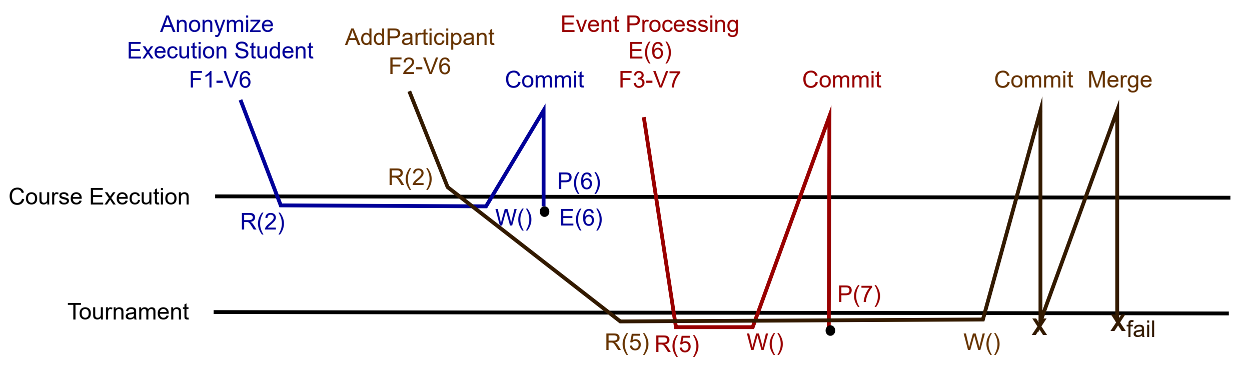

Transactional Causal Consistent Microservices Business Logic

Pedro Manuel Lopes Pereira

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: Prof. António Rito Silva

Examination Committee

Chairperson: Prof. Daniel Jorge Viegas Gonçalves
Supervisor: Prof. António Rito Silva
Member of the Committee: Prof. Luís Eduardo Teixeira Rodrigues

October 2022

This work was created using LaTeX typesetting language
in the Overleaf environment (www.overleaf.com).

Acknowledgments

I would like to thank my parents and grandparents for their emotional support and motivation.

I would also like to acknowledge my dissertation supervisor, Prof. António Rito Silva, for his all his support: for providing useful source material, for discussing core aspects of the implementation, for helping reviewing code and text and most importantly for his high and constant availability throughout all these months of development.

I'd like to thank Prof. Luís Rodrigues and Rafael Soares for providing very useful insight on TCC.

I would also like to thank the Overleaf team for providing such a valuable and easy to use tool for free, without which writing this thesis would have been much harder.

Last but not least, I'd like to thank Instituto Superior Técnico and all of my teachers throughout the years for providing me the technical skills which made developing this thesis possible.

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

Microservices architecture has been widely adopted to develop software systems, but some of its trade-offs are often ignored. In particular, the introduction of eventual consistency has a huge impact on the complexity of the application business logic design. Recent proposals to use transactional causal consistency in serverless computing seem promising, because it reduces the number of possible concurrent execution anomalies than can occur due to the lack of isolation. We propose an extension of the aggregate concept, the basic building block of microservices design, that is transactional causal consistent compliant. A simulator for the enriched aggregates is developed to allow the experimentation of this approach with a business logic rich system. From the experiment we observed a reduction of the implementation complexity. Additionally, we developed a simulator, a publicly available reusable artifact that can be used in other experiments.

Keywords

Microservices Architecture; Aggregates; Transactional Causal Consistency; Eventual Consistency; Simulator

Resumo

A arquitetura de microserviços tem sido bastante utilizada para implementar sistemas de software mas algumas das suas limitações tendem a ser ignoradas. Em particular, a introdução de consistência eventual tem um impacto considerável na complexidade no desenho da lógica de negócio. Existem propostas recentes para utilizar consistência causal transacional em sistemas computação serverless que parecem promissoras, porque reduzem número de anomalias de execução concorrentes que podem ocorrer devido à falta de isolamento. Propomos uma extensão do conceito de agregado, o principal elemento do desenho de microserviços, que seja compatível com consistência causal transacional. Foi desenvolvido um simulador para os agregados enriquecidos para permitir a experimentação desta estratégia com um sistema com lógica de negócio mais complexa. Da experiência realizada observou-se uma redução da complexidade de implementação, e produziu-se um simulador que é um artefacto reutilizável que pode ser usado noutras experiências.

