

See discussions, stats, and author profiles for this publication at: <https://www.researchgate.net/publication/335069774>

The Saga Pattern in a Reactive Microservices Environment

Conference Paper · January 2019

DOI: 10.5220/0007918704830490

CITATIONS

10

READS

1,711

3 authors, including:



Martin Štefanko
Red Hat

1 PUBLICATION 10 CITATIONS

[SEE PROFILE](#)




Bruno Rossi
Masaryk University

75 PUBLICATIONS 715 CITATIONS

[SEE PROFILE](#)

The Saga Pattern in a Reactive Microservices Environment

Martin Štefanko¹, Ondřej Chaloupka¹ and Bruno Rossi²^a

¹*Red Hat, Brno, Czech Republic*

²*Masaryk University, Faculty of Informatics, Brno, Czech Republic*
{mstefank, ochaloup}@redhat.com, brossi@mail.muni.cz

Keywords: Saga Pattern, Compensating Transactions, Reactive, Microservices, Distributed Systems.

Abstract: Transaction processing is a critical aspect of modern software systems. Such criticality increased over the years with the emergence of microservices, calling for appropriate management of transactions across separated application domains, ensuring the whole system can recover and operate in a possible degraded state. The Saga pattern emerged as a way to define compensating actions in the context of long-lived transactions. In this work, we discuss the relation between traditional transaction processing models and the Saga pattern targeting specifically the distributed environment of reactive microservices applications. In this context, we provide a comparison of the current state of transaction support in four Java-based enterprise application frameworks for microservices support: Axon, Eventuate Event Sourcing (ES), Eventuate Tram, and MicroProfile Long Running Actions (LRA).

1 INTRODUCTION

Transaction processing is widely recognized as a critical aspect of modern applications (Little et al., 2004). Particularly in a distributed environment, understanding and reaching transaction consistency based on the currently available technology stack represents a complex task. The microservices architectural pattern separates the application domain into a set of isolated services that collaborate together to model different business concepts (Sharma, 2017). Due to their distributed character, microservice systems are subject to issues associated with failure processing. To ensure that the system is able to function even in a degraded state, these applications are commonly designed with certain quality properties which are defined as "reactive systems properties" in the Reactive Manifesto: responsiveness, resilience, elasticity and asynchronous message passing (Bonér et al., 2018).


These characteristics extend to the application of transaction processing in a reactive environment. As transactions usually include multiple services while still providing ACID (Atomicity, Consistency, Isolation, Durability) guarantees (Haerder and Reuter, 1983), all participants must reach a shared uniform consensus on the transaction result. This consensus is achieved through the utilization of consensus protocol represented conventionally by the Two-phase Commit

protocol (2PC). However, due to their locking nature, these protocols may present difficulties to achieve the reactive systems properties (Stonebraker and Cattell, 2011). The Saga pattern (Garcia-Molina and Salem, 1987) represents an alternative approach to transaction processing applicable particularly to long living transactions – transactions that can span through several days. In a long running transaction, consensus protocols like 2PC may hold locks on resources for long periods – not acceptable in a reactive environment as it makes individual services less responsive.

The goal of this work is to detail the relation between traditional transaction processing models and the saga pattern in the distributed environment of reactive microservices. To understand the current support, we compare four Java-based application frameworks for microservices support (Axon, Eventuate Event Sourcing, Eventuate Tram, and MicroProfile Long Running Actions), in terms of implementation complexity, support for the saga pattern, and performance by using a common scenario.

2 THE SAGA PATTERN

A saga, as described in the original publication (Garcia-Molina and Salem, 1987), is a sequence of operations that can be interleaved with other operations. Each operation, which is a part of the saga, rep-

^a <https://orcid.org/0000-0002-8659-1520>

resents a unit of work that can be undone by the compensation action. The saga guarantees that either all operations complete successfully or the corresponding compensation actions are run for all executed operations to cancel the partial processing.

Operations. An operation represents a particular work segment that is a part of the saga. Each saga can be split into a sequence of operations in which each one individually can be implemented as a transaction with full ACID guarantees. When the operation completes, all results of the performed work are expected to be persisted in the durable storage. This means that the external observer may see the system in intermediate states of the saga execution, as well as that it may also introduce the system into an inconsistent state between the individual operation invocations.

