Masaryk University
Faculty of Informatics

Use of Transactions within a Reactive Microservices Environment

Master's Thesis

Bc. Martin Štefanko

Brno, Spring 2018

Masaryk University
Faculty of Informatics

Use of Transactions within a Reactive Microservices Environment

Master's Thesis

Bc. Martin Štefanko

Brno, Spring 2018

Replace this page with a copy of the official signed thesis assignment and the copy of the Statement of an Author.

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Bc. Martin Štefanko

Advisor: Bruno Rossi, PhD

Acknowledgement

I would like to express my sincere thanks to my supervisor Bruno Rossi, PhD for his help and support in my work on this thesis. I would also like to thank Mgr. Ondrej Chaloupka from the Narayana team for his supervision, neverending passion and interest in this topic that was inspiring to complete many of achieved results. Last but foremost, I would like to direct my thanks to my family and my beloved Magdalena who comprehensively supported me throughout the formation of this publication.

Abstract

Transaction processing is an inherent part of application development. However, particularly in the distributed environment, the utilization of transactions introduces several challenges that transaction management must be able to handle. Microservices represent an emerging architectural style for the modern distributed application design. These applications commonly address similar concerns to stay responsive, elastic and resilient which is why the utilization of traditional locking transaction commit protocols may not be acceptable. The saga pattern presents a suitable alternative solution to transaction processing that relaxes some of the ACID properties in order to promote availability. In this work, we examine how sagas differ from conventional transactions, investigate currently available saga solutions and propose a saga execution implementation for the Narayana transaction manager.

Keywords

transactions, Narayana, Long Running Actions (LRA), reactive, microservices, asynchronous, saga, compensating transactions

Contents

1 **Introduction** . . . 1
- 1.1 *Problem domain* . . . 1
- 1.2 *Research objectives* . . . 2
- 1.3 *Research contributions* . . . 3
- 1.4 *Structure* . . . 3

2 **Transaction concepts** . . . 5
- 2.1 *Transaction* . . . 5
- 2.2 *ACID properties* . . . 5
  - 2.2.1 Atomicity . . . 6
  - 2.2.2 Consistency . . . 6
  - 2.2.3 Isolation . . . 7
  - 2.2.4 Durability . . . 10
- 2.3 *Transaction manager* . . . 11
  - 2.3.1 TM types . . . 11
  - 2.3.2 JTA and JTS . . . 11
  - 2.3.3 XA specification . . . 12
- 2.4 *Transaction models* . . . 12
  - 2.4.1 Local transaction model . . . 13
  - 2.4.2 Programmatic transaction model . . . 13
  - 2.4.3 Declarative transaction model . . . 14
- 2.5 *Distributed transactions* . . . 16
- 2.6 *Consensus protocols* . . . 17
  - 2.6.1 2PC . . . 18
  - 2.6.2 3PC . . . 18
  - 2.6.3 Paxos . . . 20
  - 2.6.4 Conclusions . . . 21

3 **Microservices architecture pattern** . . . 23
- 3.1 *Architectural pattern* . . . 23
  - 3.1.1 Monolithic architecture . . . 23
  - 3.1.2 Microservices architecture . . . 24
- 3.2 *Principles of microservices* . . . 25
- 3.3 *Reactive microservices* . . . 29
  - 3.3.1 Reactive systems . . . 29
  - 3.3.2 Reactive programming . . . 31
  - 3.3.3 Reactive streams . . . 32

3.3.4 Summary . . . 33
3.4 *Challenges* . . . 34
3.4.1 Distributed systems . . . 34
3.4.2 Eventual consistency . . . 35
3.4.3 CAP theorem . . . 36
3.4.4 Operations . . . 37
3.4.5 Human factor . . . 38
4 **Saga pattern** . . . 40
4.1 *Operations* . . . 41
4.2 *Compensations* . . . 41
4.3 *BASE transaction* . . . 42
4.4 *Saga execution component and transaction log* . . . 43
4.5 *Recovery modes* . . . 44
4.6 *Distributed sagas* . . . 45
4.7 *Current development support* . . . 47
4.7.1 Axon framework . . . 47
4.7.2 Eventuate ES . . . 49
4.7.3 Eventuate Tram . . . 50
4.7.4 Narayana LRA . . . 51
4.7.5 Summary . . . 53
5 **Saga implementations comparison example** . . . 55
5.1 *Common scenario* . . . 55
5.2 *Order saga* . . . 56
5.3 *Axon service* . . . 58
5.3.1 Platform . . . 58
5.3.2 Project structure . . . 60
5.3.3 Problems . . . 61
5.4 *Eventuate service* . . . 62
5.4.1 Platform . . . 62
5.4.2 Project structure . . . 64
5.4.3 Problems . . . 65
5.5 *Eventuate Tram service* . . . 68
5.5.1 Platform . . . 68
5.5.2 Project structure . . . 68
5.5.3 Problems . . . 70
5.6 *LRA service* . . . 71
5.6.1 Platform . . . 71
5.6.2 Project structure . . . 71

5.6.3 Problems . . . 74
5.7 *Used technologies* . . . 75
5.7.1 Microservices platforms . . . 75
5.7.2 Docker . . . 77
5.7.3 Containerization platforms . . . 78
5.8 *Performance test* . . . 80
6 **LRA executor extension** . . . 85
6.1 *Motivation* . . . 85
6.2 *Design* . . . 86
6.3 *Implementation* . . . 88
6.3.1 LRA definitions . . . 88
6.3.2 LRA executor . . . 89
6.4 *Narayana LRA integration* . . . 90
6.5 *LRA executor quickstart* . . . 91
6.6 *Future work* . . . 92
7 **Conclusion** . . . 94
7.1 *Saga pattern research* . . . 94
7.2 *Narayana asynchronous LRA execution* . . . 95
7.3 *Contributions* . . . 96
7.4 *Future tasks* . . . 97
Bibliography . . . 98
A **CQRS pattern** . . . 107
B **Reactive Streams v1.0.2 API** . . . 109
C **Saga scenarios** . . . 110
C.1 *CQRS* . . . 110
C.2 *Eventuate Tram service* . . . 111
C.3 *LRA service* . . . 112
D **Example applications public APIs** . . . 113
D.1 *Axon service* . . . 113
D.2 *Eventuate service* . . . 113
D.3 *Eventuate Tram service* . . . 114
D.4 *LRA service* . . . 114
E **LRA executor extension class diagrams** . . . 116
E.1 *LRA definitions* . . . 116
E.2 *LRA executor* . . . 117
E.3 *Narayana integration* . . . 118
E.3.1 LRA REST definitions . . . 118
E.3.2 LRA coordinator . . . 119

List of Tables

2.1 Consensus protocols comparison *22*

4.1 Saga implementations comparison *54*

5.1 Performance test – scenario 1 (1 000 requests, 10 threads) *84*

5.2 Performance test – scenario 2 (10 000 requests, 100 threads) *84*

List of Figures

2.1 Dirty write (adapted from [17]) *8*
2.2 Non repeatable read (adapted from [17]) *9*

4.1 Example saga execution *40*
4.2 Distributed saga example [72] *47*

5.1 Product Information example JSON *56*
5.2 The saga model *57*
5.3 LRA definition example JSON *72*
5.4 Action example JSON *73*
5.5 Saga performance test execution *82*

6.1 Narayana LRA executor extension component diagram *88*

C.1 CQRS saga example success *110*
C.2 CQRS saga example invoice failure *110*
C.3 Eventuate Tram service saga example success *111*
C.4 Eventuate Tram service saga example invoice failure *111*
C.5 LRA service saga example success *112*
C.6 LRA service saga example invoice failure *112*

E.1 LRA definitions class diagram *116*
E.2 LRA executor class diagram *117*
E.3 LRA REST definitions class diagram *118*
E.4 LRA coordinator required integration changes *119*

1 Introduction

Transaction processing is widely recognized as a critical technology for modern applications [1]. The capability to group a sequence of operations into a logical unit of work represents a common business concept that relies on certain guarantees of correctness. Particularly in the distributed environment, the comprehension and the achievement of these requirements, based on the currently available technology stack, represents a considerably complex programming task.

1.1 Problem domain

Microservices architectural pattern defines a sophisticated development technique which separates the application domain into a set of isolated services that collaborate together to model various business concepts [2]. Each service depicts an autonomous self-maintained unit that is decoupled from other services to promote independent lifecycle management, deployment, and failure isolation. In order to cooperate their activities, microservices must employ a non-blocking remote invocation model that is frequently utilizing a form of asynchronous communication mechanisms.

Due to their distributed character, microservices applications are often inclined to many issues associated with the failure processing and the isolated development model. To ensure that the system is able to function even in a degraded state, these applications commonly provide a design that guarantees certain quality properties which are defined in the Reactive Manifesto [3] as the reactive systems. The four recognized attributes of reactive microservices systems are responsiveness, resilience, elasticity and the asynchronous message passing.

These characteristics naturally extend to the application of the transaction processing in a reactive environment. As transactions may presumably span multiple services while still providing ACID (Atomicity, Consistency, Isolation, Durability) guarantees, it is required that all impacted participants reach a shared uniform consensus on the transaction result. This consensus is achieved through the utilization of consensus protocols represented conventionally by the Two-phase

commit protocol (2PC). However, due to their commonly locking nature, these protocols may present difficulties with the achievement of properties reactive systems need to provide.

The saga pattern [4] represents an alternative approach to transaction processing applicable for long living transactions. In a long running transaction, traditional consensus protocols may hold locks on resources for long time periods which is not acceptable in a reactive environment. The saga addresses this problem by allowing participants to commit their intermediate states as local operations. Each operation is required to define a compensation action that can semantically undo the performed operation. The pattern guarantees that either all operations are completed successfully, or the compensation actions are executed for each performed operation to amend the partial processing.

1.2 Research objectives

There are three main objectives in this work:

1. The investigation of asynchronous approaches available for transaction processing in the microservices context.

2. The proposal of a proof of concept implementation utilizing the Narayana transaction manager [5] to create a service providing transaction management in reactive microservices systems.

3. The preparation of a quickstart example presenting practical considerations of the applied solution for the asynchronous execution.

The research investigation is focused on the saga pattern [4] application in reactive architectures by means of the available solutions based on the Java platform. The four studied frameworks are Axon [6], Eventuate Event Sourcing (ES) platform [7], Eventuate Tram [8] and Narayana Long Running Actions (LRA) [9].

1.3 Research contributions

As a part of the saga implementations research, we created an example quickstart project that simulates an order processing application for each investigated framework. Additionally, a performance test has been created to examine how these frameworks perform under large load. This test discovered several issues present in investigated saga solutions that have been reported to the respective platforms.

The proposed proof of concept implementation of the asynchronous LRA processing in Narayana transaction manager is called the *LRA executor extension*. This project is built on top of the current Narayana LRA coordination management to allow the asynchronous LRA execution based on the user definition. Together with this extension, an example quickstart project the *LRA executor quickstart* has been created to demonstrate asynchronous capabilities of this solution.

The research and the development conducted within this thesis have been performed in collaboration with the Narayana open source transaction manager [5] development team. All originated source code associated with this thesis is available under the open source GNU LGPL 2.1 license [10].

1.4 Structure

This thesis can be divided into three logical segments. The first part consists of the problem introduction and the motivation for this work included in the first chapter and the introduction of the basic transaction concepts and their application in the Java environment in the second chapter. Furthermore, it presents the microservices architectural style with the emphasis placed on reactive applications in the third chapter.

The second segment provides the detailed description of the saga pattern [4]. The fourth chapter describes what the saga is, how it relates to the traditional transaction processing and it presents an overview of four investigated frameworks that provide saga execution capabilities. The fifth chapter covers the detailed description of the saga implementation in each example and the study of the performance test execution performed on researched saga solutions.

The last part represents the description of the proposed solution of the asynchronous saga execution in the Narayana LRA project. The sixth chapter provides the motivation, design, and implementation of the *LRA executor extension* project together with its integration requirements and the description of the possible future inclusion of this project in the Narayana code base. The last chapter provides a summary of the performed work and concludes with the achieved results.

2 Transaction concepts

This chapter introduces the basic notions of transactions, their properties and common problems with their management both in centralized systems (e.g., databases) and in distributed systems where transactions must be coordinated across multiple service nodes connected by a computer network.

2.1 Transaction

A transaction is a unit of processing that provides all-or-nothing property to the work that is conducted within its scope, also ensuring that shared resources are protected from multiple users [1]. It represents a unified and inseparable sequence of operations that are either all performed or none of them take effect.

The transaction can end in two forms: it can be either *committed* or *aborted*. The commit determines a successful outcome - all operations within the transaction have been executed. The abort means that all performed operations have been undone and the system is in the same state as if the transaction has not been started.

From the developer point of view, it is frequently only required to start and end the transaction. All complex processing necessary for the achievement of the transaction's properties is commonly hidden by the transaction system [1] which allows developers to focus on the business processing contained in the transaction.

Generally, the achievement of above mentioned features may differ depending on the scope and the utilization of the transaction concepts in the application.

2.2 ACID properties

A transaction can be viewed as a group of business logic statements with certain shared properties [11]. Generally considered properties are one or more of atomicity, consistency, isolation, and durability.

These four properties are often referenced as ACID properties [12] and they describe the major points important for the transaction concepts.[1]

2.2.1 Atomicity

The transaction consists of a sequence of operations performed on different resources. The atomicity property means that all operations in the transaction are performed as if they were a single unit.

As the word atomicity is an overloaded term in many computational science branches, some authors prefer to reference it in the ACID context as the abortability property. The abortability is defined as the ability to abort a transaction on error and have all writes from that transaction discarded [14]. This implies that when the transaction commits successfully, all of its operations are also required to execute a valid commit. Conversely, when the transaction fails and needs to be aborted, all realized operations and effects must be undone.

2.2.2 Consistency

The word consistency refers to restrictions placed on data changes that may happen only in allowed ways. When the data is persisted, it must be valid according to all defined rules which meet the application invariants. The consistency property describes that the transaction maintains the consistency of the system and resources that it is being performed on. When the transaction is started on the consistent system, this system must remain consistent when the transaction ends – it moves from one consistent state to another.

Unlike other transactional properties (A, I, D), consistency cannot be realized by the transaction system as it does not hold any semantic knowledge about the resources it manipulates [1]. Therefore, the achievement of this property is the responsibility of the application code.

---

1. Although the ACID acronym has been associated with transactions since their beginning, Eric Brewer, the inventor of the CAP theorem (section 3.4.3), stated in his article from 2012 that it is "more mnemonic than precise"[13].

2.2.3 Isolation

The isolation property takes effect when multiple transactions can be executed concurrently on the same resources. It provides a guarantee that parallel transactions cannot interfere one with another. Therefore, each concurrent execution on the shared resource must be equivalent to some serial ordering of contained transactions. This is why the isolation is often also referred to as a serializability.

From the perspective of an external view, the isolation property means that the transaction appears as it was executed in the system entirely alone. This means that even if there are multiple transactions performed concurrently, this fact is hidden from the external perspective.

As an instinctive extension of the consistency property, the serial execution of transactions preserves the consistent state. The execution of the transactions in parallel, therefore, cannot result in an inconsistent system.

Isolation levels

The ANSI SQL-92 [15] standard distinguishes several levels that describe to which extent the isolation guarantees are provided. Levels are differentiated by simplifications of the locking mechanism in exchange for the faster processing. These levels, in decreasing order, are serializable, repeatable read, read committed and read uncommitted isolation. The standard defines the distinction between levels by the type of anomalies they are able to prevent.

In practice, this approach is not always considered as sufficient due to its locking expectations [16]. One of the well known alternative techniques which is not based on locking mechanisms is called the Multi Version Concurrency Control (MVCC). This section describes isolation levels based on the SQL-92 standard, but the MVCC is referenced where applicable to present the similarities.

Read uncommitted

The read uncommitted is the lowest level in which the transaction sets the lock only when it needs to modify the data item and releases it immediately after the transaction is completed (long write-locks). This allows that one transaction may read the not yet committed changes of any other transaction which is phenomena referred to as the *dirty read*.

However, the read uncommitted isolation level already prevents the *dirty write* conflict which represents the transaction that overwrites a value previously written by another transaction that has not yet been committed. This problem can be easily described on an example depicted in figure 2.1. In this execution, the resulting data state contains different values for variables (x = B and y = A). However, in any serial execution of these transactions the result would be consistent. The read uncommitted level prevents this phenomenon as it prohibits write access to data items before the former write locks are released.

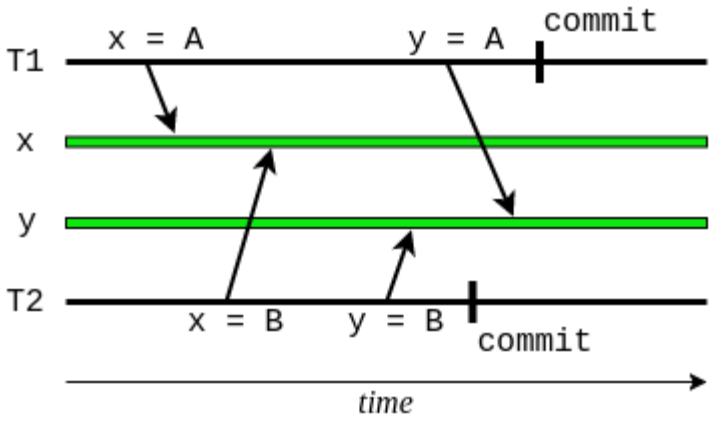

Figure 2.1: Dirty write (adapted from [17])

Read committed

Read committed level differs from the previous level in that it prohibits *dirty reads*. It allows to read the data item only when it has been committed. This directly implies that the *dirty read* phenomenon is impossible to occur in this isolation level by definition.

Repeatable read

This isolation level prevents the anomaly called the *non-repeatable read* which may still appear in read committed. The *non-repeatable read* is a problem of reading the data values in different points of time in which the whole consistent data state may not be ensured. This means that the transaction may read the data change at the later point in time which may invalidate already read information from previous processing.

For example, imagine a bank system with two accounts – both with starting balance 500 and a transfer of 100 from account 1 to 2 as a transaction. If another transaction reads the balance of the account 2 prior to the start of the transaction, it will get 500. However, the subsequent read from the account 1, after the first transaction has committed, would output 400 which results in inconsistent information.

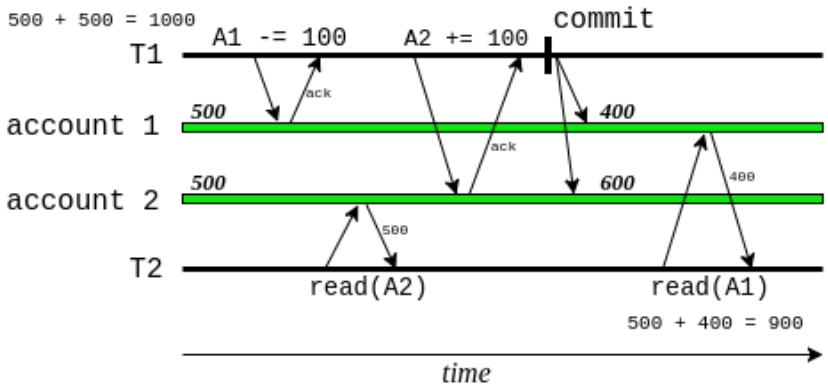

Figure 2.2: Non repeatable read (adapted from [17])

The MVCC alternative to the repeatable read is the snapshot isolation. Both the repeatable read and the snapshot isolation prevent *non-repeatable reads*, but they differ in implementation [17]. Repeatable read is based on the locking mechanisms. However, the snapshot isolation allows each transaction to read data from a snapshot of the (committed) data which is created at the time the transaction is started [16]. The database is required to internally keep track of several states and to provide each transaction with the snapshot that is appropriate for its time span.

Serializable

On top of the repeatable read, the serializable isolation prevents the system from one more race condition known as the *write skew*. The fundamental principle of the *write skew* is a transaction that reads data from the storage and then makes a decision based on this information. Different transactions may write these decision results into different parts of the database and therefore they cannot conflict. The problem is that by the time the transaction commits, the premise of the decision may no longer be valid and the resulting state may break the consistency of the system.

2.2.4 Durability

This property characterizes that all changes done by the transactions must be persistent, i. e. any state changes performed during the successfully committed transaction must be preserved in case of any subsequent system failure. How the state is preserved usually depends on the particular implementation of the transaction system. Generally, to achieve this property the use of the persistent storage like a disk drive or a cloud is sufficient. Even if this kind of storage is acceptable, it still cannot prevent data loss in the case of more critical catastrophic failures.

2.3 Transaction manager

A transaction manager (TM) is a component responsible for transaction processing, coordination and the sequential or parallel execution of transactions across one or more resources. It ensures the proper and valid completion of each transaction. It is also accountable for making the final decision whether to commit or rollback the transaction. Clients often communicate with the transaction manager only when they need to start or end the transaction.

Main responsibilities of the transaction manager are starting and ending (commit or abort) of transactions, the supervision of transactions scoped across multiple resources and rollback capabilities ensuring the failure recovery.

