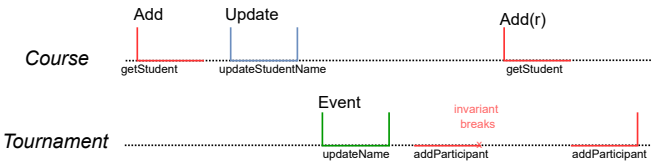
TÉCNICO LISBOA



Simulator for Saga Implemented Microservice Systems

André Martins Esgalhado

Thesis to obtain the Master of Science Degree in

Computer Science and Engineering

Supervisor: Prof. António Manuel Ferreira Rito da Silva

Examination Committee

Chairperson: Prof. Daniel Jorge Viegas Gonçalves
Supervisor: Prof. António Manuel Ferreira Rito da Silva
Member of the Committee: Prof. Rodrigo Fraga Barcelos Paulus Bruno

October 2024

Declaration
I declare that this document is an original work of my own authorship and that it fulfills all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa.

Acknowledgments

First, I would like to thank my parents and my sister for their friendship, encouragement and caring over all these years, for always being there for me, believing and supporting me and without whom this project would not be possible. I would also like to thank my grandparents, aunts, uncles and cousins for their understanding and support throughout all these years.

I would also like to acknowledge my dissertation supervisor, Prof. António Rito Silva, for his insight, support and sharing of knowledge that has made this Thesis possible. His guidance, patience and constant availability, for providing essential resources, discussing core aspects of the implementation, reviewing my work, and offering constructive feedback, has been invaluable in the successful completion of this thesis.

Additionally, I want to extend my thanks to all of my professors and the staff at Instituto Superior Técnico for equipping me with the technical knowledge and critical thinking skills that were essential to developing this thesis.

A special mention goes to my friends and colleagues, whose support, both academically and personally, helped me grow and provided me with the strength to keep going during difficult times. Whether through encouraging words, thoughtful advice, or simply being there, you all made this journey more manageable and memorable. Thank you.

To each and every one of you – Thank you.

This work was partially supported by Fundação para a Ciência e Tecnologia (FCT) through projects UIDB/50021/2020 (INESC-ID) and PTDC/CCI-COM/2156/2021 (DACOMICO).

Abstract

The development frameworks for microservice systems are complex, because they are used to develop the final system, and do not directly support the domain-driven design aspects, besides the addressing of the microservices business logic. In this thesis, we propose a sagas microservice simulator that models the system according to a domain-based design approach. The simulator supports the definition of compensating transactions, semantic locks, and event-based communication for upstream-downstream aggregates, allowing for robust conflict resolution and consistency management in distributed architectures. The results show that the simulator addresses the issues identified in the problem, it requires a minimal effort to extend, and it allows the test of complex interactions. Through these scenarios, we highlight the practical importance of implementing Sagas to handle complex interactions and ensure the system's correctness even under concurrent modifications.

Keywords

Microservices, Sagas, Simulator, Business Logic

Resumo

As frameworks para desenvolvimento de sistemas de microsserviços são complexas, porque estão pensadas para o desenvolvimento do sistema final, e não suportam diretamente os aspetos de domain-driven design, para além de endereçarem a lógica de negócio dos microsserviços. Nesta tese, propomos um simulador para microsserviços de arquitetura saga que modela o sistema de acordo com uma abordagem de domain-driven design. O simulador suporta a definição de transações compensatórias, locks semânticos, e comunicação baseada em eventos para agregados upstream-downstream, permitindo uma resolução de conflitos robusta e gestão de consistência em arquiteturas distribuídas. Os resultados mostram que o simulador aborda os problemas identificados, requer um esforço mínimo para extender, e permite testar interações complexas. Através destes cenários, destacamos a importância prática da implementação de Sagas para lidar com interações complexas e garantir a correção do sistema mesmo sob modificações simultâneas.

Palavras Chave

Microsserviços, Sagas, Simulador, Lógica de Negócio

Contents

List of Figures

List of Tables

Listings

Acronyms

ACID	Atomicity Consistency Isolation Durability
API	Application Programming Interface
CAP	Consistency Availability Partition
DDD	Domai-Driven Design
TCC	Transactional Causal Consistency

1

Introduction

Contents

The microservice architecture [1,2] is being used for the design of a large number of software systems due to its advantages for scalability and small development teams.

However, it is not without its own complexities [3], such as development complexity and the management of data consistency. The former is related to the complexity of the middleware, e.g. Kafka[1] used to implement microservice systems. The latter is related to distributed transactions that must be coordinated to support the functionalities of the microservice system, which is the result of the Consistency Availability Partition (CAP) theorem [4], which states that there is a trade-off between consistency and availability. When seen together, these two challenges result in the fact that it is possible to identify data consistency issues only late in the development process. So, it can happen that only after an extensive development effort does the team realize that the microservice architecture does not actually pay off due, for instance, to the problems and complexities associated with the coordination of distributed transactions [5].

[1] https://kafka.apache.org/

Therefore, it is worth shifting left, in the development process [6], the difficult aspects associated with the design of microservice systems. In particular, address the complexity associated with the implementation of distributed transactions in a microservice system [7].

Sagas [8] is the transactional model associated with the coordination of distributed transactions in microservice systems, which is the current engineering practice [9]. Following this model, developers must address problems resulting from the lack of isolation due to the existence of intermediate states, such as lost updates and dirty reads. Solutions like semantic locks, where transactions flag intermediate states for other transactions, require a case-by-case design that can be error-prone [9].

In this thesis, we propose a sagas microservice simulator that models the microservice system according to a domain-driven design approach [10]. It contains the minimal set of elements necessary to describe a microservice system business logic and the inter-microservice coordination, allowing for the test of the interactions in a centralized environment. In this way, there is a shifting left of the analysis and correction of the system coordination, before the actual implementation in the more expensive and troublesome context of a distributed environment. Moreover, debugging in a deterministic environment allows a faster feedback cycle for the design of the microservice system business logic. Using the simulator, software architects can assess the feasibility of the microservices architecture for the problem at hand, or even decide that some parts of the system should be implemented using an Atomicity Consistency Isolation Durability (ACID) transactional model, as some argue that not all the system has to follow the microservice architecture [11].

The simulator is evaluated against a set of canonical set of distributed transaction coordinations in microservice systems, and a large, business logic rich microservice system was implemented using the simulator.

The development of microservice systems requires a complex design of the application business logic, due to its consistency in a distributed execution context. On the other hand, there are several architectural styles for the implementation of microservice architectures. As Chris Richardson describes, what defines this type of architecture is a collection of services that are loosely coupled and are deployed independently [9]. Some of the most common challenges during microservice development include ensuring consistency, maintaining communication between services, and debugging in a distributed environment. In addition, the development of a microservice system is difficult and requires the use of complex technology.

Although various attempts have been made to simplify microservice development, especially when it comes to frameworks that help build microservice systems [12, 13], there is a gap in providing a technology-free, amenable environment for architects to experiment with architecture design and receive feedback through short cycles.

To address some of these challenges, the Transactional Causal Consistent Microservices Simula-

tor [14] was developed. This simulator extends the concept of aggregate to allow its implementation in a transactional causal consistency model, which prevents some types of anomalies resulting from lack of isolation. Although Transactional Causal Consistency (TCC) offers a stronger consistency model, most of the microservice systems use eventual consistency as a consistent model for two reasons: it is a simple model that in some cases can offer enough guarantees given the application semantics; there is no standard middleware available for more complex models that can be used in production. In such scenarios, eventual consistency, as supported by the Saga model [8], might be a better alternative. For these reasons, the proposed work is an extension of this simulator, which will support the Saga pattern [9], which implements the Saga model for the microservice architecture. Another reason to extend the simulator to accommodate the Saga pattern is the fact that it is a widely adopted approach, which will help developers make more informed decisions and mimic real-world systems.

1.1 Work Objectives

This thesis aims to achieve several objectives:

- The main objective is the development of a simulator specialized for microservice environments focused on the Saga architectural style. It needs to have a high degree of expressiveness while being simple to extend.
- Review and documentation of approaches and technologies in the existing literature on state-of-the-art simulators for distributed systems and the Saga pattern.
- Implementation of a large, business logic rich, microservice system where testing and validation can be performed with the help of the simulator.

1.2 Organization of the Document

This thesis is is organized as follows: Chapter 1 introduces the project's objectives and context. Chapter 2 provides a background on essential concepts related to microservices, the Saga architecture, domain-driven design and the problem that this thesis tackles. Chapter 3 analyzes the related work on distributed system simulators and the frameworks for the implementation of microservices systems. Chapter 4 outlines the solution proposed for extending a microservices simulator with the Saga pattern and discusses its implementation. It also shows how to describe the business logic and coordination of the microservice system in the simulator and the simulator execution engine.Chapter 5 presents the evaluation done through the design of test cases and implementation of a large microservice system.

Finally, Chapter 6 concludes the thesis with recommendations for future work and a summary of the findings.

2

Problem

Contents

This chapter contains explanations about some relevant topics for this Thesis: the Saga Architecture, Domain-Driven design, a comparison between Sagas and Transactional Causal Consistency, the state of the Transactional Causal Consistency Simulator and the problems related to these concepts.

2.1 Problem

Domai-Driven Design (DDD) [10] is being adopted as the design approach for the microservice system domain model [15, 16]. Instead of a large, interconnected domain model, the microservice is split into a set of aggregates that are atomic units of change. Therefore, accesses to an aggregate are atomic. In addition, they define the unit of consistency through a set of invariants that should be preserved by the aggregate. However, there are possible inconsistencies between the aggregates, not stated in the invariants, which should eventually be resolved.

In addition to the concept of aggregate, domain-driven design defines dependencies between the

teams that develop the aggregates, such that they can have some level of independence. Therefore, an upstream-downstream relation between two teams means that the downstream team depends on the upstream team; the downstream team is aware of the upstream aggregates, but the inverse does not occur. This corresponds to the existence of core models that other models depend on, and it is somehow related to the layered architecture style [17], where lower layers are independent of upper layers.

Listing 2.1: Quizzes Tutor Aggregates and Invariants

```
1  Aggregate Tournament {
2    Root Entity Tournament {
3      Integer id key;
4      DateTime startTime, endTime;
5      Creator creator;
6      List<Participant> participants;
7      TournamentQuiz quiz;
8    }
9    Entity Creator {
10     Integer number;
11     String name;
12   }
13   Entity Participant {
14     Integer number;
15     String name;
16   }
17   Entity TournamentQuiz {
18     Integer quizId;
19   }
20   Invariants {
21     root.startTime < root.endTime;
22     root.participants
23         .filter(p -> p.number == root.creator.number)
24         .allMatch(p -> p.name == root.creator.name);
25   }
26 }
27 Aggregate CourseExecution {
28   Root Entity CourseExecution {
29     Integer id key;
30     List<Student> students;
31   }
32   Entity Student {
33     Integer number;
34     String name;
35   }
36 }
37 Aggregate Quiz {
38   Root Entity Quiz {
39     Integer id key;
40     List<Question> questions;
41   }
42   Entity Question {
43     Integer number;
44     String question;
45   }
46 }
```

Listing 2.1 presents three aggregates, *CourseExecution*, *Quiz* and *Tournament*, which each have a root entity (it has a unique id) and a set of other entities. The *CourseExecution* and *Quiz* aggregate are core domain, and the teams that develop them should not depend on the tournament model. Actually, the *Tournament* uses them. *Tournament* has two invariants, lines 20-25: (1) tournament start time should be before end time; (2) if the tournament creator is also a participant, they should have the same name.

When mapping these concepts in the microservice architecture [9], we can associate a microservice with each aggregate because it is an atomic unit of change. The upstream-downstream relation between the microservices corresponds to an Application Programming Interface (API) of services offered by the upstream aggregate, which is used by the downstream aggregates, and a set of events published by

the upstream aggregate that can be subscribed to by downstream aggregates. The former corresponds to a dependence of the team responsible for the downstream aggregate on the API provided by the upstream aggregate. The latter supports the independence of the upstream team on the downstream teams; whenever changes occur in the upstream, they only have to publish the events; if interested, it is the responsibility of the downstream team to subscribe them and perform the necessary changes.

Considering the aggregates above, the *Course* and *Quiz* aggregates are microservices that offer service APIs that the microservice associated with *Tournament* aggregate can invoke. For example, when a student enrolls in the tournament, the *Tournament* microservice invokes a query service in the *Course* microservice to obtain the student's information. And when a tournament is created, the *Tournament* microservice invokes the *Quiz* microservice to create the tournament quiz. On the other hand, if a student changes name, the *Course* microservice publishes an event that is subscribed by the *Tournament* microservice because the team that develops *Course* is not aware of *Tournament* microservice.

We can consider two functionalities that complement this example. The *UpdateStudentName* functionality accesses the *Course* aggregate, emitting an event that is subscribed to by the downstream *Tournament* aggregate, because this is a *Course* functionality which is upstream. Upon processing of the emitted event, the creator and participant names in the *Tournament* aggregate respect their invariants. Meanwhile, the *AddParticipant* functionality reads from the *Course* aggregate to obtain student information, then writes into the *Tournament* aggregate, adding the student to the participants list. This process is feasible because the *Tournament* aggregate is downstream and this is a *Tournament* functionality. This is depicted in Listing 2.2.

Listing 2.2: *UpdateStudentName* and *AddParticipant* functionalities.

```
1  Functionality UpdateStudentName {
2      student = query CourseExecution#Student
3      update student.name
4      emit UpdateStudentNameEvent(student)
5  }
6
7  Functionality AddParticipant {
8      tournament = query Tournament
9      student = query CourseExecution#Student
10     tournamentParticipant = create TournamentParticipant(tournament,student)
11     update tournament(tournamentParticipant)
12 }
```

The behavior of the microservice system is defined by its functionalities that may correspond to the coordination of one or more microservices. When several microservice transactions are coordinated, there is a distributed transaction that may result in inconsistent states. For instance, after the change of a student name in *CourseExecution* microservice, the system can be inconsistent if that student is enrolled in *Tournament*. Eventually, the update name event will be processed and consistency restored. A

more complex interaction between transactions that can cause inconsistent states occurs in a particular interleaving of the student enrollment functionality and the student update name functionality, when the student being enrolled is the creator. The student enrollment functionality has two local transactions: read student in the *CourseExecution* microservice, and register participant in the *Tournament* microservice. When the update name functionality occurs after the read student (including the processing of its update event in *Tournament*) and before the register participant, it will result that the creator will have the updated name and the participant will have the initial name, which violates one invariant. Another situation occurs when a *Tournament* in being created, which takes several steps, and if a query is done to obtain all the tournament, an incompletely constructed tournament will be part of the result.

To address these types of problems, the Sagas transactional model was proposed, where due to the lack of isolation, it is possible that transactions execute compensating transactions to restore a consistent state in case of abort [8]. Currently, several patterns have been defined to implement the Sagas transactional model [9], such as semantic locks that local transactions write on aggregates that are in an intermediate state. In the mentioned case of a query for all tournaments, the microservice implementing the query can filter the tournaments that are not completely constructed. Note that what to do with semantic locks depends on the functionality; if the tournament has a different semantic lock indicating that is being changed, it may not be filtered. Therefore, the definition of the behavior of Sagas is done on a case-by-case basis, which increases the business logic complexity and makes it relevant to be addressed soon in the development process because it determines the application behavior.

Synthesizing, the following aspects need to be addressed by a simulator for microservice systems designed according to domain-driven design and following the Saga transactional model:

- *Atomic Aggregates*: According to the aggregate model, the aggregate is an atomic unit for change. So the simulator should ensure that at any moment an aggregate cannot be changed by two transactions, avoiding the lost update problem. For instance, two functionalities that enroll a student, both read the same tournament and then update it. The last to write the tournament aggregate should not overwrite the changes done by the first one;

- *Aggregate Invariants*: It is also necessary that the invariants are preserved at the aggregate level, for each atomic change. For instance, after a student enrolls in the tournament, it is necessary to verify the invariant associated with the names of the creator and participants;

- *Upstream-Downstream Aggregate Relations*: The processing of events is asynchronous, which raises several situations: (1) by itself there is eventual consistency in the system that the functionalities should be aware of, for instance, after the update student name, a functionality that accesses the tournament may not get the most recent name, and if that is important from the functionality business logic, it has to query the *CourseExecution*; (2) more complex is the possibility of a per-

manent inconsistencies due to the order of execution of the events, as described above, when the update name event is processed before the register of the student;

- *Aggregate Intermediate States*: When a functionality has changed an aggregate but it has not finished yet this state may be read by other functionalities, and, if the first functionality aborts it occurs a dirt read problem. To address this, semantic locks need to be supported to mark intermediate states. Therefore, in the simulator, upon accessing an aggregate with a semantic lock, it is up to the functionality (it depends on its business logic) to decide whether it aborts and retries, or it reads the aggregate, or if it ignores the semantic lock and continues processing. For example, when a tournament is being updated, the last step is to update its quiz. If the quiz update fails, the tournament update needs to be undone, and to inform other functionalities that the tournament is in an intermediate state, a semantic lock is added to the tournament.

