Enforcing Consistency in Microservice Architectures through Event-based Constraints

Anna Lesniak
University of Copenhagen
Copenhagen, Denmark
sxl492@alumni.ku.dk

Rodrigo Laigner
University of Copenhagen
Copenhagen, Denmark
rnl@di.ku.dk

Yongluan Zhou
University of Copenhagen
Copenhagen, Denmark
zhou@di.ku.dk

ABSTRACT

Microservice architectures are an emerging paradigm for developing event-driven applications. By prescribing that an application is decomposed into small and independent components, each encapsulating its own state and communicating via asynchronous events, new components and events can be easily integrated into the system. However, by pursuing a model where events are generated and processed at the application-level, developers have a hard time to safeguard arbitrary event interleavings from doing harm to application safety.

To address these challenges, we start by analyzing event-driven microservice open-source applications to identify unsafe interleavings. Next, we categorize event-based constraints to address such unsafe encodings, providing an easy-to-use guide for microservice developers. Finally, we introduce *StreamConstraints*, a library built on top of Kafka Streams designed to enforce explicit event-based constraints defined by developers. We showcase *StreamConstraints* based on the case of a popular event-driven microservice system, and demonstrate how it could benefit from event-based constraints to ensure application safety.

CCS CONCEPTS

• **Information systems → Stream management**.

KEYWORDS

microservice architecture, stream management, application integrity

ACM Reference Format:
Anna Lesniak, Rodrigo Laigner, and Yongluan Zhou. 2021. Enforcing Consistency in Microservice Architectures through Event-based Constraints. In *The 15th ACM International Conference on Distributed and Event-based Systems (DEBS '21), June 28-July 2, 2021, Virtual Event, Italy.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3465480.3467839

1 INTRODUCTION

Background. Microservices are a widely adopted architectural style for building fine-granular, lightweight, and loosely-coupled, service-oriented systems [5]. Each microservice usually enjoys the programming language and database system most appropriate for its workloads and data model. By pursuing a model where data is decentralized, each microservice encapsulates its own private state and exposes such internal state via well-defined application programming interfaces (APIs).

Although microservices are supposedly designed to operate autonomously from each other, functional dependencies across microservices are prevalent [10]. In such cases, microservices often communicate and exchange data via message passing mechanisms, such as asynchronous event-based communication [9]. Asynchronous events have been gaining increasing interest from microservice practitioners due to the benefits of loose coupling between microservices [3]. For instance, new distinct events can be easily integrated into the system since producers and consumers are truly loosely-coupled, positively influencing the ability of the system to evolve over time, limiting the faults' propagation, and possibly allowing further scalability of individual components [5].

Motivation. Although asynchronous event-based communication allows an increased level of flexibility to the system, the asynchronous nature introduces new challenges. Practitioners have to reason about the possible interleaving of event streams across and within a microservice and, since these events usually trigger updates of states, their impact on the private state of individual microservices is also a matter of concern.

Furthermore, as microservices employ substantial amount of data management logic at the application-level, practitioners often resort to handling application integrity in the application-level [10], a problematic approach that often fails to ensure that data integrity violations are missing [1]. Such an approach, together with the distributed state of microservices (e.g., distinct correlated events being emitted by different microservice sources), creates a barrier for practitioners to reason about application correctness and implement event-based constraints.

`Challenge #1:` As microservice architectures rely on message brokers for pushing events so other microservices can capture/subscribe, it is often the case where events may be delayed, duplicated, or even lost. This becomes a challenging scenario when it comes to distinct events that present a causal dependency. For instance, distinct events may present a referential constraint that inhibits a "dangling" event from being processed while its "parent" has not arrived yet.

`Challenge #2:` Another challenge comes from the case when two distinct events, with no explicit dependencies (e.g., no referential constraint), happen to arrive at the same time in a microservice and end up being processed concurrently. In the case when their interleaving leads to opposite outcomes in the application state, the order on which the two events are processed may impact on application safety.