2.3.1 TM types

A local transaction manager or a resource manager is responsible for the coordination of transactions concerning only a single resource. Because of the range of its scope, it is often built in directly into the resource. The span of the resource is defined by the managing platform, e. g., the JMS context or the JDBC database connection.

The scope of the local TM does not intend its application across multiple resources. This means that it cannot provide ACID guarantees if the transaction contains, for instance, both the database update and the JMS message send as these resources are handled by different resource managers.

The management of transactions over multiple resources is supported by a global transaction manager. It represents an operation external component that is able to coordinate several resource managers in order to provide ACID transactions spanning two or more different transactional resources.

2.3.2 JTA and JTS

The definition and the management of transactions with a guarantee of all ACID properties are in the Java environment represented by two Java specifications: the Java Transaction API (JTA) and the Java Transaction Service (JTS).

The JTA is a specification which defines high-level interfaces between a transaction manager and the parties involved in a distributed transaction system: the application, the resource manager and the application server [18]. It also determines the Java mapping of the industry standard X/Open XA protocol which allows local resource managers to participate in the global transaction managed by an external transaction manager.

The JTS represents a specification of the transaction manager implementation which supports the JTA specification at the high-level and implements the Java mapping of the Object Management Group (OMG) Object Transaction Service (OTS) specification at the low-level [19]. It utilizes the Common Object Request Broker Architecture (CORBA) OTS interfaces for the interoperability and portability of the transaction context between different JTS transaction managers over the Internet InterORB Protocol (IIOP).

2.3.3 XA specification

The eXtended Architecture (XA) standard is the X/Open Common Applications Environment (CAE) specification published in 1991 which describes the bidirectional interface between a transaction manager and a resource manager [20]. It maintains two types of components that clients can interact with: the transaction manager (TM) which defines the global TM in a sense described in the previous section and the XA resources that represent local resource managers.

The resource manager implements the XA interface in order to provide a switch that effectively delegates the transaction control to the TM. The XA TM coordination is based on the two-phase commit protocol (2PC) which means that the XA interface contains all necessary function calls that needs to available for 2PC – `xa_prepare`, `xa_commit` and `xa_rollback`. The Java mapping of the XA standard is present in the class `javax.transaction.xa.XAResource`.

2.4 Transaction models

The transaction model defines rules and semantics of how developers declare and work with transactions. From the development point of

view, there exist three distinct transaction models that may be applied in the Java environment – local, programmatic and declarative transactions. This section describes each model respectively and examines how it can be established in Java applications based on the Enterprise JavaBeans (EJB) technologies.

2.4.1 Local transaction model

The local transaction model derives its name from the fact that transactions are managed by a local resource manager which was described in the previous section. This approach represents the transaction as a connection to the individual resource. Common use-cases for this model include the Java database connectivity (JDBC) or the Java Message Service (JMS) connection providers.

The connection is usually by default configured to commit the local transaction after each operation, e.g., a database query or sending a message to a queue. The interaction with the local manager may differ depending on the underlying resource. For instance, the JMS `Session` interface provides both methods `commit()` and `rollback()` that process or destroy message operations included in the transaction respectively.

The major drawback of this model is that local transactions cannot be joined into the single ACID transaction that spans over multiple resources using the XA global transaction [21] (e. g. we need to update a database and propagate this information to the JMS topic). Another important problem is the requirement of the manual transaction management in the application code base.

2.4.2 Programmatic transaction model

The programmatic transaction model (also referenced as the Bean-Managed transaction (BMT)) is introduced by the Java Transaction API (JTA) specification. In this model, the developer handles the complete management of transactions in the source code.

Although the JTA specification provides a range of APIs, the main concerned interface for the utilization of the programmatic transactions is the `javax.transaction.UserTransaction` [18]. This interface represents an abstraction for the developer to programmatically con-

trol transaction boundaries. The only concerned methods are `begin()`, `commit()`, `rollback()` and `getStatus()`. The call to the `begin()` will start a new transaction and associates it with the current thread. As the Java platform allows only one transaction to be associated with the thread, a call to the `begin()` method may result into an exception in case the transaction has already been started in the current context. The transaction end methods (`commit()` and `rollback()`) perform their respective actions and disassociate the transaction with the thread. The `getStatus()` method returns an integer value representing the status of the current transaction derived from `javax.transaction.Status` class [18].

The most important problem introduced by the programmatic model is that the developer must ensure that the transaction is always terminated in the method that started the transaction [21]. This is often the case when the initiating method ends up with an uncaught exception and for this reason, the transaction needs to be committed or rollbacked before the method returns.

2.4.3 Declarative transaction model

The declarative transaction model is also referred to as the Container-Managed Transactions (CMT). In this model, the supplying, underlying container manages all transactions on the user's behalf. This includes starting and the administration of the end phases (either commit or rollback) of transactions. The developer is only required to set up the container with the transaction configuration that declares, for instance, that the transaction should be rollbacked on any exception.

The `javax.transaction.TransactionManager` is the main interface utilized for the declarative transaction model. However, in practice CMT transactions are generally controlled by specific annotations and the direct use of `TransactionManager` interface is discouraged. This interface is targeted for the use in the application server which allows it to control transaction boundaries on behalf of the application being managed [18].

With the declarative transaction model, users are required to configure the container with the settings of how individual transactions should be managed. This can be set up through the transaction at-

tribute represented by, for instance, `TransactionAttributeType` (Enterprise JavaBeans (EJB)), `Transactional.TxType` (Context and Dependency Injection (CDI)) or `TransactionDefinition` (Spring) classes. The supported values are – Required, Mandatory, RequiresNew, Supports, NotSupported and Never[2]:

- **Required** – If the transaction context is already present on the invocation, it will be used. Otherwise, a new transaction is started. This is the most characteristic attribute and it is usually configured as a default value.

- **Mandatory** – Similarly to the Required, mandatory transaction attribute represents that the transaction must be present on the execution. However, it requires that the transaction is already started prior to the invocation. Alternatively, it throws `TransactionRequiredException` if the transaction context cannot be found.

- **RequiresNew** – The container will begin a new transaction on every invocation. If there is already a transaction context present, it is suspended for the duration of the processing of the new transaction.

- **Supports** – This attribute represents an invocation that is not required to run under the transaction context. It tells the container to use the transaction context, if it exists before the call, or to execute the operation non-transactionally if the transaction is not present.

- **NotSupported** – The method will not be executed within the transaction context. If the transaction exists prior to the invocation, it is suspended and the method is invoked. In other case, the method is immediately started without the initiation of a new transaction.

2. The Spring framework adds one more transaction attribute called Nested which represents a single physical transaction with multiple savepoints that it can rollback to [22]

- **Never** – The container is forbidden to invoke the method if there is a transaction context present. In contrast with the NotSupported attribute which only suspends former transaction, this attribute will throw a runtime exception when the transaction is present before the invocation.

2.5 Distributed transactions

Transaction concepts, presented in the previous sections, described the transaction processing in a centralized environment. This included resource local and XA transactions that manage transactions spanning multiple resources through a global transaction manager.

However, these concepts can certainly be expanded to the distributed environment [1]. The distributed transaction represents an ACID transaction that is executed over a number of independent participants connected through a communication network[3]. The main disadvantage of these transactions is their liability to frequent failures of individual nodes or communication channels that connect them – which is something that the Distributed Transaction Processing (DTP) needs to account for.

Each node is associated with a transaction manager (TM) that manages a local transaction and communicates with other TMs in order to perform a global transaction. Generally, there is one TM selected as a global coordinator that administers TMs participating in the distributed transaction. The coordinator can be allocated with the participating node or can act as a standalone service.

The accomplishment of ACID properties with the frequent partitions failures is very difficult to achieve. In order to achieve the atomic outcome, all of the participating nodes need to reach a shared consensus on whether it is possible to execute a successful commit. The standardized protocol which guarantees the consensus for the ACID transactions is the two-phase commit protocol (2PC) which is used by the majority of modern transaction systems. The consensus protocols are discussed in detail in the following section.

3. The distributed transaction may also in some sources describe the XA transaction. In this text, it refers to the transaction spanning resources over the network.

Even if the DTP system is able to provide the distributed consensus, it often comes with a performance cost. This lead to the commonly hesitant utilization of the DTP concepts in distributed applications in the past.

However, recent network speeds and computational capacities are increasing. This allows consensus protocols (2PC, Paxos) and other DTP solutions (e.g., sagas [4]) to be easily employed in modern, scalable distributed applications.

2.6 Consensus protocols

The consensus problem represents the procedure of achieving the agreement for the shared data value between several components. It has its application in many environments including transactions where the TM needs to conclude whether a transaction can be committed depending on the participants consensus.

A consensus protocol describes a series of steps that solve the consensus problem. These steps can be typically divided into three phases – the selection of candidate values, the exchange of values between participants and the agreement. The final decision of each participant is irreversible. The consensus protocol is correct if it complies with these conditions [23]:

- **Agreement** - all non-faulty[4] nodes decide on the same single value
- **Validity** - if all non-faulty nodes have the same initial value, then the consensus must be reached on this value
- **Termination** - all non-faulty nodes eventually decide

In the transactions environment, the consensus represents the shared decision whether to commit or rollback the transaction. The following sections describe some of the most widely used consensus protocols that may be employed in (potentially distributed) transactional systems.

4. Some nodes may provide invalid or intentionally wrong information – these are known as *Byzantine failures* [24]

2.6.1 2PC

The *Two-phase commit protocol* is one of the most known employed consensus protocols used not only in the transaction processing. The procedure consists of two phases:

- **The prepare phase** - All participants send their proposals to the coordinator (TM) in which each of them states either that it is able to proceed and commit its work segment or that the transaction needs to be aborted.
- **The commit phase** – After collecting all proposals, the transaction coordinator makes a final decision – if all participants are able to commit, the transaction can be committed; conversely, if any participant stated that it needs to abort, the transaction is aborted. The final outcome is subsequently forwarded to every participant and the transaction can be finished.

The 2PC protocol is able to handle node failures to some extent through the use of transaction log. However, this does not cover every scenario and certain failures may require manual intervention.

The algorithm expects one node to act as a coordinator. This does not necessarily need to be an elected participant. Any node can act as a coordinator and initiate 2PC prepare phase by asking other participants for their votes. Furthermore, there also exits a decentralized variant but with the higher massage complexity.

The main disadvantage of the 2PC is that it is a lock based protocol. After the first phase, participants are required to hold locks on prepared resources until they receive a decision from the coordinator. If the coordinator fails after the first phase is completed, all participants will block waiting for the coordinator's decision and cannot progress (i. e. release locks) until it recovers.

2.6.2 3PC

The *Three-phase commit protocol* is a consensus protocol introduced in 1982 by Dale Skeen [25]. It extends the 2PC protocol in a non-blocking way – it allows participants to place upper time bounds on the phases completion which assures that resources are not held indefinitely. The three phases are:

- **The prepare phase** – Same as in the 2PC protocol.
- **The pre-commit phase** – If all participants voted in the first phase to commit the transaction, the coordinator sends to every participant a `preCommit` message. After the participant receives `preCommit`, it will proceed by preparing the commit by locking the required resources assuring that it is able to finish the commit. By this stage, the participant cannot execute any irreversible actions. If the preparation was successful, it responds to the coordinator with the acknowledgment message.
- **The commit phase** – After the coordinator receives preparation confirmation from all participants (the original paper [25] also allows to specify a majority vote count), it will commence the commit phase by sending the `commit` or `abort` messages, same as in the 2PC protocol.

The termination is achieved by the timeout boundaries set on every message expedition. If the coordinator timeouts, it always assumes the rollback outcome – it cannot proceed after the first phase as it did not receive votes from every participant and if the coordinator fails after the second phase, the state of the protocol would not be recoverable. However, the participant rollbacks in the same way after phase one (as it did not receive the outcome from the coordinator, it must assume abort) but if it timeouts after the second phase, it proceeds with the commit. This is allowed as the `preCommit` message is sent by the coordinator only if all participants wanted to commit the transaction in phase one.

If the coordinator fails, the new coordinator is selected by any election algorithm. This recovery node can determine the outcome of the protocol based on the state of other nodes – if any node received a `preCommit` message, the transaction can be committed (all other participants must have also received the `preCommit` message). If some node did not receive the `preCommit` message, the transaction can be aborted.

By contrast to the 2PC protocol, the 3PC is resilient to more types of failures. Nevertheless, it cannot withstand the network partition. If the partition disassociates nodes that did receive the `preCommit` message from those which did not, the newly selected coordinators on each

side of the partition will result into opposite outcomes and thereby an inconsistent system. These resilience capabilities are at the expense of the performance cost which approximately two times higher than in the 2PC protocol [25].

In 1998, Keidar and Dolev introduced an *Enhanced three-phase commit* protocol (E3PC) which maintains the consistency in the face of site failures and network partitions [26]. It is using two additional counters that impose a linear order on the majorities of system nodes while still preserving the same computational complexity as 3PC.

2.6.3 Paxos

The Paxos represents a family of distributed algorithms designed to solve the consensus problem. It was introduced by Leslie Lamport in 1998 [27].

The idea of the basic variant of the algorithm is reasonably straightforward. The Paxos distinguishes three types of system nodes: proposers, acceptors, and learners. One node can be of more than one type, even act as all of them. Proposers act as client representatives proposing values that the client wants the system to agree on. Acceptors serve as the voting mechanism and all nodes are required to know the number of acceptors that form a majority. Learners serve as the representatives that can be queried for the decided value. The algorithm runs in two phases:

- **The promise phase** – The proposer first sends its proposed value with a new unique identification number generated from a sequence to all acceptors (or the majority). When the acceptor receives a proposed value, it first checks whether the received id number is higher than the last id it promised to ignore. In that case, it will respond to the proposer with the `promise` message which denotes that it will ignore any newly received messages with lower ids. Otherwise, it already promised to the higher proposed id and therefore no action is taken.

- **The commit phase** – If the proposer collects promises from the majority of acceptors, it sends the `accept-request` message to all acceptors (or the majority) with the same id and the proposed value. When the acceptor receives the `accept-request` message,

and it did not already promise to ignore the received id, it sends the `accept` message to the proposer and all learners. If the id in the message is lower than the promised id, the message is ignored. As the `accept-request` message is sent to the majority of acceptors and all of them accept the value, the consensus is reached.

If the acceptor accepts a value, it appends the accepted id and value to each subsequent `promise` message. In this case, the proposer knows that there is some value already being decided in the system and it continues the processing with the value received in the `promise` message containing the highest of all received ids. If it wants to update this value, it needs to initiate a new run of the algorithm after the current one has ended.

Many variants of the Paxos algorithm allows it to sufficiently handle various types of failures that may influence the achievement of the consensus. In particular, even the basic variant handles problems of the 3PC algorithm, namely, network partitions and the restriction to the fail-stop model. Instead, the Paxos protocol is resilient to the fail-recover model which allows individual nodes to recover and continue processing from the point of the failure – which is expected in the modern distributed systems.

The problem that the algorithm cannot solve is two proposers that actively compete for the highest proposal number. This happens between phases as the proposer's `accept-request` message is rejected due to the higher proposal issued by the different proposer. The system is blocked until the conflict can be resolved. To mitigate this impact, systems can employ a form of the exponential back off mechanisms which allow one proposer to wait sufficiently long for the other one to finish. Another expectation of the algorithm is that acceptors are also required to have persistent storage to avoid providing misleading information in case of the fail-recovery.

2.6.4 Conclusions

This section presented in detail three consensus protocols that can be employed to solve the transaction commit / abort consensus in the distributed systems. There also exist many other algorithms, for in-

stance, the Raft or the Ark which cannot be discussed due to the space limitations. The summary of the presented algorithms is available in the table 2.1 (this table represents scenarios without any failures).

Protocol	Time (phases)	Message complexity	Client delay
2PC	2	$3(n-1)$	3 RTTs
3PC	3	$5(n-1)$	5 RTTs
Paxos	2	$4(f+1)$ (f = majority)	4 RTTs

Table 2.1: Consensus protocols comparison

The consensus is a very sophisticated and complex problem. One of its applications is the distributed transaction commit which denotes a consensus whether to commit or abort a transaction. The mostly applied protocol for this purpose in the current development is the Two-phase commit protocol [20, 28] (there is also research conducted with other mentioned protocols [29]). However, the application of consensus protocols, particularly in the distributed environments, may present problems mainly because of their locking nature. As it will be discussed in the following chapters, the saga pattern [4] provides a sophisticated alternative to the distributed transaction commit processing for long lived transactions.

3 Microservices architecture pattern

This chapter introduces the concept of microservices and it explains why modern, elastic and resilient enterprise systems should be designed and implemented according to this pattern. It provides an updated microservices status overview from my previous work publication [30].

3.1 Architectural pattern

Microservices are an architectural pattern which offers an intuitive approach to common problems following a software development. They represent a subset of a Service Oriented Architecture (SOA) [31] that advocates creating a system from a collection of small, isolated services, each of which owns its data, and is independently isolated, scalable and resilient to failure [32]. Instead of the SOA, which builds the applications around the system logical domain, microservices are focused around the application business model. Each microservice represents the separated and independent part of the system that interacts with other components only through predefined communication interfaces[1].

3.1.1 Monolithic architecture

The effective way of describing why the microservice architecture is emerging as a practical development style is to begin with the definition of the opposite pattern – the monolithic architecture. When the application is developed in the monolithic fashion, all of its content is being implemented and deployed as a single archive. Every component, i. e., a unit of software that is independently replaceable and upgradeable [33], is tightly coupled within the application. Because of the easy development of the monolithic software, this approach has been preferred by the majority of edging enterprise applications. However, when the application requires to add new functionality or to fix a software problem, any additional maintenance represents an

1. throughout the rest of this publication we will be using terms *microservice* and *service* interchangeably

issue. For instance, even because of the minor change or update in the single component, the scalability, continuous deployment and the general advancement of the whole application lifecycle can stagnate.

Monolithic applications present a few advantages – the development model is often easy to adjust to the application requirements[2], the deployment is reduced to single archive (or a small number of archives), and it is easy to horizontally scale by adding more servers behind a shared load balancer. The problems arise when the system becomes large. The monolithic code base is often complex and hard to understand which results in long learning curves [35] and developer concerns. The automatic deployment and the continuous delivery (CD) of the system also decelerate – in order to update one component you have to redeploy the entire application [36]. Although it is still able to scale horizontally, the replicated server instances take up more resources and overload the container with slower startup speeds. In general, the monolith also represents a commitment to a particular technology (or even its specific version) which makes the system difficult to maintain and also adapt to new emerging technologies.

3.1.2 Microservices architecture

Microservices introduced the application separation into the self-maintained units – services [37]. The service is a single scalable and deployable unit, which is not dependent on any context. This means that services may be deployed and scaled independently of each other, and may employ different middleware stacks for their implementation [38].

The important attribute of the microservices system is service isolation. Each microservice is responsible for the management of its own resources and it is prohibited to access resources of any other service directly. This means that each data request must be processed by the operating microservice which is allowed to accordingly control the data access and computation requirements. Services often correspond to components in the monolithic architecture.

Microservices further extends the Law of Demeter which intents to organize and reduce dependencies between classes [39]. As the service

2. the traditional development model represented as the client-server-database or the Model-View-Controller architecture [34]

presumably requires to communicate with other services in order to provide system functionality, this law naturally applies to minimize such coupling among microservices on the distributed component level.

Another standard object-oriented rule that also applies in the microservices environment is the Single Responsibility Principle (SRP) as defined by Robert C. Martin – a class should have only one reason to change [40]. There is a common misconception associated with the microservice architecture – the word *micro* should conform to the service size. Although this statement is true to some extent (there is no point in creating the microservice of the same size as the monolith), the *micro* should more resemble a scope of the service responsibility. This concept also corresponds to the Unix philosophy: Make each program do one thing well [41].

The separation and loose coupling of microservices provide an ability to deploy each individual service to the production environment autonomously, not affecting other applications or services. This allows isolated teams to develop, maintain and upgrade services independently and to form these teams around the system problem domains.

As microservices represent stateful entities, to achieve data isolation each service exposes an application programming interface (API) through which it is exclusively able to provide functionality to other services. These APIs are often technology-agnostic to ensure that the technology choices are not constrained [2]. Instead of in-process calls employed in the monolithic architecture, applications based on the microservices style utilize services by remote procedure calls which are often asynchronous. This form of segregation also facilitates the system failure recovery or resilience as each particular microservice breakdown is less prone to influence the rest of the system.

3.2 Principles of microservices

This section describes the microservices architecture from the perspective of the business use cases and the solution architecture. It is based on the work of Sam Newman [2, 42] in which he proposed to build each microservices system on a set of principles. These principles

may differ for various systems (depending on the application and microservices use cases), but in general, they can be reduced to these eight principles:

1. **Modeled around the business concepts** – When the microservices applications, together with the teams that are responsible for their maintenance, correspond to the business domain, they are generally more stable – the requirements on their functionality do not change frequently. This allows developers to focus on the particular system segment, rather than on some specific technology stack. Additionally, it also permits services to reflect the business requirements directly.

