Archer: An Event-Driven Architecture for Cyber-Physical Systems

Javier Moreno Molina
New Digital Business R&D
BBVA
Madrid, Spain
javiermoreno@ieee.org

Juan Ferrer García
New Digital Business R&D
BBVA
Madrid, Spain

Carlos Kuchkovsky Jiménez
New Digital Business - R&D
BBVA
Madrid, Spain

***Abstract*—A global Internet of Things demands new cloud architectures that can scale to support all kind of loosely coupled data and events producers and consumers. Moreover, these architectures need to be capable of offering low-latency delivery, which must not only feed real-time analytics applications, but also produce real-time actionable data that could enable operational business logic and automation triggers activation and transform microservices into cyber-physical microapplications.**

The design principles to build distributed event-driven applications are known to be complex and hard to handle at the application code level. As a result, there are several already well-known patterns that aim to provide a convenient solution and abstract the business logic development from the underlying architecture complexities.

This paper proposes a common framework that addresses all the needs of an in-production application, that are hard to solve under the event-driven paradigm. It gathers existing patterns and combines them and extends them. These patterns are implemented and comprised in a middleware library which has been tested on a proof-of-concept architecture implementation, based on Apache Kafka.

***Index Terms*—events, streaming, kafka, event-driven, Internet of Things, IoT, Cyber-Physical Systems, real-time**

I. Introduction

The Internet of Things has experimented a first phase in which connected devices have mainly become a part of information systems, feeding data to data and analytics applications. The vision of IoT however, involves a second phase in which *things* are capable of directly and openly interacting with software systems, becoming cyber-physical applications. These applications can autonomously make decisions and take actions based on available data and its analysis.

Autonomous applications performing both data analysis and operations not only require the development of cutting-edge connected and *smart* devices. While the focus has been placed on the electronics and communications industry, the truth is that cloud architectures are barely prepared to support an open internet of cyber-physical applications. The confluence of both analytics and operations is against the main design decisions and optimizations made in data architectures during the last decades, and will therefore require a deep redesign of data and software architectures for Internet services.

For instance, Online Analytical Processing (OLAP) and Online Transaction Processing (OLTP) seggregation was based on delivering service to two different groups of human users: customers and business analysts. However, that seggregation is fading due to automation.

In cyber-physical applications the extraction of actionable data from analytics must support and orchestrate autonomous decisions and operations, as depicted in Figure 1. The so far one-way flow of data from transactions to analytics becomes a closed-loop where the result of data analysis may produce new operations, therefore demanding analysis on most recent transactional data.

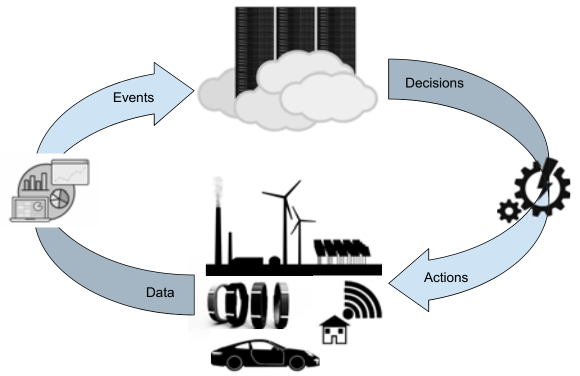

Fig. 1. Cyber-Physical Systems Closed Loop

Moreover, actionable data in many cyber-physical applications is linked to a specific context and has a very narrow window of opportunity, in which the system must make a decision. Otherwise, it will lose its value, similarly to the ”perishable insights” described by Forrester [1].

Although applications in the analytics sphere are demanding some convergence and low-latency reactions, the scalability challenge is still in place. While OLAT and OLTP seggregation enabled dealing with the CAP theorem in different ways depending on the domain, convergence requires dealing with the complexity of designing a dependable and performant system where the architecture cannot provide the same guarantees. In particular, it is known to be difficult to use and develop systems, specially in the transactional domain, with eventual consistency. The architecture and patterns proposed in this paper aim to define a framework for developers and users to deal with that complexity.