The ability to commit a partial operation breaks the isolation (serializability) property as it makes the segment changes available before the saga ends. Intermediate saga states may also introduce consistency contraventions. However, the saga utilizes the eventual consistency model which guarantees that the state will become eventually consistent after the saga completes (both successfully or by compensations calls).

Compensations. Each operation in a saga needs to have an associated compensation action. The purpose of the compensating action is to semantically undo the work performed by the original operation. This is not necessarily the contradictory action that puts the system into the same state as it was before the operation began or generally the saga started.

BASE Transaction. In contrast to the traditional transaction approach, the Saga pattern relaxes the ACID requirements to achieve availability and scalability with built-in failure management. As the saga commits each operation separately, updates of the not fully committed saga are immediately visible to other parallel operations (Gray, 1981) which directly breaks the isolation property.

Sagas utilize an alternative BASE model (Helland, 2007; Helland and Campbell, 2009) which values the availability over the consistency provided by ACID – the so-called CAP theorem (Gilbert and Lynch, 2012). The specified system properties are:

- **Basically Available** – The system guarantees availability with regards to the CAP theorem.
- **Soft state** – The state may change as time progresses even without any immediate modification request due to the eventual consistency.
- **Eventual consistency** – The state of the system is allowed to be in inconsistent states, but if the system does not receive any new update requests, then it guarantees that the state will eventually get into the consistent state (Vogels, 2009).

In practice, many modern applications are not always entirely restricted to all of the ACID transaction guarantees, so the saga pattern with the BASE model is emerging as a real alternative to traditional transactional approach.

Distributed Sagas. The notion of sagas can be naturally extended into distributed environments (Garcia-Molina and Salem, 1987). The saga pattern as an architectural pattern focuses on integrity, reliability, quality and it pertains to the communication patterns between services (Sharma, 2017; Nadareishvili et al., 2016). This allows the saga definition in distributed systems to be redefined as a sequence of requests that are being placed on particular participants invocations. These requests may provide ACID guarantees, but this is not restricted and it must be ensured by individual participants. Similarly, each participant is also required to expose the idempotent compensating request handler which can semantically undo the request that is handled by this participant in the saga. Analogously to the centralized system, the distributed saga management requires a transaction log and a Saga Execution Component (SEC) which in an optimal environment needs to be distributed and durable.

As all components are now distributed, the saga management system needs to deal with a number of additional problems that are not present in the localized environment with the main problem being network and participant failures that may happen between remote invocations. However, generally approaches from the non-distributed environment still apply.

3 SAGA IMPLEMENTATIONS COMPARISON

In this section we compare four implementations of the saga pattern that can be utilized in the enterprise Java applications – Axon¹, Eventuate Event Sourcing (ES)², Eventuate Tram³ and MicroProfile Long Running Actions (LRA)⁴ with its current implementation based on the Narayana project⁵. Table 1 summarizes the main differences of the frameworks. As of the time of this writing, these four frameworks are the

¹<https://axoniq.io>

²<https://eventuate.io>

³<https://eventuate.io/abouteventuatetram.html>

⁴<https://github.com/eclipse/microprofile-lra>

⁵<https://github.com/jbosstm/narayana/tree/master/rts/lra>

Table 1: Saga implementations comparison.

Problem	Axon	Eventuate ES	Eventuate Tram	LRA
CQRS restriction	Yes	Yes	Optional	No
Asynchronous by default	Yes	Yes	No	No
Saga tracking and definition	No	No	Yes	No
Single point of failure	No	Yes	Yes	Yes
Communication restrictions	Yes	Yes	Yes	No
Distributed by default	No	Yes	Yes	Yes

only Java projects which support saga implementations to some extent.

As a part of the investigation of the each discussed framework, we created a sample application simulating the saga processing in the production environment. The main goal of this quickstart projects is to compare the base attributes of the investigated saga solution provided by these frameworks. This includes the comparison of the development model, microservices feasibility, maintainability, scalability, performance and the applicability of the reactive principles within the saga execution. All samples are microservices applications that represent a backend processing for orders with a simple REST user interface.