2.2 Semantics

2.2.1 Saga Architecture

As explained in the book Microservices Patterns [9], one of the most difficult tasks when working with a microservice architecture is how to implement transactions that affect multiple services. In this type of architecture, the problem lies when transactions affect data in multiple services, opposed to when they happen in a single service in which we can still use ACID (Atomicity, Consistency, Isolation, Durability) transactions.

The solution to this problem revolves around the use of Sagas [8]. Richardson proposes Saga as a ”message-driven sequence of local transactions to maintain data consistency” [9]. The downside of using Sagas is that they do not have the isolation property of ACID transactions, and therefore, the application must use some countermeasures to mitigate this lack of isolation.

Neal Ford et al. [18] go even beyond Richardson and expand on the fact that there can be a multitude of Transactional Saga patterns made from the different combinations of key dimensions: communication, consistency, and coordination dimensions.

1. Communication: The mode or method of information exchange between different elements or stages of the Sagas, which can be synchronous or asynchronous.

2. Consistency: The property regarding a system’s correctness and integrity of data in which multiple operations are executed concurrently or in a distributed manner. It can be eventual or atomic.

3. Coordination: The approach used to manage and control the execution flow of a distributed transaction in a Saga. It can be orchestrated or choreographed.

2.2.2 Domain-Driven Design

In the world of microservice architecture, Domain-Driven Design is a fundamental approach used in the development process. It is not only an architectural approach, but also a design philosophy that is focused on the modeling of the domain of an application. This is particularly useful in the context of microservices, since it can help to better define and structure each service and its boundaries in the realm of the application. The principles used in DDD are often based on concepts such as aggregates, events, functionalities, entities, value objects, repositories, bounded contexts, and upstream-downstream relationships.

- **Aggregates** represent transactional consistency boundaries and dictate the rules for data modifications within a domain, which define the aggregate consistency. It is a unit of atomic change.
- **Entity and Value objects** are the elements that constitute aggregates, they contain the domain-specific behavior and data.
- **Events** act as triggers for functionalities related to upstream-downstream relationships between aggregates. They help orchestrate interactions between different components of the system, helping to maintain coherence and consistency.
- **Repositories** are responsible for managing the lifecycle of aggregates, which includes their persistence management.
- **Bounded contexts** group aggregates that are the development responsibility of a team.
- **Upstream-downstream relationships** between bounded contexts define development dependencies between the teams responsible for the bounded contexts. The downstream team is aware of the upstream team, and the inverse is not true. When a change occurs in an aggregate in an upstream bounded context, an event is published that the aggregates in downstream bounded contexts can subscribe to.
- **Functionality** defines the coordination of an interaction between aggregates through the invocation of their services. Aggregates also interact with each other via the publish-subscribe pattern.

A functionality is associated with an aggregate, known as the main aggregate of the functionality. It invokes services in its main aggregates and all aggregates that are upstream, however, it cannot invoke services in downstream aggregates because the team developing the functionality is not aware of downstream aggregates. Therefore, when an aggregate service changes an aggregate, it may publish events that are subscribed by downstream aggregates. Upon receiving an event, the downstream aggregates may trigger another functionality to handle it.

2.2.3 Sagas vs Transactional Causal Consistency

While Sagas are one of the most well-established approaches when it comes to maintaining data consistency in microservice architectures, different alternatives have recently been proposed to mitigate some of its problems. Transactional Causal Consistency (TCC) [19], for example, is another way to tackle this problem. In contrast to Sagas, TCC aims to ensure consistency based on the causal relationships between transactions. This can be relevant in scenarios where stronger consistency and isolation are more important than the flexibility provided by Sagas. In fact, the high level of complexity of TCC is not worth it for several applications, especially those that do not need strong consistency levels. Additionally, currently there are no commercial implementations of TCC, and research implementations are on top of key-value stores. In summary, at the business logic level, Sagas decentralize transaction management, prioritize eventual consistency, continue to be more appropriate for complex workflows, and are the industry standard.

In the following sections, we will explore how Sagas and TCC relate to the development of microservice systems.

2.2.4 Transactional Causal Consistent Microservices Simulator

This simulator was previously developed to test and experiment with the design of microservices built on the concept of aggregate. Aggregates are specified according to the concepts of invariants, which define their business logic. Furthermore, the simulator supported the definition of the TCC behavior associated with interactions between aggregates [20].

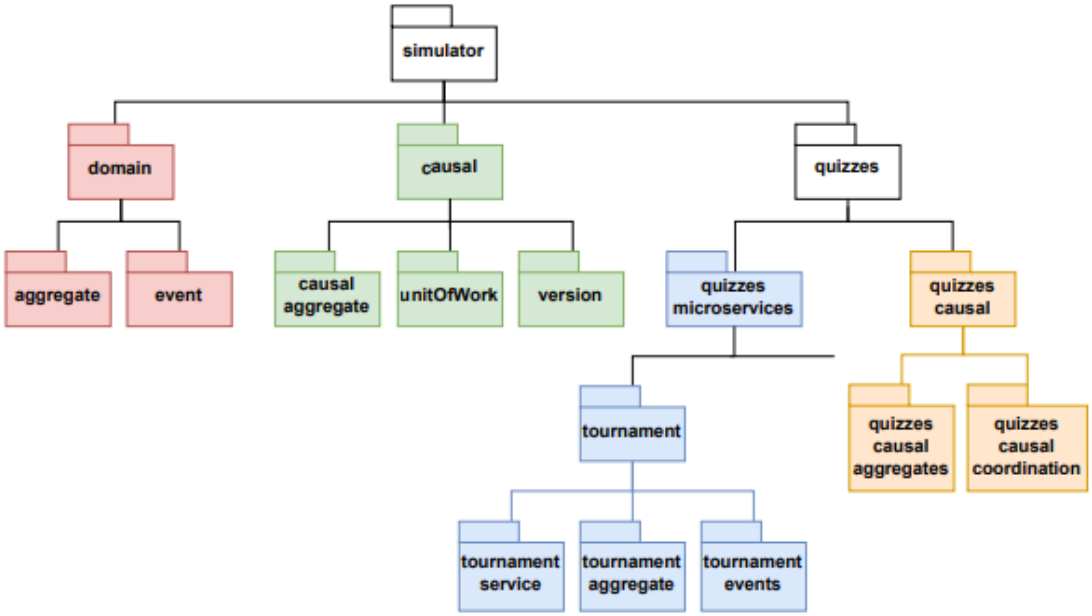


Figure 2.1: Simulator Decomposition View

The simulator consisted of three modules, as shown in Figure 2.1. There was the domain module (red), which contained the concepts of aggregate and event that are used to define a microservice. Then there was the causal module (green), where the infrastructure for TCC behavior was implemented using the unit of work pattern [21]. Finally, there was the quizzes module, which corresponds to an example of a microservice system implemented using the simulator. This module contained two sub-modules: one related to microservices business logic independent of a particular transactional model that extends the domain (blue) and the other that extends the transactional agnostic business logic with TCC behavior (orange). A detailed explanation of these modules can be found in an article submitted for publication [22].

Figure 2.2 exemplifies how the simulator could be used to implement a microservice. The diagram shows the classes that needed to be extended and the methods that needed to be redefined. Firstly, the microservice business logic was defined (blue). Then, the TCC behavior was added (orange). Note how the domain elements (red) and the TCC infrastructure (green) were used in the extension.

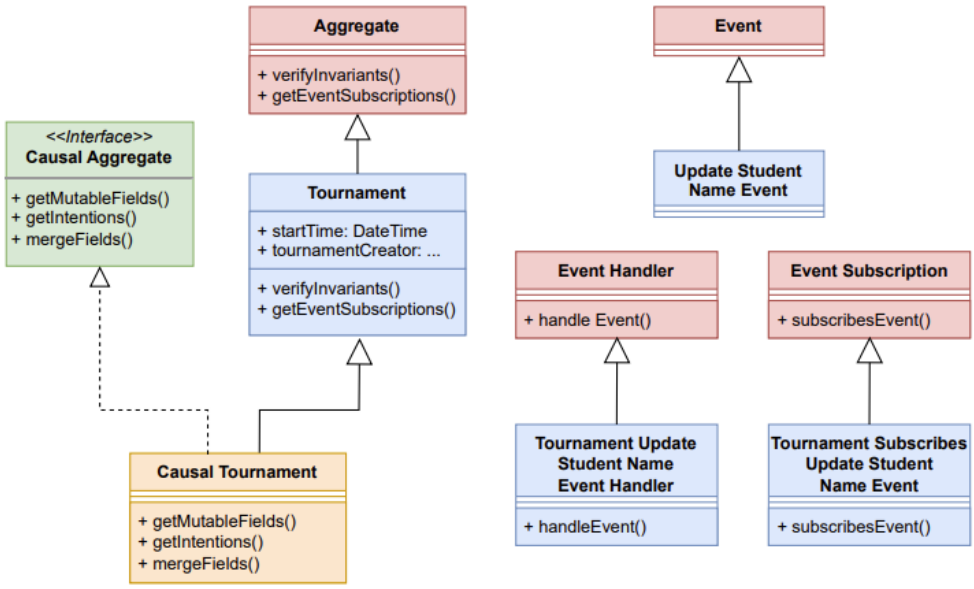


Figure 2.2: Domain Model Extension

Some of its key architectural elements were:

- Aggregates: The aggregates in the simulator are designed to be independent of any specific transactional model. This allows for a versatile foundation where the core business entities and rules are defined without being initially bound to a particular transactional model.
- UnitOfWork: The unit of work pattern in the simulator serves as a mechanism for coordinating complex transactions, especially those that span multiple aggregates or services.
- Functionality: Defines the coordination of aggregate services, using UnitOfWork to add transactional behavior to their execution.

This approach allows the simulator to maintain a modular and flexible architecture, where the core business logic encapsulated in aggregates and driven by events can be defined independently and later integrated with a specific transactional model. It is worth mentioning that the simulator could help developers experiment with these different architectures and help them mimic the impact of the implementation according to their necessities and choices. To better understand how the simulator worked, an example is described below that shows how the cases in Listings 2.1 and 2.2 were implemented. First, the tournament aggregate and its corresponding invariants are defined as shown in Listing 2.3. The Tournament *id* comes from extending Aggregate, where it is generated. Then, *invariantStartTimeBe-*

foreEndTime(), lines 8-10, represents the first invariant. After that, the *verifyInvariants()* method needs to be overridden. The same is done for the *getEventSubscriptions()* method, lines 20-25. Both methods are abstract methods defined in *Aggregate*. It should be noted that while only one case is demonstrated here, developers are expected to include all relevant cases for invariants.

Listing 2.3: Tournament aggregate definition.

```
1  @Entity
2  public abstract class Tournament extends Aggregate {
3      private LocalDateTime startTime;
4      private LocalDateTime endTime;
5
6      ...
7
8      public boolean invariantStartTimeBeforeEndTime() {
9          return this.startTime.isBefore(this.endTime);
10     }
11
12     @Override
13     public void verifyInvariants() {
14         if (!invariantStartTimeBeforeEndTime() && ...) {
15             throw new TutorException(INVARIANT_BREAK, getAggregateId());
16         }
17     }
18
19     @Override
20     public Set<EventSubscription> getEventSubscriptions() {
21         Set<EventSubscription> eventSubscriptions = new HashSet<>();
22         interInvariantCreatorExists(eventSubscriptions);
23         ...
24         return eventSubscriptions;
25     }
26 }
```

Next, the Tournament aggregate is extended by the *CausalTournament* implementing the causal aggregate. The three methods that have to be extended, *getMutableFields()*, lines 9-12, *getIntentions()*, lines 14-20 and *mergeFields()*, lines 22-37, provide the causal semantics of the aggregate. Listing 2.4 shows the class definition.

Listing 2.4: CausalTournament aggregate definition.

```
1  @Entity
2  public class CausalTournament extends Tournament implements CausalAggregate {
3      public CausalTournament(Integer aggregateId, TournamentDto tournamentDto, UserDto creatorDto,
4                              CourseExecutionDto courseExecutionDto) {
5          super(aggregateId, tournamentDto, creatorDto, courseExecutionDto);
6      }
7      ...
8
9      @Override
10     public Set<String> getMutableFields()  {
11         return Set.of("startTime", "endTime", "tournamentCreator", ...);
12     }
13
14     @Override
15     public Set<String[]> getIntentions() {
16         return Set.of(
17                 new String[]{"startTime", "endTime"},
18                 new String[]{"startTime", "numberOfQuestions"},
19                 ...);
20     }
21
22     @Override
23     public Aggregate mergeFields(Set<String> toCommitVersionChangedFields,
24         Aggregate committedVersion, Set<String> committedVersionChangedFields){
25         if (!(committedVersion instanceof Tournament)) {
26             throw new TutorException(AGGREGATE_MERGE_FAILURE, getAggregateId());
27         }
28
29         Tournament committedTournament = (Tournament) committedVersion;
30
31         mergeCreator(committedTournament, this);
32         mergeStartTime(toCommitVersionChangedFields, committedTournament, this);
33         mergeEndTime(toCommitVersionChangedFields, committedTournament, this);
34         ...
35
36         return this;
37     }
38     ...
39 }
```

Functionalities execute on transactional model-specific aggregates, in this case TCC. Listing 2.5 shows an example of how the *addParticipant* functionality, from Listing 2.2, was implemented in the simulator. First, a *unitOfWork* is created, lines 2-3. Then, the necessary data is retrieved, in this case, the tournament and the user (participant), lines 3-9. After that, a new *TournamentParticipant* is created. The service is then invoked, and the participant is added to the tournament, line 11. All of this is done in the context of the created *unitOfWork*. Finally, the *unitOfWork* service commits the changes, which are dependent on the transactional model.

Listing 2.5: Functionality implementation example.

```
1  public void addParticipant(Integer tournamentAggregateId, Integer userAggregateId) {
2      CausalUnitOfWork unitOfWork = unitOfWorkService.createUnitOfWork(
3                                      new Throwable().getStackTrace()[0].getMethodName());
4      TournamentDto tournamentDto = tournamentService.getTournamentById(
5                                                          tournamentAggregateId, unitOfWork);
6      UserDto userDto = courseExecutionService.getStudentByExecutionIdAndUserId(
7                                                 tournamentDto.getCourseExecution().getAggregateId(),
8                                                 userAggregateId,
9                                                 unitOfWork);
10     TournamentParticipant participant = new TournamentParticipant(userDto);
11     tournamentService.addParticipant(tournamentAggregateId, participant, userDto.getRole(), unitOfWork);
12     unitOfWorkService.commit(unitOfWork);
13 }
```

The functionality invokes aggregate services. The *addParticipant* service of *Tournament* aggregate is shown in Listing 2.6.

Listing 2.6: *addParticipant* service of *Tournament* aggregate.

```
1  @Retryable(
2      value = { SQLException.class },
3      backoff = @Backoff(delay = 5000))
4  @Transactional(isolation = Isolation.READ_COMMITTED)
5  public void addParticipant(Integer tournamentAggregateId,
6      TournamentParticipant tournamentParticipant, String userRole, CausalUnitOfWork unitOfWork) {
7          Tournament oldTournament = (Tournament) unitOfWorkService.
8                                      aggregateLoadAndRegisterRead(tournamentAggregateId, unitOfWork);
9
10         Tournament newTournament = new CausalTournament((CausalTournament) oldTournament);
11
12         newTournament.addParticipant(tournamentParticipant);
13
14         unitOfWork.registerChanged(newTournament);
15 }
```

It can be observed that the implementation of the service is strongly coupled with the transactional model, it creates *CausalTournament* instances, line 10. Also, note that the method *aggregateLoadAndRegisterRead()* also depends on the transactional model but it is already encapsulated inside the *UnitOfWork*.

