Operational Stream Processing: Towards Scalable and Consistent Event-Driven Applications

Asterios Katsifodimos
Delft University of Technology
Netherlands
a.katsifodimos@tudelft.nl

Marios Fragkoulis
Delft University of Technology
Netherlands
m.fragkoulis@tudelft.nl

ABSTRACT

In the last decade we are witnessing a widespread adoption of architectural styles such as microservices, for building event-driven software applications and deploying them in cloud infrastructures. Such services favor the separation of a database into independent silos of data, each of which is owned entirely by a single service. As a result, traditional OLTP systems no longer fit the architectural picture and developers often turn to ad-hoc solutions that rarely support ACID transaction consistency.

At the same time, we are witnessing the gradual maturation of distributed streaming dataflow systems. These systems nowadays have departed from the mere analysis of streaming windows and complex-event processing, employing sophisticated methods for managing state, keeping it consistent, and ensuring exactly-once processing guarantees in the presence of failures.

The goal of this paper is threefold. First, we illustrate the requirements of stateful software services in terms of consistency and scalability. Second, we present how well existing solutions meet those requirements. Finally, we outline a set of challenging problems and propose research directions for enabling event-driven applications to be developed on top of streaming dataflow systems. We strongly believe that streaming dataflows can have a central place in service-oriented architectures, taking over the execution of ACID transactions, ensuring message delivery and processing, in order to perform scalable execution of services.

1 INTRODUCTION

Event-driven Applications (EDA) are software applications that act on incoming events. EDAs nowadays span a multitude of areas. Some very common examples are GUI applications (for web services, gaming, design, etc), complex event processing (for fraud detection, pattern matching, etc.), computations on streaming graphs and machine learning, and analytical queries over streams, such as window aggregation. In this paper we focus on two emerging application types named after the architectural pattern they follow, namely *microservices* [20] and actor-based systems [1], such as Erlang [22], Akka,[1], but most importantly, higher lever abstractions such as Microsoft's Orleans [3]. Our work is motivated by the observation that these emerging architectural patterns do not receive the necessary amount of attention from the database community, although they are extremely ubiquitous and growing in popularity by the day.

Microservices and actors have a surprising number of commonalities. Microservices, like actors, are founded on the principle of separation of concerns: each microservice (or actor) manages its own data and implements a set of endpoints (actors offer function calls). The only way for a microservice to get data from another microservice is to make a call on an endpoint (a function call/shipping for an actor). However, the two have a considerable amount of differences. More specifically, microservices favor communication via synchronous REST API calls[2] and ensure fault tolerance by relying on an external database for persistence. Actors, on the other hand, communicate via asynchronous messages and typically persist state in a local data structure or even in an external storage system [3]. The local state of actors can be used for recovering after failures, but also for migrating to different machines. To alleviate data management and consistency issues from actors, Bernstein et. al. [10] proposed the concept of virtual actors, backed by actor-oriented database systems, with the goal of integrating database concepts inside actor systems.

Stateful microservices can scale particularly well, but almost always implement eventual consistency, such as SAGAS [11]. In case that strict consistency is required, orchestration and Two-Phase Commit (2PC) take place at the application level: users hard-code database logic in their applications, using APIs such as Java XA, by implementing commit, rollback, and prepare for 2PC to work. Worse, transactions today contain more and more complex business logic. However, encoding complex business logic in a stored procedure is not preferred nowadays [2]. As a result, strict consistency – an old and difficult problem that was once only the responsibility of very few database programmers – is now part of daily work for application programmers. Finally, stateful services (including virtual actors) require state locality in order to achieve low latency - not only when writing data, but most importantly when reading data. As applications become more and more interactive, reacting to state changes renders latency requirements even stricter. At the same time, serverless computing [23], an emerging trend allowing the execution of user-supplied functions as a service, is proven to be a bad fit for stateful microservices. This is because it necessitates shipping data to the code and forces communication between executing components through the storage layer, which is slow compared to a direct network connection [14].

The aforementioned shortcomings call for a principled solution that will allow implementing EDAs with innate support for transactions, loose-coupling of service modules with local state, and consistent global state. In this paper we argue that such a fabric can be based on streaming dataflows. More specifically, modern streaming dataflow systems, such as Apache Flink [7] and Samza [17], execute a topology of continuously executing operators with local state and maintain a consistent snapshot of their global state. Operators cooperate to provide analytics on bounded and unbounded data. Moreover, features like Flink's Savepoints [6] or Kafka's Streams offer a deterministic time machine for debugging and replaying dataflow executions, and can be used to hide failures from application developers by offering

[1] https://akka.io

[2] Certain microservice implementations such as reactive microservices opt for asynchronous messaging.

exactly-once processing guarantees. Finally, dataflow systems scale extremely well. As a result, at the time of writing, we are witnessing a trend towards building stateful applications on top of streaming engines.

