs, concurrent ones are far harder to reason about, making tracing a very useful tool for debugging and monitoring. Once relationships are captured, there are multiple models that can be used to represent traces, or more specifically the relationships between their components, *e.g.*, a DAG.

Tools like *X-Trace* (Fonseca et al., 2007) represent traces as DAGs of events, as exemplified in Fig. 4. This is a simplistic example of a Web request from a user's browser, where the host first looks for the *hostname* in a *Domain Name System (DNS) lookup*; if it does not have the address cached, it will do another *DNS lookup* and send the request to the IP address it gets in response. Events represent a single point in time of the computation and are connected by directional edges that represent a *happens before* relationship. Each event can connect to and be connected from multiple others, signifying that an event may cause and be caused by multiple others. Each event may contain metrics, like execution time or the value of some variable, as well as log information relative to itself, for example, accessed Uniform Resource Locator (URL) in the figure. The direct application of this expressive capability is representing map join events where a join event is caused by multiple mapped instances allowing the recording of events during execution and recording the causal relationships between these events.

The format of metadata can change according to the different tracing systems. Xiang et al. (2016) point to three different metadata types: static, fixed-width metadata; dynamic, fixed-width metadata; and dynamic, variable-width metadata. In the simplest approach, static, fixed-width metadata, the tracing system will have several messages with the *same* 64-bit identifier. However, this fails to provide an ordering for the messages, thus making it more difficult to reconstruct the inner operation of the system. The dynamic, fixed-width metadata correlates tracing messages via some scalar logical clock. Each span carries the parent's identifier, thus allowing tools to recover causality from the beginning to the end of a single request, but not making it easy to tell the order between different requests. The dynamic, variable-width is the most powerful metadata type. It uses vector clocks (Xiang et al., 2016) or some variant, like interval-tree clocks (Almeida et al., 2008), to order any pair of events in the system.

One concern with tracing, and other observation methods, is the logging and possible exposure of privileged information to operating personnel or internal users. While the logging of request payload enriches data, it needs to be weighed against security concerns. On the other hand, tracing can also be used to ensure that security practices are being followed and to detect intrusions (Sigelman et al., 2010).

3. Related work

To contextualize our proposal, we look at existing approaches and how they compare. In particular, we review work related

to system morphology extraction and root cause analysis. While the idea of fully recovering causality in a system is not new, the novelty of our approach lies in stitching together tracing and ES. By connecting state changes to runtime metadata, we generate valuable information for root-cause analysis, anomaly detection, profiling and other related goals.

3.1. Process mining

Process mining shares some of our goals, focusing on the extraction of business processes related knowledge from system observations, such as logs (Van Der Aalst et al., 2011). With the same goal, de Murillas et al. propose a schema-based tracing extraction technique leveraging the Relational Database Management System (RDBMS) data model relationships and redo log. The goal is extracting the event log to be further analyzed using process mining techniques (de Murillas et al., 2016). Still on the topic of process mining, Leemans and van der Aalst use Aspect-Oriented Programming (AOP) to log network connections, effectively collecting traces between events that can later be used to extract processes (Leemans and van der Aalst, 2015). The authors position their contribution in between reverse engineering and process mining. Van der Aalst et al. (2004) propose α - algorithm to discover workflow models from event logs without discrepancies between real workflow processes and perceived processes and represent them in the form of a Petri net. Genetic Miner (de Medeiros et al., 2007) makes use of genetic process mining algorithm for global search through causal matrices that are closely related to Petri nets. Just like Genetic Miner, AGNEs is a process discovery algorithm. Its purpose is to handle problems with incomplete event logs using event logs. So, get insights into the relationships of the dependencies (Goedertier et al., 2009).

However, there are a number of challenges in the adoption of process mining related to causal analyzes between events (Wil van der Aalst, 2011). Wil van der Aalst, Gonzalez Lopez de Murillas et al. presents a detailed list of some related challenges. In Sections 4 and 5, we discuss how our approach deals with causal analysis between events. By aligning an event log and a process model, it is possible to perform advanced analysis (Adriansyah, 2014). Leemans and van der Aalst states that the starting point for any process mining technique is an event log (Leemans and van der Aalst, 2017). This is because process mining techniques use event logs to discover, monitor, improve processes, investigate how processes are performed and how they can be improved (Van Der Aalst et al., 2011). Other process extraction techniques through causal analysis have been explored in the literature (Eiben et al., 2003; Gao et al., 2009).

In contrast, our approach does not use inference to extract processes knowledge. By using tracing, further connecting it to the application state given by ES, we can deterministically reconstruct the effective workflows. While workflows do not necessarily directly map to business processes, they are at the limit of what can be achieved automatically, without recourse to expert knowledge.

3.2. Event-based state change tracing

State auditing of systems is important to reason about their behavior, and to correct deviations. One obvious application is determining the root cause of a failure and correcting it. Industry practices and frameworks, like Lagom (2018), Eventuate Framework (2021) and Axon (2021) vividly show that ES is used in practice for this exact purpose. Furthermore academic work on ES can be found in Erb and Kargl (2014). The authors propose combining ES and CQRS with discrete event simulation, to keep a system and its simulation running in lockstep. The idea is using the event log to retrieve events, set up a certain state of the simulation and continue execution from this point on. Different branches may therefore exist in parallel, to run future simulations in background, while the main system continues to run. For example, the developer can run a simulation, capture a snapshot and then add events to a new simulation without having to re-run the simulation from scratch.