2. **Adapting a culture of automation** – Because of the service motion, failures or the communication distribution through the network, microservices brings additional complexity to the system. When the number of services increases, their maintenance, administration, and deployment can become unmanageable. The automation then presents an essential part of the service lifecycle. Practices as the automated service testing, the utilization of the continuous delivery or the unification of the deployment strategy over services, allow enterprise systems to scale more efficiently and to speed up the mechanism of the service coordination.

3. **Hiding the internal implementation details** – Every microservice generally needs to interact with other services or external systems to provide its functionality to the rest of the system. In order to keep the option of independent development, it is essential that each service hides its implementation details. This can be achieved through the motion of bounded contexts as defined for the Domain-Driven design (DDD) [43]. The bounded context delimits the applicability of a particular model, so that team members have a clear and shared understanding of what has to be consistent and how it relates to other contexts. The context is separated by an explicit interface represented as an API which allows teams to specify which utilities of the service can be shared and which must be hidden. Every request for

the service data must be subsequently processed through this public interface.

4. **Decentralizating all things** – Microservices architecture is built around the idea of self-sustaining development which means that services are maintained autonomously. This allows to delegate decision making and authority to the team that is accountable for the service maintenance. The team is then able to take full ownership of the service, which with the support of the independent deployment mechanism, results into the convenient development, testing, and life-cycle management. This principle accentuates that relevant business logic should be kept in services themselves and the communication between them must be as simple as possible. This permits to design systems in a way that adheres to Conway's law stated in 1967 [44]:

   > Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure.

   This principle also affects the system architecture and design. The purpose is to avoid approaches like enterprise service bus (ESB) or other orchestration systems, which can lead to centralization of business logic [2]. In general, architectures build on the choreography patterns rather than orchestration are preferred. The comparison of these two approaches has been investigated in many research works [45, 46, 47].

5. **Independent deployments** – This is the most important principle of the microservices architecture. When the service is being deployed, it should be the requirement that it cannot influence the lifespan of any other service. To achieve this, various techniques like consumer-driven contracts or co-existing endpoints can be used. Consumer-driven contracts make services to state their explicit expectations. These requirements are supported by the provided test suite for individual parts of the domain and they are run with each Continuous integration (CI) build. Co-existing endpoints model accommodates consumers to service changes over time. The idea is to make new endpoint which

can process updated client requests while the former endpoint still functions for a limited time. This includes techniques as blue/green releases [48] or canary deployments [49]. Customers can utilize both endpoints depending on the version their applications require which allows them to decide when to upgrade. Once the previous endpoint is no longer in use, it can be safely removed.

6. **Customer first** – Services exist to be called. It is indispensable to make these calls as simple as possible for the customers. The developers can advantage from any feedback from the clients that use their service. To ease the understanding of the service API, it should be supported by proper documentation provided by API frameworks like Swagger [50]. This also includes the service discovery mechanisms to propagate system services and to make the discovery of the service providers more apparent. To combine this information, we can use the humane registries [51] which indicate the human interaction.

7. **Failure isolation** – Even if microservices force distributed isolated development, the architecture still needs to protect against the failure propagation between services. This principle encourages resources separation to avoid the single point of failure. It is also supported by the service location distribution. As services require to communicate remotely, it is important to account for the network failures. To prevent cascading failures, various techniques like timeouts, bulkheads or circuit breakers [52] may be employed. As there are many vulnerabilities in applications which cannot be considered, there is no precise manual on how to attain this principle.

8. **High observability** – Monitoring is an important part of development and production deployment. Because of the microservices system distribution, it is not sufficient to observe actions performed by particular services apart. Instead, the monitoring solution must record the system operations altogether. To make this information more accessible, the aggregation is essential. Storing all log entries and statistics in one place can highly impact the monitoring process. Another relevant issue is to track

the service calls as each service typically communicates with other services. This can be achieved by mechanisms as semantic monitoring and techniques like synthetic transactions or correlation IDs [2]. By logging this kind of information, we can ensure traceability in the case of service failure.

3.3 Reactive microservices

Before the definition of what the reactivity means in distributed microservices environments, it is appropriate to start from the basics of what the reactivity signifies in general terms and how these principles may be applied in software architectures. This section introduces the motion behind the reactive design and why it is suitable for the use in the microservices environment.

By the definition in the Oxford dictionary, the word *reactive* symbolizes an exposure of a response to a stimulus or an action in response to a situation rather than creating or controlling it. This definition naturally translates to software systems. However, the interpretation of what the stimuli is in software applications may differ. It might be, for instance, events, messages, requests or failures. The important common property of these impulses is that the development model cannot be implemented in a way to control them. These motions differ from the traditional style of programming models in which the program functioned as a sequence of commands that were always executed in the predefined order and in the maintained controlled state.

In software systems, we distinguish three distinct classes of reactive concerns – reactive systems, reactive programming, and reactive streams [53].

3.3.1 Reactive systems

Reactive systems are an architectural style that focuses on the responsiveness. By the definition provided in the Reactive Manifesto [3], reactive systems are also resilient, elastic and message driven which makes them more flexible, loosely-coupled and scalable. Generally, this model provides straightforward programming interactions and simplified dependency management which are required in modern

applications. The following enumeration explains these essential properties in detail:

- **Responsiveness** is the most important characteristic of reactive systems. It provides a guarantee of a timely response to regular user requests, as well as the rapid failure detection. Reactive systems are expected to establish a sufficient upper bound placed on the system response times to institute an end user assurance in the system usage.

- **Resilience** covers the responsiveness of the system in the case of the system failure. The manifesto states that the system is not resilient if it becomes unresponsive after any failure. Resilience can be achieved by, e.g., replication, isolation, delegation, and loose-coupling. This ensures that the failure in one part of the system cannot affect the system as a whole which shadows the component clients from any form of the failure handling.

- **Elasticity** involves the system responsiveness in the case of an alternating load. The reactive system is expected to be able to dynamically adjust and scale system resources according to the request traffic. Elasticity also implies that the system must be able to actively replicate and regulate its components and distribute user inputs among them by the scalable, predictive (and possibly reactive) algorithms.

- **Message driven** elaborates on the asynchronous message exchange between system components that promotes the loose coupling, isolation and location transparency. Explicit message utilization has many advantages, e.g., flow control, load management or monitoring and the engagement of the back pressure. The location transparency, based on virtual addresses, decouples individual components. It may also provide a failure management mechanism, in which case the service cannot distinguish between the communication with a single component or a cluster. Additionally, the asynchronicity allows the system to utilize resources in a non-blocking way, only when they are required for the request processing.

3.3.2 Reactive programming

Reactive programming is a development model focusing on the observation of data streams, reacting to changes, and propagating them [54]. In the rest of this section, we will be referencing these data streams in the Reactive Extensions methodology as *observables*. An observable is an object that contains dynamically versatile data that represent a state which may be of interest to other objects. To consume data emitted by the observable, the interested object must *subscribe* to it.

In practice, reactive programming distinguishes three kinds of observable objects – observable data streams, singles, and completables. Observable, as a stream of data, represents an asynchronous reaction. It provides three handlers, namely, for the data value, error handling and the end of the data stream. The single is a special type of observable that depicts the stream of one value. It is associated with an execution of asynchronous operation which provides data and error callback handlers. The completable observable symbolizes the stream without any value. In contrast with the single, it does not return a data value. For this reason, the completable should be configured with the completion and error handlers.

The observable stream can be of two types – a cold or hot observable. The cold observables are lazy loaded. This means that the data stream does not process any tasks until somebody starts observing it. It represents an asynchronous action that is invoked only when there is a consumer interested in the result. When an object subscribes to the cold observable, it receives all data objects contained in the stream which allows them to be shared. Conversely, the hot observable data stream is active before the consumer subscriptions. When the consumer subscribes to the hot observable, it will receive all data values from the stream that are emitted after the subscription is created. Both cold and hot observables require the user subscription to receive the data values from the streams. If the consumer does not subscribe to an observable stream, the data is lost.

The most important concept of reactive programs is the asynchronicity. On the contrary from the traditional program invocations, this processing model is based upon notifications that are emitted when the data stream produces a value. Each asynchronous operation happens independently of the main program flow which introduces

several new aspects that need to be considered for this kind of programming paradigm. These aspects can be summarized in three simple rules: avoid side effects[3], avoid using too many threads and never block.

One of the most popular implementations of the reactive programming principles in modern systems is the Reactive eXtension (Rx). It represents a library for asynchronous and event-based programs by using observable sequences [55]. These extensions combine the observer and iterator patterns with a range of functional idioms to allow developers to easily adapt reactive methods. Reactive extensions provide a broad range of implementations for various programming languages as, for instance, Java (RxJava), C# (Rx.NET) or Kotlin (RxKotlin).

To conclude, it is important to remember that reactive programming does not build a reactive system [56]. It only provides a development model that can be used for asynchronous processing, a task based concurrency model or the non-blocking Input/Output (I/O) that may be applied with other reactive principles to create responsive and reliable distributed systems as defined in the Reactive Manifesto.

3.3.3 Reactive streams

Reactive streams represent an initiative to provide a standard for asynchronous stream processing with non-blocking back pressure[4] [57]. It addresses the issue of controlling the load placed on the stream destination in case of the consumption overload. The main focus of reactive streams is placed on mediating the stream of data among different API components without the requirement to buffer unreasonable amount of data on the receiver side.

This specification aims to provide a minimal set of interfaces and protocols that would describe the operations and entities to achieve asynchronous streams of data with non-blocking back pressure [54]. It mainly serves as an interoperability layer. The current provided Java

3. The side effect is any interaction of the function with the remainder of the program in other way than through its arguments or its return value

4. Back pressure is a form of feedback mechanism that allows consuming object to regulate the load which is being sent to it. It is typically employed in situations where the publishing object (observable) is able to emit data items more quickly than the consuming side is able to process them

virtual machine (JVM) implementation includes the Java API, the specification, the technology compatibility kit (TCK) and programming examples.

The API components that are required to be provided by the Reactive Streams implementations are publisher, subscriber, subscription, and processor. The publisher provides a potentially infinite number of elements in a sequence which are being published according to the subscriber's demands. The subscriber subscribes to the data publisher with a call to `Publisher.subscribe(Subscriber)`. The outcome of this operation is signaled to the subscribing consumer by a call to the `Subscriber.onSubscribe(Subscription)` method. The subscriber is able to request data delivery by a call to `Subscription.request(long)` in which it can specify the number of items it is able to consume. This call is then followed by a requested number of `Subscriber.onNext` calls that represent the delivery of the data item to the subscriber. If the requested number of items is `Long.MAX_VALUE`, the request is treated as effectively unbounded. The termination of the data consumption is signaled to the subscriber by the call to one of the `onComplete` or the `onError` methods. The subscriber is also allowed to request the publisher to stop sending data at any time by a call to `Subscription.cancel` method, but there still may be some data received due to the asynchronous nature of the publisher. The processor represents the component which is both the publisher and the subscriber in one instance, and it must follow the contract of both interfaces. The full API provided by the Reactive Streams in version 1.0.2 is available in the Appendix B.

All of the API component interfaces discussed above has been already included in the Java development kit (JDK) 9 in the class `java.util.concurrent.Flow`. The reactive streams initiative is supported by the companies like Netflix, Pivotal, Red Hat, Twitter and many others.

3.3.4 Summary

Previous sections described the reactive principles that may be used in software systems. As the microservices architecture supports the service isolation and loose-coupling, it can provide a reasonable environment for the achievement of properties demanded by the reactive systems [3]. Additionally, as microservices are expected to provide

asynchronous processing with a single and well-defined purpose, they can certainly benefit from the utilization of the reactive programming and reactive streams. The reactive principles naturally fit into the microservices architecture which is why these reactive microservices may form a suitable building block of modern responsive microservices systems [54].

3.4 Challenges

As it is common with every architectural style, microservices bring together with the above mentioned benefits also some drawbacks. Although the development experience showed that microservices are preferred choice over monolithic architecture, they may not be inevitably suitable for every system. This section describes some of the challenges that may impose problems when building systems based on the microservices pattern.

3.4.1 Distributed systems

The service distribution supports the architecture by the inherent model of the service boundaries. However, the communication over the network brings a few complications that microservices need to account for.

The first considerable issue is network failures. Because this is something that cannot be controlled by the invoking service, it is required that each microservice call is treated with caution (e.g., by setting an explicit timeout). Consequently, every service should always be designed in a resilient way with the possibility of failures in mind.

Another relevant problem presented by the network overhead is the communication performance. Remote calls that are required for inter-service invocations, represent additional complexity and time consumption that are not present in modular monolithic systems. There are several various techniques that may be employed to improve the general performance like, for instance, decreasing the number of calls or making them asynchronous.

The problems mentioned above have been elaborated in the work of James Gosling who in 1997 extended the draft created by Peter

Deutsch which stated wrong assumptions that are commonly being made about distributed systems. These assumptions are known as *The 8 fallacies of distributed computing*:

1. The network is reliable.
2. Latency is zero.
3. Bandwidth is infinite.
4. The network is secure.
5. Topology doesn't change.
6. There is one administrator.
7. Transport cost is zero.
8. The network is homogeneous.

Unfortunately, the original work of authors is no longer available, but Arnon Rotem-Gal-Oz provides in his publication [58] a very detailed description of each individual assumption.

3.4.2 Eventual consistency

The arguable problem present in the distributed microservices systems is the delay between the write of the data value and its actual obtainable update. Maintaining strong consistency between individual nodes is extremely difficult, which means that each service has to manage eventual consistency [59]. This may lead to the decreased system usability and customer satisfaction.

The eventual consistency is a model where after an update the system guarantees that if no new updates are made to the object, eventually all accesses (data reads) will return the last updated value [60]. It is a form of weak consistency – the system does not provide any consistency guarantees for a limited time called inconsistency window. The advantage of this model is that the maximal size of the inconsistency window can be computed from system statistics if no failure occurs.

Imagine a situation where the customer creates an order in a web interface. After the confirmation, the order request is sent to the service A which starts its processing. Right after the confirmation, the user wants to check if the order was created in the orders section which is provided by a service B. If the message about the order creation has not yet been propagated to the service B (the inconsistent window is still open), it cannot respond with the actual updated system state.

Microservices are required to manage eventual consistency to avoid making decisions based on the inconsistent information. These kind of issues are often hard to find as they are often discovered only after the inconsistent window has been closed.

The propagation delay problems are also present in the monolithic systems, but as it was mentioned in the previous chapter, the remote calls are typically remarkably less performant than in-process communication. However, in practice applications do not really depend on the strong consistency guarantees to the expected extent – for instance, the saga pattern (section 4.3) is based on the eventually consistent model. As it will be presented in the following section, it was proven that distributed systems cannot achieve both strong consistency and high availability at the same time, which is why modern distributed business applications are often willing to reasonably tolerate temporary data inconsistencies in order to promote availability.

3.4.3 CAP theorem

In 2000, Eric Brewer introduced the idea that there is a fundamental trade-off between consistency, availability and partition tolerance [61] in the distributed system. This proposal is known as *the CAP theorem* and it states that distributed systems can provide at most two of these three properties.

The consistency guarantee ensures that if some value has been written by a specific node, the query placed on any other node must return the same value or the later update. The availability states that if the node has not failed, it must always be able to respond. This does not permit error responses since the system could be trivially available by always returning an error [62]. Finally, the partition tolerance is an ability of the system to continue functioning even if the communication access between two or more nodes has been lost. In other words,

this means that services are still operating but they are not mutually reachable.

In practice, the general belief is that for wide-area systems, designers cannot forfeit the partition tolerance and therefore have to make a choice between consistency and availability [13]. The only effective way of how the system can guarantee that the partition can be avoided is to only maintain a single service.

The CAP theorem was formally proven in 2002 by Seth Gilbert and Nancy Lynch [63]. The prove is surprisingly simple and can be modeled by just two nodes.

In 2012, Daniel J. Abadi proposed an extension of the CAP theorem called PACELC theorem which is particularly applicable for distributed database systems (DDBSs). The theorem states – if there is a partition (P), how does the system trade off availability and consistency (A and C) or else (E) when the system is running normally in the absence of partitions, how does the system trade off latency (L) and consistency (C) [64]. Conversely to the CAP theorem, which implies to any read-write distributed implementation, the ELC part applies only when the system replicates data.

3.4.4 Operations

With the microservices architecture comes an inherent complexity incurred by the operational service management. As services are expected to be dynamically created and destroyed, upgraded, scaled and deployable, it is essential that the system employs techniques which simplify the operations processes. Such methods include automation, continuous delivery (CD) and integration (CI) and possibly external monitoring and orchestration tooling. Many of these approaches are useful even for the monolithic applications, but they become necessary if the system makes use of microservices [59].

In the modern move to the cloud architectures – platform as a Service (PaaS) considerably ease operations tasks. Containerization platforms like OpenShift [65] or Kubernetes [66] simplify the management of networking, automatic scaling, replication, resilience, tracing, monitoring and many other tasks.

All of the above mentioned procedures remarkably accelerate the software lifecycle process intervals. Therefore, the utilization of De-

vOps (development and operations) principles which promote increased collaboration and shared responsibilities across teams is also essential. DevOps allows not only to deliver products faster, while still maintaining the reliability assurance, but they also provide capabilities to bring the quality into the development process itself.

3.4.5 Human factor

Previous sections discussed the complexities of microservices architecture from the technical point of view. This section focuses on teams that are creating individual services and explains some of the reasons which may influence how this mind shift can affect their end performance.

- **Communication** – The team communication is essential. The opinion and problem solution discussion capabilities inside a team directly influence the deliverability of the maintained service. Although the service boundaries should prevent the requirement for inter-service agreements, it is not always possible to avoid it entirely (e.g., version management). The commitment to the DevOps culture can also support the communication process.
- **Nonuniform technology** – The possibility to choose the technology stack individually for each service may promote independence and exploring possibilities, but may also impose a maintenance overhead in a long duration.
- **Design shift** – As microservices still represent a relatively young architecture, the design composition change may be hard to utilize. Even if there are a few successful systems based on the microservices architecture (Netflix or Spotify), inexperienced teams may be hesitant to make a move to the split of the working monolithic applications. Furthermore, the transition itself impose a complex task.
- **Monitoring** – With the high number of services, collecting and processing of the monitoring information may become a very complex task. It also affects the problem traceability and debugging across services.

- **Testing** – Despite the fact that there exist several microservices testing strategies [67] which may help with the test case definitions, the asynchronous communication and the service distribution may still impose significant problems associated with test development and execution.

4 Saga pattern

The conventional transaction processing, as described in previous chapters, represents a complex challenge in the distributed microservices environment. In this chapter, we introduce the notion of sagas – an alternative approach to the traditional distributed ACID transaction execution.

A saga, as described in the original publication [4] from 1987, is a sequence of operations that can be interleaved with other operations. Each operation, which is a part of the saga, represents 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 (Figure 4.1). The original paper [4] defines the saga as the long lived transaction in the database environment.

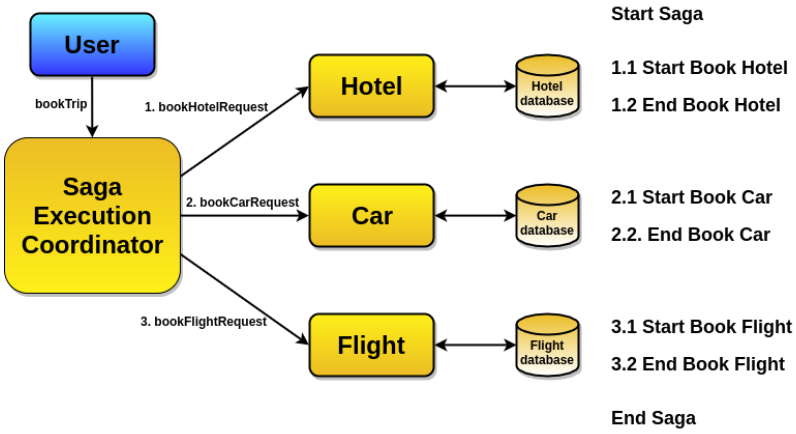

Figure 4.1: Example saga execution

In this chapter, we will describe the saga processing as it is defined by the initial publication [4], explain how the operations and compensations work in sagas, how the saga handles the failure recovery and present several modern frameworks that provide the support for the saga implementation.

4.1 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 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 (Section 3.4.2) which guarantees that the state will become eventually consistent after the saga completes.

As the definition in the original paper by Garcia-Molina and Salem [4] allows individual saga operations to interleave, it prohibits any form of dependencies between them. This would imply that the participating operation cannot depend on results committed by any previous operation in the saga sequence. However, in modern systems, the sequential execution of local operations is possible to implement if the system employs a single saga coordination process that manages the entire sequential execution.

4.2 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 activity began or generally the saga started.

Imagine that the operation consists of the sending of an email. The compensation cannot directly undo the email dispatch. Instead, it would send another email to the same destination which could explain why the previous action did fail. In this case, the system is in the state where it has two additional emails being sent. However, the

comprising system state is expected to be semantically consistent as both operation and compensation have been defined by the participant. Therefore, the consistency guarantees must be ensured by individual participants at the operation level.