Some problems that were identified with the state of the simulator before the work of this thesis was conducted include:

- Hardcoded Functionalities and Services: The implementation tightly integrated specific functionalities and services with the transactional model (TCC), limiting the simulator's flexibility to experiment with different architectural styles, such as the Saga pattern. This can be seen in the functionality implementation, that used a concrete *UnitOfWork* that implements TCC, and in the order of invo-

cations. This is visible in Listings 2.5 and 2.6 where a CausalUnitOfWork is created and used, respectively. Also, a particular class of Causal aggregate is created. This is shown in Listing 2.6, where a tournament is created from a *CausalTournament*. Note that if compensating transactional needs to be added, the functionality implementation needs to be changed. Furthermore, the services depend on the *UnitOfWork* and the transactional specific aggregates.

- Limited Communication Patterns: The simulator's design assumed synchronous communication patterns, which can be seen in Listing 2.5 where the code directly invokes service methods in a specific sequence. This may not be suitable for all microservice architectures, especially those employing asynchronous methods.

These problems may hinder the use of the simulator in more contexts and handicap its effective utilization in diverse architectural contexts and practical applications. The following sections will explore and discuss these challenges in more depth and potential solutions.

3

Related Work

Contents

There are middleware technologies for the implementation of microservice systems, such as Spring Cloud[1]. However, by using them, it is not possible to do some upfront design of the system functionalities and their business logic because the developer soon becomes immersed in the technology complexity. On the other hand, there are frameworks that support the design of microservice applications at a higher level, such as Eventuate Tram [23], Temporal [24] and SagaMAS [25]. Although these frameworks provide higher level constructs, such as service orchestration and compensating transactions, they do not explicitly provide constructs for domain-driven design, and are still focus on the development of a production system, which raise the same technology complexity problems.

There are several simulators for distributed systems that focus on the modeling of the network and the analysis of its qualities [26–29] or to describe distributed algorithms [30–32] but they do not focus on

[1] https://spring.io/projects/spring-cloud

microservice architectural style, nor address problems such as transaction management.

A new set of simulators have recently been proposed for microservice systems. They focus more on aspects such as performance [33], where a dependency graph is designed to identify performance bottlenecks, or focus in a particular domain like manufactoring [34], or simulation for edge/fog computing environments [35, 36] with service migration, dynamic distributed cluster formation, and microservice deployment, or focus on the performance of cloud native microservice chains [37], or the microservices system resilence on the occurrence of faults [38], or on the issue of integrating several simulators for embedded systems [39]. Overall, these simulators ignore the business logic complexity of some microservice systems.

Some research is done on the extension of aggregates to ensure consistency between replicas of entities in data-intensive distributed systems, such as microservices [40]. They address the same type of problems common to microservice systems that our simulator intends to identify through simulation.

3.1 Transactional Saga Patterns

As mentioned before, multiple combinations of key dimensions can generate a series of various Transactional Saga patterns. This idea is well presented in the book by Ford et al. [18] from which we can take the patterns in the table 3.1 below.

Table 3.1: Transactional Saga Patterns

Pattern Name	Communication	Consistency	Coordination
Epic Saga (sao)	Synchronous	Atomic	Orchestrated
Phone Tag Saga (sac)	Synchronous	Atomic	Choreographed
Fairy Tale Saga (seo)	Synchronous	Eventual	Orchestrated
Time Travel Saga (sec)	Synchronous	Eventual	Choreographed
Fantasy Fiction Saga (aao)	Asynchronous	Atomic	Orchestrated
Horror Story Saga (aac)	Asynchronous	Atomic	Choreographed
Parallel Saga (aeo)	Asynchronous	Eventual	Orchestrated
Anthology Saga (aec)	Asynchronous	Eventual	Choreographed

3.2 Existing tools and frameworks

There are already some tools and frameworks that help developers in the process of building microservice systems that use the Saga transactional pattern. There are already some studies that compare some of these tools [12, 13]. From there, we took inspiration to seek and propose meaningful alternatives for the simulator to explore.

3.2.1 Eventuate Tram Sagas

Eventuate Tram [23] is an open-source framework that facilitates the development of applications when it comes to transactional messaging and Sagas. The main focus of this framework are Sagas with an asynchronous communication style such as the Fantasy Fiction Saga (aao), Horror Story Saga (aac), Parallel Saga and Anthology Saga (aec) from Table 3.1. This framework provides abstract classes, interfaces and a DSL on which developers can build and use to define a Saga's state machine. These interfaces and method overriding encourage polymorphism that supports the reuse of framework components.

The aggregates not only represent the domain elements but also implement business logic, which can be described by a state machine. This consists of the definition of several groups of methods, each group corresponding to a Saga. Creating a Saga is a simple process of extending the Saga class as in Figure 3.1:

1. Define the orchestrator
   (a) Define the sequence of steps the Saga should execute in the functionality;
   (b) Specify handlers for each step with the logic for what should happen when a step is executed;
   (c) Define the compensating actions for each step, in case of failure;
   (d) Set the conditions that determine when the Saga should proceed to the next step or terminate.
2. Define the participants
   (a) Define the methods that are invoked with commands by the orchestrator.
3. Both for the orchestrator and the participants services command handlers need to be implemented.

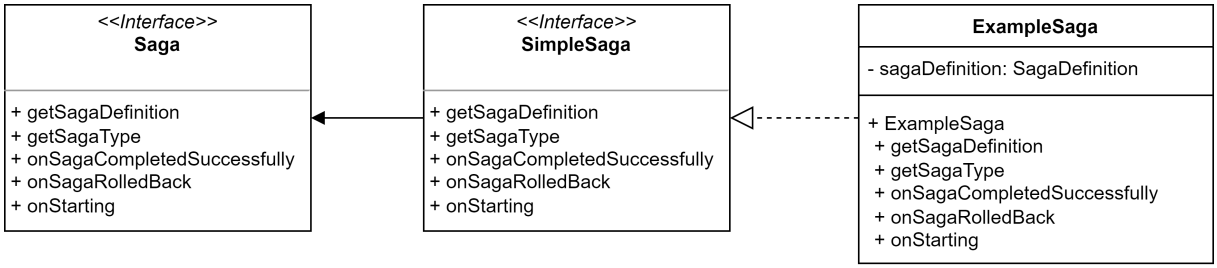


Figure 3.1: Eventuate Tram Sagas Example

Although the framework is intended for distributed systems, it can also be deployed in a centralized way. It provides the necessary means to be tested locally and, so, simulate a distributed environmnent and some of its characteristics like asynchronous communication, Saga execution and coordination

sequences and transactional messaging, failure scenarios and compensation logic and the concept of eventual consistency.

This framework allows for easy specification and implementation of microservices while maintaining a high degree of expressiveness. The framework provides a structured way to define Sagas through the orchestration of multiple microservices (participants). This orchestration includes specifying the sequence of steps, handling each step, and defining compensating actions for failures. It makes complex interactions within a distributed system clearer and more manageable. Communication typically occurs through commands and events. Commands are directives to perform an action, and events signal that something has happened. This method of communication is expressive, as it clearly delineates actions (commands) and reactions (events) within the system. It is well documented and moderately easy to use and configure. Although, it is a bit limited when it comes to the different types of Sagas that are possible to implement, especially with the high degree of specification complexity proposed by Ford et al. [18].

Listing 3.1 shows a simple version of how the previously discussed *AddParticipant* functionality could be implemented with Eventuate as a Saga. First, a compensating action is defined, lines5-6, in case the subsequent steps fail. Then the next, lines 8-9, step creates the getParticipant command message by calling *AddParticipantSagaData.getParticipant()* and sends it to the channel specified by *courseExecutionService.getParticipant*. Finally, if a successful reply is received from *courseExecutionService*, *saveTournamentCourseExecution()* is executed, lines 10-11.

Listing 3.1: AddParticipant Functionality with Eventuate

```
1  public AddParticipantSaga(TournamentServiceProxy tournamentService,
2      CourseExecutionServiceProxy courseExecutionService) {
3          this.sagaDefinition =
4              step()
5                  .withCompensation(tournamentService.rejectAddParticipant,
6                                    AddParticipantSagaData::undoAddParticipant)
7              .step()
8                  .invokeParticipant(courseExecutionService.getParticipant,
9                                     AddParticipantSagaData::getParticipant)
10                 .onReply(TournamentParticipantDto.class,
11                         AddParticipantSagaData::saveTournamentCourseExecution)
12             .step()
13                 .invokeParticipant(tournamentService.addParticipant,
14                                    AddParticipantSagaData::addParticipant)
15             .build();
16 }
```

3.2.2 Temporal

Temporal [24] is also an open-source framework to build resilient, distributed, and scalable applications. It has several features like workflow orchestration, state management or retry, and compensation logic that can be useful in the development of Saga-based patterns. Abstraction, modularity, encapsulation, and inheritance are some of the design principles that Temporal is based on and solidify it as a viable object-oriented framework that can be easily reusable in different applications and contexts.

While there is no direct equivalent of aggregates and Sagas in Temporal, the concept of a Workflow shares some similarities, particularly in managing business logic, maintaining state, and ensuring consistency and fault tolerance. Each Workflow has activities that correspond to what steps are in a Saga.

The following steps can be used to reproduce the implementation of a Saga:

1. Define Activities: These are the transaction steps and their corresponding compensations.
2. Define a Workflow Interface: This will orchestrate the activities.
3. Implement the Workflow: Define the logic to execute activities and handle failures.

Figure 3.2 contains the classes needed to implement the equivalent behavior of a Saga in Temporal.

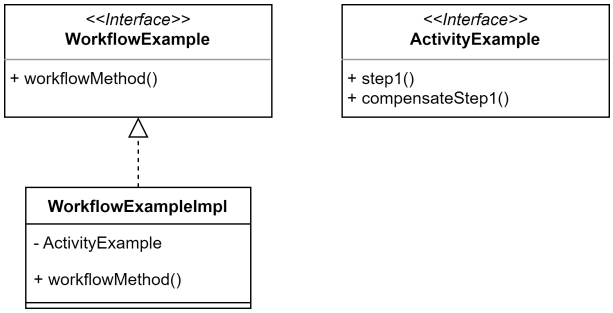


Figure 3.2: Temporal Example

Temporal can also be run in a local environment with the help of technologies like Docker for simulation purposes, but not in a single process like the Transactional Causal Consistent Microservices Simulator.

This framework is also well documented, but the nomenclature of concepts requires some adaptation, which might require additional effort from the user. Although it allows for the development of complex systems and applications, it also lacks versatility when it comes to the different types of Sagas proposed by Ford et al. [18].

The main difference between Temporal and Eventuate Tram Sagas is that Temporal focuses on orchestrating complex workflows and activities, providing a robust framework for state management and

fault tolerance, while Eventuate focuses on managing distributed transactions using the Saga pattern. Temporal is more process-oriented, focusing on workflow orchestration and stateful long-running processes, opposed to Eventuate, which is more transaction-oriented.

Listing 3.2 shows a simple version of how the previously discussed AddParticipant functionality could be implemented with Temporal. Firstly, the interface for the activities is defined, lines 2-9. These methods should then be implemented according to the business logic. After that, the interface for the workflow is defined, lines 12-16. Finally, the workflow is implemented, lines 28-40. Several options for how the activities are executed, for example, retry timers can be configured. Then, said activities and the corresponding compensating activities are executed.

Listing 3.2: AddParticipant Functionality with Temporal

```
1  //Activities Interface
2  @ActivityInterface
3  public interface TournamentActivities {
4      @ActivityMethod
5      TournamentParticipantDto getParticipant(AddParticipantSagaData data);
6      void addParticipant(AddParticipantSagaData data, TournamentParticipantDto participant);
7      void undoAddParticipant(AddParticipantSagaData data);
8      void saveTournamentCourseExecution(AddParticipantSagaData data);
9  }
10
11 //Workflow Interface
12 @WorkflowInterface
13 public interface AddParticipantWorkflow {
14     @WorkflowMethod
15     void addParticipantSaga(AddParticipantSagaData data);
16 }
17
18 //Workflow Implementation
19 public class AddParticipantWorkflowImpl implements AddParticipantWorkflow {
20     private final ActivityOptions options = ActivityOptions.newBuilder()
21         .setStartToCloseTimeout(Duration.ofHours(1))
22         .setRetryOptions(RetryOptions.newBuilder().setMaximumAttempts(1).build())
23         .build();
24
25     private final TournamentActivities activities =
26         Workflow.newActivityStub(TournamentActivities.class, options);
27
28     @Override
29     public void addParticipantSaga(AddParticipantSagaData data) {
30         Saga.Options sagaOptions = new Saga.Options.Builder().setParallelCompensation(true).build();
31         Saga Saga = new Saga(SagaOptions);
32         try {
33             TournamentParticipantDto participant = activities.getParticipant(data);
34             activities.addParticipant(data, participant);
35             Saga.addCompensation(activities::undoAddParticipant, data);
36             activities.saveTournamentCourseExecution(data);
37         } catch (ActivityFailure e) {
38           Saga.compensate();
39           throw e;
40         }
41     }
42 }
```

3.2.3 SagaMAS

SagaMAS [25] is essentially a multi-agent based framework that is being developed, where the communication and coordination aspects of Sagas are handled by multi-agent systems principles. Although

there is no specific implementation yet, the idea is that each microservice is associated with a specific agent. This agent may be located on the same server as the microservice or elsewhere. When a transaction is initiated in a microservice, it communicates the start of this transaction to its associated agent. After the agent is informed about the start of the transaction, an independent multi-agent layer takes over the handling of the transaction. This layer is also responsible for managing any errors related to the transactions and the agent layer itself. By doing this, the novelty is that there is a decoupling of the coordination aspect of transactions that becomes a responsibility of the agent layer, opening the door for future AI solutions and discoveries. Since its implementation is not publicly available yet, it is difficult to predict how the framework should be used. However, the authors specify that developers need to define the following predicates:

- *incoming_action(st_name, action)*: to define the action to be executed in the microservice given the sub-transaction.
- *compensation_action(st_name, action)*: to define action that is to be executed in case there is a problem in the transaction as a compensating mechanism.
- *next(st_name, agent)*: to define the agent to execute the given subtransaction.
- *next(st_name, new_st_name, agent)*: to link an existing subtransaction with a new sub-transaction and set the agent that will start it. This can be used to define non simple paths in the transaction without ambiguity.

Listing 3.3 shows how this translates to the AddParticipant functionality.

Listing 3.3: AddParticipant Functionality with SagaMAS

```
1  // Agent Tournament
2  incoming_action(addParticipant, createAddParticipant).
3  compensation_action(addParticipant, undoAddParticipant).
4  next(addParticipant, courseExecution).
5  incoming_action(saveTournamentCourseExecutionApproval, saveTournamentCourseExecution).
6  compensation_action(saveTournamentCourseExecutionApproval, cancelAddParticipant).
7
8  // Agent CourseExecution
9  incoming_action(addParticipant, getParticipant).
10 next(addParticipant, saveTournamentCourseExecutionApproval, tournament).
```

Although the paper does not mention other transactional models apart from Sagas, the idea of using a multi-agent system architecture that is responsible for the coordination aspects of microservices could be explored for other models like TCC for example, or even a combination where different services don't follow the same model. This becomes a possibility to explore thanks to the abstraction layer provided by the use of agents.

3.2.4 Comparison Overview

Tables 3.2 and 3.3 show comparisons between the mentioned frameworks.

It should be taken into account that in Eventuate Tram Sagas both the functionality and coordination elements are decoupled from the actual service. An attempt to implement this structure could be made in the TCC Microservices Simulator, which uses the concepts of aggregates to encapsulate business logic and data, and services to handle business operations associated with these aggregates. This would require re-writing the functionalities and services so that they would be decoupled from the transactional model.

To extend the simulator to accommodate Sagas presented a significant challenge, some considerations revolving around its architecture and mechanisms were taken into account:

- Functionalities were defined as a coordination mechanism tied to aggregate service operations. To accommodate Sagas, it is essential to conceptualize these functionalities as Saga steps, which are essentially service invocations. This shift will require rethinking how the simulator treats operations, making them a series of interrelated actions across multiple services.