Similarly to our approach, Erb et al. (2016) recognized the need to capture internal interactions, to reconstruct the state of actors in a distributed system. They used a variation of the "Dependency Vector" of Fowler and Zwaenepoel (1990), such that each actor keeps and logs the last logical clock it saw from every other actor upon receiving some message from a peer. The event log also persists incoming clock values. This makes it possible to exactly order events in the system, regardless of their presence in the log. The downside of this approach is that it is somewhat *ad hoc*, thus lacking any tool support. In Erb et al. (2017b), Erb et al. present a practical performance analysis of their previous proposal (Erb et al., 2016).

As a result of using ES, our approach does not require any form of event log extraction. By using tracing to correlate events to each other, and to services and their runtime metadata – both producers and consumers – our technique provides a more complete observation of the system. Furthermore, we propose the use of a standardized technological stack, making it possible to leverage existing, open-source, frameworks and libraries both for ES and tracing.

As future work, since our proposal improves both observability (through tracing) and controllability (through event sourcing), it can be used as a building block for autonomic systems (Nedelkoski et al., 2019).

4. Problem statement

As we mentioned in Section 2, strategies like retroactive events help dealing with incorrect states of the system. However, it is not trivial to determine which events need compensation and the breadth of events they affect. External factors, such as software updates and interaction with external systems, result in state and interaction metadata that is not captured in the event log. Additionally, some internal interactions do not immediately materialize as state change: when a sell command arrives at the aggregate (an object or set of related objects), one object might be responsible for invoking other aggregate objects to determine whether the item is available, dispatching the item, and produce the sale event. If something goes wrong in this process, developers will most likely be unable to identify if a software fault caused it and where and also which services (which consumed the event) were affected by the fault. In short, debugging may be impossible.

Let us consider a practical example to explore the limitations of event sourcing in its current form. A customer orders a watch, but receives and is charged for two. Returning and refunding one would solve the issue, but identifying the fault and its location in the system is more complicated. This could be a problem with a duplicate event, a service, an aggregate, an external system, a human factor or any combination. The event log lacks the necessary information to trace the cause and relate the involved parts (events, people, deployments, infrastructure, etc.). Furthermore, even after detecting the fault, we still lack any means to identify the extent of its effects. ES falls short of keeping the entire system history, because it provides no means of observing the meta aspects of its operation. Once any of its assumptions fails, it can no longer be relied on for debugging.

We now formalize this problem. An event in a system is an occurrence of interest for some external observer (Vernon, 2013; Jerry, 2005). Such an event could be, for example, an invocation

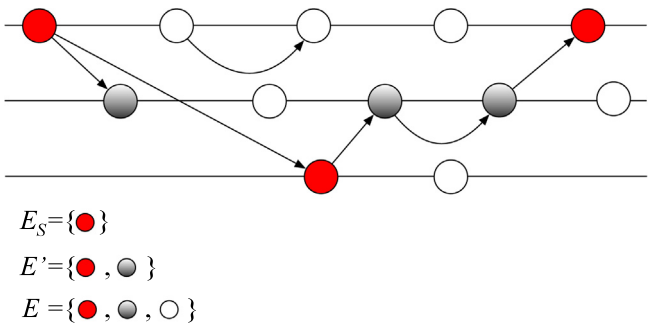


Fig. 5. Relation between the event graphs. E_S is the set of events in the log. E' is the set of traced events. E is the set of all system events.

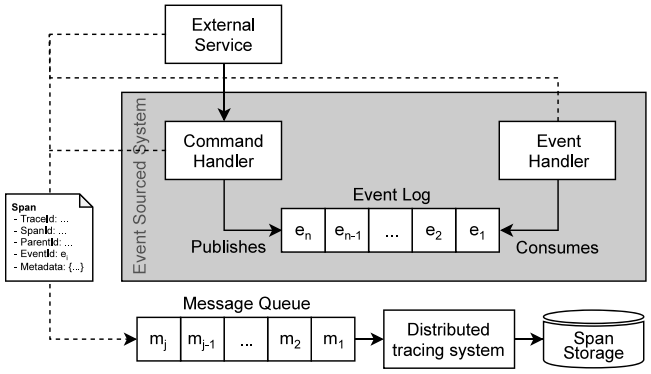


Fig. 6. A prototype architecture.

of an external service, changing the value of an internal variable, running a fork-join in a multi-threaded application, or having a sale order in the log resulting in two business domain events, to package the item and place an order to replenish the stock. We may represent a sequence of events like these as:

$$E = \langle e_1, \ldots, e_i, e_{i+1}, \ldots, e_n \rangle \quad (1)$$

The event log E_S is a subset of E, such that $e_i \in E_S \Leftrightarrow e_i \in E \wedge \pi(e_i) \in S$, where S is the set of event types that go into the event log, as defined by the architect, while $\pi(e)$ is the type of event e. Events in E_S have logical scalar timestamps defined by the event log position. Hence, we can order the events, as $e_i \rightarrow e_{i+1}$, $\forall i = 1, \ldots, |E| - 1$, according to Lamport's happens-before relation (Lamport, 1978), $\rightarrow$. If $e_i \rightarrow e_{i+1}$, e_i might have causally affected e_{i+1}.