In this paper we present the vision of *operational stream processing* whose goal is to render stream processors full-fledged data management engines, capable of executing transactions, performing analytics, and embedding complex business logic of stateful services inside dataflow operators. We organize this paper as follows: in Section 2 we focus on current best practices for implementing EDAs and their requirements for scalability and consistency. In Section 3 we review existing possible solutions to fill those requirements. Finally, in Section 4 we argue that streaming dataflow systems, such as Apache Flink [7] and Samza [17], can serve as an efficient, and scalable backend for executing EDAs, given that our community tackles a set of important challenges.

2 REQUIREMENTS OF EDAS

In this section we first briefly outline the main requirements of EDAs with respect to the backend technology required for their execution. We then focus on a set of advanced operational requirements of microservices and actors.

2.1 General Requirements

The following requirements manifest themselves in almost all EDAs. We believe that a fabric that can be used as a backend for EDAs should at least provide support for the following.

Fault-Tolerant State. State is a first-class citizen in virtually every event-driven computation. State in a streaming computation can be counters (e.g., counting elements in a stream), database contents in a microservice or the current computation state of an actor program. At the same time a distributed event-processing application, needs to ensure that, even in the presence of failures, the state of the system remains consistent and the application continues its operations from where it left off. Whenever possible, failures should be transparent to the application programmer.

Event Partitioning & Scaling Out. Computations over partitioned data are typically used for computing aggregates, and for scaling-out actor instances and load-balancing user requests (e.g., by partitioning per user id). Similarly, scalability in microservices can be typically achieved by running multiple service instances, balancing the load among them using HTTP proxies.

2.2 Advanced Requirements

Apart from the general requirements, which are needed by most EDAs, most modern applications demand a special set of requirements for the operation of the services that comprise them.

Transactions. One of the largest problems in running services is the lack of coordination schemes in order to perform *transactions* and retain consistent state across services.

State Locality. Access to local state is important for boosting the performance of services [8], yet it is not always leveraged in microservice or actor architectures in favor of keeping those services stateless. In those cases, the state is offloaded to an external storage system. However, stringent latency requirements of interactive server applications require state to remain embedded in the service.

Global State View & Analytics. Services often need to consolidate data from multiple other services in a bulk fashion. For instance consider the case of joining orders with transactions, in order to obtain insight about sales. Since each service owns its data, consolidating that state via multiple calls to service endpoints is currently slow and cumbersome.

Loose coupling. One of the most important reasons that services are so popular is that they allow developers to develop, test, and deploy them in a loosely-coupled fashion. We consider this a defining trait of services that must be respected.

Debugging and Auditing. Given their distributed nature, services inherit the difficulty to reason, understand, and debug. To this end, services need inherent support for debugging and testing in a reproducible manner.

Dynamic (re)configuration. During the lifetime of a service, the load of the service (e.g., due to churn), its assigned hardware (e.g., due to failures), and even the service itself (e.g. due to updates) can change dramatically. Services need to transparently adapt to those changes without being affected in terms of performance and availability.

3 EXISTING SOLUTIONS

Many existing solutions meet the general requirements presented in Section 2.1. The advanced requirements of Section 2.2 are a lot more challenging to support though. This section presents which of the advanced requirements are fulfilled by existing systems, namely microservices frameworks, OLTP systems, actor programming frameworks, and stream processors.

3.1 Microservices frameworks

Transactions. The most widespread solutions for performing transactions in microservices frameworks are SAGAs [11] and application-level transaction managers implementing the Java Transactions API (JTA). A transaction in the SAGA pattern allows microservices to make local state changes (e.g., book a flight/hotel) independently of the rest of the microservices taking part in the transaction. On failure, all microservices that have already updated their state, need to issue compensating actions (e.g., cancel the flight booking). SAGAs pose two main issues: i) state consistency across microservices cannot be achieved and ii) not every action can be compensated. Various web application development frameworks, such as Spring, offer eXtended Transactions (XA), an implementation of 2PC with ACID-like properties for web applications [18]. The solutions based on application-level transaction managers require a tight coupling of services, and demand from developers to implement 2PC functions (e.g., `prepare` and `abort`) in application-level code which is very error prone and necessitates deep understanding of database concepts.