The compensating actions for the individual operations are expected to be idempotent. The main reason for this requirement is the saga failure and recovery management which in detail described in section 4.5.

It is important to keep in mind that even compensating actions may fail. There are several options of how the saga management system can handle such situations. The first option is to retry the compensating operation again, but as the reason of the failure may still be valid which means that the system can get caught in an infinite retry loop. Another option is to provide a recovery block – a separated block of code which would get executed in place of the primary compensation in case of failure. The last option, which is not elegant but it is practical, is the manual intervention. This is possible to implement due to the saga nature – it does not hold any locks on resources it is being performed on. When the compensation handler is manually repaired, the saga can continue its execution where it has left off.

4.3 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 [68] which directly breaks the isolation property.

Sagas utilize an alternative BASE model [69, 70] which values the availability over the consistency provided by ACID (CAP theorem - section 3.4.3). 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 (section 3.4.2).

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.

4.4 Saga execution component and transaction log

The saga execution component (SEC) is a process that is responsible for the saga management. It communicates with the transaction manager which manages operations included in the saga. Both of these components require a transaction log to record their respective interactions[1]. The saga execution component does not require concurrency control because the saga operations can be interleaved.

The entries that may be written to the transaction log are usually associated with the saga or operation lifecycle. The saga log includes `start-saga` entry followed by one or more `start-operation` / `end-operation` entries and is finished by the `end-saga` entry. Optionally, the transaction system may also provide an ability for users to cancel the saga execution with the `abort-saga` command.

Each saga operation is channeled through the saga execution component and is recorded in the transaction log before any action may be taken. The transaction log can also contain any parameters associated with the saga execution.

1. it is convenient to share the transaction log between both components

4.5 Recovery modes

The saga paper [4] distinguishes two options to handle a failure that interrupts the saga. These two supported modes are backward and forward recovery.

Backward recovery

A backward recovery mode is the most common way of handling saga failure management as it was described in previous sections. It requires that all activities must define a compensation handler.

When the SEC component receives an `abort-saga` command in the backward recovery mode, it firstly aborts the currently executed operation. Then for every previous operation in the reverse order of the original execution, it calls its respective compensation action. After the invocation of the compensation handler corresponding to the first operation is completed, the saga may end and the system is in the semantically consistent state as it was before the saga began.

When the saga management system applies the backward recovery mode, the associated transaction log is also used to recover from the crashes of the saga coordinator. After the recovery, once all operations have been completed (committed or aborted), the saga coordinator determines the status of each saga execution by the investigation of transaction log entries.

If the log contains only both `start-operation` and `end-operation` entries for the operations comprised in the saga, then the execution is safe to continue with the next operation which has not been started. Another safe state is when the transaction log contains the `abort-saga` entry. In this case, it calls all compensation handlers for all referenced operations in the saga. This is possible due to the fact that all compensating actions are required to be idempotent.

The only unsafe state, which may be introduced after the coordinator recovers, is when the transaction log contains the `start-operation` entry without the corresponding `end-operation`. In this case, the saga coordinator selects the last successfully executed operation (contained in the transaction log) and invokes its compensation handler and compensations for all operations that have been performed before it.

As in the case of repetitive recovery for the same saga, the saga coordinator may call the corresponding compensation handlers repeatedly, the compensation actions are required to be idempotent. The original paper acknowledges that this constraint may be difficult to implement in some applications which is the reason for the introduction of the forward recovery mode discussed in the following section.

Forward recovery

For the use of the forward recovery mode, the transaction management requires that the saga itself is predefined and that the system is able to produce a checkpoint. The checkpoint represents a snapshot of the system state at the particular point in time into which the system can always be restored.

The pure forward recovery mode takes the checkpoint automatically at the beginning of each operation. Furthermore, it also disallows to abort the intermediate saga execution. This effectively eliminates the need to define any compensation actions. If the crash of the SEC occurs, it will abort the last executed operation and restart the saga from the last checkpoint. This approach effectively degrades the saga execution component to a basic persistent transaction executor, therefore losing most of the saga benefits.

Backward / forward recovery

In addition to modes defined above, it is also possible to combine these two approaches into the backward / forward recovery mode. In this mode, the transaction system takes checkpoints in predefined intervals which may be periodical or based on different criteria (e.g., the operation complexity). In case of the SEC failure, the system performs the backward recovery to the last defined checkpoint and then continues the saga execution in forward recovery mode.

4.6 Distributed sagas

The notion of sagas can be naturally extended into distributed environments [4]. The saga pattern as an architectural pattern focuses

on integrity, reliability, quality and it pertains to the communication patterns between services [71]. 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 also requires a transaction log which needs to be durable and distributed. The examples of distributed database providers include, e.g., Cassandra, RethinkDB or Apache Ignite and many others.

The saga execution coordinator (SEC) is spanned process across the participating services. This process manages and interprets the saga and it persists all processing information into the transaction log. The coordinator does not represent a single point of failure as it is allowed to fail. This is possible because the SEC process does not hold any state data, the complete saga state is contained in the distributed log. The general example of the application model employing SEC is available in figure 4.2.

As all of the above mentioned components are distributed, the saga management system needs to deal with a number of additional problems that are not present in the localized environment. The main problem is that the saga system is required to deal with network and participant failures that may happen between remote invocations. The four main locations where these failures can influence the saga execution are writes of the beginning and the end of the request to the transaction log and request and response calls to the associated participant. This may introduce unnecessary saga aborts[2], but generally approaches from the non-distributed environment still apply.

2. e.g. if the SEC fails after it has written the `start-request` entry to the transaction log – the recovery cannot determine whether the failure happened before the request, due to the network failure, or the participant response has been lost

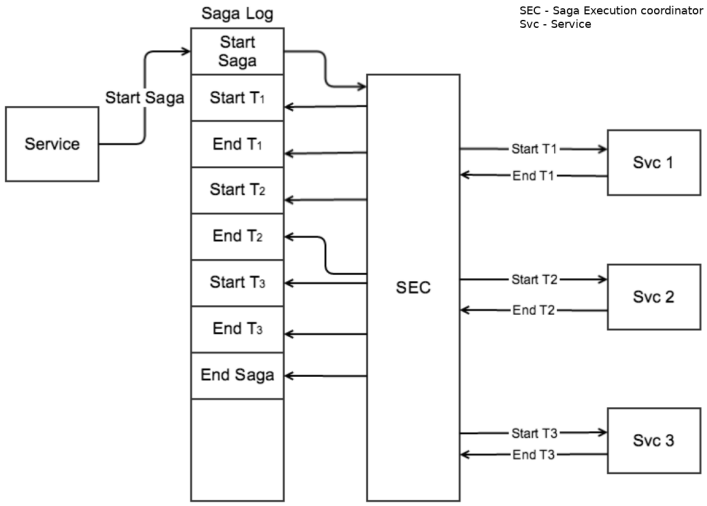

Figure 4.2: Distributed saga example [72]

4.7 Current development support

This section presents the current implementations of the saga pattern available for the enterprise use in Java applications. The four explored frameworks are Axon [6], Eventuate Event Sourcing (ES) [7], Eventuate Tram [8] and Narayana Long Running Actions (LRA) [9].

4.7.1 Axon framework

Axon is a lightweight Java framework that helps developers build scalable and extensible applications by addressing these concerns directly in the core architecture [6]. It is composed on the top of the Command Query Responsibility Segregation (CQRS) pattern which is described in the Appendix A.

The Axon framework is based upon event processing including asynchronous message passing and event sourcing. To decouple the communication between system components, it employs the mechanisms of the asynchronous message buses. This allows to design components with well-defined boundaries and therefore easy microservices development.

The CQRS architecture is directly embedded in the framework. Developers are controlling distinct CQRS components (aggregates, repositories, commands) by annotations.

Axon supports two types of message buses – the command bus and the event bus. This aligns with the CQRS pattern and allows to scale each part of the domain independently.

Saga definition

As the most of the Axon functionality, the easiest way to define sagas in an application is by annotations. The annotation `@Saga` marks the Java class as a saga implementation. In Axon, sagas are a special type of event listeners. Each object instance of the saga class is responsible for the management of a single business transaction. This includes controlling the saga state information, execution and handling of the transaction (including start and stop) or performing the corresponding compensation actions.

All interaction with the saga class happens only by triggering of events. Event handlers in saga instances are annotated with the annotation `@SagaEventHandler`.

To start a saga execution, the framework needs to receive the event with the special event handler annotated with `@StartSaga` annotation. By default, the new instance will be created only if the corresponding saga cannot be found.

The end of the saga can be defined by two means – by the event or by the API call. If the ending event is used, then the conforming event handler needs to be annotated with the `@EndSaga` annotation. Alternatively, the conditional end of the saga can be signaled by the call to the `SagaLifecycle.end()` from any method inside the saga class.

As many instances of the saga class may exist at the same time, there is a need to publish events only to the saga for which they are intended. This is done by a definition of association values. The association value is a simple key-value pair where the key is a property present in the event which forms a connection to the saga instance. The `@SagaEventHandler` annotation contains a custom attribute called the `associationProperty` which denotes the key property in the incom-

ing event. Axon also allows the definition of additional association values by a call to the `SagaLifecycle.associateWith(key, value)` and the `SagaLifecycle.removeAssociationWith(key, value)` inside any method of the saga class.

4.7.2 Eventuate ES

Eventuate is a platform for developing asynchronous microservices [7]. It focuses on the distributed data management allowing developers to focus on the business implementations. The platform consists of two products – the Eventuate ES and the Eventuate Tram. By the time of this writing, the Eventuate Tram was still in development, but as it introduced the sophisticated saga processing to the platform, it is in detail described in the section 4.7.3. This section presents the more general Eventuate ES product.

The Eventuate Event Sourcing (ES) provides the application with the programming model based on the event sourcing – a mechanism that tracks all changes to the data model as a sequence of events stored in the event log. Every model change is appended to the log and the sequence can be anytime replayed to restore the application state. The event log can also serve as an auditing solution and provide temporal queries to track previous application states. Another advantage is that the events can be replayed to the failed service after it reconnects.

The ES project is also based on CQRS principles which allow natural usage of event sourcing capabilities for aggregate entities. It is provided in two versions – as a service hosted on Amazon Web Services (AWS) or as an open source local platform.

Saga definition

The saga processing of the Eventuate platform is handled by Eventuate Tram project and therefore the ES product does not support saga implementation directly. However, the example *Eventuate service*, which is presented in the following chapter, is based exclusively on the Eventuate ES local platform providing the complete CQRS saga solution. It represents the manual user saga handling implementation not supported by the platform.

4.7.3 Eventuate Tram

The Eventuate Tram framework enables a Java/Spring application to send asynchronous messages as a part of a database transaction [8]. It utilizes the traditional Java Database Connectivity (JDBC) and Java Persistence API (JPA) based persistence model 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.

The Eventuate ES platform builds the communication exchange on top of the event sourcing. The Tram additionally provides three types of transactional messaging abstractions – messaging, events and commands. The messages are sent through dedicated named channels. The platform still supports the application development according to the CQRS pattern, but it does not enforce it.

The Eventuate Tram Sagas is a framework that provides saga processing on top of the Eventuate Tram message passing. It introduced a very sophisticated saga model that allows users to specify the saga definition as a single point of reference.

Saga definition

The saga is defined by the saga orchestrator – the service responsible for the saga handling. It is identified as an implementation of the `io.eventuate.tram.sagas.orchestration.Saga` interface. The respective saga definition is represented as an instance of the class `io.eventuate.tram.sagas.orchestration.SagaDefinition`. This instance must be returned by the `Saga.getSagaDefinition` method. It can be specified with a simple fluent API that provides methods to define operations that should be executed when the saga fails, when it needs to invoke a participant or how to process participants compensations.

The saga is defined as a sequence of steps specified by a simple Domain Specific Language (DSL) available in the `StepBuilder` class. Each step represents a local invocation, a remote participant invocation or the compensation definition. The last two return a participant builder which allows to additionally specify more actions that need to

be processed when the participant is invoked or the handler method which will be called when the participant responds.

4.7.4 Narayana LRA

Narayana Long Running Actions (LRA) is a specification developed by the Narayana team in collaboration with the Eclipse MicroProfile initiative [73]. 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 current reference implementation of 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.

The LRA is utilizing an orchestration saga model. One node is selected as a dedicated LRA coordinator that manages the saga processing. Its main responsibilities are the LRA initialization, participant enlisting and either the saga completion or compensation.

The coordinator can be a standalone service or it can be embedded within application service. For the second option, the coordinator communicates with the enclosing application by in-memory calls rather than REST.

The second model component defined in the LRA processing is the saga participant. This may be any service that is involved in the LRA. Each participant is required to provide at least one REST endpoint that serves as the compensation handler.

The execution of the LRA is started by the initiating service on the user request. The service calls the coordinator that in turn starts a new LRA and returns its unique identification to the initiating service. This id is used to enlist every participant in the saga. After the initiating service receives the LRA id, it can optionally enlist itself within the LRA and continues the saga processing with other participants invocations where each call has to contain the acquired LRA id. When a participant is invoked, it will enlist itself within the received LRA, perform its work request and subsequently return the invocation outcome to the initiator. After the LRA processing is completed or an error occurs, one of participating services (commonly the initiating

service) that knows the LRA identification, contacts the coordinator to close or compensate the LRA. The coordinator then performs the corresponding requested action for each enlisted participant.

Saga Definition

The LRA specification permits users to control the LRA lifecycle in two ways – by Java annotations or by a client API.

The MicroProfile LRA support is primarily targeted for the use of annotations which are present in the `io.narayana.lra.annotation` package. The `@LRA` is the main concerned annotation which denotes the method that should be executed within a compensatable LRA context [9]. This is specified by a required `Type` attribute which can have same values as conventional transaction attributes (section 2.4.3). Additionally, this annotation may also specify if the participant should join the incoming LRA or on which HTTP status codes the LRA should be canceled. The LRA can also be declared as scoped under another existing LRA by the `@NestedLRA` annotation. In this case, a new LRA will be started and its outcome will depend upon whether the enclosing LRA is closed or canceled [9].

In the reference implementation, if the LRA is already present on the invocation, it should be made available to the business processing through the HTTP header `Long-Running-Action`. This header contains the coordinator address and the LRA identification.

Narayana also provides an ability to control the coordinator directly by the LRA client API. These operations are contained in the `io.narayana.lra.client.LRAClient` class which allows users to perform basic client requests for the LRA services. On the background, the reference implementation contacts the LRA coordinator through its REST interface which users are also allowed to use directly. The implementation present in the `NarayanaLRAClient` class can be provided by the CDI dependency injection.

The participant is defined by annotations present on the JAX-RS resource class. It is required to expose at least one REST endpoint that handles the participant compensation. The handler resource method is defined by the `@Compensate` annotation and is automatically registered

within the coordinator when the participant is enlisted. This method will be invoked in case of the LRA cancellation.

The resource class is also allowed to specify more methods that control the participant LRA lifecycle by respective annotations:

- `@Complete` – invoked when the LRA is closed
- `@Status` – reports the participant's status to the coordinator
- `@Forget` – the coordinator allows the participant to forget information about the LRA[3]
- `@Leave` – causes the participant to leave the LRA

Both `@LRA` and `@Compensate` methods may additionally be annotated by the `@Timelimit` annotation that denotes the maximum time period after which the participant is timed out.

The participant can also be registered directly through the API. For this purpose, it must be a serializable Java class that implements the `LRAParticipant` interface. The registration is performed by a call to the `joinLRA` method of the `LRAManagement` class which can be in the reference implementation injected through the CDI.

4.7.5 Summary

Previous sections described four frameworks that provide saga implementation support. The following table 4.1 provides a simple summary of the discussed properties influencing the saga definition and execution of each respective framework applicable in microservices applications.

3. this information needs to be preserved if the participant is unable to complete or compensate immediately

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

Table 4.1: Saga implementations comparison

*there is a plan to make the coordinator flexible and resilient

5 Saga implementations comparison example

As a part of the investigation of the each discussed framework from the previous chapter, we created a sample application simulating the saga utilization. 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. The examples created for this thesis may be considered as artifacts of one iteration of design science research [74].

The sample application represents a backend processing for orders with a simple REST user interface. Users are also allowed to query persisted information through the defined microservices APIs available in Appendix D.

All examples are based on the microservices pattern. As every framework is suitable for the use in different environments, each example is achieving the same goal through the different portfolio of technologies. The exact mechanisms used in individual projects will be discussed in more detail in their respective sections.

Applications are also compared from the performance perspective in Section 5.8 by a created performance test executed in two different load scenarios.

5.1 Common scenario

A user is able to create an order by a REST call to the dedicated endpoint of the `order-service` microservice. The request must provide product information in the JavaScript object notation (JSON) format containing the product id, the commentary, and the price. For the simplicity reasons, the order always consists only of the single product. Figure 5.1 presents the example of the input JSON format for the product data. The complete REST API for each individual example is provided in the Appendix D.

The setup of each example is described in detail in their respective repositories. Generally, each microservice is a standalone Java application that must run in a separated Java environment. By default, all

```
{
    "productId": "testProduct",
    "comment": "testComment",
    "price": 100
}
```

Figure 5.1: Product Information example JSON

examples are accessible on the local address. Every microservice is also able to run in the Docker [75] platform and all quickstarts can be easily set up using the Docker compose project [76].

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

Users are also allowed to query the persisted saga data (orders, shipments, and invoices) by the respective REST endpoints described in the Appendix D. For the CQRS based examples, this information is available at the `query-service` microservice, otherwise each service is expected to be responsible for maintaining its individual persistence solution which corresponds to the microservices pattern definition.

5.2 Order saga

The saga pattern applied in this application performs the order requests. 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

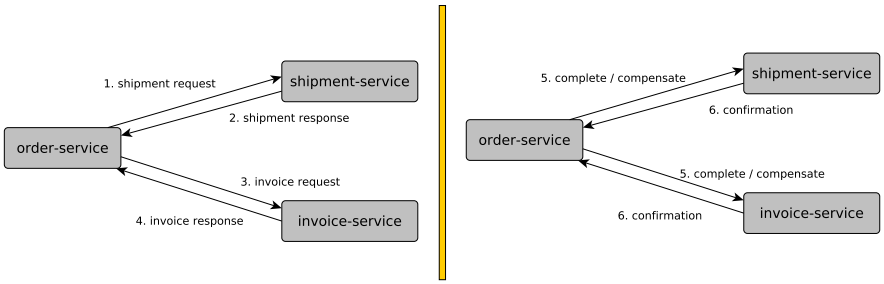

Figure 5.2: The saga model

cannot be created). The graphical representation of the saga progress is available in the figure 5.2.

Every application is able to demonstrate three testing scenarios: the valid pass, the shipment failure, and the invoice failure. In the valid scenario, after the order is requested, the saga propagation invokes requests for the shipment and the invoice. If the connection between services is stable, both participants successfully return an artificial answer and the order is completed.

As most of the applied platforms invoke participants sequentially, we distinguish separated member failures of the shipment or invoice. The shipment failure simulates the termination of saga without the full request coverage. This means that the compensations may be distributed to all services including the `invoice-service` which has not yet received the work request for the order being processed. The scenario demonstrates the need of microservices to be able to react to the requests that are not associated with any saga which means that they need to actively keep track of the sagas being currently executed.

The invoice failure scenario, on the other hand, validates that the saga compensations are executed on all participating services as the shipment is already expected to be completed. Generally, the saga pattern assumes that the compensations of the participants are called in the reverse order of the invocations because of the possible dependencies between them.

To initiate failures scenarios in examples both `shipment-service` and `invoice-service` contain injected failure conditions. To invoke the failure, the quickstarts expect product information containing a

specific product identification: `failShipment` or `failInvoice` respectively.

The graphical representation in the form of the sequence diagrams for corresponding scenarios in individual quickstarts is available in the Appendix C.

5.3 Axon service

As it was stated in the previous chapter, the Axon framework is based upon the CQRS principles. Because of this nature, it would be difficult not to follow this pattern. The individual services contain separated aggregates[1] each processing its respective commands and producing various events. Any inter-service interaction is restricted to the use of the command and event buses.

5.3.1 Platform

Axon service is a Java Spring Boot [77] microservices application. Each service is fully separated and independent Maven project [78]. Individual projects represent standalone runnable applications (fat java archive (jar)) which is the preferred distribution method for the Spring Boot applications. Axon is composed as a Java framework which is why it is included as a Maven dependency in individual microservices.

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.

1. for the definition of aggregate, please refer to the Appendix A

Command bus – Registration server service

The quickstart utilizes a variant of the distributed command bus which is based upon a different approach than the one used in the traditional single JVM Axon applications. The distributed command bus forms a bridge between separated command bus implementations to transfer commands between different JVMs [79]. Apart from the command distribution, it is also responsible for the selection of the applied communication protocol and the choice of the target destination for each transferred command.