The happens-before relationship between consecutive events in the log, defines a set of arcs T_S. In a properly designed system with event sourcing, the graph $G_S(E_S, T_S)$ should be enough to reconstruct the business model state. However, for debugging, the system may not go through the same set of internal states in different executions; more information is needed.

A natural extension for G_S is $G(E, T)$, which is the complete set of internal system events and their happens-before relation. T can be defined as an extension of T_S for E. To recover T we need more than simple logical clocks, for example vector clocks (Mattern, 1988; Fidge, 1987); $(e_i, e_j) \in T \Leftrightarrow e_i \rightarrow e_j \wedge \nexists e_k | e_i \rightarrow e_k \wedge e_k \rightarrow e_j$. E and T enable reconstruction of any state and respective transitions, thus allowing for debugging and identification of the root cause of any failure (*e.g.*, bugs in the code, configuration errors, reserving the same seat twice, writing a wrong item to package in the event log, or missing a write altogether).

A practical constraint is the impossibility of keeping E and T in the system, given their volume and collection overhead. In practice, subsets $E' \subset E$ and $T' \subset T$, must be used, such that $E_S \subset E' \subset E$ and $T_S \subset T' \subset T$. There is a trade-off to consider: larger E' and T' will enable location of more faults at the expense of processing and storage overhead. We illustrate the relation between events in Fig. 5, where the horizontal lines represent a running process, the circles represent events and the arcs represent happens-before relationships. We consider events in E_S, in red, events in E', in red and gray; E is all the previous and the white ones.

5. Approach

5.1. Formalizations

We propose to further explore the idea of Erb et al. (2016), by resorting to distributed tracing, instead of keeping vector clocks on the processes. Our argument is that, by definition, tracing is storing a relevant portion of all events and happens-before relationships, i.e. $G' = (E', T')$. Conversely, the need to have G' is fulfilled by tracing, which often is already in place in large distributed systems (Netflix, 2019; zipkin.io, 2021). Combining tracing with event sourcing is a very natural step, as both keep structured logs of the system state. Operators can thus build on the availability of tools, well-known practices and copious amount of information to locate faults. Tracing provides the ability to track all individual requests between and within different services and their relation to events in the log. Furthermore, developers can adjust the level of logging to cover as much as they want from graph G. This may involve capturing a smaller subset of E, E'. This is possible since most instrumentation libraries are open source, and configurable. Even closed-source tools may support open tracing specifications, such as OpenTelemetry (2021), leaving the possibility of implementing a custom tracing library for complete control. Keeping T is also intractable; among other things it requires dynamic variable-width metadata in tracing. In practice, dynamic fixed-width is used, restricting the preserved happens-before relationships to some subset $T' \subset T$.

We must set a connection between tracing and event logs. In particular, we must bind together event publication and subscription, together with the rest of the system. Causality metadata must thus pass from the publisher to the subscriber of the event; otherwise the tracing tools will be unable to correlate the publication of the event with the subscription. We suggest two options to pass the metadata from the publisher of the event to the subscriber. Either we write the metadata together with the event source information directly in the log; or we use the distributed tracing system, *e.g.*, zipkin.io (2021) or Jaeger (2021), to keep this information. In the former case, the event log grows larger and gets mixed with debug data; in the latter case, the subscriber must perform an additional read from the distributed tracing system before recovering the metadata, thus incurring additional latency.

Refer to Fig. 6, which illustrates the proposed architecture. In the ES system, events e_i that change the state are published to the Event Log, from where they are consumed by Event Handlers. Notice that typically the user facing front-end, as well as third party components are outside the ES scope, and therefore, their state is not contained in the Event Log. We propose leveraging distributed tracing across the entire system, to preserve metadata about executing as well as maintaining causality relationships between the events and associated execution. Each component, even the external ones, publishes *Spans* m_j to a given tracing back-end. Span publication, representing events, is asynchronous to ensure that tracing has little more overhead than logging, especially considering that data has no real-time or order delivery

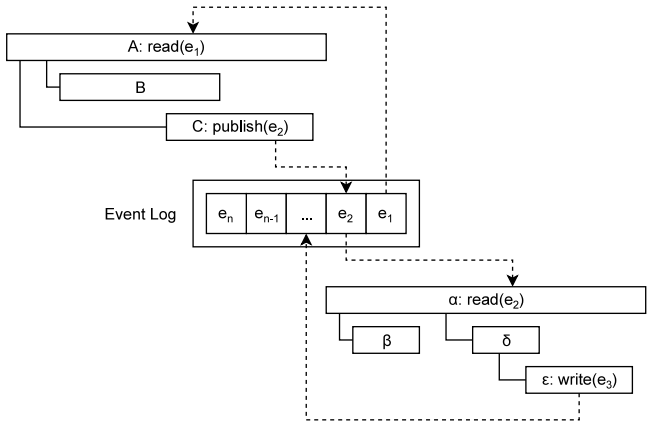


Fig. 7. Trace relationship to events.

constraints (Sambasivan, 2013). Each of these spans contains an identifier for the request (*Trace Identifier*), a unique *Span Identifier*, a reference to the span that originated it (*Parent Identifier*) as well as any arbitrary metadata the developer may wish to log about the execution. Furthermore, each span should contain the identifiers of any event *e* read or published during its execution.