- The simulator's Unit of Work supported a transactional model, not direct coordination. Extending the simulator for Sagas involved integrating an orchestration or choreography mechanism for coordinating these Saga steps. To accomplish this, another version of the unit of work was implemented, since its main purpose was to support a transactional model instead of coordination.

- Communication in the TCC simulator was primarily direct invocations and publishing and subscription of events. For Sagas, the communication strategy was made to include both direct service invocations and asynchronous messaging (commands and events), which are typical in Saga implementations. This allows for greater decoupling between services and more flexibility since services' behavior was previously hardcoded.

- One of the most significant aspects of the Saga pattern is its approach to handling failures through compensating transactions. Since the TCC simulator, in its previous form, did not inherently consider compensating logic due to TCC's atomic nature, a key part of the extension was introducing and integrating this mechanism. This involved designing ways to reverse the effects of previous operations in a Saga when a subsequent operation fails.

In conclusion, making the TCC Microservices Simulator compatible with Sagas required considerable changes to its core structure, particularly if we take into account the various Saga patterns outlined by Neal Ford et al [18].

Table 3.2: Comparison Overview

	Eventuate Tram Sagas	Temporal	SagaMAS
Communication	Asynchronous	Both	Asynchronous
Consistency	Eventual	Eventual	Eventual
Coordination	Orchestrated	Orchestrated	Semi-orchestrated
Ease of Integration	Moderate	Moderate	-
Expressiveness	Moderate	Low	-
Scalability	Good	Good	Good
Distributed environment	yes	yes	yes
Local environment	Centralized or Simulated Distribution in the same machine	Centralized or Simulated Distribution in the same machine	-
Compensating Transactions	yes	yes	yes
Automatic retry	yes	yes	no
Built-in language support	Java	Go, Java, php, python	-

Table 3.3: Design Elements

Design Element	Eventuate Tram Sagas	Temporal	SagaMas	TCC Microservices Simulator
Functionality/Operation	Saga Steps	Activities	Actions	Service Invocation
Orchestration/Coordination	Saga Orchestrator	Workflow Interface	Multi-Agent System Coordination	Functionality/Unit of Work
Communication	Commands & Events	Workflow Signal Methods	Agent Communication	Service invocation, Publish/Subscription of Events
Compensation Logic	Compensating Actions for Each Step	Activity Retries and Compensations	Compensating Actions	Merge Process
Consistency Model	Eventual Consistency	Eventual Consistency	Eventual Consistency	Transactional Causal Consistency
Testing Environment	Local & Distributed	Local (via Docker) & Distributed	-	Local & Simulated Distributed
Additional Features	-	State Management, Workflow Orchestration	-	Invariants

3.3 Comparison with the developed framework

In this section, we compare the proposed microservice framework with other existing solutions to highlight its strengths and areas for improvement. While many frameworks provide tools for microservice architecture, such as handling distributed transactions and communication, the framework developed in this thesis brings unique advantages in terms of flexibility, domain-driven design, and ease of testing.

3.3.1 Eventuate Tram Sagas

Eventuate Tram Sagas is a well-known framework for implementing distributed transactions using the Saga pattern. Like our framework, it supports compensating transactions and event-based communication, which helps ensure eventual consistency across services. However, Eventuate Tram Sagas is focused primarily on asynchronous communication and lacks direct support for domain-driven design principles.

Our framework, on the other hand, integrates domain-driven design deeply into the architecture, making it easier for developers to map their business logic directly into the microservices without additional complexity. Additionally, the support for upstream-downstream aggregate relationships in our framework encourages cleaner separations of responsibilities between teams, reducing dependencies and enhancing productivity.

3.3.2 Temporal

Temporal is another framework that allows developers to build resilient, distributed systems. It focuses on workflow orchestration and offers strong state management and retry mechanisms for long-running processes. Temporal's main strength lies in its ability to handle complex workflows with fault tolerance, which is beneficial for certain use cases, but it can also introduce complexity in terms of configuration and setup.

Compared to Temporal, our framework offers a more lightweight approach, with a simpler configuration process, making it easier for smaller teams or projects that do not require complex state management to get started. While Temporal excels at handling highly complex workflows, our framework strikes a balance between flexibility and ease of use, providing sufficient robustness for most business applications without unnecessary overhead.

3.3.3 SagaMAS

SagaMAS is a multi-agent-based system that manages distributed transactions through agents. It presents an innovative approach by decoupling transaction coordination from the services themselves,

allowing the coordination to be handled independently. However, SagaMAS is still in its early stages, and its practical applications are yet to be fully explored.

In contrast, our framework is already fully implemented and tested, providing concrete tools for handling distributed transactions with well-defined patterns. By incorporating both semantic locks and compensating transactions, our framework delivers a reliable solution for ensuring consistency while allowing teams to work independently on their services.

3.3.4 Overall Comparison

Each of the compared frameworks has its strengths, but our framework offers a unique combination of features that make it particularly suitable for development teams following domain-driven design and aiming to simulate complex microservice interactions early in the development process.

3.4 Simulation in Microservices

With the increasing popularity of microservice architectures, there is also a need for good and reliable simulation environments and tools for these systems [41, 42]. In this section, we will explore some of the work that has already been developed in this area.

3.4.1 FERAL

FERAL [39], as the name indicates, is a Framework for Efficient simulator coupling on Requirements and Architecture level. Feral attempts to address several challenges when it comes to accurately predicting the behavior of embedded systems in complex environments. Communication behavior, system environments, and functional behavior of system components are some characteristics that usually involve the coupling of specialized simulators. This introduces restricted flexibility in choosing simulators and an inability to create integrated scenarios efficiently. To address this, FERAL enables the integration of heterogeneous simulators and simulation models, offering high reuse potential, simplifying simulator integration, and enabling the rapid development of simulator coupling scenarios. The framework aims to address the problems of semantic integration between simulators, correct synchronization, and the need for extensibility to accommodate additional simulators and models. Although FERAL is focused on embedded systems, dealing with hardware-software interaction, there are some key ideas that could be taken into consideration when developing the microservices simulator. Both incorporate multiple components. Feral integrates multiple simulators in a single framework; similarly, the microservices simulator intends to integrate different transactional models, architectural styles and patterns within microservice systems. The simulator would allow for a different combination of aspects to be simulated, such as

Sagas or TCC and synchronous or asynchronous communication. Both attempt to be modular and flexible, allowing for the reuse of components and adapting to different simulation needs. To achieve a high level of modularity, like in FERAL, the simulator could try to maximize the number of independent components/modules, so that they could be easily replaced and operate independently without affecting each other. This would also facilitate the addition of new features, models, and simulating components contributing to better extensibility. As FERAL allows for different execution models that are integrated as directors, the simulator could try a similar approach where the execution and communication are also managed in independent layers. FERAL's correct synchronization, which refers to the precise coordination of multiple simulation components, is a concept that could also be applied to the simulator. This would ensure that multiple microservices interact cohesively, even if they follow different transactional patterns. This is also important in the context of distributed systems, where services may operate asynchronously or depend on one another's outputs.

3.4.2 μqSim

μqSim [43] is a scalable and validated queueing network simulator for interactive microservices. The main problems addressed by μqSim are the complexity introduced by changing from monolithic designs to microservices and the difficulty in studying microservices at scale. μqSim provides detailed intra- and inter-microservice models, allowing for accurate reproduction of complex applications. μqSim is modular, supporting the reuse of individual models across different microservices and applications. Similarly to what was discussed in 3.4, although the focal point of μqSim is microservices and its domain is slightly different from the microservices simulator, there are some points that could be considered. μqSim focuses on microservices, emphasizing scalability and performance in interactive systems, while the simulator focuses on exploring transactional models in microservices. Even though μqSim specifically targets performance metrics such as throughput and tail latency, it was also developed with adaptability and modularity in mind, allowing component reuse, such as the microservices simulator. μqSim also provides users with the possibility of using high-level, declarative specifications of the microservices dependency graph and server platforms. Something like this could be implemented in the simulator to allow for configuration specification or easier testing development.

These works solidify the increasing adoption of microservices and reinforce the need for simulators and tools that help with their comprehension and development. Both works provide valuable insights on how to approach and model a simulation environment and how to handle complexity in microservice architectures.

4

Microservice Simulator

Contents

This chapter contains the details and explanations about the implementation and the work that was done to transform the Transactional Causal Consistent Microservices Simulator into a Microservice Simulator Framework that allows for both Transactional Causal Consistent and Sagas implemented microservice simulation.

4.1 Microservice Sagas Simulator

The microservice simulator supports domain-driven design concepts and their integration in a microservice architecture. In addition, it provides a centralized execution engine that simulates the execution of Sagas in a distributed environment.

This section presents the implementation and adaptation of key concepts within the microservice architecture, focusing on the integration of the Saga pattern into the simulator framework. To achieve this the previous TCC simulator was extended and adapted to allow for both transactional models.

The next sections explain how aggregates were adapted to allow for the Saga architecture, how workflows and the unit of work pattern were used to manage functionalities and the remaining structural changes that were made in the simulator.

4.1.1 Overview

In order to accommodate for the Sagas architecture and simulation, some changes and refactorizations were made to the structure of the simulator. These can be seen in Figure 4.1. In this updated view, a new module for coordination is implemented. It contains the abstractions for *unitOfWork* and *workflow*, which will be used to define the transactional logic. The *unitOfWork* is responsible for the management of the transactional aspect of operations. The *workflow* is responsible for defining and managing the sequence in which business operations are performed. Each of these is also extended according to a specific transactional model. These abstract modules separate the transactional behavior from the execution logic of the functionality. Regarding the implementation of quizzes, a new module equivalent to *quizzes causal* appears, but for the Saga model. These modules contain the aggregate extension and the definitions of the workflows for each functionality behavior that are specific to each model. Finally, *quizzes microservices* contains all the components that are common to both transactional modules.

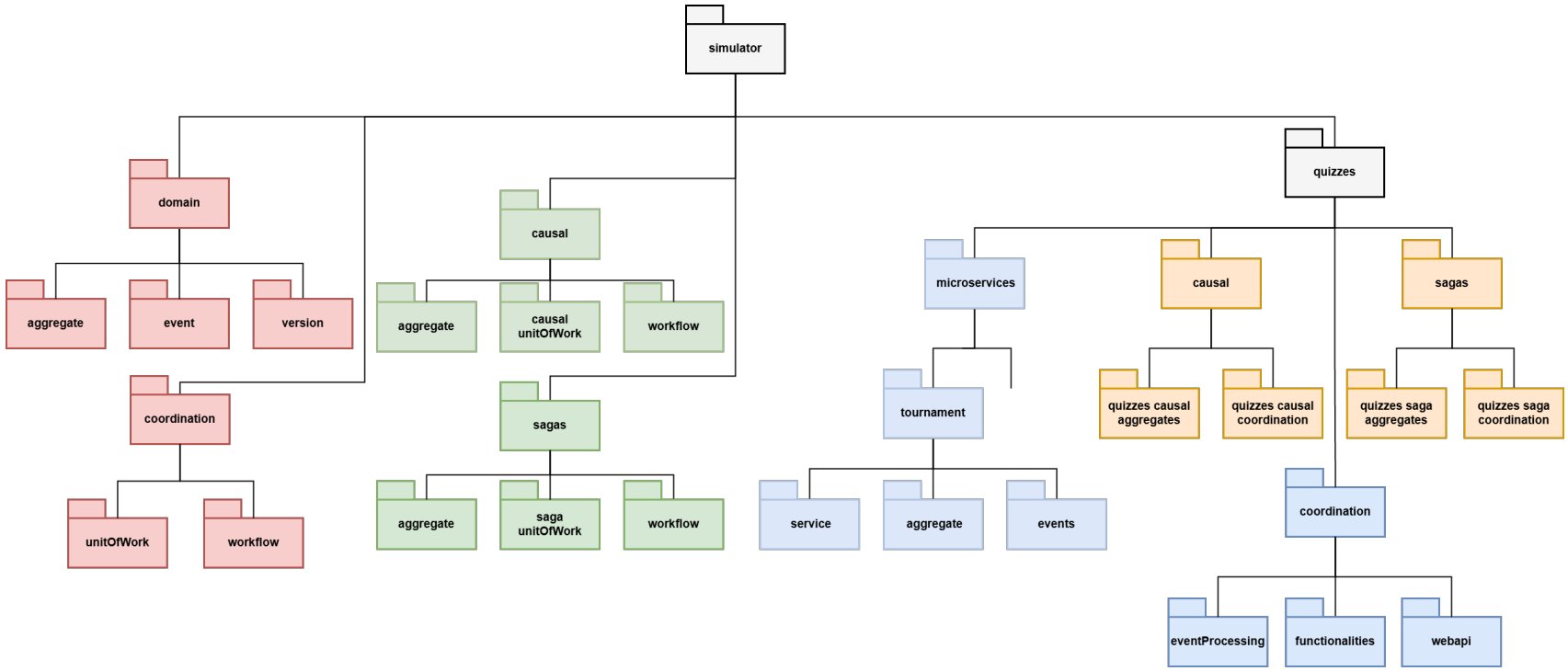


Figure 4.1: Updated Simulator Decomposition View

4.1.2 Aggregate Design

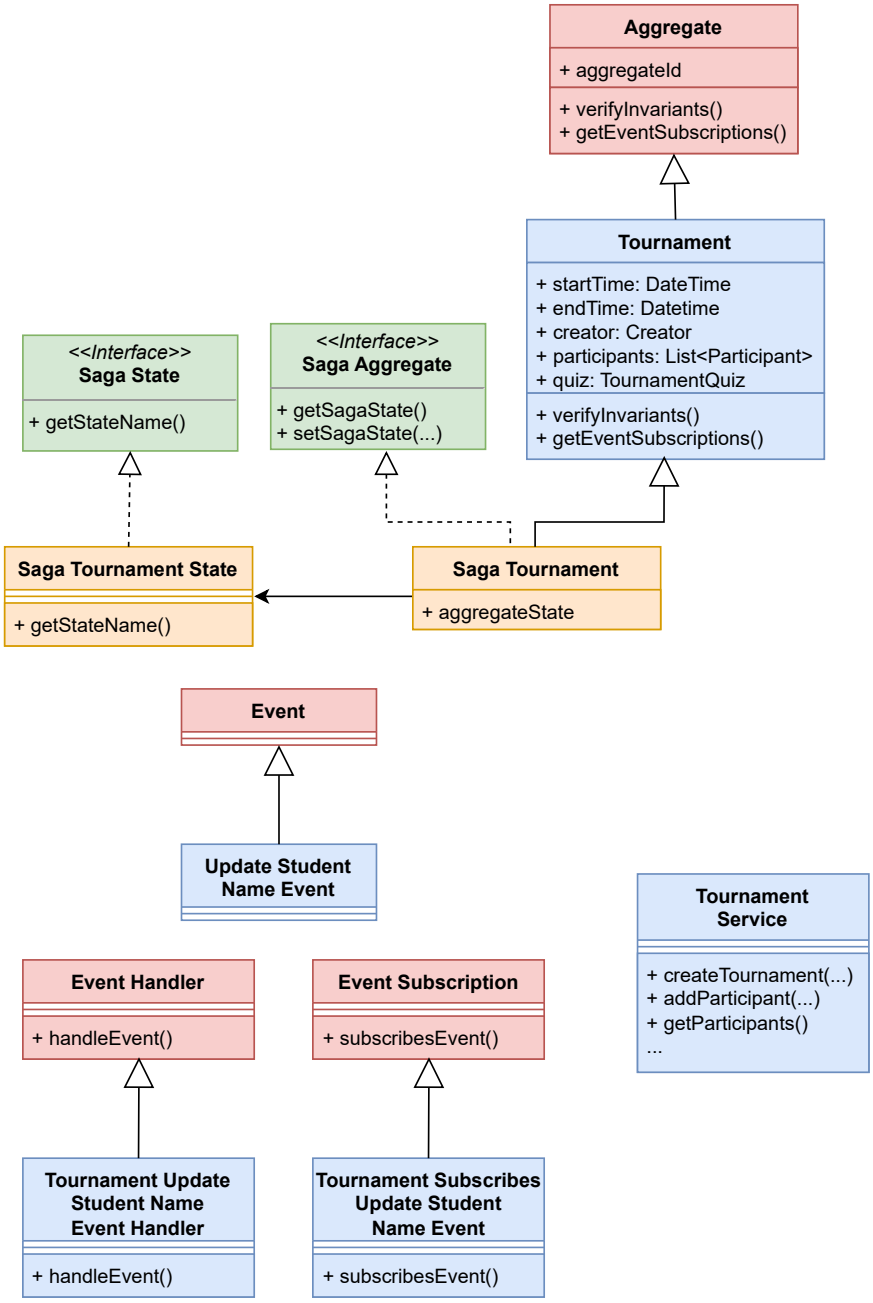


Figure 4.2: Aggregate Domain Model.