The article is organized as follows: in Section II the state-of-the-art on event-driven architectures is discussed. Afterwards, Section III presents the Archer architecture proposal, including the adopted patterns and infrastructure. Next, Section IV describes an implementation of the proposed architecture, which is evaluated in Section V before concluding in Section VI.

II. Related Work

The need for new architectures for Internet of Things and Cyber-Physical Systems is widely accepted by the state-of-the-art and has been a matter of study in recent years. The complexity of event-driven architectures has been addressed in different domains. Brooks et al. [2] propose an actor-oriented streaming model of computation to handle the interactions between the heterogeneous components of IoT Cyber-Physical applications, but it focuses more on defining proxy interfaces (accessors) and more deterministic semantics, based on time-stamping, but it does not define the model for data and event storage. Actor-oriented streaming is also the basis for some frameworks for *Kappa* architectures in real-time big data analytics use cases, such as Akka Streams [3]. The main focus however, is to deliver results in a database for later consumption.

Data Oriented Architectures have also been proposed in [4], where specifically, Data-Distribution System (DDS) [5], was discussed as a standard and the relevance of middleware infrastructure is valuated.

Consistency models are also a rather old topic, that has also driven research in other areas, such as multiprocessor systems. In [6], the memory consistency and the ordering problems are explored and different models are described and discussed.

The complexity of distributed transactions is also present in the literature. In [7], distributed transaction pitfalls and useful patterns to relieve developers from failing are explored, in order to let developers focus on business logic development.

Patterns for event-driven microservices have also been studied and implemented in [10], which proposes a scalable architecture based on Event Sourcing, Command Query Responsibility Segregation (CQRS) and Domain-Driven Development (DDD). However, it disregards some operational difficulties, such as event reprocessing. The concept of event sourcing and CQRS has practically been explored in [8]. To be capable of ordering events, there is a topic for commands, however, there is a single common topic for all commands in the platform.

The concept of IoT microservices is explored in [9]. Different patterns and best practices are discussed, as for instance, the self-containment of microservices, which should contain everything they need to fulfill their tasks.

In [11], tools for building event-driven microservice architectures are presented, based on Apache Kafka. The implementation presented here is based on those tools at the infrastructure layer.

III. Archer Architecture

To satisfy the needs for a Cyber-Physical Internet of Things (CPIoT), this paper proposes an architecture, which, unlike ordinary web services or big data architectures, must consider several critical aspects:

- **Interoperability:** The idea of the proposed architecture is to provide a common dataspace where anything can easily interact, either by contributing with its own data, or by consuming different data sources in order to take some actions.
- **Flexibility:** The IoT must be naturally extensible. An application contributing with data to the system shall not make any assumption about who and how will make use of that data.
- **Low Coupling:** In order to provide maximum flexibility and reduce integration cost and maintenance, one of the objectives is to enable loosely coupled services. This goes hand in hand with the aforementioned interlocutor unawareness.
- **Asynchrony:** A synchronous response of the service providers cannot be assumed. IoT is essentially decentralized and devices connectivity is not always stable, so operations on the system will not always have an immediately visible response.

Reactive systems already offer a solution to most of these aspects [12]. The Reactive Manifesto, however, focuses in message-driven architectures, as it considers them to be more resilient than event-driven architectures. However, this statement is based on the ephemeral nature of event consumption chains. In the prospused architecture, event consumption chains are actually persisted, and therefore disaster recovery and resilience becomes possible also in an event-driven architecture, as events can be made available to new consumers or can be re-processed for error correction.

A. Data Architecture

The foundation of the proposed architecture is an Event Store, which is where all events in the system are logged. The Event Store must support concurrency and be loosely coupled with the services writing or reading data to and from it. It acts as a common shared-memory for all processing components and it can be seen as a Data Bus, where new publishers or subscribers can be initiated or removed with no configuration. A component may emit events, but is not aware of other components listening to them.