Fig. 7 exemplifies how this mechanism can be leveraged to ask questions about the relationships between different events. This ensures that it becomes possible to recover the set of traces, and therefore logs and other metadata of the entire workflow that originated a specific event. The reverse is also true, given an observation of interest, such as an error, it becomes possible to determine all events involved it its origin as well as all that were generated as its consequence. For debugging purposes, this operation if much cheaper than having to deploy an alternative replica or rewinding the event log.

6. Evaluation

To validate our proposal we consider an Event-Sourced (ESd) online Guitar Store – shown in Fig. 2 – to demonstrate the advantages of tracing for observing the state of, and reasoning about, ESd systems.

The sample system is composed by three aggregates: payment, warehouse and supply management; as well as an orchestrator in charge of coordinating the aggregates. Aggregates produce the following log events:

- `GuitarReserved`: upon receiving the instruction to purchase a guitar, the Warehouse aggregate writes the reservation event to the log.
- `PaymentReceived`: upon successfully charging a user, the Payment aggregate writes this event to the log.
- `GuitarSent`: the Warehouse aggregate writes this event after sending a guitar.
- `StockOrdered`: the Supplier aggregate writes this event after ordering stock.

We focus on a single workflow, the acquisition of a guitar, depicted in Fig. 2. The orchestrator is responsible for managing the purchase process, by triggering commands to the appropriate aggregates. Once the orchestrator receives a purchase command from a user it tells the Warehouse aggregate to reserve the guitar. Once the Warehouse aggregate does this reservation, the next step is for the Payment aggregate to charge the client, followed by the actual shipping of the item, again in the Warehouse. Finally, if necessary, the orchestrator tells the Supplier aggregate to restock. Under correct operation, and assuming the case where restock is necessary, this will produce the following events:

```
Event Log:
[
 [GuitarReserved, 1, [orderID], baggage],
 [PaymentReceived, 2, [orderID], baggage],
 [GuitarSent, 3, [orderID], baggage],
 [StockOrdered, 4, [quantity], baggage]
]
```

Each event includes its type, identifier, data, as well as baggage for tracing (`baggage`). As we discuss in Section 2.3, this metadata can assume multiple forms, like static or dynamic, fixed or variable width. Industry standard implementations usually resort to dynamic fixed-width metadata, thus being limited to intra-request causality (Xiang et al., 2016).

In the following subsection, we reason about how tracing would be helpful in four common scenarios, such as debugging runtime issues or process mining.

6.1. Scenarios

An ESd system can still experience failures that are not trivial to solve, even with compensatory actions and replays. Beforehand, the root cause must be identified by inspecting the event log and available monitoring data, typically, logs and monitoring.

We categorize anomaly causes as Static and Dynamic. Static causes are related to system design mistakes, such as incorrect process implementation, *e.g.*, charging for an item before making sure it is available. Dynamic causes are related to runtime failures, possibly intermittent, such as software or hardware failures *e.g.*, a software bug may cause an event not to be published. While ES creates the mechanisms to correct state errors, tracing opens the opportunity to use richer, causality-preserving information for dynamic analysis. Events can be causally related, and grouped by process and even individual request. Furthermore, this information can be linked to runtime metadata, such as logs. As an example, one could detect anomalies by comparing event sequences against the known baseline for their respective workflows.

Consider the following scenarios:

6.1.1. Missing event

As a result of an intermittent software failure, events may be missing from some requests. In this scenario, the Warehouse aggregate failed to emit the `GuitarSent`, represented in the event sequence below.

```
Event Log:
[
 [GuitarReserved, 1, [orderID], baggage],
 [PaymentReceived, 2, [orderID], baggage],
 [StockOrdered, 3, [quantity], baggage]
]
```

This missing event results from an exception in the Warehouse aggregate. Normally this is logged, but if tracing is available, the log is correlated to the specific faulty request.

6.1.2. Duplicate event

Some aggregate could write an event to the log and crash. Due to faulty logic, it may execute again and duplicate the event. In our sample system, an example would be a double shipping order.

```
Event Log:
[
 [GuitarReserved, 1, [orderID], baggage],
 [PaymentReceived, 2, [orderID], baggage],
 [GuitarSent, 3, [orderID], baggage],
 [GuitarSent, 4, [orderID], baggage],
 [StockOrdered, 5, [quantity], baggage]
]
```

6.1.3. Event with incorrect data

An event is faulty, as a consequence of representing wrong state or being erroneously created. As exemplified in the event log below, if the Payment aggregate emits a `PaymentReceived` event with the wrong order identifier, the customer will not get his or her order, and some other unpaid order may be shipped instead.

```
Event Log:
[
 [GuitarReserved, 1, [orderID], baggage],
 [PaymentReceived, 2, [wrongOrderID], baggage],
 [GuitarSent, 3, [wrongOrderID], baggage],
 [StockOrdered, 4, [quantity], baggage]
]
```

In this situation, we want to find the event publisher and determine if there were faulty conditions associated with it.