Common scenario. A user is able to create an order by a REST call to the dedicated endpoint of the `order-service` microservice. This endpoint expects a product information in the JavaScript object notation (JSON) format containing the product id, the commentary, and the price. For simplicity reasons, an order always consists only of a single product.

Generally, each microservice is a standalone Java application that must run in a separated Java environment. By default, all examples are accessible on the local address. Every microservice is also able to run in the Docker platform and all quickstarts can be easily set up using the Docker Compose project. Details can be found in repositories of individual projects.

Every order request (saga invocation) is asynchronous - the REST call to the order endpoint immediately returns a response. All of the following interactions are documented by messages that are logged by the individual microservices. The overall saga process can be examined in the `order-service` or in the case of LRA in the `api-gateway` modules.

The persisted saga data (orders, shipments, and

invoices) can be queried by the respective REST endpoints in individual services. For the CQRS based examples, this information is available at the `query-service` microservice, otherwise each microservice is expected to be responsible for maintaining its individual persistence solution which corresponds to the microservices pattern definition.

Order saga. The saga pattern applied in this application performs the order requests (Fig. 1). The order saga consists of three parts – the production of shipping and invoice information and if both invocations are successful, the actual order creation. If any part of the processing fails, the whole progress is expected to be undone. For instance, if the shipment is successfully created but the invoice assembly is not able to be confirmed, the persisted shipment information, as well as the order, must be canceled (also optionally notifying the user that the order cannot be created). To initiate failures scenarios in examples both `shipment-service` and `invoice-service` expect product information containing a specific product identification: `failShipment` or `failInvoice` respectively.

3.1 Axon service

Axon is a lightweight Java framework that helps developers build scalable and extensible applications by addressing these concerns directly in the core architecture. It builds upon the Command Query Responsibility Segregation (CQRS) pattern (Fowler, 2018). The individual services contain separated aggregates each processing its respective commands and producing events. Any inter-service interaction is restricted to the use of the command and event buses.

Platform. Axon service is a Java Spring Boot mi-

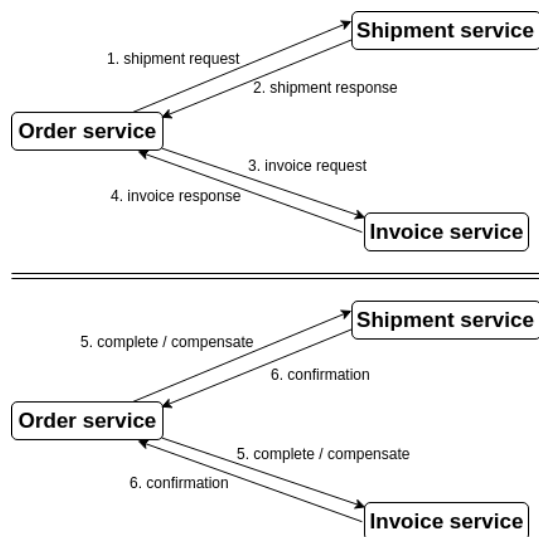


Figure 1: The saga model

crosservices application. Individual projects represent standalone runnable applications (fat java archive (jar)) which is the preferred distribution method for the Spring Boot applications.

As a CQRS based quickstart, Axon service uses two different and separated communication channels to exchange information between services: the command bus and the event bus. By default, Axon framework constraints both channels to a single JVM and therefore one microservice. However, developers are also able to specify several specific ways of the configuration to distribute messages between different services which is used in this Axon quickstart.

3.1.1 Problems

Maintenance of the saga structure. The main substantial problem of the saga processing in Axon is the missing structure of the internal lifecycle of the saga. The only operations provided by the platform are the start and the stop of the saga. The actual invocation of participants, collecting of responses and handling of the compensations is up to the developer as the only way of communication with the saga is through events.

In this application, the `OrderManagementSaga` contains two internal classes – `OrderProcessing` and `OrderCompensationProcessing` which are responsible for tracking of the saga execution and compensation respectively. This kind of the saga definition can be harder to maintain and more error prone as sagas as expected to be generally more complex processes.