The Event Store is built following a topic-based publish/subscribe pattern. Therefore, every event is written in the Event Store under a specific topic. Depending on the kind of data being written on the system, the architecture considers the following type of topics:

- **Data Changelog:** This is the foundation of system data. It follows the event sourcing pattern, where every change in system data is logged as an event and appended to the data stream. Every entity of the system must have its own independent data changelog topic.

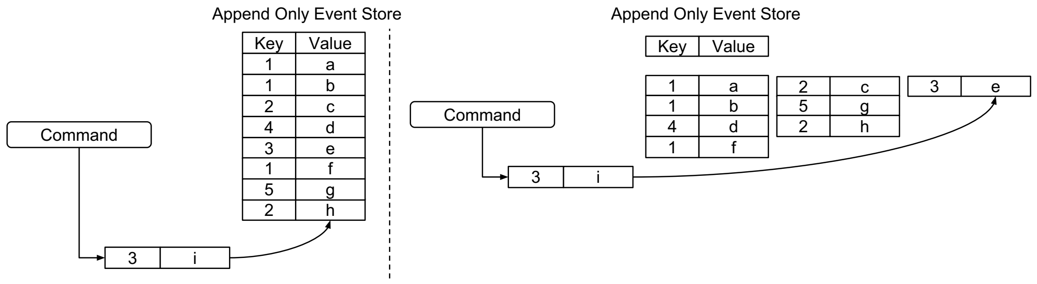

Fig. 2. Key-Value Event Log, single (left) and distributed by key (right)

- **Raw Datalog:** they contain real-time datastreams where every input is timed and independent, and is not an incremental change over previous data. E.g. telemetry data.
- **Command Log:** It registers any input to the system from an external application, e.g. a client website or a mobile app, where a user introduces some data. The interaction needs to be registered as it is, in order to provide traceability and error correction.
- **Transaction Log:** It is used whenever data is exchanged beyond the system boundaries. It records communication initiated from within the system using a two way commit procotol, to detect any failure in external read/write operations and to keep ephemeral data within the system to enable reprocessing system state with data sources that may have expired or may not be accessible anymore.
- **Event Log:** The event log aims to support complex event processing through an acyclic dataflow model paradigm. Event log does not introduce new information in the system, but contains derived data that can be obtained from processing several data changelogs.
- **Snapshot:** In order to reduce the need of reprocessing and reduce recovery time in case of local-state reconstruction, fixed pictures of entities are saved as snapshots from time to time.

Figure 2 shows the event log without distribution and with three partitions, distributed by key. All events modifying entity "1" are sent to the first partition. This way, ordering is only required within the same partition and there is no need for synchronization or ordering consensus among separate data partitions.

B. Processors and Protocols

The proposed architecture defines semantically independent contexts and bind them to a specific processor unit. Due to data parallelism restrictions, there can be at maximum one processor per data partition. Depending on the protocol executed there are four kind of processors:

1) **Microapplications:** Unlike microservices, microprocessors do not "serve" the data. They enable the input of newly generated data into the system and may also serve a materialized view that conforms to its own application-specific domain, which is called the microapplication *local-state*.
2) **Gateways:** They are in charge of connecting to external systems. They receive commands from other components. They process those commands and start a two way commit protocol with the external service they connect to.
3) **Data Ingestors:** They just write raw data in a raw datalog topic as long as it fulfils the defined schema and security requirements.
4) **Event Processors:** They process complex data to generate high level events and write them on the event log.

1) Microapplication Protocol: Microapplications offer an interface to external entities to interact with system data, e.g. create or modify entities. In order to dependably achieve this, there is a micrapplication protocol that ensures consistency and reprocessability, shown in Figure 3:

1) Every authorized and valid external interaction, which fulfils the defined interfaces, is recorded in the system as fast as possible to prevent data loss. These interactions are treated as commands rather than events, which essentially means that they are only interpretable by the microapplication offering the interface and therefore, they have a specific destination.
2) The destination microapplication must then subscribe to its corresponding commands stream and process them, according to its internal business logic, to transform them into a platform event, which is finally interpretable by any other processor.