Axon provides two options for the connection of different services through the distributed command bus – `JGroups Connector` and `Spring Cloud Connector`. The one used in this quickstart is the Spring Cloud [80] method. Axon also allows users to choose which particular Spring Cloud implementation will be used to distribute commands. The underlying implementation utilized in this quickstart is based on the Netflix Eureka Discovery client and Eureka Server combination [81].

Eureka server is a REST based service that is primarily used in the Amazon Web Services (AWS) cloud for locating services for the purpose of load balancing and failover of middle-tier servers [82]. In this application, it is represented by the `registration-server` service.

Eureka is also distributed as a Java-based client library Eureka Client which is included in all application business services – each service, as a part of its initialization, registers itself with the Eureka server. Axon is then able to redirect commands to the right service chosen by the value of parameters in the command class that is annotated by the `@TargetAggregateIdentifier` annotation.

Event bus - RabbitMQ service

For the distribution of the event bus, Axon service uses an external messaging system based on the Advanced Message Queuing Protocol (AMQP) protocol. The utilized message broker implementation is the RabbitMQ [83] which is contained in the `rabbitmq` service.

RabbitMQ is an open source message broker that supports multiple messaging protocols and is easy to deploy both on premises and in the

cloud [83]. Main provided features include asynchronous messaging, high availability cluster deployment or various tooling that supports continuous integration, management and monitoring.

This quickstart provides a separated message queue for each business service and one special queue for the `query-service` microservice which is subscribed to all processing events. Axon platform provides the direct support for the AMQP, so no specific handling of the produced events is required – Axon automatically distributes events to all connected queues.

Both the Spring Cloud registration server and the RabbitMQ message broker are required external providers that need to be running before the business services can be deployed. Unfortunately, by the time of this writing, there is no way to distribute commands or events directly in the Axon platform.

5.3.2 Project structure

The application is composed of six microservices – the `order-service`, the `shipment-service`, the `invoice-service`, the `query-service`, the `registration-server` and the `rabbitmq`. Furthermore, it also contains a separated project `service-model` which serves as a support library for individual microservices.

The `order-service` project is a business microservice responsible for the saga handling. It contains the logic for the order request, the saga initiation, and the saga compensation.

Both `shipment-service` and `invoice-service` are business services functioning only as saga participants. Their only obligation is to provide their respective computations.

The `query-service` is a specific microservice included for purposes of the CQRS pattern. It collects information about prepared orders, shipments and invoices, and provides an external APIs for users to query the collected data.

Both `registration-server` and `rabbitmq` services are summarized in the previous section. They provide capabilities to distribute command and event buses respectively.

Finally, the `service-model` is a Kotlin and Java Maven application providing the core API for commands and events used by various business services. This is required as all classes must match in order

for particular handlers to be invoked. Furthermore, it also provides common utilities and the logging support. It is mandatory to include this project on the classpath of every other service.

5.3.3 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. Axon only provides ways to indicate 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. As production ready sagas can be expected to run in a number of days, this kind of saga specification can quickly become the bottleneck of the saga maintenance.

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. Like it was already presented, 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.

5.4 Eventuate service

Similarly to Axon, Eventuate service is also based on the event sourcing and the CQRS pattern. For this reason, 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.

5.4.1 Platform

Eventuate service is a microservices application consisting of a set of Spring Boot [77] business services, one backing module and a number of support services provided by the Eventuate platform. In this section, we will focus on the Eventuate platform and services it provides, the business part of the application is described in the following section.

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 and the Eventuate console service. The example employs these ser-

vices as Docker images included in the provided docker-compose configuration.

Apache Zookeeper service

Apache Zookeeper is an open-source project which enables highly reliable distributed coordination [84]. It maintains a centralized service which supervises various functionalities like the handling of configuration information, synchronization, naming or grouping. The Eventuate Local platform provides its own Docker image tagged as `eventuateio/eventuateio-local-zookeeper`.

Apache Kafka service

The Apache Kafka streaming platform is the service which is responsible for the administration of subscription and publishing mechanisms controlling the event processing for business microservices. As it is based on the Streams API, it allows the platform to react to events in real time. Eventuate manages Kafka as the notification service for the event propagation. Eventuate ships its own Kafka version in the `eventuateio/eventuateio-local-kafka` docker image.

MySQL database service

The SQL database used in this application for the event persistence is the MySQL open-source database which is currently the only database supported by the platform. The Eventuate Local maintains two tables – `EVENTS` and `ENTITIES`. This database also serves as a transaction log maintained as a base for the event sourcing. The containerized version is located under `eventuateio/eventuateio-local-mysql` tag.

CDC service

This service represents the change data capture (CDC) component. The CDC service has two main responsibilities – it follows the transaction log and it publishes each event which is inserted into the `EVENTS`

table to the Kafka topic that corresponds to the aggregate for which the event is intended. Eventuate Local supports two options of the execution of the CDC – internally in each business service or as a standalone application. This quickstart applies the Eventuate CDC service `eventuateio/eventuateio-local-cdc-service` as a standalone Docker container.

Eventuate console

The last support service is the `consoleserver`. It provides a simple interface for accessing collected information about created aggregate types and the event log. The supplied Docker container image is the `eventuateio/eventuateio-local-console`.

5.4.2 Project structure

This section describes the set of services composing the business side of the application. This set contains four services that cover the saga execution and data processing (`order-service`, `shipment-service`, `invoice-service` and `query-service`), one service representing the persistent storage (`mongodb`) and the support module (`service-model`).

All of the business services are Spring Boot applications based on the Gradle [85] build system. Each microservice is represented as an independent module capable of being separately built and deployed. Even if Spring Boot projects can be executed directly from the command line as regular Java applications, this quickstart leverages the Docker approach of the Eventuate Local platform and containerize all of its services.

To connect to the Eventuate platform, each service defines a set of environment properties that are utilized in their Spring Boot configuration. This information includes connection and authentication details for the MySQL database, the CDC component, and the connection Uniform Resource Locators (URLs) for Kafka and Zookeeper services. These properties are specified in the container specification for each individual business service in the `docker-compose.yml` file.

The actual saga execution is managed in the `order-service` microservice. The saga realization implementation is contained in three classes – the `OrderSagaAggregate`, the `SagaEventSubscriber` and the

`OrderSagaService`. The first class is an ordinary CQRS aggregate that handles commands for the saga initialization and participants outcomes. Conversely, the second class represents the event processor listening for the events produced by the aggregate. It is basically a wrapper around the `OrderSagaService` – the class responsible for the remote REST calls to other services and the command dispatching for the `OrderSagaAggregate`. The usage of the separated event listener is required because Eventuate does not allow aggregates to be declared as Spring components. The reason of this drawback is described in more detail in the following section.

Except for the normal order API, the `order-service` also provides a management API for the participants to be able to share the information about their processing. These endpoints are hardcoded in the application which may not be acceptable for a production realization.

Both the `shipment-service` and the `invoice-service` contain a simple aggregate together with its associated event listener which control participant interactions within the saga. Each service accommodates REST endpoints for the saga request and its possible compensation.

Similarly to Axon service, the `service-model` project acts as a support library for other services. It contains a core API for each business service which needs to be shared and a few utilization classes.

The last business microservice is the `query-service`. It performs as a response service providing the information about persisted orders, shipments, and invoices. It contains an event listener for each designated microservice which in turn preserves the achieved information in the Mongo NoSQL database. This service also provides a simple Swagger interface to ease the user application interaction.

5.4.3 Problems

Complexity

As this project represents a plain CQRS based example, it completely 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 first 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 a few additional steps.

The full saga administration is handled by the project from the very beginning. It covers the support of starting, stopping and following the saga execution as well as saga compensations. Restrictions placed by the CQRS pattern furthermore put additional requirements on the saga processing which may not be demanded by other frameworks. Before the introduction of the Eventuate Tram framework, the Eventuate platform did not provide any saga support.

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. The aggregate class is required to extend the `ReflectiveMutableCommandProcessingAggregate` specifying the type of the command interface which allows the classpath instantiation. The event listener is defined

by the `@EventSubscriber` annotation on the target class and by the `@EventHandlerMethod` annotation on respective methods consuming dispatched 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 aggregate class is defined as a value of the `entity` attribute of the `@EventEntity` annotation. This annotation is usually placed on the event interface which implementation represents produced events.

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 specification. 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 manually replicated in each service (restricted to system properties) and therefore predisposed to errors.

In the end, it is worth mentioning that as the Eventuate service is the pure CQRS saga example, it has a few problems which have been reduced or removed in the later Eventuate Tram implementation that is in detail described in the following section.

5.5 Eventuate Tram service

The Eventuate Tram service quickstart is based on the new recently introduced Eventuate Tram framework. This framework provides several solutions to problems associated with the saga management that are present in the Eventuate platform.

5.5.1 Platform

Similarly to the full Eventuate 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. As all mentioned services represent the same functionality as they are responsible for in the full Eventuate platform, individual descriptions of each service are defined in detail in the section 5.4.1.

All services are deployed by the `docker-compose` configuration distributed with the framework. This setup is based on the same Eventuate docker images for `zookeeper` and `kafka` services, and with `mysql` and `cdcservice` redefined by Tram.

5.5.2 Project structure

As Eventuate Tram does not restrict its services to the CQRS pattern, this project, in contrast to Eventuate service, contains only three business microservices and one support module. Every service is configured in a similar way as for the full platform containing the references and authentication details for the MySQL database, Kafka framework, and Apache Zookeeper service. The quickstart is distributed with

the predefined `docker-compose` configuration file that enables one command startup of all specified services.

The last service which is not required for the saga execution and handling is the `mongodb` NoSQL server. This service is present only for the demonstration purposes to allow the data retrieval from different storage than the one that is used by the Eventuate platform.

The first microservice `order-service` is responsible for the order requests and saga processing. It also provides the ability to query persisted orders from the `mongodb` service.

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

The step provides an ability to advance the saga execution in three ways – by invoking of the local function, by a call to the remote participant or by the definition of the compensation method for the saga. Furthermore, the participant is able to define individual actions that comprise its engagement in the saga, the compensation handler and several reply handlers that are distinguished by the data object class that is received in the reply. This definition provides a simple in one place saga specification which is suitable for easier maintenance and distribution. This declarative approach provides many advantages that will be detailed in the following chapter.

The last two business services that contribute to the saga execution are `shipment-service` and `invoice-service`. Both of these microservices define several command handler methods associated with the channel that is dedicated to their respective functionality.

The channel is a main communication mechanism used in Eventuate Tram platform. It is denoted by a string name that needs to be specified in the command message as a target destination. The definition of channel associates commands that it is able to receive with command handling methods that are invoked when the corresponding commands are delivered. The command handler returns a `Message` object identifying the outcome of the invocation. The failure outcome of any participant will immediately result in the saga compensation.

Both services are also connected to the `mongodb` server in order to provide the browsing of created shipments or invoices respectively.

In conclusion, Tram remarkably simplified the Eventuate platform for the usage of sagas. Most importantly, it introduced a simple fluent API for the saga definition and the loss of CQRS restrictions. Altogether, Eventuate Tram platform makes a suitable saga solution for microservices based environment.

5.5.3 Problems

Destination identification

The destination channels in Tram are distinguished by a simple string which may cause problems in the case of name conflicts. Currently, the choice of the handler to be invoked depends on two resources – 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 interact with the same participant in several different commands. This may cause problems with definitions of compensation and reply handlers as the developer needs to mind the logical grouping of participant invocations.

5.6 LRA service

The LRA service is the first example which differs from previous quickstarts as it does not restrict its services to any particular conventions. Individual services are connected through the exposed REST routes.

5.6.1 Platform

This quickstart is composed as a set of WildFly Swarm microservices applications. Every microservice is designed to be easily deployed to the OpenShift container application platform provided by Red Hat, Inc. [65] Both of these platforms are in detail described in the following technology section.

Each service uses the `fabric8-maven-plugin` for the build and the deployment to the OpenShift platform. This plugin provides a straightforward way of declaring the necessary configuration information the platform requires to orchestrate the service according to the user demands. This project applies the source to image (S2I) toolkit that builds imminent Docker images which can be immediately deployed to the OpenShift.

As it was already presented in the previous chapter, Narayana's long running actions are not composed as a platform, but rather as a standalone coordination service. This project employs the standalone Narayana LRA coordinator service 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 checking (failure discovery) which is why this configuration is not included in example services.

5.6.2 Project structure

This project consists of five WildFly Swarm microservices and one support module. Each service is designed and configured with the `fabric8-maven-plugin` providing simple deployment to the OpenShift platform. Additionally, services contain a customized Dockerfile

specifying environment properties and the target Swarm uberjar file which is used for the source-to-image (S2I) builds in OpenShift.

The support module is called the `service-model`. This module is responsible for the specification of the LRA definition and exchanged JSON data formats, the description of the communication model used in other services and the administration of common utilities.

The `LRADefinition` class denotes the JSON format of the LRA representation. It presents a simplified version of the LRA capabilities for example purposes. The definition includes only required attributes – the name of the LRA, a list of individual actions that form the LRA and the unspecified object containing the user defined data associated with the LRA. The example LRA definition JSON format is available in the figure 5.3.

```
{
    "name": "testLRA",
    "actions": [],
    "info": {}
}
```

Figure 5.3: LRA definition example JSON

The individual actions that compose the LRA are incident to the pattern services in this quickstart use for the communication. The information exchange is based upon the REST architectural style which expects that services adhere to predefined endpoint rules.

The action definition consists of the name, the action type and the service for which the invocation is intended. The `service-model` project contains both `ActionType` and `Service` enumerations that denote possible values. The example action JSON is included in the figure 5.4.

The actual deployable microservices project consists of five services – `api-gateway`, `order-service`, `shipment-service`, `invoice-service` and `lra-coordinator`. Every service is configured with the addresses of other services as they reflect the OpenShift/Kubernetes application names. All of exposed APIs are defined in the Appendix D.

```
{
    "name": "testAction",
    "actionType": "request|status",
    "service": "order|shipment|invoice"
}
```

Figure 5.4: Action example JSON

The services that provide the LRA execution capabilities are the `order-service`, the `shipment-service` and the `invoice-service`. Every service provides a simple computation that contributes to the LRA realization. Additionally, `order-service` also provides a user invocation endpoint that can initiate the LRA. As all three services are eventually subscribed to the LRA, they all provide REST endpoints annotated by the LRA annotations for completion and compensation invocations. Each service is configured with an in-memory H2 SQL database for the data persistence.

The `lra-coordinator` project is provided by the Narayana framework. Although the LRA specification does not require the application of the REST architectural style, the `lra-coordinator` operates a set of REST endpoints that maintain the start of the LRA, gathering the information about active and recovering LRAs, the management of the nested LRAs, and the ability of participants to join (enlist in) or leave the LRA. Narayana provides this project already as a WildFly Swarm distribution. However, for the investigation purposes, this quickstart still builds it as a part of the S2I deployment.

Even if the `lra-coordinator` presents the REST endpoints for the LRA management, this quickstart invokes the coordinator by the client module provided by Narayana. The `lra-client` dependency provides the `LRAClient` class that is configured with the coordinator location and serves as a proxy separating the user from the actual REST invocations. This class is defined as a CDI bean to enable simple use through the dependency injection. The participant enlisting and invocations are defined by annotations present in the `io.narayana.lra.annotation` package.

The last service is the `api-gateway`. This module functions as an interface that makes the LRA execution transparent for the invoking services. The current state of the Narayana handling of LRA requests will be described in the following section.

The `api-gateway` uses the LRA and action definition classes from the `service-module` to handle the LRA processing on behalf of the initiating service. It exposes a REST interface that consumes the LRA definition JSON.

The actual LRA execution is managed in the `LRAExecutor` class. This class provides one public method `processLRA(LRADefinition lraDefinition) : void` that is responsible for starting and performing of the LRA, collecting the participants results and closing or compensating the LRA. This method processes the LRA asynchronously – each execution is propagated to the new thread from the cached thread pool defined by the executor service present in the `LRAExecutor` class.

As this module was designed for this particular LRA scenario, it is configured to execute the LRA actions (order, shipment and invoice requests) independently and in parallel. It collects the result of each action and eventually closes or cancels the LRA with methods provided by the `LRAClient`. Certainly, this is an area which could be in a more general execution module further extended with, e.g., the sequential configuration or the LRA nesting.

5.6.3 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.

In the LRA service is this problem addressed by the `api-gateway` service which functions as a proxy for the `lra-coordinator` that handles the saga execution. This approach does not affect the Narayana LRA management. The only change from the traditional processing is that series of LRA actions are performed by the `api-gateway` instead of the initiating service. This allows the service to be immediately available for the subsequent user requests and to scale the LRA processing independently from the application services that utilize the saga actions execution.

Single point of failure

By the time of this writing, the LRA coordinator represents a single point of failure for the LRA processing as 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.

5.7 Used technologies

This section provides a point of reference that summarizes the technologies applied in all of the above described quickstart examples.

5.7.1 Microservices platforms

Spring Boot

Spring Boot is a framework built on top of the Spring Framework [86]. Its main focus is on the creation of standalone runnable applications that are easily employed in microservices architectures. It favors

convention over configuration to allow the usage of Spring features with a little of the Spring configuration [77].

The Spring Boot platform is composed into aggregate modules known as *starters*. The starter is a dependency descriptor which contains dependencies that are required to provide some functionality to the application. In order to use Spring Boot, the `spring-boot-starter` core module must be incorporated. Other useful modules, for instance, `spring-boot-starter-web` or `spring-boot-starter-data-jpa` can be provided to support the microservice adoption. Starters can be packaged with the application using Maven or Gradle build configurations.

Another responsibility of this framework is the packaging of the application. The preferable way for the microservice is to create an executable fat Java archive (jar) which contains all of the application dependencies. The application can then be simply executed by a `java -jar` command.

A spring Boot microservice must fulfill two requirements – it must follow a Maven layout convention and it must provide an entry point. This can be any class annotated with the `@EnableAutoConfiguration` annotation that starts the Spring Boot context.

Spring provides various ways for exposing available services. Registration servers like Eureka or Consul which are integrated within the Spring cloud [80] can be used as the service discovery mechanisms. For a manual approach, the microservices functionality can be exposed through the RESTful API.

WildFly Swarm

WildFly Swarm is the Red Hat microservices initiative designed to enable deconstructing the WildFly application server and pasting just enough of it back together with the application to create a self-contained executable jar [87]. It is compatible with the MicroProfile project.

The traditional Java Enterprise Edition (EE) approach follows the development of the application and its subsequent deployment to the application server which includes necessary dependencies which the application requires to run. On the other hand, WildFly Swarm creates a fat Java archive which packages all needed dependencies. This emu-

lates the packaging of only requisite parts of the application server. The resulting jar is a standalone runnable Java application which can be executed by the `java -jar` command. It also provides Maven and Gradle plugins to ease the development of Swarm applications.

The default fat jar (also called the uberjar) contains the user application and the needed parts of the WildFly server. Swarm also supports the packaging of the necessary server parts separately from the application. This method is known as the hollow jar and is particularly useful in the containerized environment as the server may be placed in the lower layers that do not require frequent rebuilds.

The individual server parts are being delivered in packages named fractions. The fraction represents a precise selection of capabilities that can be included in the application. It may denote the exact WildFly subsystem as JAX-RS or CDI, or a more sophisticated set of facilities to provide some additional functionality like Red Hat Single Sign-On (RHSSO).

5.7.2 Docker

Docker is an open source container platform designed to make it easier to build, secure and manage the widest array of applications from development to production both on premises and in the cloud [75]. Docker containers allow applications to run on top of the kernel services provided by the hosting system which considerably affects the application performance. However, it still builds containers on top of the generalized interface which warrants straightforward portability between different machines.

A container image is a lightweight, stand-alone, executable package of a piece of software that includes everything needed to run it – code, runtime, system tools, libraries or settings [75]. All docker containers that run on the same machine share the kernel services of the host. The images are built on the concept of layers. The layer provides an abstraction to share common filesystems, configuration and other data that can be reused by several docker containers.

Containers isolate applications from the operating system they are running on and also provide the separation from other docker containers running concurrently on the same computer. Instead of virtual machines which provide similar functionality, Docker virtualizes the

operating system, not the hardware. Docker provides abstraction at the application level.

Docker as a tool is targeted for the simple utilization. It provides a unified environment for both developers and administrators supporting the DevOps (development and operations) practices. Particularly, developers profit from portable code that is able to run on almost any operating platform supporting Docker[2], while operations gain visibility and management services from comprehensive control panel covering all containerized applications.

Docker compose

Docker compose is a tool for the defining and running multi-container Docker applications [76]. It allows to specify configuration for several containers as YAML file that can be passed as an argument to the command line tool to run all services with a single command.

The YAML configuration includes the same options that are available as switches to native Docker commands. Additionally, it also provides functionality to orchestrate the specified services once they are started through the `docker-compose` command line utility. These functions include the network definition which allows easy addressing and the location transparency of employed containers, the preservation of the container volumes, recreation of only changed containers or the specification of environment variables directly in the configuration file.