AMQP usage with sagas. When the distributed event

bus is processing events from an AMQP queue which the saga class is listening to, the framework does not deliver events correctly to the attached handlers. This issue has been reported to the Axon framework and it will be fixed in the next release. The workaround that was used in the quickstart was to artificially wait 1000 milliseconds before delivering the event from the queue to the framework.

CQRS restrictions. As CQRS is a pattern that manages the domain formation of the application, Axon can place hard requirements for the projects that do not follow the CQRS domain separation. Sagas in Axon are only a specialized type of the event listener. The only way Axon produces events is through an interaction with the aggregate instance - events are produced purely as a response to the received command. Therefore, the use of Axon sagas in the non-CQRS environment may be too restrictive to the user implementation.

3.2 Eventuate service

Eventuate is a platform for developing asynchronous microservices with the main focus placed on the distributed data management. The platform consists of two products – Eventuate Event Sourcing (ES) and Eventuate Tram. Similarly to Axon, Eventuate service is also based on the event sourcing and the CQRS pattern. The business execution is managed in the aggregates which correspond to the respective microservices projects. The communication is as a result restricted to the command processing and the event application. However, conversely to Axon, the command and event buses are not distributed. The remote messaging is restricted to the REST protocol. This quickstart represents the pure CQRS approach to the saga processing. This means that the whole saga implementation is created by the developer using the platform only for the event and command distribution. For this reason, the Eventuate service is more complex than any other quickstart but for the example purposes, it distinctively demonstrates how sophisticated is the saga administration provided by all remaining platforms.

Platform. Eventuate service is a microservices application consisting of a set of Spring Boot business services, one backing module and a number of support services provided by the Eventuate platform.

This quickstart is established as an Eventuate Local version of the platform. This means that it uses underlying SQL database for the event persistence and the Kafka streaming platform for the event distribution. Eventuate Local provides five services used by the quickstart that are managed by the platform,

namely, Apache Zookeeper, Apache Kafka, MySQL database, the change data capture component (CDC) and the Eventuate console service. The example employs these services as Docker images included in the provided docker-compose configuration.

3.2.1 Problems

Complexity. As this project represents a plain CQRS based example, it demonstrates the background process required for the saga execution. Therefore, the complexity of this quickstart may appear more critical than in other projects as the background saga execution often contains many optimizations. The main complexity problem is that the project contains a high number of command and event classes. This is required as aggregate classes are only able to consume commands and produce events. For this reason, the communication between components often demands additional steps.

Command bus distribution restrictions. This quickstart uses the REST architectural style for the remote distribution of commands between services. Even if all of the microservices are connected to the same MySQL database, they cannot directly propagate commands between each other. This is due to the way Eventuate dispatches commands through the aggregate repository. The aggregate repository represents the database table that is restricted to one aggregate and it is provided by the platform through the dependency injection. For this reason, it must declare the target aggregate class and the command type. The sharing of the aggregate class type may be very restrictive, especially for microservices applications.

Aggregate instantiation. The Eventuate framework creates the instances of aggregate classes by a call to the default constructor. This effectively prohibits the use of aggregate instance managed by the underlying server container. For this reason, each aggregate in this project is separated into two classes – the actual aggregate responsible for the command processing and the event subscriber instance managing incoming events. This restriction is seconded by the rule stating that each produced event from the aggregate's command processing method must also be applied by the different method of the same aggregate. This limitation exists because of the event sourcing feature providing the ability to replay already executed commands to reconstruct the aggregate's state in the case of failure. The aggregate then may contain unnecessary empty methods as the saga also requires the propagation of the information to different components (e.g., the REST controller).

Event entity specification. As well as the command type, Eventuate also requires the definition of

the event type each aggregate is able to apply. The problem rises when the events need to be shared between several modules. This is a common requirement as the CQRS pattern requires the query domain to be separated. The event interfaces are therefore included in the shared library module same as the `service-model` used in this project. The hard coded information of the full name of the aggregate class used in the `@EventEntity` annotation then may become hard to maintain.

Platform structure. The platform structure places the obligation on each developed microservice to conduct with the connecting requirements. This means that every service must provide linking information for the Eventuate platform services described in the previous section, namely, MySQL database, Apache Kafka, Apache Zookeeper and CDC component. This information is replicated in all individual services and therefore predisposed to errors.