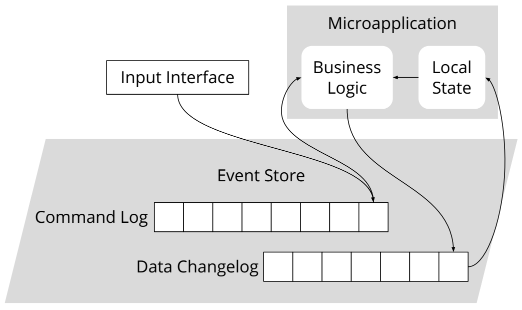

Fig. 3. Microapplication protocol steps diagram

Therefore the microapplication protocol is a combination of command sourcing and event sourcing patterns. Changes in the business logic should not affect commands writing. Reprocessing the commands stream should produce a deterministic events stream output, enabling consistent error correction.

2) Gateway Protocol: Gateways are used to interact with external systems from within the system. To ensure consistency and to manage asynchronous interactions with external services, there is a need for controlling that interaction within the system, so that ongoing operations are not lost in case of failure. Thus, any gateway processor must implement the following protocol, shown in Figure 4:

1) Gateways may initiate a external interaction either based on a command received through their respective command stream or based on a scenario recognition materialized in a system event. In either case they must open an interaction and enclose the trigger and record it with a unique identifier in their respective transaction log.
2) The following interaction events, in most simple case, the response, must be recorded in the transaction log under the same interaction identifier. The transaction log, however, is not intended to be consumed by other processors, but just as interaction control.
3) Any resulting information that should be read and interpreted by other processors needs to be written in a data changelog or event log topic.

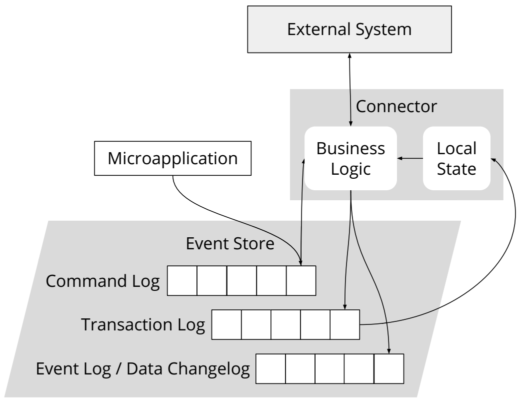

Fig. 4. Gateway protocol steps diagram

As a result, any transaction that occurs between the system and any external service is traceable and any resulting information is recorded as an event.

IV. Archer Event-Driven Framework

The proposed architecture has been implemented as part of the so-called Archer Event-Driven Framework.

The foundation of the proposed architecture is the event store, which is a distributed transaction log, optimized for writing. Apache Kafka, although widely-used as a message queue, is architected, at the storage layer, as a massive transaction log and has a growing ecosystem of tools and connectors that makes it a very powerful tool. Therefore, the implementation uses Kafka as Event repository and configures it with practically infinite retention, so that event logs are immutable and never deleted.

Kafka implements a publish/subscription pattern through their basic structure: the Kafka topic. Topic partitions can be consumed in parallel, but the ordering of events cannot be ensured across different partitions. In the proposed implementation, in order to obtain the desired consistency, all events referring to the same entity must be written on the same partition. This is achieved by setting the entity identifier as the key of the message, as all messages with the same key are sent to the same partition.

Every new piece of data is just appended in the transaction log. However, in order to read the current state of your data, processing the whole data stream is required. To make reading data more efficient, materialized views of entities current state need to be computed. There are two main approaches to creating these queryable states:

1) Propagate new messages to a database, e.g. using Kafka-Connect. These messages are then processed to build the database with system current state.
2) Create an in-memory local-state repository. Every new message is consumed by the microapplication itself, which updates a table with current states.