Docker compose represents a simple way to provide container automation that may ease development and support of the continuous delivery (CD) and integration (CI) pipelines.

5.7.3 Containerization platforms

OpenShift platform

Red Hat OpenShift is an open source container application platform that brings Docker and Kubernetes to the enterprise [65] which is generally built on top of the Red Hat Enterprise Linux. It provides

2. e.g., the image built on AMD cannot always run on Intel processors

deployment, management and monitoring capabilities of the containerized software. OpenShift provides automation in the cloud environment that enables simple development workflow including easy provisioning, building, and deployment of enterprise applications allowing faster delivery to end customers.

The platform provides extensive set of features like self-service maintenance, polyglot (language independent) application support, container-based environment and the automation of application builds, scaling and health management. It can also administer persistence, the application centric networking and multiple interaction models, e.g., command line tools or the web console.

OpenShift is provided in several variants. The upstream open source community project is OpenShift Origin which is a distribution of Kubernetes optimized for continuous application development and multi-tenant deployment [65]. On top of the Kubernetes platform, Origin provides the developer and operations centric tooling, security, logging or pipelining and many other capabilities. Origin is also available as the all in one virtual machine called Minishift which utilizes a local single-node OpenShift cluster targeted for the local development.

The second alternative is the OpenShift Online. Currently distributed in version 3, it serves as a Red Hat multi-tenant public cloud application development and hosting service. OpenShift Online provides an integrated environment that allows developers to focus on the application development instead of its management through the set of facilities like source-to-image builds eliminating the Dockerfiles creation, one click deployments through git hooks, automatic scaling according to the traffic and the integration with many Integrated Development Environments (IDEs).

OpenShift Dedicated offers a private, highly available OpenShift cluster provided as a cloud service managed by Red Hat that is dedicated to a single customer (single-tenant). It can be hosted on public cloud services platforms like Amazon Web Services (AWS) or the Google Cloud Platform (GCP).

The Red Hat OpenShift Container Platform provides a solution to operate OpenShift on the customer's own infrastructure – in the data center or in the private cloud. It is based on the same code base as the OpenShift Dedicated.

The last OpenShift variant it the OpenShift.io. It provides an open online end-to-end development environment for planning, creating and deploying hybrid cloud services [88]. It supports an integrated approach to DevOps, including tools as one-click container management, machine learning system and the integration with many practical projects like fabric8 or Eclipse Che.

Kubernetes

Kubernetes is an open source project providing automation, scaling and management of containerized applications [66]. It groups the application containers into logical units for easier management and discovery.

The features of Kubernetes include the service discovery, load balancing, automatic container placement or the self-healing for the automated failure recovery and rollbacks. It also manages the storage orchestration, scaling of containers, secrets, container configuration and batch capabilities.

Kubernetes platform is suitable and portable to any cloud environment involving public, private, hybrid clouds and the multi-cloud. It allows application containers to be run in the clusters of physical or virtual machines. Kubernetes is not a traditional PaaS (Platform as a Service) solution but it provides the platform that many PaaS systems build upon, e.g., OpenShift or Deis.

5.8 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.

As these projects represent only quickstart examples which were not adjusted for performance problems, this test serves only as a reference to inspect the behavior of the simple starting application. It does not correspond to the real production environment, but it still references some of possible performance improvement points present in each inspected framework.

The test has been run in the cloud computing platform Digital Ocean [89]. The virtual machine specification:

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

The performance test is using the PerfCake [90] testing framework to generate requested number of order requests. As all 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 (Figure 5.5).

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 later test because of various problems discussed in following sections. The 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.

Axon service

The major performance problem of this example was the artificial one second delay placed on the shipment and invoice response receival 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 was synchronous REST calls for the inter-service command dispatching over distributed command bus. This issue has

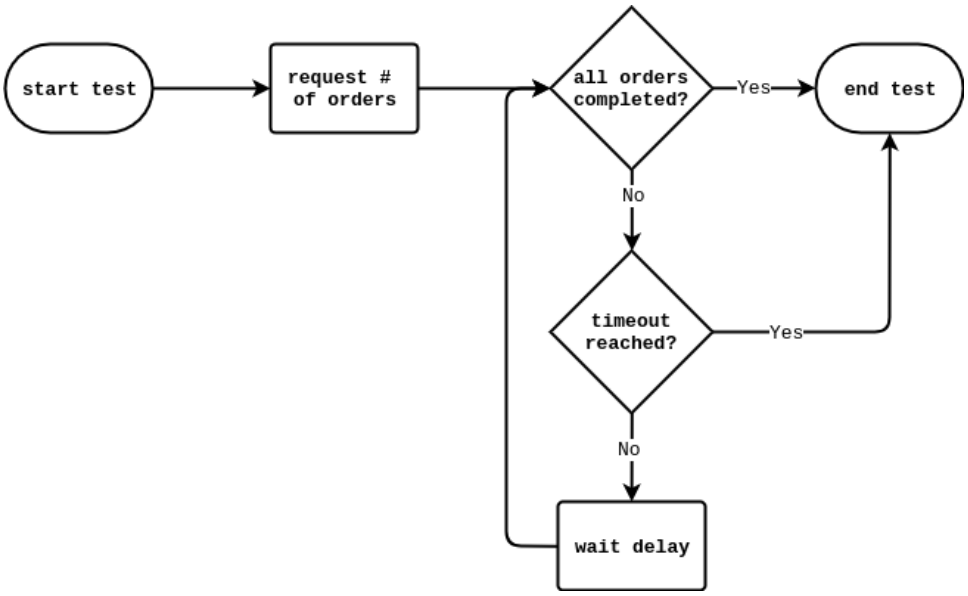

Figure 5.5: Saga performance test execution

been reported and it was still being 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 request has been created in 58 seconds, but because of the above mentioned issue the final count reached 2000 orders. The 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 19791 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, the scenario 2 in some cases produced an exception in 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 minutes and 56 seconds.

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

Tables 5.1 and 5.2 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 5.2 represent the best result of three performed test executions in scenario 2.

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 5.1: Performance test – scenario 1 (1 000 requests, 10 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

Table 5.2: Performance test – scenario 2 (10 000 requests, 100 threads)

6 LRA executor extension

One of the main goals of this thesis was the investigation of how the saga pattern implementation in the Narayana LRA can be updated for the better support of the LRA utilization in reactive microservices environments. The *LRA executor extension* has been created as a proof of concept implementation of the proposed solution which extends the saga orchestration capabilities of the LRA coordinator with the saga execution according to the specified user definition.

6.1 Motivation

With the current LRA implementation, the LRA coordinator is an orchestrating service that manages the saga processing spanning multiple user services. When the service wants to start a new saga, it contacts the coordinator which creates and starts a new LRA and returns its identification (an URL combining coordinator address and transaction object unique identifier) to the initiating service. This service then begins the saga execution by invoking other services that perform individual operations passing them the saga identification in the HTTP header. The participant upon invocation contacts the coordinator to enlist itself within the received saga. After the processing is completed or an error occurs, the coordinator is requested by any (commonly initiating) service that knows the saga identification to complete or compensate the LRA execution.

This approach is suitably employable in any application with a minimal impact on the existing code base. The coordinator ensures that the saga guarantee is preserved – either all enlisted participants complete their operations successfully or their compensating actions are invoked. It does not influence the actual saga execution which is in full control of the initiating and other included services.

The current LRA implementation expects at some point a certain system service to be able to make a decision of how the saga execution should be finished. This means that the system needs to track the saga execution in one or more of its services. The LRA coordinator maintains the information only about enlisted participants and has no further knowledge about the performed saga semantics. The LRA

then commonly consists of a sequence of service invocations and the finalizing outcome decision depending on collected responses.

Certainly, this processing is something that can be implemented by the user services[1], but as it represents an expected use case in many applications, the *LRA executor extension* has been created to present a feature of the LRA execution on the user's behalf. It extends the coordinator orchestrating capabilities with the actual saga execution based on the user description which allows developers to focus on the saga business tasks rather than on its processing.

Another advantage of this approach is the decrease of the number of exchanged messages required for the saga processing. As the coordinator / executor receives all participating services in the user's saga definition, it is possible to enlist each participant within the LRA prior to its actual invocation. This means that the participating service is not required to contact the coordinator. It also allows the coordinator to manage participant invocations uniformly (number of retries, failure policies). After the coordinator collects responses from all specified participants, it makes the decision of whether to complete or compensate the LRA and optionally informs the initiating service about the result.

This approach is definitely not appropriate for every user saga scenario. For instance, if the saga should be formed dynamically depending on certain intermediate results, some services may be only occasionally enlisted within the saga execution. However, long running transactions have generally well-defined purpose which is why the *LRA executor extension* may be effective in many applications utilizing the Narayana LRA implementation.

6.2 Design

The *LRA executor extension* project consists of two main modules – the `lra-definitions` and the `lra-executor`. It is designed in a general way to ensure the portability and the simplicity of further extensions. In particular, users are encouraged to customize most of the classes to accommodate executor to their specific communication requirements.

1. e. g. LRA service or a new Saga enterprise integration pattern recently introduced in Apache Camel [91]

These modules do not have any direct dependency on the Narayana project. However, this proof of concept includes custom code base changes that adjust the Narayana `lra-coordinator` project to the executor capabilities providing the implementation based on the REST protocol.

The `lra-definitions` module aims to provide a uniform interface for the user LRA definition. This definition is intended to fully describe the saga execution with all necessary information that the initiator needs to propagate to participating services. The definition can be supplied to the executing service as the Java class or as a provided JSON representation.

This module also maintains a set of builder classes that focus on the ease of the saga development experience in a similar way as it is implemented in the Eventuate Tram project (section 4.7.3). It provides a fluent API that promotes the readability and the presentation of the LRA Domain Specific Language (DSL).

The `lra-executor` module represents a simple executor definition and both synchronous and asynchronous default processing implementation. Both execution methods consume the LRA definition as an argument. The default implementation executes individual LRA actions in the same order as they were specified in the LRA definition.

The executor has been designed in a way that is intended for an extension. This mainly includes the action invocation implementation that is associated with communication conventions utilized in targeted applications.

The current design also requires that the executor implementation specifies how the LRA is started as the existing LRA orchestration logic is available only in the Narayana's `lra-coordinator` module. The inclusion of this module in the *LRA executor extension* would result into cyclic dependencies as the `lra-coordinator` has been modified as the presently only executor implementation. This problem can be easily resolved by a refactoring of the `lra-coordinator` module that would extract the LRA orchestration from the LRA REST processing to allow further extensions with different protocols. The current integration implementation is described in section 6.4.

6.3 Implementation

This section describes the implementation of two modules included in the *LRA executor extension*: LRA definitions and the LRA executor. Each component contains a set of unit tests that verify the solution applicability and also present basic usage examples. Figure 6.1 provides an overview of the *LRA executor extension* utilization together with its integration in the Narayana LRA project which is in detail discussed in the following section. Class diagrams of individual modules are available for reference in the Appendix E.

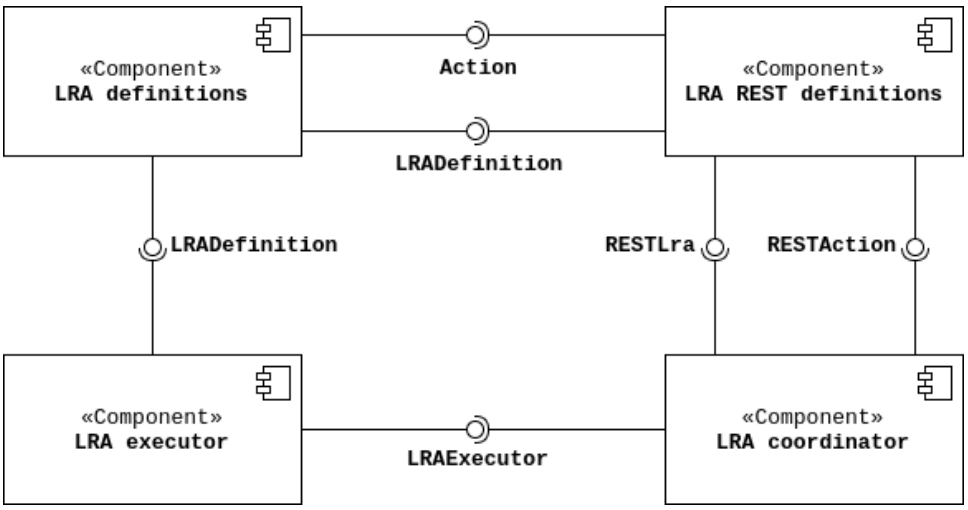

Figure 6.1: Narayana LRA executor extension component diagram

6.3.1 LRA definitions

The essential part of this module is the `LRADefinition` interface. It encapsulates all necessary information describing the saga execution. The full list of included properties is available in the Appendix E.

The `lra-definitions` module contains one implementation of this interface called the `LRADefinitionImpl` which represents the JSON representation of the `LRADefinition` interface. This is required as the

automatic JSON object creation can be determined only from the Java classes, not interfaces. This representation has been included as the JSON format is commonly utilized in the microservices applications, but certainly it is not required in every environment (e.g., in-JVM implementation).

The main concerned segment of the `LRADefintion` interface for the saga executor is the list of `Action` objects that define individual saga participant invocations. The `Action` represents a functional interface[2] with one included method `ActionResult invoke(LRAData lraData)`. This interface is intended to be implemented by the respective executor implementation to define the communication procedures required to invoke the participant (e. g. REST calls or JMS messages). It consumes an `LRAData` object that encapsulates the optional saga data provided by the initiating service which are to be included in invocations. The returned `ActionResult` class provides information about the outcome of the invocation with the potential failure description.

6.3.2 LRA executor

The main definition of the executor capabilities is defined by the `LRAExecutor` interface. This interface contains three methods (the full API specification is available in the Appendix E):

- **LRAResult executeLRA(LRADefinition)** – This method starts and executes the LRA according to the provided definition. It will return the result once all actions are processed.
- **Future<LRAResult> executeLRAAsync(LRADefinition)** – Similarly as above, but the invocation of this method is asynchronous. It returns a `Future<LRAResult>` which allows users to optionally wait for the end of the execution.
- **URL startLRA(LRADefinition)** – This method starts a new LRA based on the information provided in the definition. This method can be removed once the *LRA executor extension* is included in the Narayana project.

2. an interface with exactly one abstract method

The `AbstractLRAExecutor` class represents a default abstract implementation of the `LRAExecutor` interface. Both execution methods process provided actions in the order they are defined in the received LRA definition. This class provides three abstract protected methods:

- **ActionResult executeAction(Action, LRAData)** – By default, it represents a wrapper around the action invocation. Users can override this method to customize the action processing.
- **void completeLRA(LRAResult)** – Signalizes the successful LRA execution.
- **void compensateLRA(LRAResult)** – Indicates that the LRA needs to be compensated.

The `LRAResult` class contains all information about the saga execution, namely, the LRA id, definition and optionally the failure cause.

The current implementation functions exclusively as the LRA executor. The actual orchestration and tracking are left to the implementations of the `LRAExecutor` interface. The following section describes how it is integrated on top of the current LRA implementation.

6.4 Narayana LRA integration

The *LRA executor extension* solution provided in this thesis has been integrated and tested with the Narayana transaction manager 5.8.1.Final.

A new module `lra-rest-definitions` provides a REST extension of the LRA definitions project. The `RESTLra` interface declares the REST `LRADefinition`. It additionally specifies only the callback endpoint which can be optionally notified about LRA completion. The implementation of the action interface is included in the `RESTAction` class. It is defined by the participant URL and an optional callback resource address which specifies where the LRA operations handlers are expected. If the callback address is not provided, the participant target address is used[3]. This module also provides a custom builder

3. in future, it may be possible to add support for the retrieval of this information from LRA annotations upon the participant invocation

class and the JSON representation for both the LRA definition and the action respectively.

The executor implementation is contained in the `lra-coordinator` module which contains the main concerned class `RESTLRAExecutor`. It represents the `LRAExecutor` implementation based on the REST protocol. Internally, it also utilizes the Narayana `LRAService` class to manage the saga lifecycle (start, complete and compensate of the LRA).

The `RESTLRAExecutor` provides a capability to enlist saga participants prior to the invocation. This allows to lose the requirement of the enlisting call from participants and therefore the need to declare LRA processing on their endpoints. However, participants are still expected to expose endpoints for LRA handlers (e. g. complete, compensate). For this reason, the `lra-rest-definitions` module contains a `ParticipantCallback` interface that declares the required methods for the RestEasy JAX-RS resource. The callback URL may be provided as a part of the `RESTAction` definition.

The integration also includes some minor modifications to ease the use of the definition within the coordinator. This mainly concerns a separate starting endpoint in the `lra-coordinator` which consumes the `RESTLra` and the corresponding client method declaration in the `NarayanaLRAClient`.

6.5 LRA executor quickstart

To demonstrate asynchronous LRA execution capabilities of the proposed solution, we created a sample quickstart application *LRA executor quickstart* that utilizes the LRA coordinator with incorporated *LRA executor extension*.

The example consists of four microservices: a saga service, two participant services, and the LRA coordinator. The execution simulates artificial asynchronous processing. The LRA is initiated on user request by the saga service which sends the predefined LRA definition to the coordinator and immediately returns a response. The coordinator processes the definition by two invocations of respective participants and completes the LRA (processing can be monitored in the coordinator's output). Both participants only wait for specified time periods to simulate asynchronous tasks.

For the ease of use, the LRA coordinator is provided as a docker image under `xstefank/lra-coordinator:5.8.1.Final` tag available on Docker hub. The example is accommodated to be run with the Docker compose or OpenShift configuration.

As the LRA start request to the saga service returns a response immediately, it allows the saga service to be instantly available for subsequent requests. This extraction of the LRA execution from the invoking service enables the microservices system to stay responsive in a way aligned with the reactive principles [3].

6.6 Future work

The current implementation of the *LRA executor extension*, discussed in previous sections, provides a working proof of concept of potential Narayana LRA execution capabilities. However, there are still some tasks that need to be addressed before it can be included in the project.

The first assignment of the executor inclusion would be the refactoring of the current LRA code base to extract the orchestration logic from the LRA coordinator. This will allow to employ different communication patterns and protocols for the LRA processing on top of the same service layer.

Apart from the refactoring, the main problem that needs to be addressed is the recovery capabilities of the executor. If the executor fails in the middle of the execution, there is currently no failure handling that may allow it to continue processing once it is recovered. One possible solution can be implemented by persisting the LRA definition before the execution starts and also by marking each participant invocation start and completion.

As a part of the failure handling, it may also be useful to allow developers to declare a strategy that would be applied in case of the participant failure. These strategies can be mapped exactly to saga recovery modes.

Another issue that may present an improvement point is that currently, the default implementation executes actions sequentially in the same order as they are specified in the definition. However, when the actions do not depend on each other, it may be effective to execute them in parallel. This information can be included directly in the LRA

definition to permit users to specify the processing mode for separated parts of the saga execution.

7 Conclusion

This thesis had three main objectives – to investigate the use of transaction processing in the reactive microservices environment, to propose an asynchronous saga execution solution for the Narayana transaction manager and to present an example quickstart application utilizing the proposed solution.

7.1 Saga pattern research

The initial study covered the examination of the basic transactions concepts and the description of the well-known consensus protocols that can be currently utilized for the distributed transaction commit problem in modern architectures. Furthermore, it provided a detailed description of the microservices architectural style and the application of the reactive ideology in this environment.

The Saga pattern [4] methodology has been introduced as an alternative approach to the traditional transactional processing. It presented that by the relaxation of ACID requirements, the distributed system can provide availability instead of consistency guarantees which is commonly desirable in reactive architectures. Because of its non-locking nature and the simplicity of its programming model, it was selected as the primary research subject for the utilization of the transaction management in reactive microservices applications.

The current development support of the saga pattern for the production environment was explored through the implementation of an order processing microservices application that utilized the saga processing in four Java frameworks – Axon, Eventuate ES, Eventuate Tram and Narayana LRA. Every example provides a straightforward quickstart introduction to the saga configuration, definition, and execution in each respective framework. Almost all examples have already been recognized in the community for their contribution.

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 interesting discoveries of possible improvement points in the saga processing of individual frameworks. In total, there were

two issues reported to the Axon framework, one problem reported to the Eventuate ES and one feature request submitted to the Eventuate Tram framework.

7.2 Narayana asynchronous LRA execution

The prepared Narayana LRA quickstart microservices application introduced a simple API gateway that abstracted business services from the saga processing by the asynchronous execution capability based on the service provided LRA definition. This processing implementation presented that Narayana LRA does not influence how the user structures or employs the LRA realization, provided that coordinator is available.

However, the saga definition and execution logic were required to be provided by the application employing LRA. As the asynchronous saga execution represents an expected use case in many reactive applications, the *LRA executor extension* was created to extract this saga processing implementation and introduce it as an extension to the current LRA coordination capabilities.

The project was structured into two logical parts – the LRA definitions and the LRA executor.