The simulator supports the specification of aggregates that are prepared to execute according to a Sagas transactional model. Figure 4.2 presents the main entities of the simulator domain model to represent Sagas. For modularity and extensibility, the model has different levels of cohesion. The classes in red and in green are independent of the application domain; they belong to the simulator core and are reused for all simulations. The red classes are related to the definition of aggregates, and the green classes to the definition of a Sagas transactional model; therefore, it is possible to extend the simulator to support other transactional models. The blue and orange classes are application dependent, whereas the blue is independent of the transactional model and the orange is specific of the Sagas transactional model.

The central concept is the aggregate that comprises a set of entities and has a unique identifier, such that other microservices can refer to it. Abstract class *Aggregate* has a unique *aggregateId* and defines two abstract methods: *verifyInvariants* which define the invariant aggregates; and *getEventSubscriptions* that define the events to which it subscribes from upstream microservices. In the figure, these methods are overriden in *Tournament*. In addition, *Tournament* refers to the other entities that comprise the aggregate, *Participant* and *TournamentQuiz*. Note that these entities are not aggregates.

The *SagaAggregate* interface provides methods to support semantic locks (*getSagaState* and *setSagaState*). These methods are implemented in *SagaTournament* to enrich *Tournament* with Sagas transactional management. In addition, the implementations of the *SagaState* interface define the semantic locks for each type of aggregate.

Abstract classes *Event*, *EventSubscription* and *EventHandler* define the simulator core for upstream-downstream asynchronous processing of events. Upstream microservices emit their events during the execution of services, and downstream events implement *EventSubscription* and *EventHandler* to, respectively, subscribe and process the events they are interested in. The instances of *EventSubscription* are returned by the *getEventSubscriptions* of the aggregate.

Below, we present the code for some of the simulator extensions for the Quizzes Tutor system.

Listing 4.1: verifyInvariants() and getEventSubscriptions() methods

```
1  public void verifyInvariants() {
2    if (!(this.startTime.isBefore(this.endTime) &&
3          ... {
4      throw new TutorException(INVARIANT_BREAK, ...);
5    }
6  }
7
8  public Set<EventSubscription> getEventSubscriptions() {
9    Set<EventSubscription> eventSubscriptions = new HashSet<>();
10     eventSubscriptions.add(
11       new TournamentSubscribesUpdateStudentName(this));
12     ...
13   return eventSubscriptions;
14 }
```

The code in Listing 4.1 shows the definition of the two methods in aggregate *Tournament* for invariant verification, lines 1-6, and subscription of events, lines 8-14. For the event subscription it registers an instance of *TournamentSubscribesUpdateStudentName* which *subscribesEvent* method is defined to verify if there is an event of *UpdateStudentNameEvent* for one of the tournament participants, as shown in the following code in Listing 4.2.

Listing 4.2: subscribesEvent method

```
1  public boolean subscribesEvent(Event e) {
2    return
3      getEventType().equals(e.getEventType) &&
4      participants.stream().anyMatch(p ->
5        p.getAggregateId.equals(((UpdateStudentNameEvent) e)
6            .getStudentAggregateId()))
7  }
```

Handling events triggers the execution of a functionality. The simulator periodically handles the events by invoking the *handleEvent* method of *TournamentUpdateStudentNameEventHandler*.

Finally, class *TournamentService* defines the methods of the microservice API, in this case for the *Tournament* microservice. The simulator expects that a class is defined for each microservice, and although it does not inherit from an abstract class, common to all microservice APIs, it reuses some generic methods provided by the simulator coordinator part, such as, for instance, a method to emit events (*registerEvent*) or update changed aggregates (*registerChanged*) in class *UnitOfWorkService* in Figure 4.3).

Listing 4.3: addParticipant method

```
1 @Transactional(isolation = Isolation.SERIALIZABLE)
2 public void addParticipant(...) {
3   Tournament oldTournament = (Tournament)
4     unitOfWorkService.aggregateLoadAndRegisterRead(...);
5   Tournament newTournament = oldTournament.clone();
6   newTournament.addParticipant(tournamentParticipant);
7   unitOfWorkService.registerChanged(newTournament);
8 }
```

In the method *addParticipant* of the *Tournament* microservice API, shown in Listing 4.3, the tournament aggregate is read using the unit of work service, lines 3-4, then it is duplicated, line 5, and the participant is added, line 6. Finally, the new aggregate is registered in the Saga unit of work service to be committed, line 7.

4.1.3 Coordination Design

The coordination design describes how the functionality of a system is achieved through the interaction of several microservices. The execution of a system functionality is triggered by an external interaction with the system, such as updating the name of a student, for instance.

The design applies the Unit of Work pattern to define the transactional context associated with the execution of a functionality [44] and the Workflow patterns to define the coordination of several services [45]

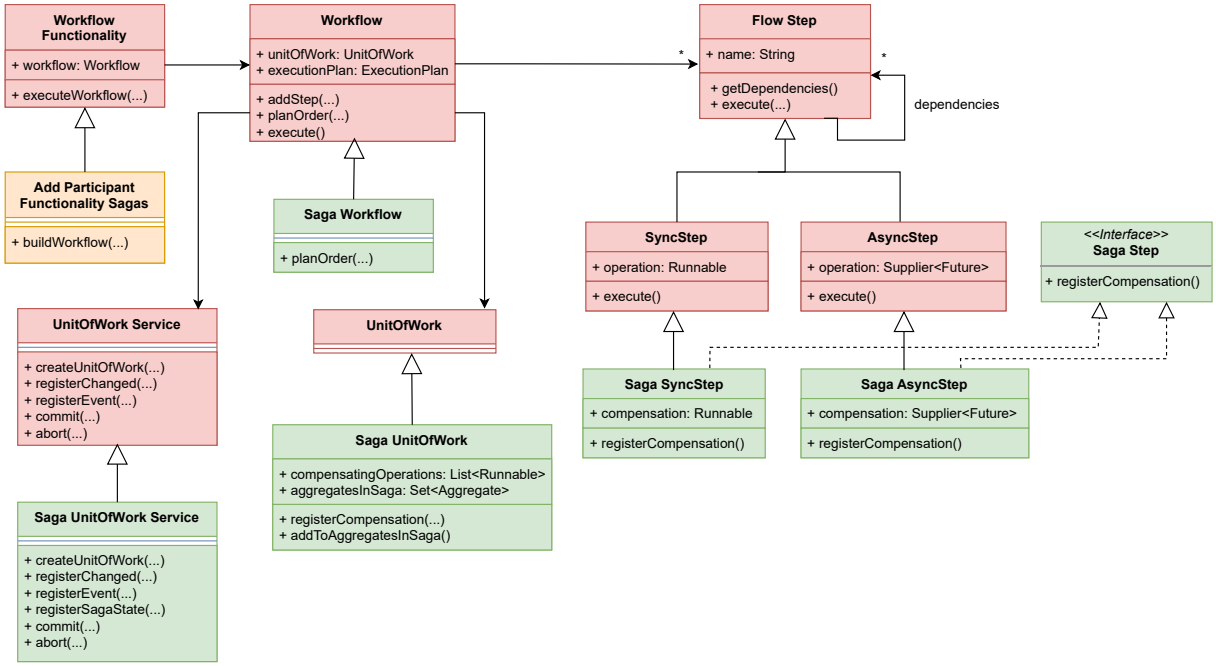


Figure 4.3: Functionality Domain Model.

To define a functionality, it is necessary to implement a single class, like *AddParticipantFunctionalitySagas* in Figure 4.3 for adding a student as a participant of the tournament. This class uses the core coordination classes of the simulator that will manage the execution of the functionality. The architect only has to define the coordination implementation method *buildWorkflow* and indicate the events that may have to be handled during functionality execution, method *handleEvents*. The latter is due to the fact that during the execution of a Saga the functionality may be interested in some event that occurred in an aggregate which it has interacted with.

By invoking the *executeWorkflow* method in *WorklowFunctionlity* a concrete functionality triggers the execution of the functionality according to the coordination defined in *Workflow*. The execution starts with the creation of an *SagaUnitOfWork* that registers the aggregates that are changed and the events to emit as a result of the execution of the functionality. It also stores compensation operations in case the functionality execution has to abort. The *SagaUnitOfWorkService* is the transactional manager responsible for initializing the unit of work and deciding on its commit or abort.

The simulator implements a simple workflow, which comprises a set of *FlowStep* which have an execution dependence relation between them. The dependencies are used to generate an execution plan. Each flow step executes on its own thread. There are two types of *FlowStep*: *SyncStep*, where the workflow waits for the callee to finish before executing the next step; and *AsyncStep*, where the workflow executes the steps asynchronously and uses futures to synchronize results availability [46]. The two Java used constructs, *Runnable* and *Supplier*, in their *operation* attributes, encapsulate the

invocation of the microservice API, such as the method *addParticipant* of class *TournamentService* in Figure 4.2. Finally, the subclasses *SagaSyncStep* and *SagaAsyncStep* define the compensating operations associated with each step, which are added to the *UnitOfWork* when the step is executed using the *registerCompensation* method.

Listing 4.4: Method *buildWorlflow* of *AddParticipantFunctionalitySagas* (partial)

```
1  public void buildWorkflow(...) {
2    this.workflow = new SagaWorkflow(...);
3
4    SagaSyncStep getStudentStep =
5          new SagaSyncStep("getStudentStep", () -> {
6      StudentDto studentDto =
7          courseService.getStudentById(...);
8      setStudentDto(studentDto);
9      this.currentStep = "getStudentStep";
10   });
11
12   SagaSyncStep addParticipantStep =
13         new SagaSyncStep("addParticipantStep", () -> {
14     Participant participant =
15               new Participant(getStudentDto());
16     tournamentService.addParticipant(...);
17     this.currentStep = "addParticipantStep";
18   }, new ArrayList<>(Arrays.asList(getStudentStep)));
19
20   this.workflow.addStep(getStudentStep);
21   this.workflow.addStep(addParticipantStep);
22 }
```

Listing 4.4 presents the *buildWorkflow* method of the *addParticipantFunctionalitySagas* class. It starts by creating a 2 step Saga workflow. The first step, lines 4-10, gets the student information from the *Course* microservice API by invoking *getStudentById*. The second step creates the *Participant* entity using the information received from a previous step through *getStudentDto*; then invokes the *addParticipant* method of the *Tournament* microservice API. The creation of the step also indicates that *getStudentStep* should precede it (added to the step using an array list), line 18. Finally, the steps are added to the workflow in the last three instructions, lines 20-21.

Another example, in which semantic locks and compensation operations are used, is the *UpdateTournamentFunctionalitySagas* functionality, which updates two aggregates *Tournament* and *Quiz*. This is shown in Listing 4.5

Listing 4.5: Method *buildWorlflow* of *UpdateTournamentFunctionalitySagas* (partial)

```
public void buildWorkflow(TournamentDto tournamentDto,...) {
  this.workflow = new SagaWorkflow(...);

  SagaSyncStep getOriginalTournamentStep =
      new SagaSyncStep("getOriginalTournamentStep", () -> {
    TournamentDto originalTournamentDto =
        tournamentService.getTournamentById(...);
    switch (originalTournamentDto.getSagaState()) {
      case IN_UPDATE_TOURNAMENT -> {
        throw new TutorException(...);
     }
     ...
    }
  });

  SagaSyncStep updateTournamentStep =
      new SagaSyncStep("updateTournamentStep", () -> {
    TournamentDto newTournamentDto =
        tournamentService
            .updateTournament(tournamentDto,...);
    unitOfWorkService
        .registerSagaState(IN_UPDATE_TOURNAMENT,...);
    this.setNewTournamentDto(newTournamentDto);
  }, new ArrayList<>(
      Arrays.asList(getOriginalTournamentStep)));

  updateTournamentStep.registerCompensation(() -> {
    tournamentService
        .updateTournament(originalTournamentDto,...);
    unitOfWorkService.registerSagaState(NOT_IN_SAGA,...);
  }, unitOfWork);

  SagaSyncStep updateQuizStep =
      new SagaSyncStep("updateQuizStep", () -> {
    QuizDto quizDto =
        new QuizDto(this.getNewTournamentDto());
    quizService.updateGeneratedQuiz(...);
  }, new ArrayList<>(Arrays.asList(updateTournamentStep)));

  workflow.addStep(getOriginalTournamentStep);
  workflow.addStep(updateTournamentStep);
  workflow.addStep(updateQuizStep);
}
```

In this functionality, it can be seen that the original tournament is kept in the Saga, obtained in the first step (*getOriginalTournamentStep*), such that if an abort occurs, it is used in the compensation operation (*registerCompensation*) of the second step (*updateTournamentStep*). This can occur if the third step *updateQuizStep* fails. In the second step, a semantic lock (*IN_UPDATE_TOURNAMENT*) is written in

the tournament, lines 21-22. The lock is released (*NOT_IN_SAGA*) in the compensation operation, line 30. According to this functionality business logic, it is not possible to have two simultaneous updates of the same tournament, which is verified in *getOriginalTournamentStep* and if that is the case, an exception is thrown, lines 8-11.

Once the workflow associated with the functionality is built, it is executed according to the Sagas transactional behavior, which is implemented applying the *Unit of Work* pattern. To apply the pattern, two classes are defined *SagaUnitOfWorkService* and *SagaUnitOfWork*, where the former corresponds to the API provided to the functionality and the latter contains the state of a Saga execution.

A new *SagaUnitOfWork* instance is passed to the *buildWorkflow* method and is passed to each of the services that are executed by the functionality. The services register the updated aggregates and events that have to be emitted due to their business logic execution (methods *registerChanged* and *registerEvent* of the *SagaUnitOfWorkService*, see the code of *addParticipant* service for a case of *registerChanged*). For *registerChanged*, the *SagaUnitOfWorkService* verifies aggregate invariants, and if they pass, it writes the updated aggregate, otherwise it aborts the execution of the functionality. Note that, this way, the intermediate state of the aggregate becomes visible before the functionality commits, which is the expected lack of isolation of Sagas behavior. For the implementation of *registerEvent*, the event is written and can be subscribed to. Since the *registerChanged* method can result in an abort, in a service, it should always be invoked before *registerEvent* to guarantee that the event is published only if the service does not abort. On the other hand, when a semantic lock is written in the context of a workflow execution, the method *registerSagaState* of *SagaUnitOfWorkService*, the aggregate is written, so that other functionalities can read the semantic lock, and it is also added to the set of *aggregatesInSaga* in *SagaUnitOfWork*.

At the end of a functionality execution, the workflow invokes the method *commit* of *SagaUnitOfWorkService*. It releases all the semantic locks for the set of *aggregatesInSaga*. If an abort occurs during the execution of the functionality, the compensation operations are invoked by the inverse order of their register. Note that when a *SagaStep* executes it registers its compensating operation in *SagaUnitOfWork*.

The simulator is implemented as a Spring Boot[1] application and is publicly available.

4.1.4 Workflow Design

The workflow design in the simulator implements the coordination of multiple microservices, ensuring that distributed transactions are handled properly, especially in the context of Sagas. The workflows are realized using an execution plan that organizes the steps required to complete a functionality depending on the dependencies specified by the developer when building the workflow. The core components of the workflow include the *ExecutionPlan*, *planOrder*, and *execute* methods, which manage the ordering

[1] https://spring.io/projects/spring-boot

and execution of the steps in the workflow.

The *ExecutionPlan* class is responsible for managing the workflow steps and their dependencies. It maintains a list of steps and their execution statuses, allowing the simulator to control when each step can be executed based on its dependencies. Steps without dependencies are executed immediately, while steps with dependencies wait until all prerequisite steps have been completed.

The method *planOrder* shown in Listing 4.6 defines the ordering of steps in the workflow by analyzing their dependencies. It computes the in-degree (number of dependencies) for each step, lines 3-5, and uses a queue to manage steps that are ready for execution. This method ensures that steps are executed in the correct order while avoiding cyclic dependencies, which could lead to deadlocks or inconsistent states.