3.3 Eventuate Tram service

The Eventuate Tram service quickstart is based on the recently introduced Eventuate Tram framework. The Eventuate Tram framework extends the platform with asynchronous messages that can be sent as a part of a database transaction. It utilizes the traditional Java Database Connectivity (JDBC) and Java Persistence API (JPA) to provide the transactional messaging. This enables the microservice to atomically update its state and to publish this information as a message or as an event to other services.

Platform. Similarly to the Eventuate ES distribution, the Eventuate Tram establishes four services that form the Tram platform: Apache Zookeeper, Apache Kafka, MySQL database and CDC component. The fifth service, Console server, is not included as the platform does not present this functionality by the time of this writing. All services are deployed by the `docker-compose` configuration distributed with the framework.

The most important element in this application is the saga definition that is located in the `OrderSaga` class of the `order-service`. This definition uses the declarative approach with the fluent API to denote the saga in steps of execution. Every step declares a handler method representing a participant request that will be invoked when this step is reached by a reference to the private method in this class as well as the reference to its compensation handler.

3.3.1 Problems

Destination identification. The destination channels in Tram are characterized by a simple string which

may cause problems in the case of name conflicts. The choice of the handler to be invoked when message is received depends on two values – the name of the channel and the command dispatcher id. When both strings match the same destination even in different services, the platform delivers the commands between handlers in the random fashion which may become a complex issue in larger projects.

Command handlers. Handler methods that are referenced in the definition are restricted to the communication model provided by the platform. This allows a single command to be sent to the required destination. Unfortunately, the platform does not allow the saga to perform any other operation without the participant invocation which may lead to unnecessary empty commands and channels declarations.

Similarly, the saga may need to send multiple different commands to the same participant. This may cause problems with definitions of compensation and reply handlers as the developer needs to mind the logical grouping of participant invocations.

3.4 LRA service

MicroProfile Long Running Actions (LRA) is a specification developed in collaboration with the Eclipse MicroProfile initiative (Štefanko, 2017). There are several companies currently participating and contributing to it (e.g., Red Hat or Payara). It proposes a new API for the coordination of long running activities with the assurance of the globally consistent outcome and without any locking mechanisms. The LRA specification is based on the Context and Dependency Injection (CDI) and Java API for RESTful Web services (JAX-RS) Java EE specifications. The communication is handled over HyperText Transfer Protocol (HTTP) and the Representational State Transfer (REST) architectural style.

Platform. This quickstart is composed as a set of WildFly Swarm microservices. Every service is designed to be deployed to the OpenShift container application platform provided by Red Hat, Inc.

Narayana's implementation (<http://narayana.io>) of the Long Running Actions specification is not composed as a platform, but rather as a standalone coordination service. This project employs the standalone Narayana LRA coordinator which is constructed as a WildFly Swarm microservice called `lra-coordinator`. All other traditional microservices requirements are handled by the OpenShift platform. This covers service discovery and location transparency, monitoring, logging, resiliency and health checks (failure discovery) which is why this configuration is not included in example services.

3.4.1 Problems

REST restriction. The Narayana Long running actions are a very efficient development model for the microservices applications. Although it mainly aims for the compatibility with the MicroProfile specification (REST and CDI), it does not restrict microservices to essentially any other particular implementation restrictions. Even if the MicroProfile is restricted to REST invocations, Narayana LRA specification does not require the usage this architectural style for the communication with the coordinator. However, the only implementation that is currently available is based on REST, but it certainly can be extended to other communication protocols in the future.

LRA execution. The current implementation is that the LRA framework provides only coordination and management capabilities, it does not handle the saga structuring and execution. Extraction of these capabilities directly to the LRA processing would be certainly applicable in many common specifically reactive applications use cases which can ease the development and orchestration of the saga execution.

Single point of failure. By the time of this writing, the LRA coordinator represents a single point of failure for the LRA processing in Narayana because it contains the object store that is used for storing the LRA information. User services also need to be adjusted for the situations when the coordinator is not available.