6.1.4. Dead code pruning

To eliminate unused code, services, or events, one may wish to extract the dependencies between the publishers, subscribers and events. In other words, the ability of tracing to perform dynamic architectural/morphological analysis is also useful in ESd systems.

6.2. Comparison

For the missing, duplicate, and incorrect event scenarios from Sections 6.1.1, 6.1.2, and 6.1.3, the comparison is similar. We start by pointing out that, without tracing, the way to find the issue involves waiting for an end user report, searching through the event log to find events containing the correct order identifier, and then filtering logs by service and time-stamp, to find the runtime metadata of that particular request. I.e., operators would need to manually connect the events in the log, associate events with runtime exceptions, and detect missing or duplicate events by themselves, using time consuming *ad hoc* approaches.

If the system is using instrumentation for the sake of tracing, dynamic analysis can detect an anomaly for that sequence of events (most executions of that workflow have exactly one `GuitarSent` event). Additionally, the tracing baggage present in the events allows for an immediate collection of all related events, services, and logs, thus providing a much better view of the system to the operator and enabling a faster identification and correction of the anomaly. This is much easier than using logs to find the culprit, as traces preserve causal relationships.

For the last scenario, in 6.1.4, tracing is even more important. Without the causality preserving properties of tracing stitching events together with publishers and subscribers, this would have to be solved organizationally. In other words, it would require inquiring other developers, teams and even organizations, or producing and maintaining up to date documentation.

7. Discussion

Distributed tracing is highly complementary to ES as it adds a new dimension, by connecting it to execution metadata. To explore the proposed approach and its implications, this sections discusses the current limitations of ES and how tracing can mitigate them. In particular, we discuss a few example use cases and how they are affected.

7.1. Event sourcing limitations

Event Sourcing is unable to keep an explicit relation between events, handlers and services. For example, if a handler consumes some event e_i, is this event the result of a previous event e_j, or are they concurrent? Knowing this may help to track the root and propagation of an incorrect event.

A second limitation of ES is that it cannot encompass the entire system. In theory, any single component should keep all its state as a set of events in the log. However, at some point, the ES system must interact with the real world or third party systems. This requires wrapping external calls, to ensure that they are controlled by a gateway that can properly handle replaying and consistently repeat responses whenever the external system is not idempotent. This results in reduced observability without the ES advantages for the external systems. ES systems are thus hard to develop and maintain and their scope is normally restricted to the business and data model.

The main selling point of ES systems is the ability to replay the event log to recover any previous state up to a given point, as well as enabling Retroactive Events (Fowler, 2005c) and Parallel Models (Fowler, 2005b‘). However, for debugging, these mechanisms are overwhelmingly expensive. Even with snapshots, rebuilding the state on a high throughput production system is complicated. Furthermore, if multiple teams wish to debug at once, they will require multiple parallel deployments.

7.2. Advantages of tracing

Tracing is greatly complementary to ES and can help overcome its limitations. It usually enables recovery of the causal flow of execution of a particular request, as well as arbitrary metadata relative to the system internal state, like variables and timestamps. Tracing can integrate into the same causal timeline all data from other observability tools, such as logging and monitoring; for example, monitoring can be integrated *a posteriori* based on real-time clock or, in alternative, one can record a sample at execution time on the span. The same is valid for logs: they can be later retrieved based on the span identifier (which must be included with the log message), or immediately added to the spans.

ES requires the system to strictly follow the Event Log format and its architectural pattern, such as logging all state changing events. Tracing requires no particular design, and its data types are standardized, being relatively easy to use on the whole system. Even though traces are not as general as vector clocks (causality can only be ascertained inside a trace and not inter-trace), they preserve the most relevant relationships for debugging purposes. This addition makes it possible to recursively explore the relationships between Events in the Log, as well as the metadata of their execution (both by publishers and all subscribers).

Due to its low cost of adoption and industry acceptance, one can reasonably expect external systems to either be instrumented or possible to instrument, thus shedding light into previously unobserved sections of the system. Tools and standards for tracing, like OpenTracing (2021), OpenCensus (2021) or the more recent OpenTelemetry (2021) have been successfully deployed to mitigate the complexities of large-scale microservice architectures. One can observe this relation in Fig. 6: tracing information can stitch together interactions with external services and log events back into a global consistent view that breaks the boundaries of a highly partitioned system.

Even within the boundaries of the ES system, tracing can enrich observability, by restoring the causality between events in the log, the services that generate and especially services

that process them, – something that was not available before – as well as the metadata resulting from that execution. Tracing enables operators to determine the log event dependency graph, i.e., which events generate which ones. This information may also serve to determine which services are responsible for producing or consuming any type or any instance of an event. The consequences of faulty code deployments can be isolated to the events they produce and, with the help of tracing, to the events that derive from them. Tracing also enables identification of unused services and events, as these do not trigger any state changes. Another interesting feature of mixing ES and tracing is the ability to identify non-deterministic behavior during replays. ES assumes that state is a pure function of the log; if this is not the case, the internal component state may differ on future replays of the log, thus generating different spans/traces. This can serve as a cheap way of checking for determinism and detecting problems sooner.