State Locality. The state locality requirement contradicts the design of stateless microservices, which dictates handing over the responsibility of persisting state to an external database system. Thus, current microservice frameworks fail to fulfill this requirement. Scaling-out microservices requires very sophisticated design of the backing database architecture. The service's state in these cases is external; accessing it requires a call to an external system, introducing latency.

Global State View. Microservices frameworks are not suitable for providing global state views. Since each microservice owns its data and state, having a complete view over the complete system's data state requires manually making a snapshot of all services' individual databases and importing those in a central database to perform analytics. Such a process cannot be done without special software and protocols; naively dumping all individual

databases into a data warehouse without taking down all services in advance will most certainly result into data inconsistencies.

Debugging & Auditing. Developing microservices using event-sourcing and Command Query Responsibility Segregation (CQRS) [5] allows developers to replay message exchanges between services and debug their applications by rebuilding the state of an application at the time that a bug occurred.

3.2 Distributed OLTP Systems

Distributed OnLine Transaction Processing (OLTP) systems such as VoltDB [21] and H-Store [15], can be used as a backend for EDAs since they provide scalability, consistency, global view of the application state, and analytics. However, they introduce a number of issues. First, OLTP systems require that transactional code is included in the database as a stored procedure. However, transactional code is often an indivisible part of application code, a microservice's business logic for instance. Pushing the business logic into the database is largely disliked [2]. Second, distributed OLTP systems partition their state as they see fit, yet in order to achieve low latency, the state of each service should be locally available and even in the same memory space as the service itself. Finally, having one distributed database that manages the state of all applications goes against loose-coupling and requires that different services agree on a given schema, database system, etc.

3.3 Actor programming frameworks

Actors are good examples of loosely-coupled systems, that can be reconfigured in the presence of failures and environment changes. In addition, effective debugging can be achieved by use of the event sourcing and CQRS patterns, like in reactive microservices. Erlang, Akka, and Microsoft Orleans are popular actor-based programming frameworks. In the rest of this section, we focus on Orleans, which offers the highest level of abstraction among actor-based systems.

Transactions. Actors in general do not provide means for executing distributed transactions. Orleans appears to be implementing some form of ACID transactions with 2PC based on batching [10].

Global State View & Analytics. Actors can offer state locality but they lack the capability to provide a consolidated view of the global state and perform analytics on that state. For instance, Orleans can save the persistent state of actors locally or in inexpensive cloud storage that can be replicated for scalability and fault tolerance. Adopting the encapsulation principle, each actor has local access to its own data and state and restricted access to the state of other actors as per their public interface. This organization works for established actor-local operations, but leaves much to be desired in terms of a consistent global state view that can be used to answer queries on the complete state of an application. Performing analytics on global state in an actor system would be similar to a microservices.

3.4 Stream Processing Systems

So far stream processors are primarily known for their capacity to support high-throughput and low-latency analytics. However, modern stream processors such as Apache Flink [7], also support distributed state consistency via exactly-once state processing guarantees. Stream processors can serve as a platform for running EDAs in a scalable [16] and consistent fashion [6]. We argue that additional research needs to be performed in order for stream processors to be able to satisfy the operational requirements of EDAs, namely transactions, query-able global state, and loose coupling at the API level. Current stream processors lack appropriate programming models that allow developing microservice architectures. At present, they only offer functional APIs focused on bulk event processing, rather than message exchange that allows for loose microservice coupling. Moreover, stream processor systems need transactional facilities to support advanced business logic and coordination. The only two exceptions that are available at the time of writing are i) a closed-source implementation of multi-key transactions in Apache Flink, as well as S-Store [19], which provides ACID guarantees on shared mutable state on a single machine. Having a global state view of an application is also missing. Apache Flink, for instance, only supports external querying of a single operator's state.

Finally, stream processors can serve as a good basis for debugging distributed services. For instance, message brokers, such as Apache Kafka [17], can be used for storing all messages that are exchanged among microservices and replaying them during debugging. Moreover, Apache Flink's savepoints[3] can be used to replay events from a specific past consistent state of the stream topology in order to debug an application.

4 THE ROAD AHEAD

This section summarizes the research directions our community needs to take in order to realize the operational stream processing vision. We envision EDAs to be authored in a service-oriented API. Such an API would enable a service to have custom business logic, to communicate with other services via (a)synchronous message exchange, and to have access to local state.