3.5 Performance test

To compare examples for their applicability in real systems from the performance perspective, a simple performance test has been created to investigate how they behave under large load. Tables 2 and 3 present a summary of the performance test executions. The processing delay defines the time between the last sent request and the last processed order and the total time denotes time from the first request to the last completed order. The format for both times is `mm:ss`. Results presented in Table 3 represent the best result of three performed test executions in scenario 2. The test has been run in the cloud computing platform Digital Ocean (www.digitalocean.com). The virtual machine specification:

Fedora 27 x64, Kernel: 4.13.9-300.fc27.x86_64
6 vCPUs (Intel(R) Xeon(R), E5-2650 v4 2.20GHz)
RAM: 16 GB, SSD: 320 GB, OpenJDK 1.8.0_144,
Maven 3.5.0, Gradle 2.13

The performance test is using the PerfCake (<http://perfcake.org>) testing framework to generate requested number of order requests. As all

Table 2: Performance test, time in mm:ss – scenario 1 (1 000 requests, 10 threads)

Project	Processing delay	Total time	Completed requests
Axon service	00:46	00:51	1 000
Eventuate service	00:49	00:58	2 000
Eventuate Tram service	00:27	00:34	1 000
LRA service	00:04	01:10	1 000

Table 3: Performance test, time in mm:ss – scenario 2 (10 000 requests, 100 threads)

Project	Processing delay	Total time	Completed requests
Axon service	06:53	07:44	5 657
Eventuate service	14:05	14:46	19 791
Eventuate Tram service	03:20	03:56	10 000
LRA service	00:22	08:58	10 000

investigated frameworks perform the saga execution asynchronously, the test first requests the number of orders and then performs the get orders call to the respective service in periodic intervals. The test ends when all orders are processed or the defined timeout is reached (Fig. 2).

Every example has been tested in two modes – 1000 order requests with 10 threads (scenario 1) and 10 000 order requests with 100 threads (scenario 2). The reason was that some of the frameworks are not able to handle the second test because of various problems discussed in the following sections. Scenario 2 has been run three times for each individual example, and the presented results are taken from the best execution. Each test execution has been run in the same setup on a new virtual machine.

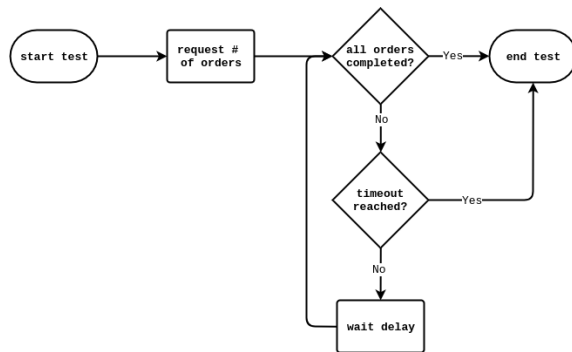


Figure 2: Saga performance test execution

Axon service. The major performance problem of this example was the artificial one second delay placed on the shipment and invoice response retrieval in the `order-service`. This problem has been reported to the Axon framework and this quickstart has been provided with a fixed configuration that removed the need for the timeout. This setting will be included by default in the Axon next release.

Another problem were synchronous REST calls for inter-service command dispatching over distributed command bus. This issue has been reported and was still investigated by the time of this writing.

The performance test in the scenario 1 has been executed successfully in 51 seconds. The scenario 2 met the limits of the platform when the database lock for the first command could not be taken after about 5000 requests. This scenario was run three times with the best result of 5657 completed orders in 7 minutes and 44 seconds.

Eventuate service. This performance test discovered a new issue in the Eventuate Local platform – when order requests are created in fast succession, the database update may fail which results in redelivery of the same events and therefore several orders for one request. This problem has been reported to the Eventuate project.

In the scenario 1, the 1000 requests have been created in 58 seconds, but because of the mentioned issue the final count reached 2 000 orders. Scenario 2 was executed three times where each run produced different amount of completed orders. The presented test

execution finished after 14 minutes and 46 seconds with 19 791 completed orders.

Eventuate Tram service. The Eventuate Tram project performed well in the scenario 1 which has been finished in 34 seconds with all completed orders. However, scenario 2 in some cases produced an exception in the Kafka service that timed out on the `session.timeout.ms` – a timeout that is used to detect consumer failures. The default value for this property is hardcoded in the Eventuate code base and cannot be customized. This issue has been reported to the platform as the feature request. The representing successful run has been completed in 3 min. 56 sec.