We can explore the deterministic nature of ES systems in function of the event log, to check new software versions with production data. As it gets up to the present state, a new version will consume all the previous log events. Tracing can help to check whether the new version goes through the same internal states and outputs the same derivative events.

We believe that additional metadata that helps developers track code versions can lay the foundation for powerful features. Knowing the version of deployed artifacts and knowing their timeline in the system is necessary, as developers can only determine the internal state of the system if they can pinpoint where the system was changed or scaled. For example, a direct comparison of traces is only possible if they were generated by the same or compatible versions of the artifact, as instrumentation may change according to business needs. Another case can happen if the data schema changes with a new version. Should this happen, we must know the correspondence between events and the software version that generated them.

Consequently by preserving causality relationships, tracing considerably simplifies complex procedures. Thus, it strongly reduces the need for extremely expensive rewinds and replays of the event log that aim at recovering the same metadata that tracing already stores.

8. Conclusion

In this paper we argue that an event log does not have enough information for a complete audit of a distributed system. As far as we know, we are the first to suggest adding end-to-end tracing to event sourcing, as a way to mitigate the traceability problems felt by practitioners.

While ES offers many advantages in terms of replayability, isolation and data schema flexibility, for the purposes of debugging, it still provides an incomplete view of the system. Normally, debugging requires instrumenting the relevant part and going through the expensive process of replaying the events and collecting the metadata. Instead, we provide better system observability, thus reducing the need for replaying the event log.

Our proposal solves many practical issues that arise during operation and debugging of ES systems. We can easily answer questions, such as: (1) Which services consumed a given event? (2) Which service, in which version, produced a given event? (3) Which events are no longer used? (4) What is the service dependency graph?

A great advantage of adding tracing to ES is that it is widely accepted by the industry, thus adding little cost to the ES system. It is often already in place and the tools and standards are already there. Furthermore, since ES imposes a particularly coherent design style made of small components communicating through the event log, it is relatively inexpensive to add or interpose instrumentation points between communication. This opens up a new research direction: creating frameworks to systematize and simplify the development, validation, debugging and deployment of systems integrating ES and tracing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work was supported by research grants of the programs: Science Without Borders (Ciências sem Fronteiras - CsF), Brazil, Brazilian Space Agency (Agência Espacial Brasileira - AEB), Brazil and by Portuguese funds through the Foundation for Science and Technology, I.P., within the scope of the project CISUC – UID/CEC/00326/2020 and by the European Social Fund, through the Regional Operational Program Centro 2020.

References

Adriansyah, Arya, 2014. Aligning observed and modeled behavior (Ph.D. thesis). Technische Universiteit Eindhoven.

Almeida, Paulo Sergio, Baquero, Carlos, Fonte, Victor, 2008. Interval tree clocks. In: International Conference on Principles of Distributed Systems. Springer, pp. 259–274.

Axon, 2021. URL: https://axoniq.io/.

Clayman, Stuart, Galis, Alex, Chapman, Clovis, Toffetti, Giovanni, Rodero-Merino, Luis, Vaquero, Luis Miguel, Nagin, Kenneth, Rochwerger, Benny, 2010. Monitoring service clouds in the future internet. In: Future Internet Assembly. Valencia, Spain, pp. 115–126.

de Medeiros, Ana Karla A., Weijters, Anton J.M.M., van der Aalst, Wil M.P., 2007. Genetic process mining: an experimental evaluation. Data Min. Knowl. Discov. 14 (2), 245–304.

de Murillas, Eduardo González López, van der Aalst, Wil M.P., Reijers, Hajo A., 2016. Process mining on databases: Unearthing historical data from redo logs. In: International Conference on Business Process Management. Springer, pp. 367–385.

Debski, Andrzej, Szczepanik, Bartlomiej, Malawski, Maciej, Spahr, Stefan, Muthig, Dirk, 2017. In search for a scalable & reactive architecture of a cloud application: CQRS and event sourcing case study. IEEE Softw. (99).

Eiben, Agoston E., Smith, James E., et al., 2003. Introduction To Evolutionary Computing. Vol. 53, Springer.

Erb, Benjamin, Habiger, Gerhard, Hauck, Franz J., 2016. On the potential of event sourcing for retroactive actor-based programming. In: First Workshop on Programming Models and Languages for Distributed Computing. ACM, p. 4.

Erb, Benjamin, Kargl, Frank, 2014. Combining discrete event simulations and event sourcing. In: Proceedings of the 7th International ICST Conference on Simulation Tools and Techniques. ICST, pp. 51–55.

Erb, Benjamin, Meißner, Dominik, Habiger, Gerhard, Pietron, Jakob, Kargl, Frank, 2017a. Consistent retrospective snapshots in distributed event-sourced systems. In: 2017 International Conference on Networked Systems (NetSys). IEEE, pp. 1–8.

Erb, Benjamin, Meissner, Dominik, Habiger, Gerhard, Pietron, Jakob, Kargl, Frank, 2017b. Consistent retrospective snapshots in distributed event-sourced systems. In: Networked Systems (NetSys), 2017 International Conference on. IEEE, pp. 1–8.