In our architecture proposal, although the first option is also considered for some use cases, e.g. archive, advanced queries or analytics use cases which are not affected by latency, the main option for reading data is to make use of in-memory local state stores, which are kept in the microapplication components. Therefore, not only it is not necessary to compute all events in a stream for every query, but it is also not necessary to query a remote database, as all interesting states are held in-memory.

This local state serves two main purposes:

1) Serve materialized views of the system current state, e.g. through HTTP endpoints.
2) Minimize latency of event processing by provisioning an always up-to-date version of the event processing required contextual data. This context provisioning reduces the need for querying remote databases.

An in-memory local-state can be implemented using the Kafka Streams library, which uses RocksDB as local storage. However, unlike microservices, which are architected as stateless components that can be replaced and horizontally scaled almost instantly, local-state provisioning makes microapplications stateful, which means that a microapplication replacement will require rebuilding local-state.

Moreover, microapplications may make use of data partitioning for parallelization and throughput improvement. Microapplication instances may then write only in their assigned data partition. The may however keep a cross-partition local-state using Kafka Streams Global KTables, or may just contain the local state for their assigned partition. In this case the system must either wisely select the instance to query, for instance, based on the entity key, or request the data to the instance that owns it, using Kafka interactive queries.

Local-states are therefore a double-edged sword. On the one hand they may help reduce latency in event processing. On the other hand, they lead to stateful microapplications, that need more complex management, specially in case of failure.

A. *Archer Library*

In order to lead with complexity of event-driven architecture, the implementation includes a library, which is actually divided into three sublibraries, to serve as an abstraction layer for the application developer: Domain, States and Common.

1) Domain Library: The domain library includes the base classes and interfaces for defining a domain. It implements the repository and the aggregate patterns.

The repository implementation enables the developer to write and retrieve entity instances. In the former case, it propagates events to the corresponding Kafka data changelog topic. In the latter case it retrieves the current state from the microapplication local storage (KTable).

There is in principle only one Root Aggregate per microapplication. Depending on the different scopes and possible transactions within the domain, aggregates can be added to the root aggregate and to the aggregate map that conforms the domain. Every aggregate writes events automatically on its own topic, which shall only be written by itself.

2) States Library: In order to easily conform a microapplication context, the states library allows the developer to easily build local stores that can be queried within the microapplication to obtain its self-defined current state.

The local-state is not actually local, as it is the result of processing events that have already been sourced to the Event Store (Kafka). It can be seen better as a cached view of the part of the system that is relevant for the application.

3) Common Library: This library includes some common utils that can be used in different contexts. For instance, it enables to easily configure kafka clients with a consistent configuration across the whole system. It also implements some patterns, such as creating a pool of kafka clients to be reused within a microapplication instead of instantiating and populating a new one every time it is required, which is very inefficient.

V. Evaluation

The presented case study is not only a first opportunity to implement the proposed architecture, but also enables evaluating several properties that any architecture to be used in productive environments needs to tackle. In this case, the evaluation focuses on scalability, robustness and consistency, which are all actually closely related.

A. *Scalability*

In any event-driven architecture, the main scalability limitation is the ordering of events. Ordering is only guaranteed within command logs data partitions, while the system may accept events in multiple topics and data partitions concurrently.

In Figure 5, it is shown how the AKF Scale Cube [13] is mapped into the proposed architecture:

- **X:** Every microservice can be replicated. In order to preserve the ordering guarantees, however, the number of microservice replicas is limited to the number of data partitions. For instance, in Figure 5, the microservice A has 3 replicas and 3 data partitions, therefore it is at full capacity, while microservice B has 2 replicas for 3 data partitions. In case scaling is required, it could be incremented in one more replica.
- **Y:** The microservice architecture established different independent semantic contexts that enables scaling them indepdendently. In Figure 5, there are two semantic contexts: A and B, which have different event stores and different services, served through a common API.
- **Z:** Data can be scaled by distributing it on different partitions that can be consumed in parallel. In Figure 5, both event stores A and B have three data partitions each.