A set of services authored in that API can be compiled into a dataflow graph, an example of which is depicted in Figure 1. Edges represent channels of message exchange among services. Vertices execute custom business logic, and are given access to managed local state. The dataflow engine takes care of the state's consistency, as well as the routing and exactly-once processing of messages within services. For the services executing in the dataflow graph to send and receive requests to/from external systems, they have to write or read to a message queue/log (e.g., Apache Kafka). This, not only enables exactly-once processing and delivery of messages, but it can also hide failures from the application programmers who can assume that, if a message has been sent, their application will receive an answer to that message exactly-once. Finally, certain parts of the inputs can be replayed in order to debug/audit a given service deployment.

One would argue that such an architecture could be implemented on existing systems such as Apache Flink, Samza, or Kafka. However, a set of requirements that we introduced in the previous section remain unsatisfied and require further research.

Transactions. Current dataflow systems guarantee consistency of single-event changes on a given partition of state. In order to guarantee consistency during multi-key, multi-partition state changes, we need to extend existing approaches of consistent snapshots [6] drawing ideas from distributed systems [9], and traditional OLTP systems. Some form of timestamp-based concurrency control could be employed [4, 13], especially given that processing- or event-time is a first-class citizen of all events entering a stream processor. Furthermore, we can invent 2PC-like protocols that take advantage of the FIFO connections between computation vertices in dataflow graphs, and ensure the order in which transactions arrive and state changes are applied.

[3] https://ci.apache.org/projects/flink/flink-docs-stable/ops/state/savepoints.html

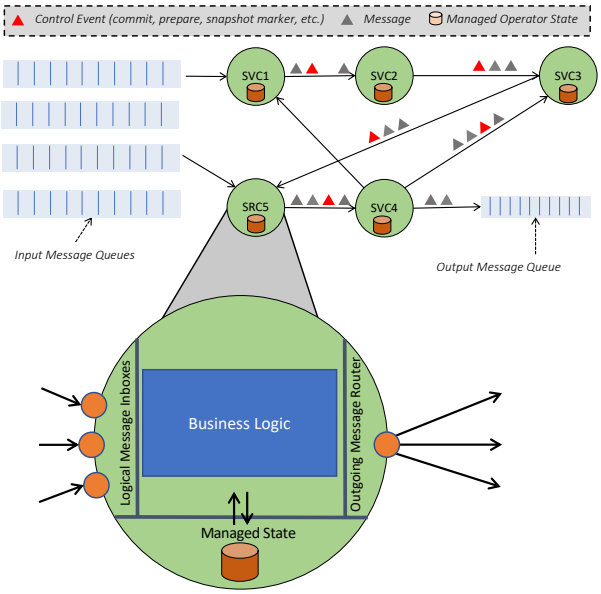

Figure 1: A set of services running inside a streaming dataflow graph. The business logic of services runs as an operator that processes messages and produces responses, the state of the service is managed by the dataflow engine, and all inputs and outputs of a given system are logged.

(Non-Dataflow) APIs for Loosely-coupled Services. Current dataflow systems only allow developers to author data pipelines, by defining data dependencies among operators and user-defined functions such as `Map` and `Reduce` in an explicit manner. To allow developers to develop loosely-coupled data-intensive services, we need novel APIs which will allow developers to individually develop, test, and debug services. Those services can be automatically compiled into a single, efficient, and scalable dataflow graph. To this end, we can derive the data dependencies from the defined messages/endpoints that applications send to each other, building the dataflow graph dynamically.

Consistent Access to Global State. Consistent snapshots [6] of transactional dataflows need to provide safe and consistent read access to global state, i.e., we need the means to execute distributed queries over the state of different operators. More specifically, we could provide the means for services to publish (views over) their state on top of which other services can build materialized views. Materialized views can maintain fresh results [12] and guarantee locality.

Dynamic (re)configuration. Currently, dataflow systems build a dataflow graph statically and then parallelize and deploy it, since all the data dependencies among operators are pre-defined. However, frequent and independent updates of services necessitate highly dynamic graphs with network channels that can be created or destroyed at any given time during the execution of a service. The ultimate goal is that the performance of existing services is not affected by changes in the dataflow graph. An important use case of dynamic configuration is the automatic parallelization of services. As we mentioned earlier, each service obtains access to managed local state. However, both the state and the input messages that are directed to a given service should be partitioned whenever possible, in order to ensure parallel execution. To this end, given a key for each state object, we aim at automatically parallelizing and even replicating service deployments and optimizing them for throughput and latency, without sacrificing consistency.