Listing 4.6: planOrder method in SagaWorkflow

```
1  public ExecutionPlan planOrder(HashMap<FlowStep, ArrayList<FlowStep>> stepsWithDependencies) {
2      // Calculate how many dependencies each step has
3      for (HashMap.Entry<FlowStep, ArrayList<FlowStep>> entry: stepsWithDependencies.entrySet()) {
4          inDegree.put(entry.getKey(), entry.getValue().size());
5      }
6
7      // Steps without dependencies are ready to be ordered
8      for (HashMap.Entry<FlowStep, Integer> entry: inDegree.entrySet()) {
9          if (entry.getValue() == 0) {
10             readySteps.add(entry.getKey());
11         }
12     }
13
14     // Order steps based on dependency resolution
15     while (!readySteps.isEmpty()) {
16         FlowStep step = readySteps.poll();
17         orderedSteps.add(step);
18
19         for (HashMap.Entry<FlowStep, ArrayList<FlowStep>> entry: stepsWithDependencies.entrySet()) {
20             if (!entry.getKey().equals(step) && entry.getValue().contains(step)) {
21                 inDegree.put(entry.getKey(), inDegree.get(entry.getKey()) - 1);
22                 if (inDegree.get(entry.getKey()) == 0) {
23                     readySteps.add(entry.getKey());
24                 }
25             }
26         }
27     }
28
29     if (orderedSteps.size() != stepsWithDependencies.size()) {
30         throw new IllegalStateException("Cyclic dependency detected in steps");
31     }
32
33     return new ExecutionPlan(orderedSteps, stepsWithDependencies, this.getFunctionality());
34 }
```

Once the steps are ordered using the *planOrder* method, the *execute* method shown in Listing 4.7

is responsible for running the workflow. It executes each step sequentially or concurrently, depending on whether the steps are synchronous or asynchronous. This method also handles compensations in case a step fails, ensuring that the system remains consistent by invoking the registered compensation actions in reverse order.

Listing 4.7: execute method in Workflow

```
1  public CompletableFuture<Void> execute(UnitOfWork unitOfWork) {
2      this.executionPlan = planOrder(this.stepsWithDependencies);
3      try {
4          return executionPlan.execute(unitOfWork)
5              .thenRun(() -> {
6                  unitOfWorkService.commit(unitOfWork);
7              })
8              .exceptionally(ex -> {
9                  Throwable cause = (ex instanceof CompletionException) ? ex.getCause() : ex;
10                 unitOfWorkService.abort(unitOfWork);
11                 throw new RuntimeException(cause);
12             });
13     } catch (TutorException e) {
14         unitOfWorkService.abort(unitOfWork);
15         throw e;
16     }
17 }
```

The *execute* method initiates the workflow's execution, commits the transaction if successful, or aborts and compensates if an exception occurs.

To simulate concurrency within the microservices system, the simulator allows developers to use the *executeUntilStep* and *resume* methods. These methods allow the workflow to be paused and resumed at specific steps, simulating scenarios where multiple workflows are executing concurrently.

The *executeUntilStep* method allows the execution of the workflow up to a particular step. This is useful for simulating partial executions, where a process is interrupted, and the system must handle concurrent modifications or other workflows operating in parallel. Once a workflow has been paused at a certain step, the *resume* method can be used to continue the execution from that point. This simulates real-world scenarios where a process is resumed after waiting for external conditions or parallel transactions to complete. Both these methods are shown in Listing 4.8.

Listing 4.8: executeUntilStep method in Workflow

```
1  public CompletableFuture<Void> executeUntilStep(FlowStep targetStep, UnitOfWork unitOfWork) {
2      ArrayList<FlowStep> stepsToExecute = new ArrayList<>();
3      for (FlowStep step : plan) {
4          stepsToExecute.add(step);
5          if (step.equals(targetStep)) {
6              break;
7          }
8      }
9      return executeSteps(stepsToExecute, unitOfWork);
10 }
11
12 public CompletableFuture<Void> resume(UnitOfWork unitOfWork) {
13     ArrayList<FlowStep> remainingSteps = new ArrayList<>();
14     for (FlowStep step : plan) {
15         if (!executedSteps.get(step)) {
16             remainingSteps.add(step);
17         }
18     }
19     return executeSteps(remainingSteps, unitOfWork);
20 }
```

By combining the use of *executeUntilStep* and *resume*, the simulator can simulate complex interleavings between workflows, allowing developers to test concurrent scenarios where multiple workflows affect the same set of microservices. This is particularly useful in distributed systems where concurrency can introduce challenges like race conditions, inconsistent states, and the need for compensating transactions.

Concurrency in microservice systems can result in inconsistent states if not handled correctly. By leveraging the simulator's ability to pause and resume workflows, along with the compensating mechanisms of Sagas, it is possible to resolve these inconsistencies. For instance, if two workflows attempt to modify the same aggregate, the simulator can simulate one workflow completing, while the other compensates for any conflicts.

The simulator ensures that the workflow steps are executed in the correct order and that compensations are applied whenever necessary, providing a robust framework to test concurrent microservice interactions.

4.2 Microservice Simulator Framework

With these adaptions made to the simulator, the previous TCC simulator was enriched and now also allows for the simulation of Sagas. The structural changes that were made allow for the developers to reuse most of the code having only as a task to rewrite the specifications of the aggregates and

functionalities according to the desired transactional model. By using workflows, the simulator now provides a unified framework where different transactional behaviors can be simulated, ensuring that both models can coexist, and complex transactions can be handled flexibly.

5

Evaluation

Contents

5.1 Evaluation

To evaluate the simulator, we show that it supports aspects identified in the problem, it requires a minimal effort to extend, and that it allows the test of complex interactions. Finally, a large microservice system was implemented using the simulator, and the lessons learned are reported.

5.1.1 Completeness

The solution addresses all identified aspects of microservice systems using a domain-driven design approach in Section 2.1.

- *Atomic Aggregates*: Atomic aggregates are supported because each service is executed in an ACID transactional context (@Transactional in the service implementation). Moreover, when the

service reads an aggregate it reads the complete aggregate, clones it, does the changes in the clone, and then writes it (inside *registerChanged*);

- *Aggregate Invariants*: Aggregate invariants are preserved because for each atomic change of an aggregate, all the invariants are verified. The method *verifyInvariants* of the aggregate is invoked by *registerChanged*, and if a verification fails the functionality aborts;
- *Upstream-Downstream Aggregate Relations*: The downstream-upstream relations between the aggregates are supported by the service APIs and the communication between events. The service emits events by invoking *registerEvent* at the end of its execution, and each aggregate defines its events subscriptions and handlings;
- *Aggregate Intermediate States*: A state is added to all Saga aggregates through the implementation of the *SagaAggregate* and *SagaState* interfaces, where the latter allows the definition of aggregate-specific intermediate states. Intermediate states are managed by the functionalities and are decoupled from the service business logic, because Saga management is performed at the level of functionality execution. Therefore, the functionality is responsible for reading and writing semantic locks.

5.1.2 Ease of Extension

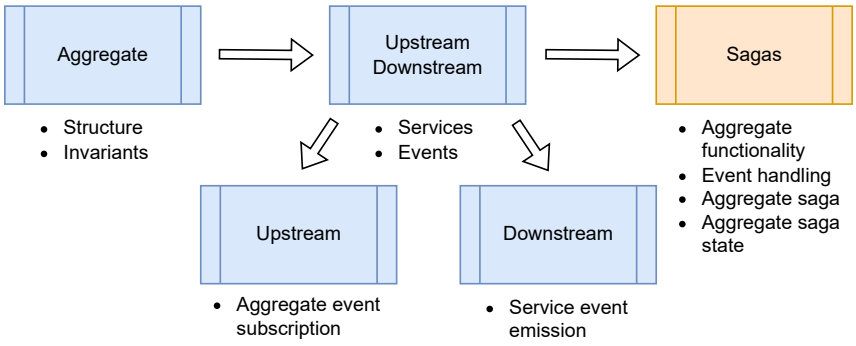


Figure 5.1: Simulator extension process

Figure 5.1 shows the steps to implement an aggregate by extending the simulator. This is an iterative process that starts by defining for each aggregate its structure and invariants. The former requires the extension from *Aggregate* class and the implementation of the classes for each of the aggregate entities. Additionally, some queries may be needed to be implemented to retrieve the aggregate from the repository. The complexity is proportional to the number of entities, but each entity definition only requires the definition of its attributes. With regard to aggregate invariants, it is only necessary to extend

the *verifyInvariants* method. The complexity of this task is equal to the complexity of the business logic, but since the goal of the simulator is to verify the transactional behavior of the microservice system given its business logic, their implementation is required.

In the second step, the focus is on the aggregate upstream-downstream relationships. The aggregate microservice API, provided for downstream aggregates, is defined through a set of methods in a class service. These methods only have to change the aggregate state and register the changed aggregate in the *UnitOfWorkService*. In addition, it may be necessary to emit events required by downstream aggregates, also using the *UnitOfWorkService*. Events are defined by extending class *Event*. Furthermore, it is necessary to subscribe to events published by upstream aggregates by extending *EventSubscription* class and redefining aggregate method *getEventSubscriptions*. The complexity associated to this step is proportional to the number of services and events, and no extra work need to be done. The simulator allows event subscription based on the aggregate state, see the example above, which requires the definition of some logic.

Finally, in the third step, the aggregate functionalities are implemented using the Sagas transactional model. To do so, it is defined for each aggregate its set of Saga states by extending *SagaState*. Then, by implementing each aggregate functionality, using the simulator workflow constructs, new states are added to its Saga state. It is also necessary to handle the subscribed events by extending from *EventHandler* and implement the handling functionality. This is the most complex part in terms of the Sagas business logic that deals with semantic locks, but the developer can focus on the core problem, while the simulator provides the mechanisms for functionality steps and transactional management.

To extend the simulator and implement an application, such as the Quizzes tutor example, we can follow a structured approach. This involves defining microservices and establishing interactions between them, using the tournament microservice as a reference.

The process begins with defining the microservice's core structure, which in the simulator is represented by aggregates. Each aggregate encapsulates related data and behavior, ensuring that changes adhere to business rules and consistency requirements. Figure 5.2 illustrates the following procedure that demonstrates how simple it is to implement a microservice in the simulator.

1. **Defining the Aggregate**

   The first step is to define the aggregate class, such as *Tournament*, which represents the microservice's central concept. This class includes essential properties and defines invariants that must be maintained (e.g., tournament start and end times). The aggregate should also specify event subscriptions to react to relevant domain events, thereby ensuring that the microservice remains consistent with changes occurring in other parts of the system.

2. **Creating Repositories**

   Repositories, such as *TournamentRepository*, are necessary for persisting aggregates in the

database. They provide methods for querying and storing aggregates, thus supporting the transactional operations performed during the Saga execution.

3. **Defining Services**

   Aggregate services, such as *TournamentService*, are created to encapsulate business logic operations related to the aggregate. These services are used within functionalities to manipulate the aggregate in a consistent and controlled manner. For example, *TournamentService* might offer methods for creating, updating, and retrieving tournament data.

4. **Handling Events**

   Event handling plays a crucial role in the simulator's architecture. Implementations such as *TournamentEventHandling* and *TournamentEventProcessing* manage domain events that affect the aggregate. These components ensure that the microservice can react to changes in other parts of the system, allowing for asynchronous processing and eventual consistency.

5. **Implementing the Saga Logic**

   To support distributed transactions, the Saga pattern is employed. Each aggregate requires a corresponding Saga implementation, like *SagaTournament*, which manages the aggregate's state throughout the transaction lifecycle. The Saga implementation defines possible states (*TournamentSagaState*) that act as semantic locks, allowing the system to handle intermediate states and ensure data consistency across multiple services.

6. **Implementing Functionalities**

   Functionalities represent the business processes that coordinate the execution of workflows, leveraging the defined services. Each functionality, like *CreateTournamentFunctionalitySagas*, uses workflow steps to define the sequence of operations required for a business process, including compensation actions in case of failure. This allows the system to manage distributed transactions robustly.

7. **Exposing the Functionality via Web API**

   To make the functionalities accessible to the application, web API controllers are implemented, such as *TournamentController*. These controllers expose RESTful endpoints that clients can call to execute functionalities (e.g., creating a tournament). The controllers interact with the underlying services and functionalities, providing a bridge between the client and the business logic.

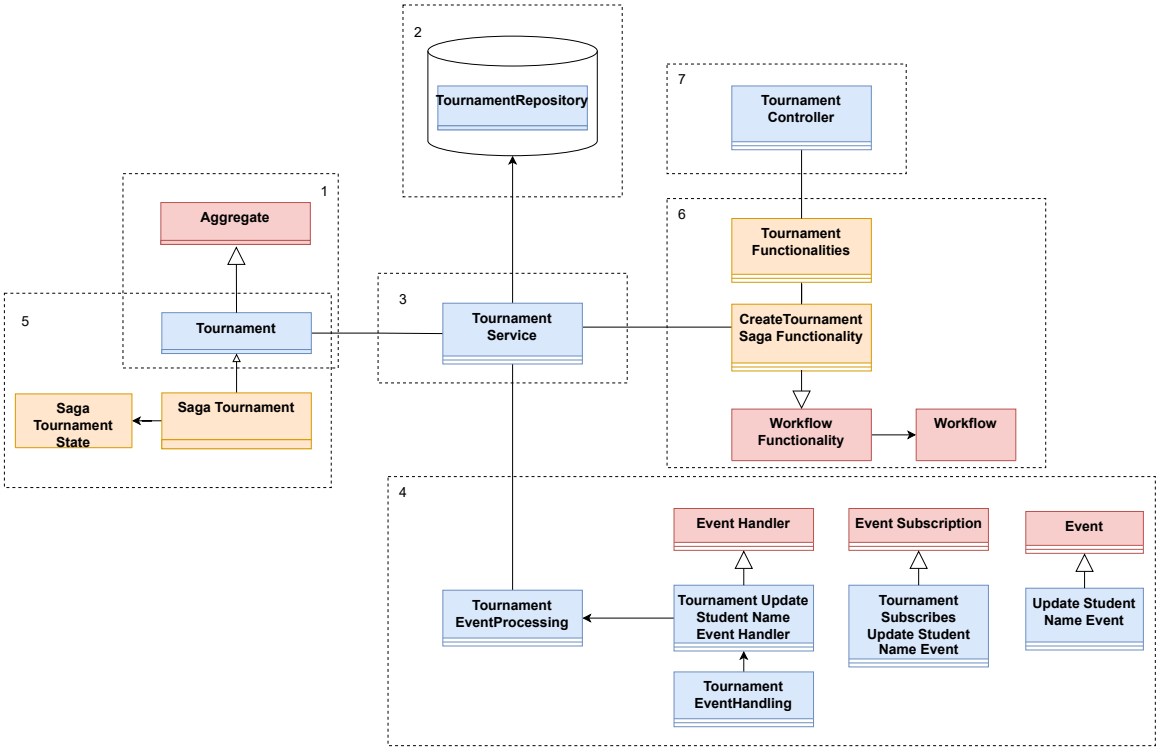


Figure 5.2: Example of the implementation of a microservice in the simulator.

By following these steps, the simulator can be easily extended to implement new microservices or enhance existing ones. As we have seen, the developer doesn't need to change any logic related to the core concepts of the simulator like the *UnitOfWork* and the *Workflow*, instead, he can focus on the implementation of the business logic. The use of aggregates, Sagas, and workflows provides a flexible and modular approach, enabling developers to introduce new functionalities while maintaining consistency and adhering to business rules.

5.1.3 Simulate Interleavings

To use the simulator and test the microservices business logic implemented using Sagas, it is necessary to provide two features: (1) the ability to partially execute a functionality; (2) control the when to process events. The former allows to control when the steps of a functionality are executing, making it possible to interleave their execution with the steps of another functionality. The latter allows to decide when the handling of an event occurs, because events are handled periodically, but that way it is possible to make their handling deterministic.

Listing 5.1 shows a Spock[1] test that exemplifies the use of these features for an interleaving of

[1] https://spockframework.org/

UpdateStudentName and *AddParticipant* functionalities.

