JETIR.ORG ISSN: 2349-5162 | ESTD Year : 2014 | Monthly Issue
JOURNAL OF EMERGING TECHNOLOGIES AND INNOVATIVE RESEARCH (JETIR)
An International Scholarly Open Access, Peer-reviewed, Refereed Journal

Review on Event Sourcing Systems in Distributed Systems

[1] Abhinav Parhad, [2]Jharna Chopra
[1]Research Scholar, [2]Associate Professor

Abstract: The handling of data by computer systems is revolutionized with the event sourcing methods. It represents discrete occurrences or actions that have happened within the application. These events are appended to an event log or a journal, which serves as a historical record of all the events that have taken place. The current research reviews the potential drawbacks and complexities associated with event sourcing, including the management and storage of the event log, resource-intensive event replay, and design considerations for distributed systems.

***IndexTerms* –Event sourcing, CQRS**

I. INTRODUCTION

Event sourcing is an incredibly powerful software design pattern that revolutionizes the way applications handle data. It's all about capturing every single change or event that occurs within the application and storing them as a series of unalterable events. Unlike traditional approaches that only focus on the current state of an object, event sourcing goes above and beyond, preserving the entire history that led to the present. Software systems are increasing in complexity, used in increasingly critical processes, and serve increasing numbers of end-users. Architectural patterns enable engineers to build these systems using knowledge acquired by other engineers.

II. LITERATURE REVIEW

Kabbedijk et al. (2012) [1] describes event sourcing as a sub pattern of Command Query Responsibility Segregation (CQRS) in his work on the improved variability and scalability of systems applying CQRS.

Fowler (2017) [2] Event sourcing differs from traditional event-driven architectures (EDAs) in that it maintains a record of all state changes as an append-only sequence of events. Event sourcing diverges from event-driven methodologies such as stream processing, transactional processing, and blockchain in two notable manners. In the context of Event Sourced Systems (ESS), events are initially stored as the state of the application. In contrast to Enterprise Social Systems (ESSs), which prioritise communication as a secondary function, alternative systems employ events as a mode of communication. The second differentiation lies in the close association between events and those transpiring in authentic business operations.

Brandolini (2018) [3] has proposed event storming as a group design methodology. Similar to the concept of ideation, this approach centers on the occurrences that take place within a given software system.

Overeem et al., (2017) [4] identified a lack of literature on the development of ESSs during their research. The present research endeavors to address this gap in the literature by means of conducting a series of 25 interviews with engineers, with the aim of constructing a comprehensive depiction of event-sourced systems. Furthermore, we enumerate four categories of rationales for utilizing event sourcing in applications, one of which pertains to the mitigation of intricacy.

Adolph et al. (2011) [5] According to the author, Grounded Theory (GT) is a viable approach for conducting research in underexplored areas. A grounded theory elucidates the manner in which individuals employ a particular approach to address their primary issue. The pivotal element of the Grounded Theory methodology is the "central category," which is fundamental in defining and shaping the research process. The primary focus of this work pertains to the development and construction of event-driven systems by software engineers. The formal definition of event sourcing facilitates comprehension, validation, and pedagogy of the pattern and its ramifications among experts and learners. Section 2 of the manuscript discusses the three most salient components, and elucidates the utilisation of Grounded Theory (GT) to construct a conceptual framework for Energy Storage Systems (ESSs) predicated on 25 interviews. Based on the gathered data, it is possible to discern the prevailing pattern and its significance.

Anh et al. (2018) [6] address blockchain as a singular data format that only allows for appending. Event sharing and crowdsourcing employ similar data styles, albeit with distinct purposes. The blockchain technology endeavours to address issues pertaining to the sharing of information, consensus building, and establishing trust. In contrast, event sourcing endeavours to address issues

pertaining to historical data, temporal intricacies, and the tracking of changes over time. In contrast to event source, which operates autonomously, the blockchain methodology employs this approach as a means of addressing its challenges. A blockchain methodology can be employed to construct systems that rely on the occurrence of events. The operational mechanics of blockchain technology with respect to information dissemination and validation do not facilitate the attainment of the objectives of event sourcing.

The event source method needs immutability because it makes sure that every change in state is saved. Helland (2015) [7] says that the fact that data can't be changed is a key part of distributed systems. Some people think that immutability is what makes event sourcing unique, but it is not guaranteed like it is with block chains. In some of the systems being looked at, immutability is given up for a model creation process that is easier. Different levels of changelessness were found. Eight of the first-degree ESSs are unchangeable, which means they never change what happens. The second level of immutability, which lets cut-off events, is used by three of the nineteen systems. The event storage is changed at this point, but backups make sure that no data is lost. These backups will always be kept because they are needed to meet rules or service-level agreements. This amount of immutability still allows for an audit trail, since all changes to the state can be found in the records.