LRA service. This quickstart was for the performance test adjusted to be run directly in the Docker platform to simulate a similar environment for all examples. The test presented that the LRA coordinator orchestration does not influence the processing performance. The scenario 1 has been finished with exceptional overhead only 4 seconds and total time 1 minute and 10 seconds while the scenario 2 finished in 8 minutes and 58 seconds with just 22 seconds spent on the additional saga processing. Both scenarios successfully completed all order requests.

Summary. In general, both Eventuate Tram and LRA performed in this test better than Axon or Eventuate ES because the main focus of these frameworks is placed on saga development. However, Axon and Eventuate ES still present a simplified integration of the saga pattern in CQRS based applications.

4 CONCLUSIONS

This paper covered transactions concepts that can be utilized for the transaction commit problem in modern software architectures. It covered conventional transactional approaches utilizing two-phase commit protocol and application to the distributed environment of microservices. The saga pattern (Garcia-Molina and Salem, 1987) has been introduced as an alternative approach to traditional transaction processing.

Development support of the saga pattern for production environments was explored through the investigation of saga processing in four Java frameworks – Axon, Eventuate ES, Eventuate Tram and Narayana LRA. All of the frameworks were examined in terms of the implementation of an order processing microservices application utilizing the saga execution. Both Axon and Eventuate ES provide simplified definitions of the saga pattern, however, at the expense of the manual saga execution tracking and the mandatory CQRS pattern application. Conversely, Eventuate Tram and LRA are frameworks specifically

designed for saga executions. Both frameworks provide easy integrations and transparent executions of sagas in enterprise Java applications. These quickstart applications were also compared from a performance perspective through a created test that examined saga processing under large applied load. These performance experiments resulted into several suggestions of possible improvement points in the saga processing of individual frameworks. In conclusion, the saga pattern (Garcia-Molina and Salem, 1987) provides a sophisticated alternative to conventional transaction processing by means of its non-blocking nature – useful in modern microservices applications.

Acknowledgements. The research was supported from ERDF/ESF "CyberSecurity, CyberCrime and Critical Information Infrastructures Center of Excellence" (No. CZ.02.1.01/0.0/0.0/16_019/0000822).

REFERENCES

- Bonér, J., Farley, D., Kuhn, R., and Thompson, M. (2018). Reactive manifesto. <https://www.reactivemanifesto.org>.
- Fowler, M. (2018). Cqrs. <https://martinfowler.com/bliki/CQRS.html>.
- Garcia-Molina, H. and Salem, K. (1987). Sagas. *ACM SIGMOD Record*, 16(3):249–259.
- Gilbert, S. and Lynch, N. (2012). Perspectives on the CAP theorem. *Computer*, 45(2):30–36.
- Gray, J. (1981). The transaction concept: Virtues and limitations. In *Proceedings of the Seventh Int. Conference on Very Large Data Bases - Volume 7*, VLDB '81, pages 144–154. VLDB Endowment.
- Haerder, T. and Reuter, A. (1983). Principles of transaction-oriented database recovery. *ACM Computing Surveys*, 15(4):287–317.
- Helland, P. (2007). Life beyond distributed transactions: an apostate's opinion. In *CIDR*, pages 132–141.
- Helland, P. and Campbell, D. (2009). Building on quicksand. *CoRR*, abs/0909.1788.
- Little, M., Maron, J., and Pavlik, G. (2004). *Java transaction processing*. Prentice Hall.
- Nadareishvili, I., Mitra, R., McLarty, M., and Amundsen, M. (2016). *Microservice architecture: aligning principles, practices, and culture*. "O'Reilly Media, Inc."
- Sharma, U. R. (2017). *Practical Microservices*. Packt Publishing Ltd., 1 edition.
- Stonebraker, M. and Cattell, R. (2011). 10 rules for scalable performance in 'simple operation' datastores. *Commun. ACM*, 54(6):72–80.
- Vogels, W. (2009). Eventually consistent. *Communications of the ACM*, 52(1):40.
- Štefanko, M. (2017). Saga implementations comparison. <http://jbossts.blogspot.cz/2017/12/saga-implementations-comparison.html>.