DEBS '21, June 28-July 2, 2021, Virtual Event, Italy
© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-8555-8/21/06…$15.00
https://doi.org/10.1145/3465480.3467839

In both cases, the weak semantics observed on the eventual delivery of events creates a barrier for ensuring event ordering semantics. Through state of the practice abstractions, developers find it hard to express event-based constraints, such as causal consistency (*Challenge #1*) and event processing order (*Challenge #2*). Thus, practitioners are forced to resort to complex and error-prone solutions at the application-level and usually end up eschewing off the enforcement of event-based constraints [10].

Related Work. Although stream processing engines are designed to perform queries that run continuously over unbounded data streams [3], the abstraction targets operations that are different from the ones performed in microservices. ksqlDB [8] allows applications to maintain materialized views over streams, emit views' updates as events, and query data on demand as in a traditional database. Kafka Streams [6] enables data transformations over streams at the application-level, leveraging guarantees (e.g., fault tolerance) provided by a Kafka cluster. Complex Event Processing (CEP) [12] engines focus on detecting complex event patterns over event streams, as well as transforming and aggregating events. However, all aforementioned engines do not provide abstraction for the specification and enforcement of event-based constraints over streams.

Contributions. The complex interaction patterns found in microservices force developers to deal with the possible interleaving of event streams. Built on the observation that microservice applications already implement substantial data management logic at the application-level, we take advantage over this fact to allow developers to explicitly define the relevant dependencies of events [2]. Through a library abstraction on top of Kafka Streams API [6], we allow developers to specify dependencies cutting across distinct events, thus enjoying the benefit of not having to track all potential causality of events that present no dependencies across each other. Our contributions are listed as follows:

(i) We analyze popular open-source repositories that exhibits event-driven semantics for interactions between microservices [4, 11] in order to investigate application logic encodings that are unsafe under arbitrary event interleaving. These unsafe encodings highlight the existence of implicit dependency constraints over streams and the lack of proper event-based constraint enforcement raises a risk to application safety.

(ii) Based on the identified unsafe encodings, we categorize a set of event-based constraints that represent different scenarios where distinct, but correlated events can interleave and negatively affect data consistency. The constraints categorized represent an easy-to-use guideline for developers implementing event-driven microservices in the wild.

(iii) We provide system-level support to enforce event-based constraints on top of Kafka Streams API, thus alleviating developers' burden on handling complex and dependent interactions between microservices.

(iv) To demonstrate our solution, we select a popular open-source microservice application as a baseline and guide the audience on the process of specifying event-based constraints, thus refraining the application from experiencing unsafe event interleaving.

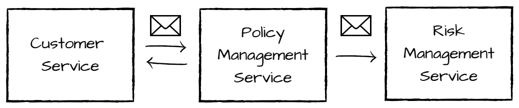

Figure 1: A partial view of *Lakeside Mutual* application

2 CASE STUDY

We illustrate the problems brought about by the complex interplay of event-driven microservices through a case study of *Lakeside Mutual*, an open-source microservice application for managing insurance policies [11]. Although other open-source applications present similar features [4], we decided for *Lakeside Mutual* due to its popularity, the event-driven nature, and the presence of heterogeneous event-based implicit constraints that are unsafe under arbitrary interleavings.

The application consists of several microservices, each serving a specific-purpose concern, such as managing insurance policies, handling customer data, and calculating risks associated to insurance emission. The microservices are interconnected via defined asynchronous events, as exhibited in Figure 1 (adapted from [11]), and each microservice relies on a specific-purpose database for its own data management.

`Use Case #1: Policy Management`

Context. The *Risk Management* microservice is responsible for maintaining a materialized view of the active policies of the system based on observed policy-related streams reflecting policy data items updates. As seen in Figure 1, the *Risk Management* microservice subscribes to policy update events generated by the *Policy Management* microservice. A policy start with the status `CREATED` and, depending on an agent's analysis (an application user), can evolve to `ACCEPTED` or `DENIED`. In any case, updates to the internal structure of the policy object are also propagated (e.g., fixing the name of the client). Eventually, after further analysis by an agent, a policy may also be deleted. In this case, the policy is removed from the materialized view and users can no longer observe such policy.