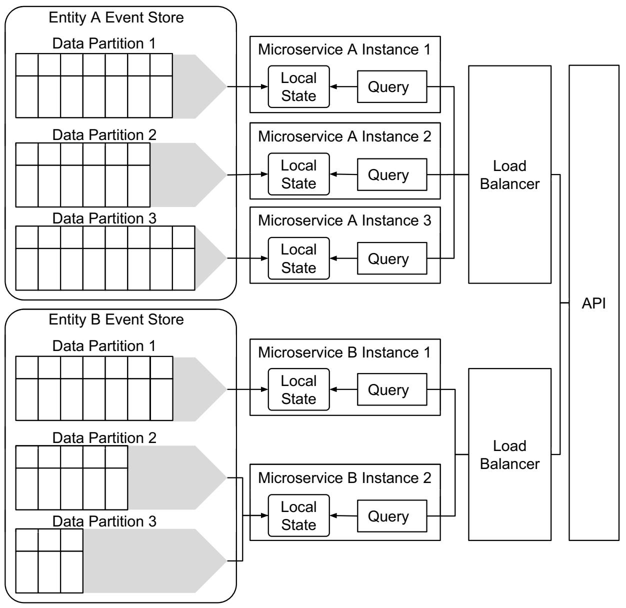

Fig. 5. Scale Cube: Replication (X), Microservices (Y) and Data Partitioning (Z)

Regarding local-state, fast reconstruction is facilitated by periodic checkpoints, which are persisted in snapshot topics. Local-state can then be reconstructed from the snapshot and only the events produced after the snapshot need to be reprocessed.

B. *Error Correction and Reprocessing*

One of the well-known difficulties of the event-sourcing pattern is the error correction, as events are immutable (they cannot be modified). However, this difficulty also encloses one of the most powerful features of event-sourcing. As data in the system is the result of processing events. If there is a bug in any processing code, as the whole event history is kept in the event store, it becomes possible to fix the processing bug and reprocess all the events that may have been affected by it. This way, correcting the code, automatically corrects all the data in the whole system history that has been affected by it.

Reprocessing however, introduces several difficulties:

- **Data Consistency:** In order to achieve a stable and robust system, the architecture must guarantee some determinism in event processing, so that re-processing all the events provide consistent results.
- **External Interactions:** If an operation shall not be repeated at reprocessing, the gateway protocol will not execute the operation and will instead provide the same response provided by the external system during the original execution, which, according to the gateway protocol, must have been logged in the gateway transaction log.

C. Consistency

The most basic consistency offered by event driven architectures is *eventual consistency*. However, it provides weak guarantees and, without a global ordering of events, it cannot assure consistency across different event processors. Stronger consistency models are required in order to build a robust and efficient distributed system [14].

In addition to eventual consistency, there are two properties, as defined in real-time editing system [15], that need to be guaranteed by the architecture to ensure system scalability and error correction: convergence and causality preservation.

The main consistency model used in the framework is processor consistency, achieved for all contexts with an assigned processor. In this case, the architecture requires an unique independent processor to be in charge of writing data into the event store. Processor consistency establishes an ordering of all the events produced by a common processor. However, events written by different processors (i.e. microservices), may not be read consistently by themselves or other processors. For those cases, in order to enable establishing an order among events processed in different processors, every event in the system is timestamped. Timestamps however, are limited by clock skew and granularity.

The system also provides means to achieve causal consistency. Every event in the system has a unique identifier, which can be used to annotate the triggering events into any newly written event, so that a causal relationship can be easily established.