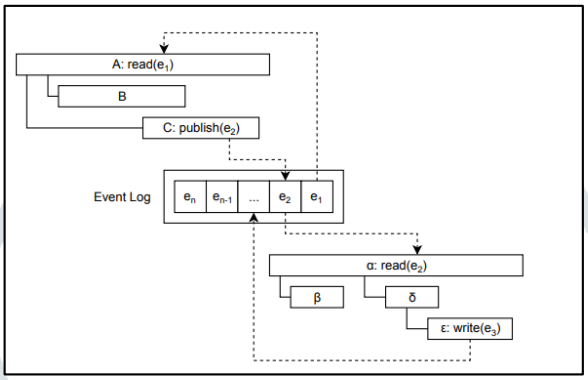

Figure 1: Trace relationship to events [7]

According to Martin et al. [8], one advantage of having the event log and the option to perform partial and branching replays is the ease of debugging. In this context, ES is frequently promoted as a solution to some distribution-related issues. This procedure leaves no system information, which is required to monitor errors and demonstrate the validity of the ES mechanisms themselves. Frequently, developers must use unstructured logging to identify and resolve these problems. It is essential to remember that branching or replaying the log incurs a considerable cost in terms of resources and source code complexity, as some of the events will need to be processed twice.

Christopher and others [9] ES is a system design approach that maintains a number of desirable functional and non-functional qualities, such as auditability, when it comes to managing state change in distributed systems. APIs read Query Model Command Model fetch instruction Changes Write-side Read-side A data storage enhancement (a) CQRS configuration made simpler. The acquire instruction is interpreted by APIs for the domain model. DB Event Subscribe Read Configuration for the Store Event (b) CQRS and Event Sourcing. Scalability, decoupling, and CQRS architecture with and without ES are shown in Fig. 2. ES integrates concepts from a variety of techniques and programming paradigms, such as CQRS and publish-subscribe systems.

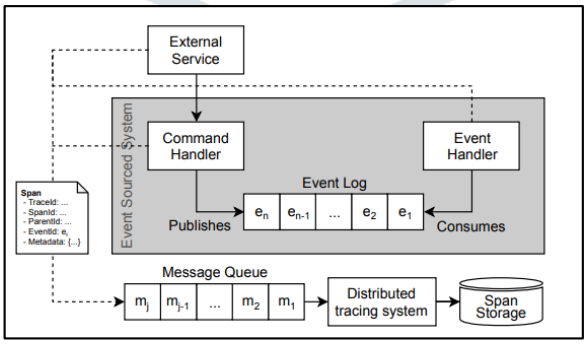

Figure 2: A prototype architecture [9]

Using ES, it is possible to audit the status, enabling corrective actions such as publishing an event to the journal in order to dispatch the lost parcel. Occasionally, it may be necessary to switch or rearrange the occurrences. For instance, an incorrect invoice may need to be replaced. It may be necessary to rerun the log after making the necessary modifications. In Retroactive Event [10] (which derives from Parallel Model [11]), Martin Fowler suggests an automated method for attaining this. A single copy that is aware of multiple embedded parallel models (i.e., a single copy that maintains account of multiple states) or allowing the system to execute copies of itself are the two most common methods for achieving this.

Xiang et al. [12], "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 aims to increase the observability of concurrent systems by using sequential observations to characterize their non-sequential behavior.

Tools like X-Trace [13] represent traces as DAGs of events, as exemplified in Figure 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.

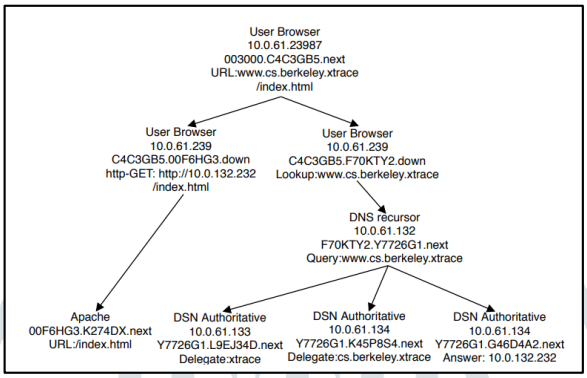

Figure 3: A prototype architecture [13]

Using the sophisticated techniques of formal language theory, Mazurkiewicz attempts to formalise tracing and develop a mathematical framework for describing concurrent system behaviour [14, 15]. Mazurkiewicz's definition makes plain that this method becomes significantly more useful for concurrent programmers by enabling operators to monitor state changes across multiple execution threads and processes. Because concurrent programmers are significantly more difficult to reason about than linear ones, tracing is an especially effective monitoring and debugging technique for them. Once connections are gathered, traces, or more precisely the interactions between their components, can be represented using a variety of models, including a Directed Acyclic Graph (DAG).