Erb, Benjamin, Meißner, Dominik, Ogger, Ferdinand, Kargl, Frank, 2018. Log pruning in distributed event-sourced systems. In: Proceedings of the 12th ACM International Conference on Distributed and Event-Based Systems. ACM, pp. 230–233.

Eugster, Patrick Th, Felber, Pascal A, Guerraoui, Rachid, Kermarrec, Anne-Marie, 2003. The many faces of publish/subscribe. ACM Comput. Surv. 35 (2), 114–131.

Evans, Eric, 2004. Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley Professional.

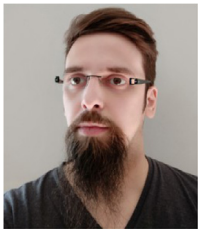
Eventuate Framework, 2021. URL: https://eventuate.io/.

Fidge, Colin J., 1987. Timestamps in Message-Passing Systems that Preserve the Partial Ordering. Australian National University. Department of Computer Science.

Fonseca, Rodrigo, Porter, George, Katz, Randy H., Shenker, Scott, Stoica, Ion, 2007. X-trace: A pervasive network tracing framework. In: Proceedings of the 4th USENIX Conference on Networked Systems Design & Implementation (NSDI'07). (April), USENIX Association, p. 20.

Fowler, Martin, 2005a. Event sourcing. https://martinfowler.com/eaaDev/EventSourcing.html.
Fowler, Martin, 2005b'. Parallel model. https://martinfowler.com/eaaDev/ParallelModel.html.
Fowler, Martin, 2005c. Retroactive event. https://www.martinfowler.com/eaaDev/RetroactiveEvent.html.
Fowler, J., Zwaenepoel, W., 1990. Causal distributed breakpoints. In: Proceedings., 10th International Conference on Distributed Computing Systems. pp. 134–141. http://dx.doi.org/10.1109/ICDCS.1990.89277.
Gao, Zhi-peng, Jian, Chen, Qiu, Xue-song, Meng, Luo-ming, 2009. Qoe/qos driven simulated annealing-based genetic algorithm for web services selection. J. China Univ. Posts Telecommun. 16, 102–107.
Garcia-Molina, Hector, Salem, Kenneth, 1987. Sagas. In: Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data. In: SIGMOD '87, Association for Computing Machinery, New York, NY, USA, pp. 249–259. http://dx.doi.org/10.1145/38713.38742, https://doi.org/10.1145/38713.38742.
Goedertier, Stijn, Martens, David, Vanthienen, Jan, Baesens, Bat, 2009. Robust process discovery with artificial negative events. J. Mach. Learn. Res. 10, 1305–1340.
Gonzalez Lopez de Murillas, E., 2019. Process mining on databases: extracting event data from real-life data sources (Ph.D. thesis). Technische Universiteit Eindhoven.
Gousios, Georgios, Loverdos, Christos KK, Louridas, Panos, Koziris, Nectarios, 2012. Aquarium: An extensible billing platform for cloud infrastructures. Article.
Hohpe, G., 2005. Your coffee shop doesn't use two-phase commit [asynchronous messaging architecture]. IEEE Softw. 22 (2), 64–66. http://dx.doi.org/10.1109/MS.2005.52.
Jaeger, 2021. URL: https://www.jaegertracing.io/.
Jerry, Banks, 2005. Discrete Event System Simulation. Pearson Education India.
Lagom, 2018. URL: https://www.lagomframework.com/.
Lamport, Leslie, 1978. Time, clocks, and the ordering of events in a distributed system. Commun. ACM 21 (7), 558–565.
Leemans, Maikel, van der Aalst, Wil M.P., 2015. Process mining in software systems: Discovering real-life business transactions and process models from distributed systems. In: 2015 ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS). IEEE, pp. 44–53.
Leemans, Maikel, van der Aalst, Wil M.P., 2017. Modeling and discovering cancelation behavior. In: OTM Confederated International Conferences on the Move To Meaningful Internet Systems. Springer, pp. 93–113.
Mattern, Friedemann, 1988. Virtual time and global states of distributed systems. In: Parallel and Distributed Algorithms. North-Holland, pp. 215–226.
Mazurkiewicz, Antoni, 1977. Concurrent program schemes and their interpretations. In: DAIMI Report Series. Vol. 6, http://dx.doi.org/10.7146/dpb.v6i78.7691, https://tidsskrift.dk/daimipb/article/view/7691.
Mazurkiewicz, Antoni, 1987. Trace theory. In: Brauer, W., Reisig, W., Rozenberg, G. (Eds.), Petri Nets: Applications and Relationships To Other Models of Concurrency: Advances in Petri Nets 1986, Part II Proceedings of an Advanced Course Bad Honnef, 8.–19. September 1986. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 278–324. http://dx.doi.org/10.1007/3-540-17906-2_30.
Meyer, Bertrand, 1988. Object-Oriented Software Construction. Vol. 2, Prentice Hall, New York.
Molina, Javier Moreno, García, Juan Ferrer, Jiménez, Carlos Kuchkovsky, 2018. Archer: An event-driven architecture for cyber-physical systems. In: 2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion). IEEE, pp. 335–340.
Nadareishvili, Irakli, Mitra, Ronnie, McLarty, Matt, Amundsen, Mike, 2016. Microservice Architecture: Aligning Principles, Practices, and Culture. O'Reilly Media, Inc..
Nedelkoski, Sasho, Cardoso, Jorge, Kao, Odej, 2019. Anomaly detection and classification using distributed tracing and deep learning. In: 2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID). IEEE, pp. 241–250.
Netflix, 2019. URL: https://medium.com/@NetflixTechBlog/inca-message-tracing-and-loss-detection-for-streaming-data-netflix-de4836fc38c9.
Newman, Sam, 2015. Building Microservices: Designing Fine-Grained Systems, first ed. O'Reilly Media, p. 280.
OpenCensus, 2021. URL: https://opencensus.io/.
OpenTelemetry, 2021. URL: https://opentelemetry.io/.
OpenTracing, 2021. URL: https://opentracing.io/docs/overview/tracers/.
Overeem, Michiel, Spoor, Marten, Jansen, Slinger, 2017. The dark side of event sourcing: Managing data conversion. In: Software Analysis, Evolution and Reengineering (SANER), 2017 IEEE 24th International Conference on. IEEE, pp. 193–204.
Pautasso, Cesare, Zimmermann, Olaf, Amundsen, Mike, Lewis, James, Josuttis, Nicolai M, 2017. Microservices in practice, part 1: Reality check and service design. IEEE Softw. 34 (1), 91–98.
Ritchie, Peter, 2016. Design and architecture: Patterns and practices. In: Practical Microsoft Visual Studio 2015. Springer, pp. 79–100.
Sambasivan, Raja R., 2013. Diagnosing performance changes in distributed systems by comparing request flows (Ph.D. thesis). Carnegie Mellon University.
Sigelman, Benjamin H, Barroso, Luiz Andre, Burrows, Mike, Stephenson, Pat, Plakal, Manoj, Beaver, Donald, Jaspan, Saul, Shanbhag, Chandan, 2010. Dapper, a large-scale distributed systems tracing infrastructure.
Sun, Hongkai, 2016. General baggage model for end-to-end tracing and its application on critical path analysis. In: M.Sc. Esis. Brown University.
Van Der Aalst, Wil, Adriansyah, Arya, De Medeiros, Ana Karla Alves, Arcieri, Franco, Baier, Thomas, Blickle, Tobias, Bose, Jagadeesh Chandra, Van Den Brand, Peter, Brandtjen, Ronald, Buijs, Joos, et al., 2011. Process mining manifesto. In: International Conference on Business Process Management. Springer, pp. 169–194.
Van der Aalst, Wil, Weijters, Ton, Maruster, Laura, 2004. Workflow mining: Discovering process models from event logs. IEEE Trans. Knowl. Data Eng. 16 (9), 1128–1142.
Vernon, Vaughn, 2013. Implementing Domain-Driven Design. Addison-Wesley.
Wil van der Aalst, M.P., 2011. Process Mining. Discovery, Conformance and Enhancement of Business Processes. Springer Heidelberg.
Xiang, Yong, Li, Hu, Wang, Sen, Xu, Wei, 2016. Debugging openstack problems using a state graph approach. In: Proceedings of the 4th USENIX Conference on Networked Systems Design & Implementation. In: APSys '16, Vol. 14, ACM, New York, NY, USA, pp. 49–64. http://dx.doi.org/10.1145/2967360.2967366, http://arxiv.org/abs/1606.05963.
zipkin.io, 2021. URL: https://zipkin.io/.