Problem. The eventual delivery of policy streams creates a challenging scenario for developers. When there is no tracking of the order of updates of a same object, a late arriving update may be overwritten by a precedent update, which would not be applied if causal ordering was observed between these two events. Although one may argue that eventual consistency yields a compelling model for distributed applications like microservices, some interleavings may lead to an inconsistent view. As an example, consider the case where an agent unintentionally assigns an incorrect policy to a given client. A few seconds later, the agent realizes that such operation is wrong and then corrects the fact by issuing a compensation, assigning the policy to the correct client. It is easy to observe that an incorrect interleaving of these streams decidedly leads to a wrong view of the state in the *Risk Management* microservice.

`Use Case #2: Insurance Quote Expiration`

Context. The *Policy Management* microservice also interacts with the *Customer* microservice in the process of managing insurance quotes. The typical scenario consists of an insurance quote client request being submitted to the *Policy Management* through the *Customer* microservice. An agent can then offer a quote and send it back to the client. If the client accepts the provided quote, the policy is created. Besides, every proposed quote has an associated expiration date. The *Policy Management* microservice runs a periodical batch job, triggered by an internal event, that retrieves all the insurance quotes which expiration date has passed and, for each quote, sends an `InsuranceQuoteExpired` event to the *Customer* microservice.

Problem. However, it is possible that the expiration process is interleaved with the customer accepting the provided quote. Consider the case that the policy creation process is initiated in *Customer* microservice before the insurance quote expiration event is issued by the *Policy Management* microservice. The correct behavior of the system would be acknowledging that the creation process started before the expiration and thus discarding the expiration event. However, a scheduling of functions that react to these distinct events that does not obey such ordering constraint decidedly lead to an inconsistent application state.

3 SYSTEM CONSTRAINTS AND DESIGN

As seen in Section 2, the arbitrary interleaving of event streams poses a risk to data integrity. However, the lack of intuitive and effective abstractions for expressing event-based constraints in microservices force developers to either eschew fending off unsafe behaviors or resort to complex and error-prone implementations at the application-level.

To address this gap, we start by proposing a categorization of event-based constraints that reflects the identified unsafe interleavings of event streams on microservice applications. Furthermore, we also present our system-level design built on top of Kafka Streams to effectively address the identified gaps.

Causal Constraints. A common case where events are produced in microservices concern propagating updates of data items, so subscribed microservices can build materialized views over external data. **`Challenge #1`** and **`Use Case #1`** presented examples where, when causality within the streams of an event and across dependent distinct events, respectively, are not observed, application integrity may be compromised. In other words, a precedence, or *happen-before* relation must be observed between events in some cases. To allow developers to specify causal constraints, we provide an interface where one can define a pair of events, where the former is considered a required prerequisite for processing the latter.

Terminal Constraints. Built from our **`Use Case #1`**, consider again the scenario involving the *Policy Management* and the *Risk Management* microservices. A delete policy operation marks that a policy should not be seen anymore by any of the application users, which also implicates on how other microservices perceive the system state. For instance, no subsequent updates to a deleted policy should be observed by microservices subscribed to such events. In this context, a delete policy can be understood as a terminal constraint, since no other updates to such a policy are allowed to be observed. However, in cases when updates to deleted policies continue being issued afterwards (e.g., due to a bug introduced in the producer microservice), the absence of an event-based constraint related to this terminal constraint in every consumer microservice entails a risk to application integrity. When defining terminal constraints, a developer also has the possibility to specify whether events incoming after the terminal event should be dropped or redirected to another topic.

Listing 1: Window constraint specification example

```
1  new WindowConstraintBuilder[String, InsuranceQuoteEvent]
2    .before((
3      (_, e) => e.isInstanceOf[InsuranceQuoteExpiredEvent],
4      "insurance-quote-expired"))
5    .after((
6      (_, e) => e.isInstanceOf[PolicyCreatedEvent],
7      "policy-created"))
8    .window(Duration.ofSeconds(1))
9    .dropBefore
```

Window Constraints. We now turn our attention to the scenario presented in **`Challenge #2`** and **`Use Case #2`**. Developers have a hard time on specifying at the application-level the conditions on which an expiration event should be processed.