Finally, in case of events that need ordering guarantees that cannot be satisfied with any of the aforementioned mechanisms, e.g. externally triggered events (commands), the system makes use of a 2-phase commit mechanism, in which a single processor must log, in a self-owned topic, all the events it depends of, in order to commit to an specific event processing order. The resulting ordered stream can then be processed, or reprocessed, consistently.

VI. Conclusion and Outlook

This paper proposes an event-driven architecture with a data centric approach, in which every component is connected to a streaming platform and exchange data with other components following a publish/subscribe pattern.

Event-Driven Architectures, however, are known to provide weak consistency guarantees that increase architecture and coding complexity. The proposed architecture provides a middleware, which makes use of well-known patterns, such as Domain-Driven Design, to handle that complexity through a set of underlying protocols. An implementation based on Apache Kafka has been used to demonstrate the architecture and the framework.

Common issues in this kind of architectures have been evaluated for the proposed framework, i.e. scalability, error correction and consistency. Although all of them have been collected in the literature, the present work is the only one, to our knowledge, that proposes an architecture and framework that addresses all three and provides a practical implementation.

Although the architecture is conceived for an Internet of Things, there are open challenges related mainly to offering a satisfactory user experience in event-driven asynchronous platforms for the human customers, as they are used to synchronous interactions and read-after-write consistency.

References

[1] M. Gualtieri and R. Curran, "The forrester wave: Big data streaming analytics, q1 2016," *Forrestor. com, Cambridge MA*, 2016.

[2] C. Brooks, C. Jerad, H. Kim, E. Lee, M. Lohstroh, V. Nouvellet, B. Osyk, and M. Weber, "A component architecture for the internet of things," *Proceedings of the IEEE*, April 2018.

[3] "AKKA documentation," https://akka.io/docs/, 2018, accessed: 2018-05-04.

[4] R. Joshi, "Data-oriented architecture: A loosely-coupled real-time soa," *Real-Time Innovations, Inc*, 2007.

[5] Object Management Group, "Data distribution service for real time systems (DDS) v1.4," http://www.omg.org/spec/DDS/1.4, 2015.

[6] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy, *Memory consistency and event ordering in scalable shared-memory multiprocessors*. ACM, 1990, vol. 18, no. 2SI.

[7] P. Helland, "Life beyond distributed transactions: an apostate's opinion." in *CIDR*, vol. 2007, 2007, pp. 132–141.

[8] D. Betts, J. Dominguez, G. Melnik, F. Simonazzi, and M. Subramanian, "Exploring cqrs and event sourcing: A journey into high scalability, availability, and maintainability with windows azure," 2013.

[9] B. Butzin, F. Golatowski, and D. Timmermann, "Microservices approach for the internet of things," in *Emerging Technologies and Factory Automation (ETFA), 2016 IEEE 21st International Conference on*. IEEE, 2016, pp. 1–6.

[10] A. Debski, B. Szczepanik, M. Malawski, S. Spahr, and D. Muthig, "In search for a scalable & reactive architecture of a cloud application: Cqrs and event sourcing case study," *IEEE Software*, 2017.

[11] B. Stopford, "Building a microservices ecosystem with kafka streams and KSQL," https://www.confluent.io/blog/building-a-microservices-ecosystem-with-kafka-streams-and-ksql/, 2017, accessed: 2018-05-07.

[12] J. Bonér, D. Farley, R. Kuhn, and M. Thompson, "The reactive manifesto," 2014.

[13] M. L. Abbott and M. T. Fisher, *The art of scalability: Scalable web architecture, processes, and organizations for the modern enterprise*. Pearson Education, 2009.

[14] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "Conflict-free replicated data types," in *Symposium on Self-Stabilizing Systems*. Springer, 2011, pp. 386–400.

[15] C. Sun, X. Jia, Y. Zhang, Y. Yang, and D. Chen, "Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems," *ACM Transactions on Computer-Human Interaction (TOCHI)*, vol. 5, no. 1, pp. 63–108, 1998.