The LRA definitions module provided uniform LRA saga definition capabilities that fully described the LRA with all necessary information required for its successful execution. The implementation provided LRA definition in the form of the Java class or the JSON format, but its universal design may promote further expansions utilizing various data formats in the future.

The LRA executor represents a consistent method of the LRA processing according to the specified definition. The specification supports both synchronous and asynchronous LRA invocation based on the strategy defined by the implementation. Due to its general design, implementations are encouraged to adjust the execution to their particular LRA use cases.

Both modules abstract the communication methods the executor employs to contact LRA participants in the form of actions which specify how the individual invocations are performed. The current proof of concept was based on the REST protocol.

7.3 Contributions

This thesis provides a general overview of approaches that are available for the transaction processing in the microservices environment. With regards to the reactive architectures, it in detail describes the saga pattern as a suitable solution for the asynchronous transaction execution in distributed applications.

The first important contribution is the research of the saga solutions available for the Java platform. The created quickstart examples function as a stable starting point for the saga definition, configuration, and execution demonstration in each respective framework. This thesis additionally also contributes with the detailed comparison and the summary of advantages and disadvantages regarding saga processing in implemented examples. The produced performance test revealed several problems that have been properly reported to the respective frameworks main developers.

The main contribution is concluded with the implementation of *LRA executor extension*. This project provides a proof-of-concept prototype that illustrates the LRA execution capabilities that are built on top of the Narayana LRA implementation. The project is also promoted by a created *LRA executor quickstart* application that presents the asynchronous LRA processing.

The full list of projects produced within this thesis:

- *Axon service* – 2 reported issues
- *Eventuate service* – 1 reported issue
- *Eventuate Tram service* – 1 feature request
- *LRA service*
- *Saga example performance test*
- *LRA executor extension*
- *LRA executor quickstart*
- *Narayana LRA integration*

All of these projects are available as open source contributions.

7.4 Future tasks

The *LRA executor extension* provides a stable base prototype implementation of the LRA synchronous and asynchronous processing based on the user specified definition. Even if it can be applied as the extension on top of the current Narayana LRA implementation, it may be beneficial to adjust the Narayana code base to include it directly in the future.

As the Narayana LRA specification is still in the pre-released state [9], the inclusion of this project would require a proper specification, implementation and API finalization, and more comprehensive testing support. Additionally, due to the design of the executor, there are many possibilities of how it can be extended including support of new LRA definition formats, processing strategies and most importantly by the promotion of new communication protocols facilitating the Narayana LRA execution in microservices environments.

Bibliography

[1] M. Little, J. Maron, and G. Pavlik, *Java transaction processing*. Prentice Hall, 2004.

[2] S. Newman, *Building Microservices*, 1st ed. O'Reilly Media, Inc., 2015.

[3] J. Bonér, D. Farley, R. Kuhn, and M. Thompson, "Reactive manifesto," 2018. [Online]. Available: https://www.reactivemanifesto.org/

[4] H. Garcia-Molina and K. Salem, "Sagas," *ACM SIGMOD Record*, vol. 16, no. 3, pp. 249–259, 1987.

[5] Narayana, "Narayana," 2018. [Online]. Available: http://narayana.io/

[6] A. framework, "Axon framework," 2017. [Online]. Available: http://www.axonframework.org/

[7] Eventuate.io, "Eventuate.io," 2017. [Online]. Available: http://eventuate.io/

[8] Eventuate, Inc, "Eventuate tram," 2018. [Online]. Available: http://eventuate.io/abouteventuatetram.html

[9] Narayana, "Narayana LRA," 2017. [Online]. Available: https://github.com/eclipse/microprofile-sandbox/tree/master/proposals/0009-LRA

[10] GNU / FSF, "GNU Lesser General Public License, version 2.1," 1999. [Online]. Available: https://www.gnu.org/licenses/old-licenses/lgpl-2.1.en.html

[11] M. Musgrove, "Narayana + wildfly," 2015. [Online]. Available: https://developer.jboss.org/servlet/JiveServlet/download/53044-3-129391/jbug-brno-transactions.pdf

[12] T. Haerder and A. Reuter, “Principles of transaction-oriented database recovery,” *ACM Computing Surveys*, vol. 15, no. 4, pp. 287–317, 1983.

[13] E. Brewer, “CAP twelve years later: How the "rules" have changed,” *Computer*, vol. 45, no. 2, pp. 23–29, feb 2012.

[14] M. Kleppmann, *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*. O’Reilly Media, 2017.

[15] ISO, “Information Technology - Database Language SQL,” International Organization for Standardization, Maynard, Massachusetts, Standard, july 1992. [Online]. Available: http://www.contrib.andrew.cmu.edu/~shadow/sql/sql1992.txt

[16] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O’Neil, and P. O’Neil, “A critique of ANSI SQL isolation levels,” *ACM SIGMOD Record*, vol. 24, no. 2, pp. 1–10, 1995.

[17] M. Kleppmann, “Transactions: myths, surprises and opportunities,” 2018. [Online]. Available: https://martin.kleppmann.com/2015/09/26/transactions-at-strange-loop.html

[18] Sun Microsystems Inc., “Java Transaction API,” 2002. [Online]. Available: http://download.oracle.com/otn-pub/jcp/jta-1.1-spec-oth-JSpec/jta-1_1-spec.pdf

[19] Sun Microsystems Inc., “Java Transaction Service,” 1999. [Online]. Available: http://download.oracle.com/otn-pub/jcp/7309-jts-1.0-spec-oth-JSpec/jts1_0-spec.pdf

[20] X/Open Company Ltd., “Distributed Transaction Processing: The XA Specification,” X/Open CAE Specification, Dec. 1991. [Online]. Available: http://pubs.opengroup.org/onlinepubs/009680699/toc.pdf

[21] M. Richards, *Java transaction design strategies*, 1st ed. C4Media, 2006.

[22] Spring community, "Spring documentation/Transaction Management," 2018. [Online]. Available: https://docs.spring.io/spring/docs/current/spring-framework-reference/data-access.html#transaction

[23] A. D. Kshemkalyani and M. Singhal, *Distributed Computing: Principles, Algorithms, and Systems*. Cambridge University Press, 2008.

[24] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem," *ACM Transactions on Programming Languages and Systems*, vol. 4/3, pp. 382–401, July 1982. [Online]. Available: https://www.microsoft.com/en-us/research/publication/byzantine-generals-problem/

[25] D. Skeen, "A quorum-based commit protocol," Ithaca, NY, USA, Tech. Rep., 1982.

[26] I. Keidar and D. Dolev, "Increasing the resilience of distributed and replicated database systems," *J. Comput. Syst. Sci.*, vol. 57, no. 3, pp. 309–324, Dec. 1998. [Online]. Available: http://dx.doi.org/10.1006/jcss.1998.1566

[27] L. Lamport, "The part-time parliament," *ACM Trans. Comput. Syst.*, vol. 16, no. 2, pp. 133–169, May 1998. [Online]. Available: http://doi.acm.org/10.1145/279227.279229

[28] OASIS, "Web Services Atomic Transaction 1.2," 2018. [Online]. Available: http://docs.oasis-open.org/ws-tx/wstx-wsat-1.2-spec.html

[29] J. Gray and L. Lamport, "Consensus on transaction commit," *ACM Trans. Database Syst.*, vol. 31, no. 1, pp. 133–160, Mar. 2006. [Online]. Available: http://doi.acm.org/10.1145/1132863.1132867

[30] M. Štefanko, "Messaging providers in the SilverWare microservices platform," Bachelor's thesis, Masaryk University, Faculty of Informatics, Botanická 68a, Brno, Czech republic, 6 2016.

[31] A. Rotem-Gal-Oz, *SOA Patterns*. Manning Publications, 2012.

[32] J. Bonér, *Reactive microservices architecture*, 1st ed. O'Reilly Media, Inc., 2016.

[33] J. Lewis and M. Fowler, "Microservices," 2014. [Online]. Available: http://martinfowler.com/articles/microservices.html

[34] I. H. Sarker and K. Apu, "Mvc architecture driven design and implementation of java framework for developing desktop application," *International Journal of Hybrid Information Technology*, vol. 7, no. 5, pp. 317–322, 2014.

[35] E. Stump P.E., "All about learning curves," *Galorath Incorporated*, 2014.

[36] C. Richardson, "Pattern: Monolithic architecture," 2017. [Online]. Available: http://microservices.io/patterns/monolithic.html

[37] C. Richardson, "Introduction to Microservices | NGINX," 2015. [Online]. Available: https://www.nginx.com/blog/introduction-to-microservices/

[38] W. Hasselbring and G. Steinacker, "Microservice architectures for scalability, agility and reliability in e-commerce," *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, 2017.

[39] K. Lieberherr and I. Holland, "Assuring good style for object-oriented programs," *IEEE Software*, vol. 6, no. 5, pp. 38–48, 1989.

[40] R. C. Martin and M. Martin, *Agile principles, patterns, and practices in C#*, 1st ed. Prentice Hall, 2006.

[41] M. D. McIlroy, E. N. Pinson, and B. A. Tague, "Unix time-sharing system: Foreword," *Bell System Technical Journal*, vol. 57, no. 6, pp. 1899–1904, 1978.

[42] S. Newman, "Principles of Microservices," 2016. [Online]. Available: http://samnewman.io/talks/principles-of-microservices/

[43] E. Evans, *Domain-driven design*. Addison-Wesley, 2003.

[44] M. Conway, "Conway's law," 2018. [Online]. Available: http://www.melconway.com/Home/Conways_Law.html

[45] N. Busi, R. Gorrieri, C. Guidi, L. roberto, and G. Zavattaro, "Choreography and orchestration conformance for system design," *Lecture Notes in Computer Science*, p. 63–81, 2006.

[46] R. Dijkman and M. Dumas, "Service-oriented design: A multi-viewpoint approach," *International Journal of Cooperative Information Systems*, vol. 13, no. 04, pp. 337–368, dec 2004. [Online]. Available: https://doi.org/10.1142%2Fs0218843004001012

[47] K. Benghazi, M. Noguera, C. Rodríguez-Domínguez, A. B. Pelegrina, and J. L. Garrido, "Real-time web services orchestration and choreography," in *Proceedings of the 6th International Workshop on Enterprise & Organizational Modeling and Simulation*, ser. EOMAS '10. Aachen, Germany, Germany: CEUR-WS.org, 2010, pp. 142–153. [Online]. Available: http://dl.acm.org/citation.cfm?id=1866939.1866952

[48] V. Farcic, *The DevOps 2.0 Toolkit*. Packt Publishing Ltd., 2016, pp. 222–252.

[49] J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley, 2010, pp. 263–265.

[50] Swagger community, "Swagger framework," 2016. [Online]. Available: http://swagger.io/

[51] M. Fowler, "Humaneregistry," 2008. [Online]. Available: http://martinfowler.com/bliki/HumaneRegistry.html

[52] M. T. Nygard, *Release It!: Design and Deploy Production-Ready Software*. Pragmatic Bookshelf, 2018.

[53] C. Escoffier, "The reactive landscape," 2018. [Online]. Available: https://www.slideshare.net/RedHatDevelopers/the-reactive-landscape

[54] C. Escoffier, *Building Reactive Microservices in Java*, 1st ed. O'Reilly Media, Inc., 2017.

[55] RxJava community, "Rxjava," 2018. [Online]. Available: https://github.com/ReactiveX/RxJava

[56] C. Escoffier, "5 things to know about reactive programming," 2017. [Online]. Available: https://developers.redhat.com/blog/2017/06/30/5-things-to-know-about-reactive-programming/

[57] R. S. community, "Reactive streams," 2018. [Online]. Available: http://www.reactive-streams.org/

[58] A. Rotem-Gal-Oz, "Fallacies of distributed computing explained." [Online]. Available: http://www.rgoarchitects.com/Files/fallacies.pdf

[59] M. Fowler, "Microservice trade-offs," 2015. [Online]. Available: https://martinfowler.com/articles/microservice-trade-offs.html

[60] W. Vogels, "Eventually consistent," *Communications of the ACM*, vol. 52, no. 1, p. 40, jan 2009.

[61] S. Gilbert and N. Lynch, "Perspectives on the CAP theorem," *Computer*, vol. 45, no. 2, pp. 30–36, feb 2012.

[62] H. Robinson, "The cap faq," 2013. [Online]. Available: https://henryr.github.io/cap-faq/

[63] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *ACM SIGACT News*, vol. 33, no. 2, p. 51, jun 2002.

[64] D. Abadi, "Consistency tradeoffs in modern distributed database system design: CAP is only part of the story," *Computer*, vol. 45, no. 2, pp. 37–42, feb 2012.

[65] Red Hat, Inc., "OpenShift container application platform," 2018. [Online]. Available: https://www.openshift.com/

[66] Kubernetes, "Kubernetes," 2018. [Online]. Available: https://kubernetes.io/

[67] T. Clemson, "Testing strategies in a microservice architecture," 2014. [Online]. Available: https://martinfowler.com/articles/microservice-testing/

[68] J. Gray, "The transaction concept: Virtues and limitations (invited paper)," in *Proceedings of the Seventh International Conference on Very Large Data Bases - Volume 7*, ser. VLDB '81. VLDB Endowment, 1981, pp. 144–154. [Online]. Available: http://jimgray.azurewebsites.net/papers/thetransactionconcept.pdf

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

[70] P. Helland and D. Campbell, "Building on quicksand," *CoRR*, vol. abs/0909.1788, 2009. [Online]. Available: http://arxiv.org/abs/0909.1788

[71] U. R. Sharma, *Practical Microservices*, 1st ed. Packt Publishing Ltd., 2017.

[72] C. McCaffrey, "Applying the saga pattern," 2015. [Online]. Available: https://speakerdeck.com/caitiem20/applying-the-saga-pattern

[73] M. Štefanko, "Saga implementations comparison," 2017. [Online]. Available: http://jbossts.blogspot.cz/2017/12/saga-implementations-comparison.html

[74] R. J. Wieringa, *Design Science Methodology for Information Systems and Software Engineering*. Springer, 2014.

[75] "Docker," 2018. [Online]. Available: https://www.docker.com/

[76] "Docker compose," 2018. [Online]. Available: https://docs.docker.com/compose/

[77] Spring community, "Spring Boot," 2018. [Online]. Available: http://projects.spring.io/spring-boot/

[78] B. Porter, J. Zyl, and O. Lamy, "Maven," 2018. [Online]. Available: https://maven.apache.org/

[79] Axon community, "Axon framework reference guide," 2018. [Online]. Available: https://docs.axonframework.org/v/3.1/

[80] I. Pivotal Software, "Spring cloud," 2018. [Online]. Available: https://projects.spring.io/spring-cloud/

[81] Spring community, "Netflix service registration and discovery," 2018. [Online]. Available: https://spring.io/guides/gs/service-registration-and-discovery/

[82] N. Eureka, "Eureka at a glance," 2014. [Online]. Available: https://github.com/Netflix/eureka/wiki/Eureka-at-a-glance

[83] Pivotal software, "Rabbitmq," 2018. [Online]. Available: https://www.rabbitmq.com/

[84] Apache software foundation, "Apache zookeeper," 2018. [Online]. Available: https://zookeeper.apache.org/

[85] Gradle Inc., "Gradle," 2018. [Online]. Available: https://gradle.org/

[86] D. Woods, "Building Microservices with Spring Boot," 2016. [Online]. Available: http://www.infoq.com/articles/boot-microservices

[87] A. Gupta, "Wildfly swarm: Building microservices with java ee," 2018. [Online]. Available: http://blog.arungupta.me/wildfly-swarm-microservices-javaee/

[88] Red Hat, Inc., "Openshift.io," 2018. [Online]. Available: https://openshift.io/

[89] Digital Ocean, LLC, "Digital ocean," 2018. [Online]. Available: https://www.digitalocean.com/

[90] PerfCake community, "Perfcake," 2016. [Online]. Available: https://www.perfcake.org/

[91] N. Ferraro, "The saga pattern in apache camel," 2018. [Online]. Available: https://www.nicolaferraro.me/2018/04/25/saga-pattern-in-apache-camel/

A CQRS pattern

The Command Query Responsibility Segregation pattern describes the separation of the application domain into two distinct parts – the command model which is responsible for the application processing that changes the system state and the query model that provides information about the current system state to the user including various transformations or filtering.

This pattern extends a base given by the Command-query separation (CQS) which was introduced by Bertrand Meyer in his book *Object-Oriented Software Construction*. The main idea is to split the object's methods into two categories – queries which just return a value without changing the state, and commands that change the state of the object and do not return any result.

The commands are typically illustrated as simple objects identified by their respective names which are always expected to be in an imperative tense. They contain all necessary information that is needed to perform the request. Each command is delivered to a specified aggregate's command handler method that matches its identifier.

The query segment is responsible for the presentation of data to the end user. This typically represents methods that return data transfer objects (DTOs) or other data model entities. It can also provide several representations of the presented information or prevent users from multiple round trips by the data accumulation.

An aggregate is a main building block in the CQRS architecture. It represents a data entity that is always kept in a consistent state. The state can be changed by a reaction to the published event. Events are produced (applied) by the aggregate as a reaction to the received command.

To create more resilient systems, most of the CQRS frameworks also employ an event sourcing mechanism. Every published event is being persisted to the durable storage which allows the aggregate to rebuild its state in the case of failure. This can be done just by replaying (reapplying) of the already published events. It also allows to persist and redeliver events that cannot be transmitted to the aggregate during the failed state.

Another advantage of the domain separation is the performance increase. As both sides are allowed to scale up independently, the system can perform more operations concurrently.

The CQRS pattern may be particularly suitable for service (microservices) oriented systems. Because of their distributed nature, it is easy to separate concerns and allow customers to interact with different services depending on their performed activities.

To finalize, although the CQRS pattern has its benefits (ease of complexity and performance support), the practice showed that systems usually need to share the model between command and query sides. If the system is not built with this patter in mind, the CQRS may place very hard restrictions that may not be applicable in every application.

B Reactive Streams v1.0.2 API

org.reactivestreams.Publisher

```
public interface Publisher<T> {
    public void subscribe(Subscriber<? super T> s);
}
```

org.reactivestreams.Subscriber

```
public interface Subscriber<T> {
    public void onSubscribe(Subscription s);
    public void onNext(T t);
    public void onError(Throwable t);
    public void onComplete();
}
```

org.reactivestreams.Subscription

```
public interface Subscription {
    public void request(long n);
    public void cancel();
}
```

org.reactivestreams.Processor

```
public interface Processor<T, R>
    extends Subscriber<T>, Publisher<R> {
}
```