Listing 5.1: Concurrent add participant and update name test

```
1  def 'concurrent add participant and update name' () {
2    given: 'two functionalities'
3      def addParticipantFunctionality =
4          new AddParticipantFunctionalitySagas(uow1,...)
5      def updateStudentNameFunctionality =
6          new UpdateStudentNameFunctionalitySagas(uow2,...)
7    and: 'add participant reads student from course'
8      addParticipantFunctionality
9          .executeUntilStep("getStudentStep", uow1)
10   and: 'update name is executed'
11     updateStudentNameFunctionality.executeWorkflow(uow2)
12
13   when: 'add participant executes remaining steps'
14     addParticipantFunctionality.resumeWorkflow(uow1)
15   then: 'student is added with old name'
16     def result = tournamentFunctionalities.findTournament(...)
17     result.getParticipants().find{it.aggregateId ==
18                         aggregateId}.name == ORIGINAL_NAME
19
20   when: 'update name event is processed'
21     tournamentEventHandling.handleUpdateStudentNameEvent();
22   then: 'the student name is updated'
23     result = tournamentFunctionalities.findTournament(...)
24     result.getParticipants().find{it.aggregateId ==
25                         aggregateId}.name == UPDATED_NAME
26 }
```

In the example, the first step to add the participant functionality is executed using the auxiliary method *executeUntilStep*, lines 8-9. Then comes the complete execution of the update student name functionality, line 11, and then the remaining steps of the add participant functionality are executed using the auxiliary method *resumeWorkflow*, line 14. The first check verifies that the student is added as a participant using the original name. Then the handling of the update name event is triggered using the method *handleUpdateStudentNameEvent*, line 21, and the verification returns that it has the updated name (eventual consistency). Note the use of two different units of work, *uow1* and *uow2*, for each of the functionalities, *addParticipantFunctionality* and *updateStudentNameFunctionality*, respectively.

5.1.4 Complex Interleavings

Complex interactions that can arise in distributed microservice systems coordinated by the Sagas transactional model. The simulator supports test case design to force their occurrence in a deterministic context and assess its behavior. These test cases explore scenarios in which different functionalities are executed in parallel or in sequence, affecting the state of multiple microservice aggregates. The test

scenarios can illustrate concurrency issues such as lost updates, dirty reads, and eventual consistency. We highlight how the Sagas model addresses these issues using semantic locks, compensations, and event-based synchronization to maintain consistency across the system.

In Figures 5.3 to 5.8, 6 scenarios are described for the update student name and add participant functionalities, represented, respectively, as *Update* and *Add*. Moreover, an additional functionality is considered, resulting from the handling of the update name event by the *Tournament* aggregate, identified as *Event*. The functionalities are composed of steps that invoke aggregate services. For instance, the add participant functionality comprises two steps that invoke two services: *getStudent* of *Course* and *addParticipant* of *Tournament*. Two aggregates are represented with their timelines: *Course* and *Tournament*.

The initial state of the scenarios, corresponding to their test setup, is the one where the student being added to the tournament as a participant, is the tournament creator, because we intend to exercise the invariant that states that the participant and the creator must have the same name. In addition, we want to verify in which situations there are eventual consistency of the name between the *Course* and the *Tournament*.

The scenarios cover all interleavings to be tested and are split into sequential and concurrent, where in the former there is no interleaving of the functionality steps. The eventual consistency delay occurs between the end of the update student name functionality and the event handling.

In scenario (a) the creator name is updated first, and then the participant is added with the updated name.

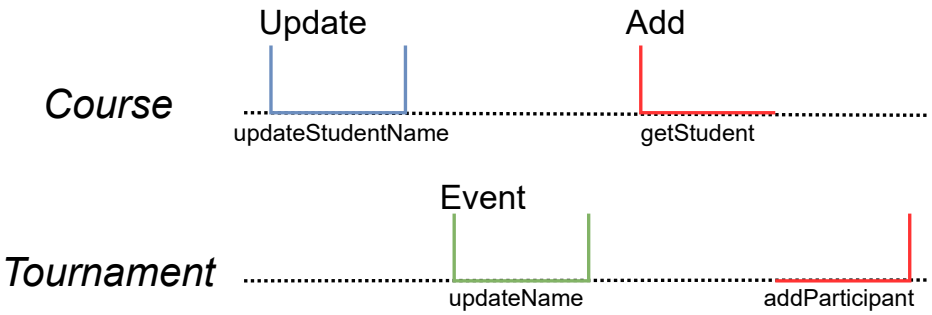


Figure 5.3: Sequential Add Participant and Update Student Name interleavings - (a)

In scenario (b) an invariant violation occurs because the add reads the updated name and tries to add the participant before the event is handled, in which situation the creator name still has the old name, the violation is detected, when, on *registerChanged* of the tournament, the *verifyInvariants* is invoked;

as a result, and also shown in the scenario, the add participant functionality is retried and succeeds because it reads from the updated course.

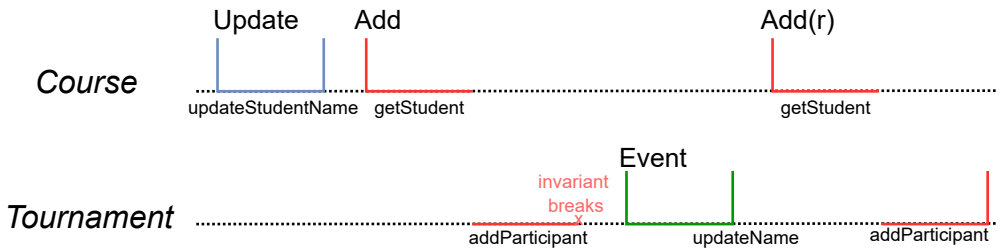


Figure 5.4: Sequential Add Participant and Update Student Name interleavings - (b)

In scenario (c) the add participant is done before the update and when the event is handled both, creator and participant are updated.

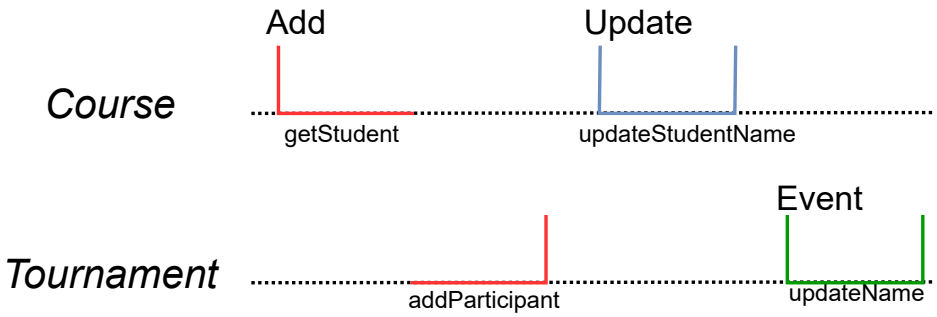


Figure 5.5: Sequential Add Participant and Update Student Name interleavings - (c)

In scenario (d) the name update occurs first, the add reads the update name and the event is processed before the the participant is added, which updates the creator name and the invariant is preserved when the participant is added.

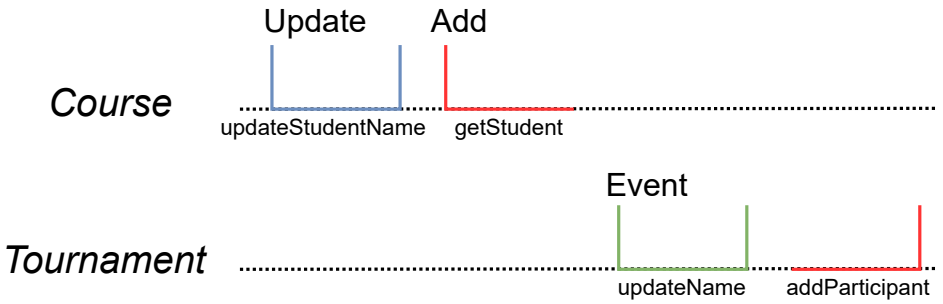


Figure 5.6: Concurrent Add Participant and Update Student Name interleavings - (d)

In scenario (e) an invariant violation is detected because the add participant reads the student before it is updated, and the update name, as well as the event handling occur before the participant is added; after adding the participant, the creator has the updated name while the corresponding participant has the old name.

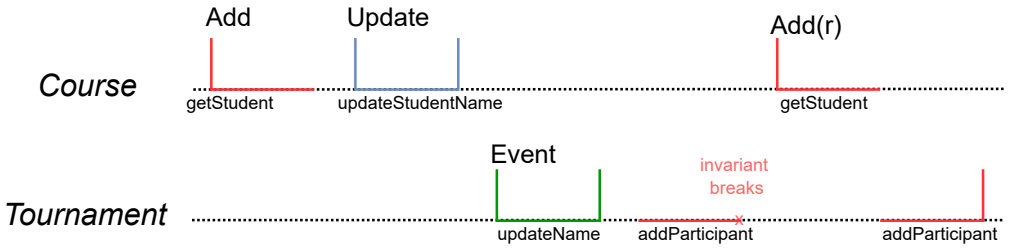


Figure 5.7: Concurrent Add Participant and Update Student Name interleavings - (e)

In scenario (f) the update name occurs in between the add participant, but since the event is only handled in the end, the invariant is not violated because both the creator and the participant have the old name.

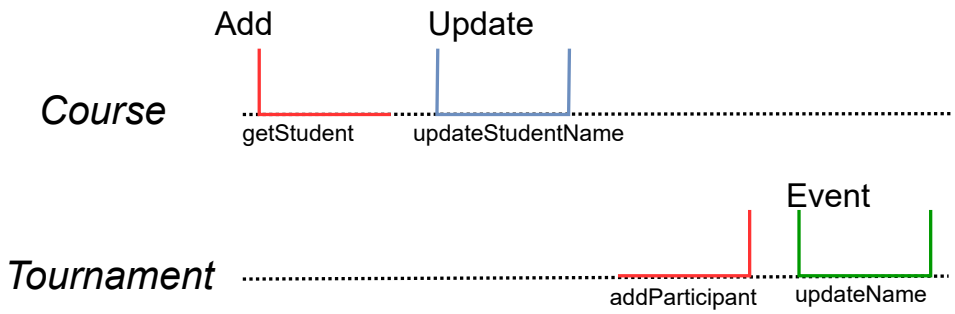


Figure 5.8: Concurrent Add Participant and Update Student Name interleavings - (f)

Figure 5.9 shows a scenario in which there is the interleaving of two executions of an update tournament functionality in the same tournament. The scenario tests that the semantic lock avoids lost updates. Suppose that the first execution is the first to write the tournament and the second to write the quiz, which would result in a quiz that is not consistent with the tournament. A dirty read is also avoided by the semantic lock in the cases where the *updateQuiz* service fails and other functionalities have read the updated tournament (interaction between functionalities not shown in the figure).

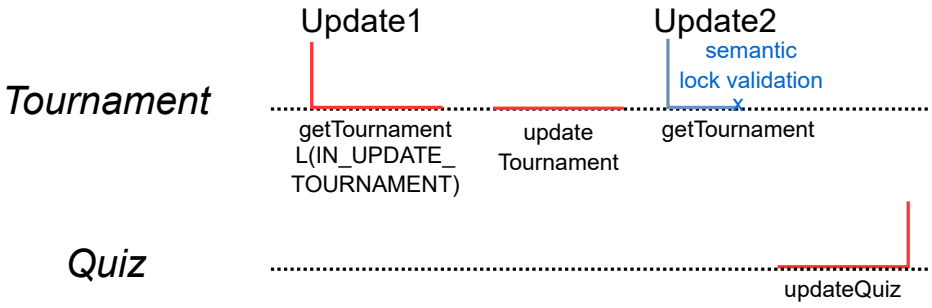


Figure 5.9: Update Tournament interleaving.

5.1.5 A Large Monolith System

In this section, we describe an experiment involving a large microservices system, focusing on the interactions between functionalities and aggregates and the implications for maintaining consistency in a distributed microservices system. The microservices system we analyzed was the Quizzes Tutor application, which consisted of 8 aggregates, and respective microservices: *Course*, *User*, *Topic*, *Question*, *CourseExecution*, *Quiz*, *Answer*, and *Tournament*.

Each of these aggregates was studied in detail to identify the number of aggregate invariants, eventual consistency dependencies between aggregates, which correspond to their event subscriptions, and API methods. Table 5.1 shows a summary:

Table 5.1: Aggregate invariants, event subscriptions, and services

Aggregates	Invariants	Event Subscription	API Methods
Course	2	0	2
User	2	0	6
Topic	0	0	5
Question	0	2	7
CourseExecution	4	1	10
Quiz	7	3	7
Answer	5	5	6
Tournament	13	11	11

The *Course* aggregate, for example, is managed by the *Course* microservice and maintains two invariants to ensure the integrity of its entities. There are no eventual consistency dependencies for this aggregate, because it operates independently within its microservice. In contrast, the *Tournament* aggregate is significantly more complex, containing 13 invariants and 11 event subscriptions, and involves multiple microservices such as *User*, *Quiz*, and *CourseExecution*.

The data above reveals the variety and distribution of invariants and eventual consistency dependencies in a typical microservice system, providing insight into the complexity of enforcing consistency within and across aggregates.

The experiment clearly demonstrates that handling large aggregates with a rich set of business logic in a microservices system requires sophisticated consistency mechanisms, especially when compared to a monolithic architecture where all data consistency is managed in a centralized manner.

A total of 45 functionalities were implemented in the experiment. A functionality is associated with an aggregate, referred as the main aggregate of the functionality, and may invoke the API of upstream aggregates or publish events to be subscribed by downstream aggregates, the secondary aggregates. The following list defines the levels of complexity associated with a functionality, which depend on the number of read and writes it performs on different aggregates:

- A: *Query*: The functionality reads the main and/or secondary upstream aggregate.
- B: *Simple Functionality*: The functionality writes its main aggregate only and can read its main aggregate and other upstream aggregates.
- C: *Complex Functionality*: The functionality writes its main and other secondary upstream aggregates, and it can read the main of other upstream aggregates.
- D: *Event Functionality*: The functionality writes a secondary downstream aggregate. This is an

indirect write because the upstream aggregate does not know the downstream aggregate, but the downstream aggregate may hold some state of the upstream aggregate.

Note that it is possible that a functionality belongs to more than one case, for instance, it can be a simple and event functionality, which is the case of the update student name.

In order to better understand the complexity and dependencies in the quizzes application, we categorized and analyzed the 45 different functionalities based on the previous complexity classification 5.10.

By analyzing Figure 5.10 we can observe a diverse range of complexities across the implemented functionalities of Quizzes Tutor, including cases from all categories. This proves that the simulator is able to support various levels of interaction and coordination, from simple queries to complex operations involving multiple aggregates and event-driven behaviors. The considerable amount of cases C and D, shows that the simulator is capable of testing complex scenarios where distributed transactions and eventual consistency play a critical role. The coverage across all cases also proves the robustness of the simulator when it comes to handling different types of microservice interactions, ensuring that both straightforward operations and more intricate workflows involving distributed transactions and asynchronous event handling are effectively tested.

	A	B	C	D
	FindParticipant FindQuestionByAggregateId FindQuestionsByCourse FindQuiz FindTopicsByCourse FindTournament FindUserById GetAvailableQuizzes GetClosedTournamentsForCourseExecution GetCourseExecutionById GetCourseExecutionsByUser GetCourseExecutions GetOpenedTournamentsForCourseExecution GetStudents GetTeachers GetTopicById GetTournamentsForCourseExecution	ActivateUser AnonymizeStudent AnswerQuestion AddParticipant AddStudent CancelTournament ConcludeQuiz CreateCourseExecution CreateQuestion CreateQuiz CreateTopic CreateUser DeleteTopic DeleteUser LeaveTournament RemoveCourseExecution RemoveQuestion RemoveStudentFromCourseExecution RemoveTournament StartQuiz UpdateQuestion UpdateQuiz UpdateStudentName UpdateTopic	CreateTournament SolveQuiz UpdateQuestionTopics UpdateTournament	AnonymizeStudent AnswerQuestion DeleteUser DeleteTopic RemoveCourseExecution RemoveQuestion RemoveStudentFromCourseExecution UpdateQuestion UpdateStudentName UpdateTopic
Total	17	24	4	10

Figure 5.10: Complexity of the Quizzes Tutor functionalities implemented in the simulator.