Stanley Lima is a Ph.D. student at the University of Coimbra, Portugal, as well as a Senior Cloud Solution Architect at Ânima Educação group. He is affiliated to the Centre for Informatics and Systems of the University of Coimbra, Portugal, and to the Service Prototyping Lab (SPLab) at the Zurich University of Applied Sciences (ZHAW), Institute of Applied Information Technology in Winterthur Switzerland. His main research interests are reliable software and distributed systems, namely observability and tracing, cloud environments as well as consistency versus availability and related open challenges in cloud computing.



Jaime Correia is a Ph.D. student at the Department of Informatics Engineering of the University of Coimbra and has previously received a B.Sc. in Informatics Engineering and an M.Sc. in Software Engineering from the same institution. His main research interests are distributed systems with focus on observability of distributed systems, particularly tracing, cloud environments as well as autonomic and elastic systems.

Filipe Araujo is a Tenured Assistant Professor at the University of Coimbra, Portugal. He received his Ph.D. in 2006 from the University of Lisbon, Portugal. His main research topic is observability of fine-grained distributed systems. His research interests include cloud computing, microservices, monitoring, security, and other distributed systems topics. He is the author of the blog "Enterprise Application Integration" (http://eai-course.blogspot.com), which has over 100,000 page views from all over the world.

Jorge Cardoso his currently Chief Architect for AIOps (artificial intelligence for IT operations) at Huawei Munich Research Center and Associate Professor at the University of Coimbra, Portugal. His current research involves the development of the next generation of AI-driven IT Operations tools and platforms. He has a Ph.D. from the University of Georgia, US and a M.Sc./B.Sc. in Informatics Engineering from the University of Coimbra.