A compelling abstraction widely used in stream processing to reason about the inter-relation of distinct events are windows [6]. A window specifies a bounded fragment of event streams within a time interval. Windows are particularly useful in the case of the insurance quote expiration. Differently from **`Use Case #1`**, where a microservice is the single source of events, in insurance quote expiration, an unsafe interleaving can be caused by different sources (the *Policy Management* and *Customer* microservices). In this sense, by simply stating a window duration and an action to be applied on one or both of the observed streams, we believe developers are better equipped to reason about such unsafe interleaving. An example snippet of such specification is exhibited in Listing 1. For instance, a window constraint that could address this scenario is dropping the insurance expiration event (lines 3 and 4) if a policy creation event (lines 6 and 7) was detected within a 1 second period (line 8). Besides, as the developer may prefer swapping the order of events or dropping the last one observed, our system also provides such support.

System Design. To allow event-based constraints specification, we built *StreamConstraints*, a system built on top of Kafka Streams, a client library for stream processing [6]. *StreamConstraints* allows for specifying event-based constraints for the presented categories and enforces these constraints on the topic where the event stream is mapped into. Developers have the possibility to define a single, primitive constraint or combine multiple different categories into a complex invariant definition. The implementation of *StreamConstraints* relies on the low-level Processor API [7] that allows for creating custom stream processors and interacting with state stores. *StreamConstraints* maintains an internal projection of the state imposed by the processed events. In this sense, the materialized view is used to enforce causal and terminal constraints while other state stores temporarily buffer events involved in window constraints. We believe that taking advantage of an industry-strength stream processing framework like Kafka allows for an easy encoding of constraints and decreased learning curve for microservice practitioners, at the same time taking advantage over the guarantees provided by Kafka, such as fault-tolerance.

An example of how our system is used within the *Lakeside Mutual* microservices is presented in Figure 2. As can be seen, the library can be directly integrated into the *Customer* microservice and is able to supply it with a correct ordering of events concerning

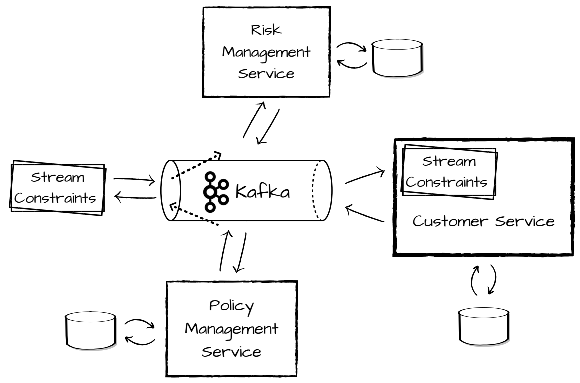

Figure 2: *Lakeside Mutual* microservice architecture diagram with *StreamConstraints*

insurance quote expiration and policy creation. The *Customer* microservice is a Java application and, as this programming language is well supported by Kafka Streams, the library can be directly used in the microservice. In the case of *Risk Management* microservice, which is a *NodeJS* application, the library can enforce the constraints apart from the application, publishing the resulted corrected ordered streams back to the message broker for consumption of a consumer microservice.

4 DEMONSTRATION

In our demonstration, we would like to showcase two major points of the presented system: (i) the ability to specify different classes of event-based constraints at the application-level, meeting the state of practice abstractions adopted by microservice developers, and; (ii) The enforcement of event-based constraints through a system-level abstraction built on top of Kafka Streams, that aims to refrain developers from implementing complex and error-prone mechanisms at the application-level.

To ease of reasoning, our demonstration setup is based on our case study (§ 2), inviting the audience to navigate through the *Lakeside Mutual* microservice application. We will start by demonstrating how the lack of event-based constraint enforcement affects application integrity by allowing any arbitrary order of events. The example will consist of the steps necessary to reproduce an instance of data integrity anomaly caused by an interleaving of expiration and policy creation events, as described in Section 2.