5.1.6 Usage

The framework offers a powerful and flexible solution for implementing microservices using a domain-driven design approach. Besides that, the usage of the framework can present some challenges. One of the main difficulties developers can encounter is defining the dependencies between steps when writing a functionality. This process requires a moderate amount of knowledge of the system due to the involvement of semantic locks and event-based communication mechanisms. These locks help prevent lost updates or inconsistent reads but add complexity to the implementation, as they need to be carefully defined based on the system's business logic.

Furthermore, developers must handle events correctly to ensure smooth inter-service communication, which is not straightforward. Different teams working on various microservices might have different interpretations of the events and locks, which could lead to misunderstandings or conflicting implementations. Therefore, proper coordination between teams is essential to ensure the framework is used effectively.

5.1.7 Threats to Validity

Some aspects need to be considered in terms of the threats to the validity of the evaluation:

- *Orchestration vs Choreography* - the simulator implements a functionality in terms of a workflow of services that follow an orchestration approach. This is a limitation of the simulator because it does not support a set of microservices where the coordination of services is a choreography. However, given the complexity of business logic in an eventual consistent environment, applications that have a rich business logic should adopt the orchestration strategy [9].

- *Asynchronous Communication* - the communication between microservices is usually asynchronous, using message queues, to increase the system robustness to temporary failures in microservices. Although we support both synchronous and asynchronous steps, the evaluation was performed using the synchronous version. Although this is a limitation of the evaluation, it is also recommended that the coordinator waits for the response from the microservice before progressing to the next, to simplify the business logic, although the communication between the coordinator and the microservices is done asynchronously [9]. Note that the events in the simulator are asynchronous, but relate to a different type of interaction between microservices (upstream-downstream), and those are evaluated.

6

Conclusion

Contents

6.1 Conclusions

In conclusion, this thesis' proposed microservice system simulator offers a useful tool for handling the challenges involved in developing microservices, especially when utilizing the Sagas transactional model. Through the use of a domain-driven design approach, the simulator considerably lowers the risks and expenses associated with problems that arise later in the development process by enabling the early identification and resolution of possible difficulties related to data consistency and coordination. The simulator's evaluation shows that it can handle intricate relationships between aggregates, guaranteeing that invariants are maintained and conflicts are successfully resolved via compensating transactions and semantic locks.

One of the simulator's key strengths is its flexibility in simulating different transactional models. While it is primarily designed to simulate the Sagas pattern with eventual consistency, it also supports Trans-

actional Causal Consistency (TCC), a stronger consistency model that ensures causal relationships between transactions are respected. This dual capability allows the simulator to model both eventual consistency systems, which prioritize performance and availability, and systems that require stricter consistency guarantees. By providing support for TCC, the simulator extends its applicability to a wider range of use cases, enabling developers to evaluate the trade-offs between different consistency models and choose the most appropriate one for their system.

The ease of extension and support for simulating intricate interleavings further highlights the practical utility of this tool for developers seeking to optimize the design and functionality of their microservice-based systems. Ultimately, before committing to expensive implementations in distributed environments, developers can use this simulator to investigate various coordination mechanisms, evaluate the viability of microservice architectures, and make well-informed design decisions. In order to further increase the simulator's usefulness in simulating real-world microservice behaviors, future development will concentrate on expanding it to better accommodate asynchronous step executions.

Moreover, while the current focus is on synchronous transactions, the simulator is poised for future development to incorporate asynchronous step executions, which will more accurately reflect real-world microservice environments where message delays and network latencies are common. Expanding the simulator to better accommodate asynchronous operations will further enhance its ability to simulate realistic system behaviors and evaluate their performance in distributed contexts.

Overall, the proposed simulator not only simplifies the early stages of microservice system design but also contributes to the broader effort of developing reliable and consistent distributed systems using state-of-the-art architectural patterns. By supporting both Sagas and TCC models, it provides a comprehensive platform for testing and improving microservice architectures, making it a highly versatile tool for developers working on distributed applications.

The simulator and the Quizzes Tutor system experiment are available in a public repository.

6.2 System Limitations and Future Work

The development of the microservice simulator, while successful in achieving its goals of simulating distributed systems using the Saga architectural pattern, has certain limitations. One of the most notable constraints is the reliance on synchronous communication. While asynchronous communication, typical in real-world microservices, is supported by the simulator, the evaluation largely depended on synchronous transactions. This creates a gap when attempting to fully simulate environments where microservices operate asynchronously, particularly when considering message queuing systems that delay responses or introduce network latency.

Another limitation is that the simulator currently supports orchestration-based coordination, which

simplifies workflow management by centralizing control. However, many microservice systems prefer choreography-based coordination, where services manage interactions independently. This divergence from real-world usage patterns may result in a simulation that does not fully capture all interaction nuances, especially in systems that emphasize high service autonomy.

The handling of compensating transactions also presents challenges. While the simulator incorporates compensating actions for failure recovery, these are primarily orchestrated. Future improvements should extend support to hybrid models, where not all transactions rely on a single transactional pattern. This would include scenarios where ACID transactions coexist with Saga-based eventual consistency models, better representing a broader range of distributed systems.

Moreover, the usage of the framework presents certain challenges, especially when developers need to define dependencies between steps in a functionality. These dependencies often involve the use of semantic locks and event-based communication mechanisms, which prevent issues like lost updates and inconsistent reads but add complexity to the design. Developers must carefully define these mechanisms based on the system's business logic, which requires a moderate to high level of system knowledge. However, a potential improvement would be to enhance the simulator to automatically calculate dependencies between steps in a functionality. By analyzing the events, locks, and services involved, the simulator could suggest or enforce the correct sequence of actions, helping developers reduce errors and streamline the implementation process.

6.2.1 Future Work

Several enhancements could address the identified limitations:

- **Asynchronous Communication Enhancements**: Expanding the evaluation to incorporate asynchronous communication steps will allow the simulator to more accurately reflect real-world message queuing and delayed responses.
- **Choreography-based Coordination**: Introducing choreography alongside orchestration will enable developers to explore decentralized coordination strategies, allowing greater flexibility in simulating real-world systems.
- **Support for Hybrid Transaction Models**: Extending the simulator to manage hybrid transactional environments where some services utilize strict ACID transactions while others rely on eventual consistency models like Sagas.
- **Dependency Calculation Between Steps**: Developing a feature within the simulator that calculates the dependencies between steps in a functionality based on semantic locks, events, and service interactions. This feature could help automate the detection of correct step sequences and prevent inconsistencies during development.

- **Integration with Cloud Environments**: Adding the ability to simulate microservices running on distributed cloud platforms to enable testing under more realistic distributed conditions and provide insights into cloud-native system behavior.

By addressing these areas, the simulator could become a more powerful tool for designing and testing large-scale, resilient, and efficient microservice-based systems.

Bibliography

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

[2] J. Thönes, “Microservices,” *IEEE Software*, vol. 32, no. 1, pp. 116–116, 2015.

[3] N. Ford, M. Richards, P. Sadalage, and Z. Dehghani, *Software Architecture: The Hard Parts*. O’Reilly Media, Inc., 2021.

[4] A. Fox and E. A. Brewer, “Harvest, yield, and scalable tolerant systems,” in *Proceedings of the The Seventh Workshop on Hot Topics in Operating Systems*, ser. HOTOS ’99. USA: IEEE Computer Society, 1999, p. 174.

[5] N. C. Mendonça, C. Box, C. Manolache, and L. Ryan, “The monolith strikes back: Why istio migrated from microservices to a monolithic architecture,” *IEEE Software*, vol. 38, no. 5, pp. 17–22, 2021.

[6] N. Nader-Rezvani, *Agile Quality Test Strategy*. Berkeley, CA: Apress, 2019, pp. 121–138. [Online]. Available: https://doi.org/10.1007/978-1-4842-3751-9_7

[7] N. Santos and A. Rito Silva, “A complexity metric for microservices architecture migration,” in *2020 IEEE International Conference on Software Architecture (ICSA)*, 2020, pp. 169–178.

[8] H. Garcia-Molina and K. Salem, “Sagas,” in *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD ’87. New York, NY, USA: Association for Computing Machinery, 1987, p. 249–259.

[9] C. Richardson, *Microservices Patterns*. Manning Publications Co., 2019.

[10] E. Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison Wesley, 2003.

[11] D. Haywood, “In defense of the monolith,” in *Microservices vs. Monoliths - The Reality Beyond the Hype*. InfoQ, 2017, vol. 52, pp. 18–37. [Online]. Available: https://www.infoQ.com/minibooks/emag-microservices-monoliths

[12] M. Štefanko, O. Chaloupka, and B. Rossi, "The saga pattern in a reactive microservices environment," in *Proceedings of the 14th International Conference on Software Technologies - Volume 1: ICSOFT,*, INSTICC. SciTePress, 2019, pp. 483–490.

[13] K. Dürr, R. Lichtenthäler, and G. Wirtz, "An evaluation of saga pattern implementation technologies," in *Central-European Workshop on Services and their Composition*, 2021.

[14] "Blcm simulator," https://github.com/socialsoftware/business-logic-consistency-models, accessed: 2023-10-13.

[15] H. Vural and M. Koyuncu, "Does domain-driven design lead to finding the optimal modularity of a microservice?" *IEEE Access*, vol. 9, pp. 32 721–32 733, 2021.

[16] O. Özkan, Önder Babur, and M. van den Brand, "Domain-driven design in software development: A systematic literature review on implementation, challenges, and effectiveness," 2023.

[17] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, P. Merson, R. Nord, and J. Stafford, *Documenting Software Architectures: Views and Beyond, Second Edition*, ser. SEI Series in Software Engineering. Upper Saddle River, NJ: Addison-Wesley, 2010.

[18] Neal Ford, Mark Richards, Pramod Sadalage, Zhamak Dehghani, *Software Architecture: The Hard Parts*. O'Reilly Media, Inc, 2022.

[19] D. D. Akkoorath, A. Z. Tomsic, M. Bravo, Z. Li, T. Crain, A. Bieniusa, N. Preguiça, and M. Shapiro, "Cure: Strong semantics meets high availability and low latency," in *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*, 2016, pp. 405–414.

[20] P. Pereira and A. R. Silva, "Transactional causal consistent microservices simulator," in *Distributed Applications and Interoperable Systems*, M. Patiño-Martínez and J. Paulo, Eds. Cham: Springer Nature Switzerland, 2023, pp. 57–73.

[21] M. Fowler, *Patterns of Enterprise Application Architecture*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[22] P. Pereira and A. R. Silva, "Microservices simulator: An object-oriented framework for transactional causal consistency," submitted.

[23] "Eventuate tram," https://github.com/eventuate-tram/eventuate-tram-core, accessed: 2023-10-13.

[24] "Temporal," https://temporal.io/, accessed: 2023-10-13.

[25] X. Limón, A. Guerra-Hernández, A. J. Sánchez-García, and J. C. Peréz Arriaga, “Sagamas: A software framework for distributed transactions in the microservice architecture,” in *2018 6th International Conference in Software Engineering Research and Innovation (CONISOFT)*, 2018, pp. 50–58.

[26] W. Kun, D. Han, Y. Zhang, S. Lu, D. Chen, and L. Xie, “Ndp2psim: a ns2-based platform for peer-to-peer network simulations,” in *Proceedings of the 2005 International Conference on Parallel and Distributed Processing and Applications*, ser. ISPA’05. Berlin, Heidelberg: Springer-Verlag, 2005, p. 520–529. [Online]. Available: https://doi.org/10.1007/11576259_57

[27] H. Wan and N. Ishikawa, “Design and implementation of a simulator for peer-to-peer networks: optimal-sim,” in *PACRIM. 2005 IEEE Pacific Rim Conference on Communications, Computers and signal Processing, 2005.*, 2005, pp. 105–108.

[28] J. Pujol-Ahullo and P. Garcia-Lopez, “Planetsim: An extensible simulation tool for peer-to-peer networks and services,” in *2009 IEEE Ninth International Conference on Peer-to-Peer Computing*, 2009, pp. 85–86.

[29] A. Montresor and M. Jelasity, “Peersim: A scalable p2p simulator,” in *2009 IEEE Ninth International Conference on Peer-to-Peer Computing*, 2009, pp. 99–100.

[30] P. Urban, X. Defago, and A. Schiper, “Neko: a single environment to simulate and prototype distributed algorithms,” in *Proceedings 15th International Conference on Information Networking*, 2001, pp. 503–511.

[31] L. Leonini, E. Rivière, and P. Felber, “Splay: distributed systems evaluation made simple (or how to turn ideas into live systems in a breeze),” in *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, ser. NSDI’09. USA: USENIX Association, 2009, p. 185–198.

[32] Y. A. Liu, S. D. Stoller, and B. Lin, “From clarity to efficiency for distributed algorithms,” *ACM Trans. Program. Lang. Syst.*, vol. 39, no. 3, May 2017. [Online]. Available: https://doi.org/10.1145/2994595

[33] Y. Zhang, Y. Gan, and C. Delimitrou, “uqsim: Scalable and validated simulation of cloud microservices,” 2019. [Online]. Available: https://arxiv.org/abs/1911.02122

[34] M. Ciavotta, M. Alge, S. Menato, D. Rovere, and P. Pedrazzoli, “A microservice-based middleware for the digital factory,” *Procedia Manufacturing*, vol. 11, pp. 931–938, 2017, 27th International Conference on Flexible Automation and Intelligent Manufacturing, FAIM2017, 27-30 June 2017, Modena, Italy. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2351978917304055

[35] R. Mahmud, S. Pallewatta, M. Goudarzi, and R. Buyya, “ifogsim2: An extended ifogsim simulator for mobility, clustering, and microservice management in edge and fog computing environments,” *Journal of Systems and Software*, vol. 190, p. 111351, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121222000863

[36] H. Shi, X. He, T. Wang, and Z. Wang, “Servicesim: A modelling and simulation toolkit of microservice systems in cloud-edge environment,” in *Service-Oriented Computing*, F. Monti, S. Rinderle-Ma, A. Ruiz Cortés, Z. Zheng, and M. Mecella, Eds. Cham: Springer Nature Switzerland, 2023, pp. 258–272.

[37] M. G. Khan, J. Taheri, A. Al-Dulaimy, and A. Kassler, “Perfsim: A performance simulator for cloud native microservice chains,” *IEEE Transactions on Cloud Computing*, vol. 11, no. 2, pp. 1395–1413, 2023.

[38] S. Frank, L. Wagner, A. Hakamian, M. Straesser, and A. van Hoorn, “Misim: A simulator for resilience assessment of microservice-based architectures,” in *2022 IEEE 22nd International Conference on Software Quality, Reliability and Security (QRS)*, 2022, pp. 1014–1025.

[39] T. Kuhr, T. Forster, T. Braun, and R. Gotzhein, “Feral — framework for simulator coupling on requirements and architecture level,” in *2013 Eleventh ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2013)*, 2013, pp. 11–22.

[40] S. Braun, A. Bieniusa, and F. Elberzhager, “Advanced domain-driven design for consistency in distributed data-intensive systems,” in *Proceedings of the 8th Workshop on Principles and Practice of Consistency for Distributed Data*, ser. PaPoC ’21. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3447865.3457969

[41] M. R. S. Sedghpour, A. O. Duque, X. Cai, B. Skubic, E. Elmroth, C. Klein, and J. Tordsson, “Hydragen: A microservice benchmark generator,” in *2023 IEEE 16th International Conference on Cloud Computing (CLOUD)*, 2023, pp. 189–200.

[42] H. H. A. Valera, M. Dalmau, P. Roose, J. Larracoechea, and C. Herzog, “Draceo: A smart simulator to deploy energy saving methods in microservices based networks,” in *2020 IEEE 29th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*, 2020, pp. 94–99.

[43] Y. Zhang, Y. Gan, and C. Delimitrou, “uqsim: Scalable and validated simulation of cloud microservices,” 2019.

[44] M. Fowler, *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2003.

[45] W. M. van Der Aalst, A. H. Ter Hofstede, B. Kiepuszewski, and A. P. Barros, “Workflow patterns,” *Distributed and parallel databases*, vol. 14, pp. 5–51, 2003.

[46] D. Caromel, L. Henrio, and B. P. Serpette, “Asynchronous sequential processes,” *Information and Computation*, vol. 207, no. 4, pp. 459–495, 2009. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0890540108001582