Various techniques employed for monitoring can alter the form of the data. In the event of utilizing static and fixed-width data, the tracking mechanism shall detect numerous messages bearing an identical 64-bit identifier. The lack of a specific order in the messages renders the comprehension of the system's internal workings more challenging. Messages that are traced are connected through a scalar logical clock and dynamic, fixed-width data. Each subsequent span is numbered with the identifier of the preceding span. This feature enables tools to determine the sequence of requests within a single request, yet poses a challenge in determining the sequence between requests. Metadata that possesses the capability of adjusting its width is considered the most effective form of dynamic material. By utilizing vector clocks [12] or a modified version such as interval-tree clocks [29], it is possible to establish a relationship between any two events within a system.

V. CONCLUSION

Event sourcing, distinguished from traditional event-driven architectures, maintains a record of all state changes as an append-only sequence of events. It is employed as a mode of communication in event-sourced systems (ESSs) and closely associated with authentic business operations. The use of event sourcing in applications offers several rationales, including the mitigation of intricacy. The research conducted in this field fills a gap in the literature by providing a comprehensive depiction of event-sourced systems. Grounded Theory (GT) methodology is employed to construct a conceptual framework for ESSs, facilitating comprehension and validation of the pattern.

REFERENCES

[1] Kabbedijk, J., Jansen, S., Brinkkemper, S., 2012. A case study of the variability consequences of the CQRS pattern in online business software. In: Proceedings of the 17th European Conference on Pattern Languages of Programs. ACM, p. 2. http://dx.doi.org/10.1145/0000000.0000000.

[2] Fowler, M., 2013. Schemaless data structures. http://martinfowler.com/articles/ schemaless/.

[3] Brandolini, A., 2018. Introducing Event Storming. Leanpub.

[4] Overeem, M., Spoor, M., Jansen, S., 2017. The dark side of event sourcing: Managing Data conversion. In: IEEE 24th International Conference on Software Analysis, Evolution and Reengineering, SANER, pp. 193–204.

[5] Adolph, S., Hall, W., Kruchten, P., 2011. Using grounded theory to study the experience of software development. Empir. Softw. Eng. 16 (4), 487–513. http://dx.doi.org/10.1007/s10664-010-9152-6.

[6] Anh, D.T.T., Zhang, M., Ooi, B.C., Chen, G., 2018. Untangling blockchain: A data processing view of blockchain systems. IEEE Trans. Knowl. Data Eng. 4347 (c), 1–20. http://dx.doi.org/10.1109/TKDE.2017.2781227, arXiv:1708.05665.

[7] Helland, P., 2015. Immutability changes everything. Commun. ACM 59 (1), 64–70. http://dx.doi.org/10.1145/2844112, arXiv:arXiv:1011.1669v3.

[8] Martin Fowler. Event Sourcing. 2005.

[9] Patrick Th Eugster et al. "The many faces of publish/subscribe". In: ACM computing surveys (CSUR) 35.2 (2003), pp. 114–131.

[10] Martin Fowler. Retroactive Event. 2005.

[11] Martin Fowler. Parallel Model. 2005.

[12] Yong Xiang et al. "Debugging OpenStack Problems Using a State Graph Approach". In: Proceedings of the 4th USENIX conference on Networked systems design & implementation. Vol. 14. APSys '16 5. ACM. New York, NY, USA: ACM, 2016, pp. 49–64. ISBN: 9781450342650. DOI: 10 . 1145 / 2967360. 2967366. arXiv: 1606. 05963. URL: http:// arxiv. org/ abs/ 1606. 05963.

[13] Antoni Mazurkiewicz. "Concurrent Program Schemes and their Interpretations". In: DAIMI Report Series 6.78 (July 1977). ISSN: 2245-9316. DOI: 10.7146/dpb.v6i78. 7691. URL: https://tidsskrift.dk/daimipb/article/view/ 7691

[14] Antoni Mazurkiewicz. "Trace theory". In: 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. Ed. by W Brauer, W Reisig, and G Rozenberg. Berlin, Heidelberg: Springer Berlin Heidelberg, 1987, pp. 278–324. ISBN: 978-3-540-47926-0. DOI: 10.1007/3-540-17906-2 30. URL: https://doi.org/ 10.1007/3-540-17906-2%7B%5C %7D30.

[15] Rodrigo Fonseca et al. "X-trace: A pervasive network tracing framework". In: Proceedings of the 4th USENIX conference on Networked systems design & implementation (NSDI'07). April. USENIX Association. 2007, p. 20. DOI: 10.1.1.108.2220. URL: http://portal.acm. org/citation.cfm?id=1973450.

[16] Paulo Sergio Almeida, Carlos Baquero, and Victor Fonte. "Interval tree clocks". In: International Conference On Principles Of Distributed Systems. Springer. 2008, pp. 259–274.