C Saga scenarios

C.1 CQRS

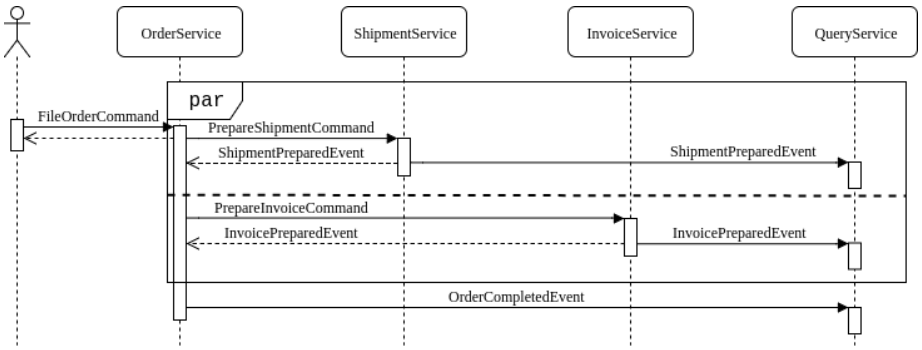

Figure C.1: CQRS saga example success

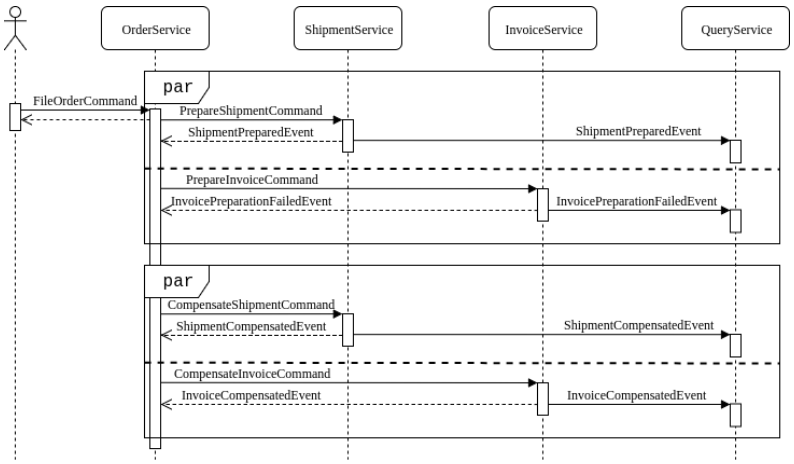

Figure C.2: CQRS saga example invoice failure

C.2 Eventuate Tram service

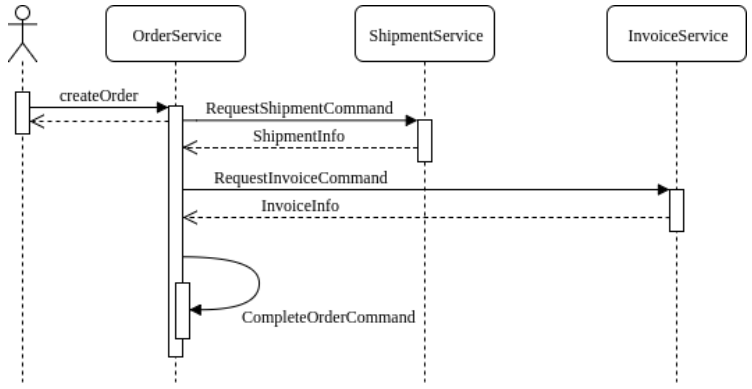

Figure C.3: Eventuate Tram service saga example success

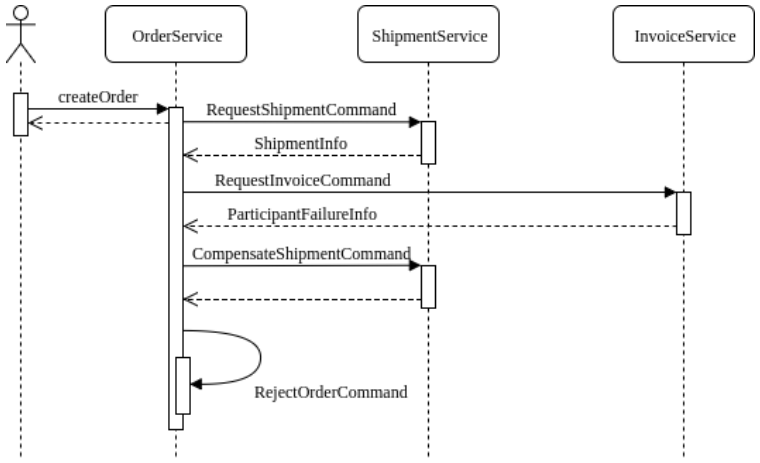

Figure C.4: Eventuate Tram service saga example invoice failure

C.3 LRA service

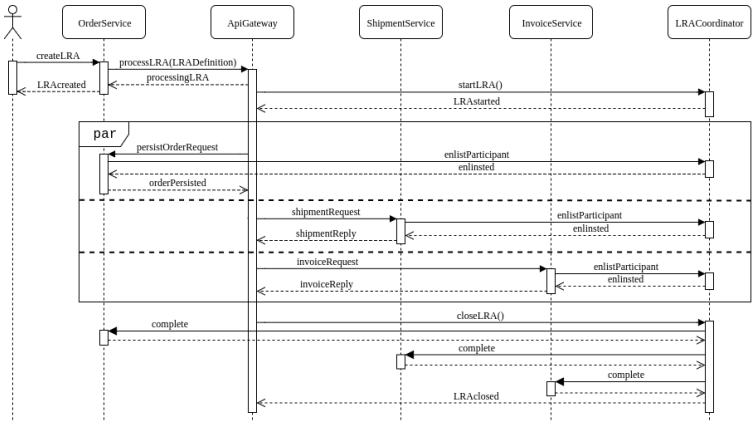

Figure C.5: LRA service saga example success

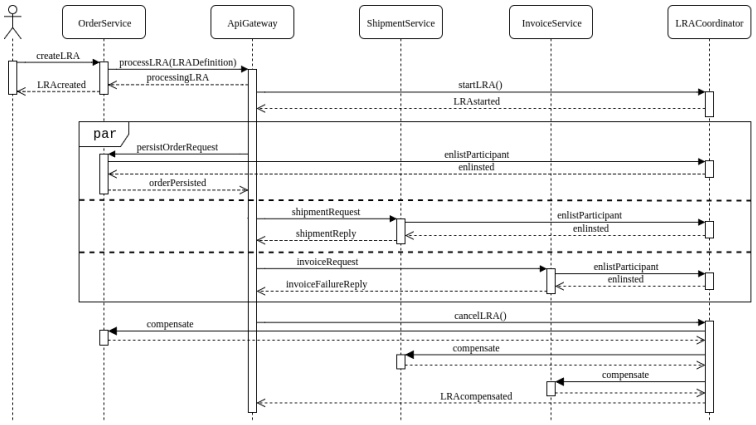

Figure C.6: LRA service saga example invoice failure

D Example applications public APIs

D.1 Axon service

Order service

```
POST /api/order
```

Query service

```
GET /api/orders
GET /api/order/{order_id}
GET /api/shipments
GET /api/shipment/{shipment_id}
GET /api/invoices
GET /api/invoice/{invoice_id}
```

D.2 Eventuate service

Order service

```
POST /api/order
POST /management/shipment
POST /management/shipment/fail
POST /management/shipment/compensation
POST /management/invoice
POST /management/invoice/fail
POST /management/invoice/compensation
```

Shipment service

```
POST /api/request
POST /api/compensate
```

Invoice service

```
POST /api/request
POST /api/compensate
```

Query service

```
GET /api/orders
GET /api/order/{order_id}
GET /api/shipments
GET /api/shipment/{shipment_id}
GET /api/invoices
GET /api/invoice/{invoice_id}
```

D.3 Eventuate Tram service

Order service

```
POST /api/order
GET  /api/orders
GET  /api/order/{order_id}
```

Shipment service

```
GET /api/shipments
GET /api/shipment/{shipment_id}
```

Invoice service

```
GET /api/invoices
GET /api/invoice/{invoice_id}
```

D.4 LRA service

Order service

```
POST /api/order
GET  /api/health
```

Shipment service

```
POST /api/request
PUT  /api/complete
PUT  /api/compensate
GET  /api/health
```

Invoice service

```
POST /api/request
PUT  /api/complete
PUT  /api/compensate
GET  /api/health
```

LRA coordinator

```
GET  /lra-coordinator
GET  /lra-coordinator/{LraId}
GET  /lra-coordinator/status/{LraId}
POST /lra-coordinator/start
PUT  /lra-coordinator/{LraId}/renew
GET  /lra-coordinator/{NestedLraId}/status
PUT  /lra-coordinator/{NestedLraId}/complete
PUT  /lra-coordinator/{NestedLraId}/compensate
PUT  /lra-coordinator/{NestedLraId}/forget
PUT  /lra-coordinator/{LraId}/close
PUT  /lra-coordinator/{LraId}/cancel
PUT  /lra-coordinator/{LraId}
PUT  /lra-coordinator/{LraId}/remove
GET  /api/health
GET  /lra-recovery-coordinator/{LraId}/{RecCoordId}
PUT  /lra-recovery-coordinator/{LraId}/{RecCoordId}
GET  /lra-recovery-coordinator/recovery
```

API gateway

```
PUT  /api/complete
PUT  /api/compensate
GET  /api/health
POST /api/lra
```

E LRA executor extension class diagrams

E.1 LRA definitions

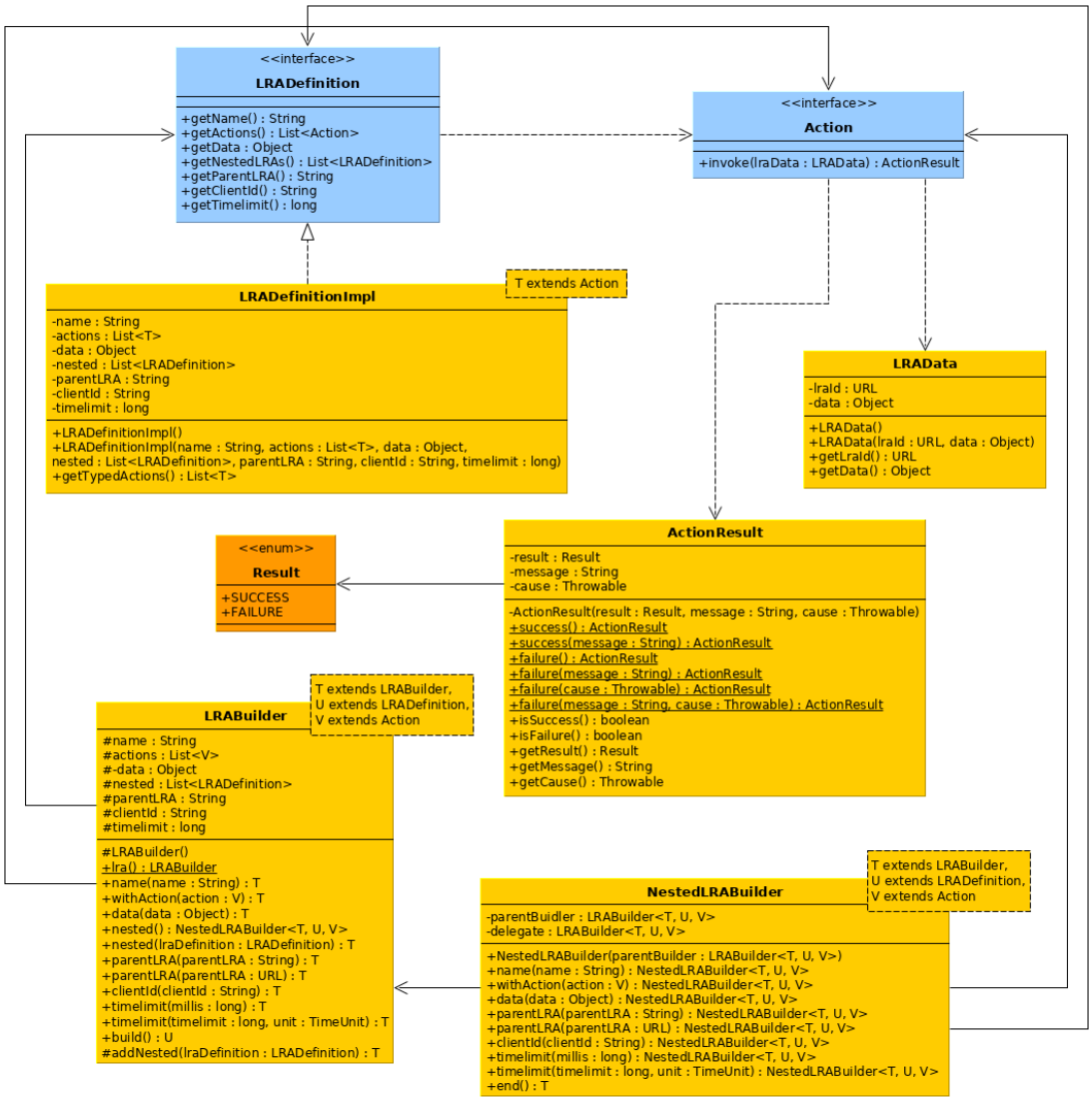

Figure E.1: LRA definitions class diagram

E.2 LRA executor

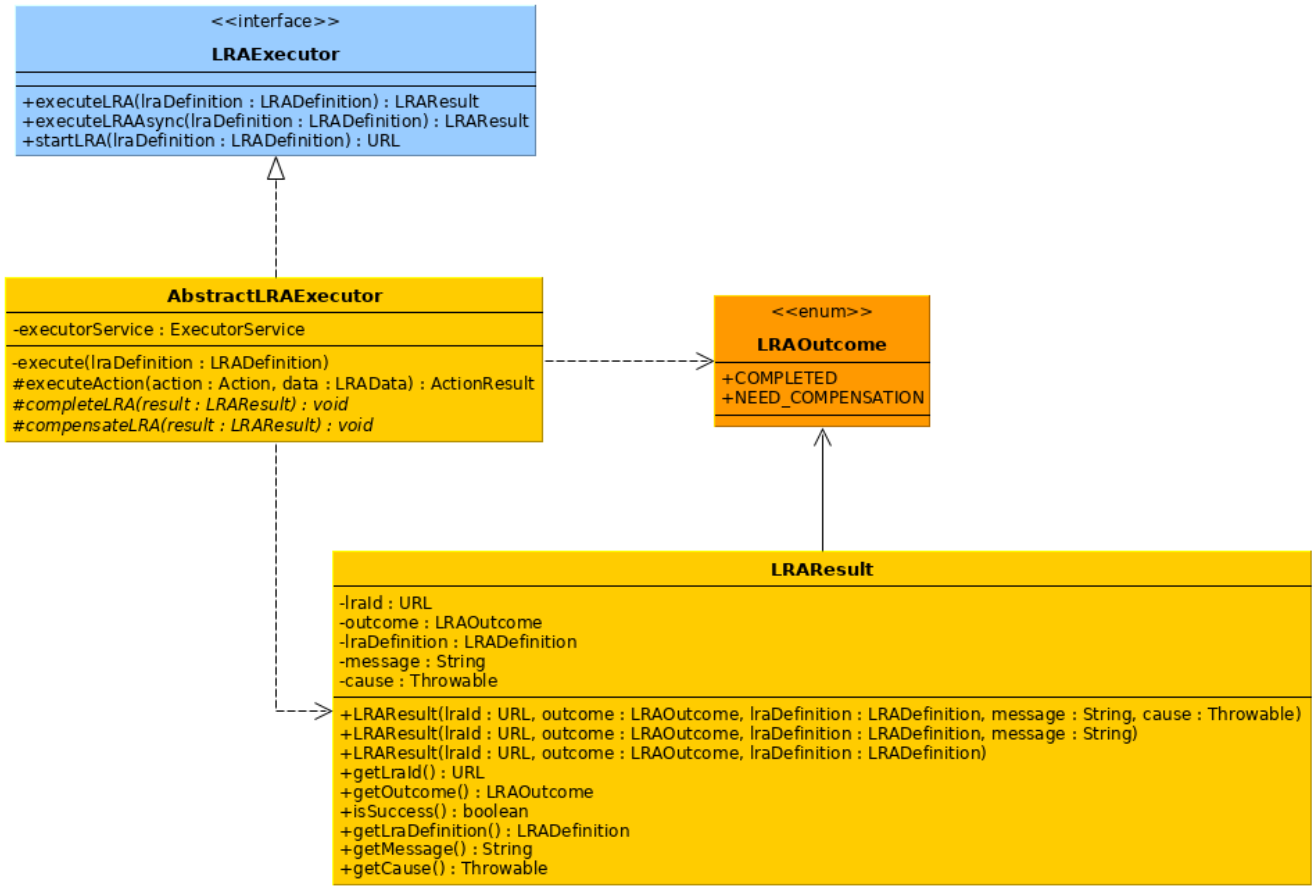

Figure E.2: LRA executor class diagram

E.3 Narayana integration

E.3.1 LRA REST definitions

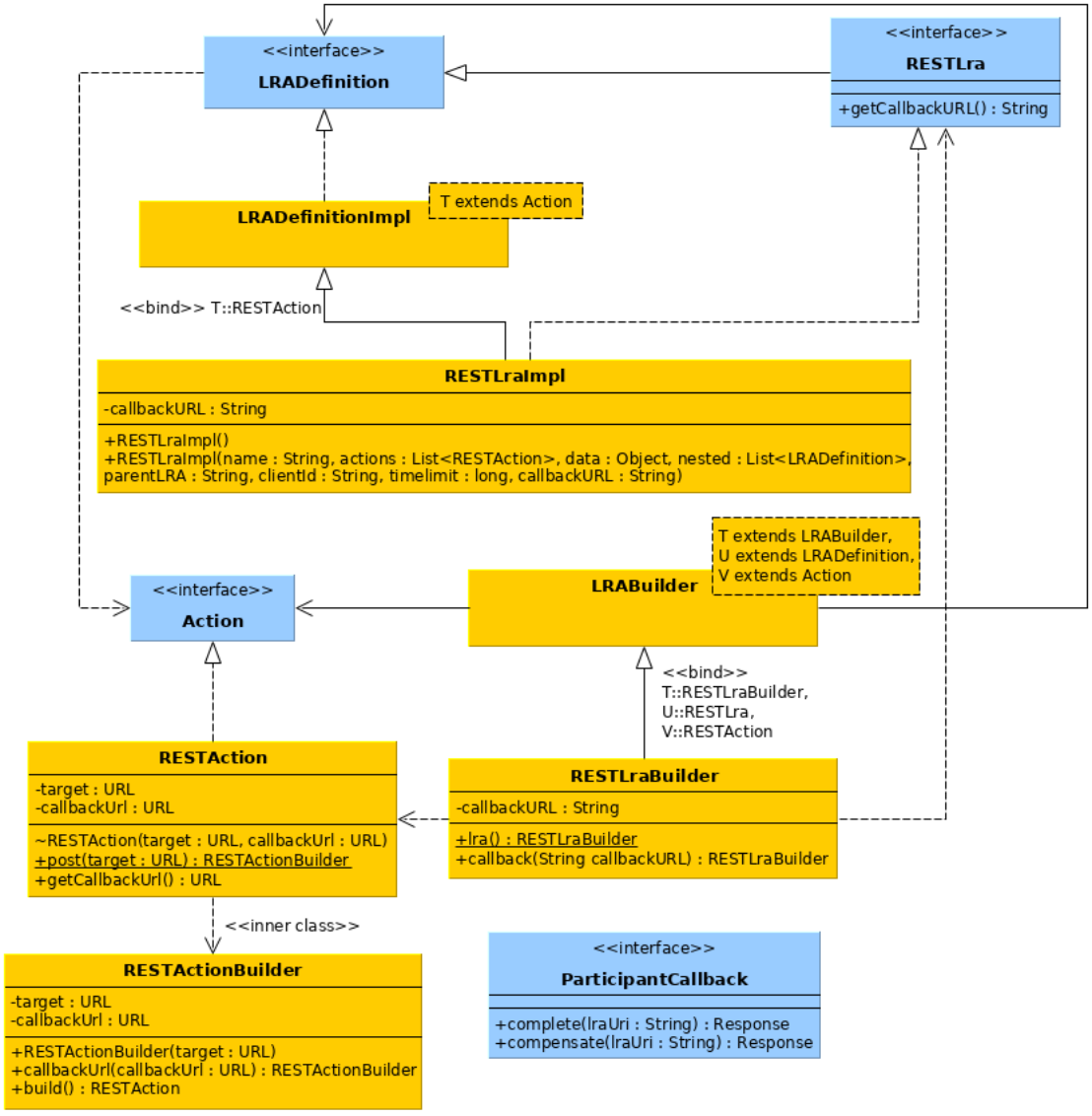

Figure E.3: LRA REST definitions class diagram

E.3.2 LRA coordinator

Required integration changes

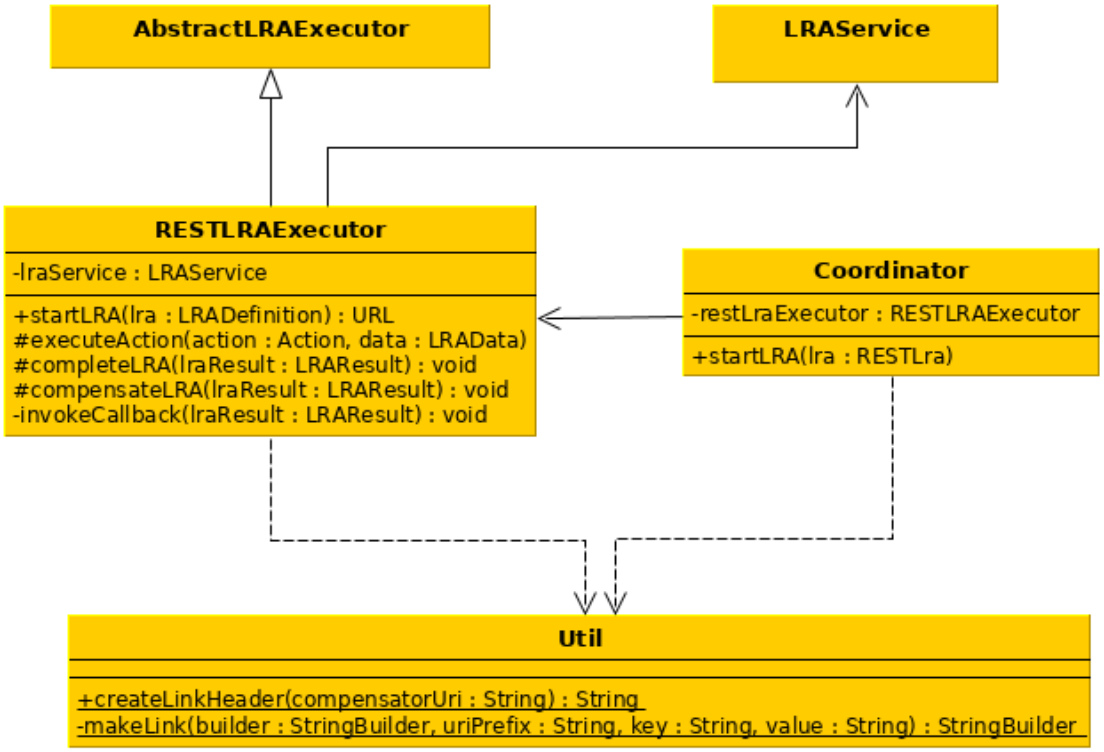

Figure E.4: LRA coordinator required integration changes