Next, we demonstrate how constraints are specified using our system. In this sense, the audience will be invited to propose event-based constraints based on the categorization provided in §3 to address the identified data integrity anomalies presented earlier. To support the reasoning, we will present a graph representation of the inter-relation of the constraints proposed, like the one shown in Figure 3. In the end, we expect to exhibit that the constraint enforcement provided by our system avoids the unsafe interleavings mapped in §2, showing that no data integrity anomalies on event processing are observed, thus ensuring application integrity.

5 CURRENT STATE AND NEXT STEPS

StreamConstraints performs validation of the developer-defined specifications based on cycle detection. In the case of `Order`-related

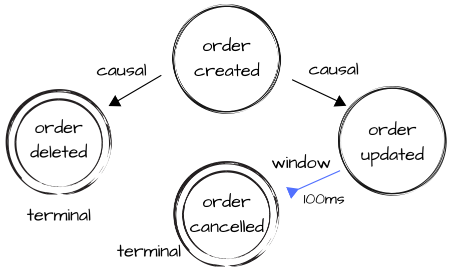

Figure 3: Dependence graph in `Order`-related events

events processing, Figure 3 exhibits an example dependence graph built based on an input specification, where edges represent causal and window relations. For instance, as a safety measure, given a terminal constraint (e.g., `OrderDeleted`), the system inhibits the specification of any dependence relation on which an event occurs after a terminal event. Allowing developers to specify their event constraints using graphical abstractions may be a worthy avenue.

As of this witting, *StreamConstraints* provides application-level abstractions for specifying pairs of event types that present a dependence relation according to application semantics through causal and window constraints. In future, we intend to allow for more complex relations, such as involving more than two distinct types of events.

ACKNOWLEDGMENTS

This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie agreement No 801199 and Independent Research Fund Denmark grant No 9041-00368B.

REFERENCES

[1] Peter Bailis, Alan Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2015. Feral Concurrency Control: An Empirical Investigation of Modern Application Integrity. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 1327–1342.

[2] Peter Bailis, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2012. The Potential Dangers of Causal Consistency and an Explicit Solution. In *Proceedings of the ACM Symposium on Cloud Computing*. Article 22.

[3] Confluent. 2021. *Event-Driven Microservices Architecture.* https://www.confluent.io/resources/event-driven-microservices (Accessed on 2021-05-01).

[4] Dotnet-Architecture. [n.d.]. dotnet-architecture/eShopOnContainers. https://github.com/dotnet-architecture/eShopOnContainers

[5] Pooyan Jamshidi, Claus Pahl, Nabor C. Mendonça, James Lewis, and Stefan Tilkov. 2018. Microservices: The Journey So Far and Challenges Ahead. *IEEE Software* 35, 3 (2018), 24–35.

[6] Apache Kafka. 2021. *Kafka Streams.* https://kafka.apache.org/27/documentation/streams (Accessed on 2021-05-02).

[7] Apache Kafka. 2021. *Kafka Streams Processor API.* https://docs.confluent.io/platform/current/streams/developer-guide/processor-api.html

[8] ksqlDB. 2021. *ksqlDB: The database purpose-built for stream processing applications.* https://ksqldb.io (Accessed on 2021-05-01).

[9] Rodrigo Laigner, Marcos Kalinowski, Pedro Diniz, Leonardo Barros, Carlos Cassino, Melissa Lemos, Darlan Arruda, Sérgio Lifschitz, and Yongluan Zhou. 2020. From a Monolithic Big Data System to a Microservices Event-Driven Architecture. In *2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. 213–220.

[10] Rodrigo Laigner, Yongluan Zhou, Marcos Antonio Vaz Salles, Yijian Liu, and Marcos Kalinowski. 2021. Data Management in Microservices: State of the Practice, Challenges, and Research Directions. arXiv:2103.00170 [cs.DB]

[11] Microservice-API-Patterns. [n.d.]. Microservice-API-Patterns/LakesideMutual. https://github.com/Microservice-API-Patterns/LakesideMutual

[12] Eugene Wu, Yanlei Diao, and Shariq Rizvi. 2006. High-Performance Complex Event Processing over Streams. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 407–418.