5 CONCLUSIONS & FUTURE WORK

In this paper we made a case for using streaming dataflow systems as a backend for stateful event-driven applications, such as microservices. We listed a set of requirements that include ACID transactions, global state consolidation, and the need for debugging and auditing. We then used those requirements to draw a rough outline of the future work that we believe has to take place, to materialize the vision of operational stream processing.

REFERENCES

[1] Gul Agha. 1986. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, MA, USA.
[2] 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 *SIGMOD '15.*
[3] P. A. Bernstein and S. Bykov. 2016. Developing Cloud Services Using the Orleans Virtual Actor Model. *IEEE Internet Computing* 20, 5 (2016).
[4] Philip A. Bernstein and Nathan Goodman. 1981. Concurrency Control in Distributed Database Systems. *ACM Comput. Surv.* 13, 2 (June 1981).
[5] Dominic Betts, Julian Dominguez, Grigori Melnik, Fernando Simonazzi, and Mani Subramanian. 2013. *Exploring CQRS and Event Sourcing: A Journey into High Scalability, Availability, and Maintainability with Windows Azure* (1st ed.). Microsoft patterns & practices.
[6] Paris Carbone, Stephan Ewen, Gyula Fóra, Seif Haridi, Stefan Richter, and Kostas Tzoumas. 2017. State management in Apache Flink®: consistent stateful distributed stream processing. *PVLDB* 10, 12 (2017).
[7] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink®: Stream and Batch Processing in a Single Engine. *IEEE Data Eng. Bull.* 38 (2015).
[8] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin Levandoski, James Hunter, and Mike Barnett. 2018. FASTER: A Concurrent Key-Value Store with In-Place Updates. In *SIGMOD.* ACM.
[9] K. Mani Chandy and Leslie Lamport. 1985. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Trans. Comput. Syst.* 3, 1 (1985).
[10] Tamer Eldeeb and Phil Bernstein. 2016. *Transactions for Distributed Actors in the Cloud.* Technical Report.
[11] Hector Garcia-Molina and Kenneth Salem. 1987. Sagas. In *SIGMOD '87.*
[12] Jon Gjengset, Malte Schwarzkopf, Jonathan Behrens, Lara Timbó Araújo, Martin Ek, Eddie Kohler, M. Frans Kaashoek, and Robert Morris. 2018. Noria: dynamic, partially-stateful data-flow for high-performance web applications. In *OSDI '18.*
[13] Rachael Harding, Dana Van Aken, Andrew Pavlo, and Michael Stonebraker. 2017. An evaluation of distributed concurrency control. *PVLDB* 10, 5 (2017).
[14] Joseph M. Hellerstein, Jose M. Faleiro, Joseph E. Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. 2019. Serverless Computing: One Step Forward, Two Steps Back. In *CIDR '19.*
[15] Robert Kallman, Hideaki Kimura, and Jonathan Natkins et al. 2008. H-store: A High-performance, Distributed Main Memory Transaction Processing System. *PVLDB* 1, 2 (Aug. 2008).
[16] Jeyhun Karimov, Tilmann Rabl, Asterios Katsifodimos, Roman Samarev, Henri Heiskanen, and Volker Markl. 2018. Benchmarking Distributed Stream Data Processing Systems. In *ICDE '18.*
[17] Martin Kleppmann and Jay Kreps. 2015. Kafka, Samza and the Unix Philosophy of Distributed Data. *IEEE Data Eng. Bull.* 38, 4 (2015), 4–14.
[18] X/Open Company Ltd. 1991. "Distributed Transaction Processing: The XA specification", X/Open CAE Specification.
[19] John Meehan, Nesime Tatbul, Stan Zdonik, Cansu Aslantas, Ugur Cetintemel, Jiang Du, Tim Kraska, Samuel Madden, David Maier, Andrew Pavlo, et al. 2015. S-Store: streaming meets transaction processing. *PVLDB* 8, 13 (2015).
[20] Sam Newman. 2015. *Building microservices: designing fine-grained systems.* " O'Reilly Media, Inc.".
[21] Michael Stonebraker and Ariel Weisberg. 2013. The VoltDB Main Memory DBMS. *IEEE Data Eng. Bull.* 36 (2013), 21–27.
[22] Robert Virding, Claes Wikström, and Mike Williams. 1996. *Concurrent Programming in ERLANG (2Nd Ed.).* Prentice Hall International (UK) Ltd.
[23] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. 2018. Peeking Behind the Curtains of Serverless Platforms. In *USENIX ATC '18.*