Palavras Chave

Arquitetura de Microserviços; Agregados; Consitência Causal Transacional; Consistência Eventual; Simulador

Contents

List of Figures

List of Tables

Listings

Acronyms

CRDT	Conflict-free Replicated Data Type
DDD	Domain Driven Design
JPA	Java Persistance API
TCC	Transactional Causal Consistency

1 Introduction

Contents

Microservices have become increasingly adopted [1] as the architecture of business systems, because it promotes the split of the domain model into consistent pieces that are owned by small agile development teams, and facilitates scalability [2, 3].

These systems are implemented using the saga pattern [4] to handle the concurrency anomalies, like lost update and dirty reads, which result on extra complexity to the implementation of the system business logic [5]. It has been identified a trade-off between the business logic complexity and use of microservices [6], which is also confirmed by the type of systems where the use of microservices has been successfully reported, where there is the need for high scalability, but the domain business logic is less complex, e.g. Netflix.

Recent research has proposed the use of transactional causal consistency to support serverless computing [7, 8], which reduce the number of anomalies by providing a causal snapshot to support the distributed execution functionalities. However, as far as we know, there is no experimentation of this approach with systems that have complex business logic. On the other hand, the two implementations provide a low level API which is not friendly for the software developer.

Therefore, we intend to answer the following research questions:

1. Does the use of transactional causal consistency simplify the microservices implementation of business logic rich system?

2. Can the use of a transactional causal consistency simulator ease the experimentation with large domain models?

To answer the research questions, we extend the domain-driven concept of aggregate [9], which is the basic building block of microservices systems [4], with causal consistency semantics. Afterwards, it is designed, and implemented, a simulator for aggregates enriched with transactional causal consistency. Then, a large software system, with rich business logic, is implemented. Finally, the results are evaluated.

As result of this work, a set of constructs are proposed to enrich aggregates, a transactional causal consistency simulator is made available, and a classification on how to implement business logic with transactional causal consistency is defined.

After the introduction in this Chapter, related work is presented in Chapter 2. The constructs for aggregate definition are presented in Chapter 3, that also defines the semantics of transactional causal consistency for microservices systems build with aggregates. Chapter 4 describes the simulator design and implementation. Ch Chapter 5 describes the case study complexity and analyses of the implementation of the case study using the simulator, as well as the simulator semantics. Finally, Chapter 6 draws the conclusions.

1.1 Goals

We intend to leverage on existing work by enriching the concept of aggregate to support TCC semantics and to develop our own custom semantics, inspired by CRDT's and Avanced Domain Driven Design (DDD) mechanisms, to maintain system consistency. Additionally, develop a TCC simulator that eases the experiment with the implementation of complex microservices systems, applying TCC semantics to the functionalities execution.

2 Related Work

Contents

2.1 Microservices

Eventual consistency [10] has been adopted in the implementation of microservices, using the sagas [4, 11]. However, writing application business logic in the context of the eventual consistency requires an extra effort [5] to deal with anomalies like lost updates and dirty reads. This is due to the intermediate states created by the functionality execution in each one of the microservices. Due to this lack of isolation, the business logic is intertwined with the handling of incomplete states associated with the current execution with other functionalities. The complexity depends on the number of these intermediate states, which depend on the number of functionalities [5].

Therefore, implementing a system using microservices is not trivial, depends on the complexity of the business logic [6] and sometimes is too complex and systems are migrated back to a monolith architecture [12].

2.2 Eventual Consistency VS Transactional Causal Consistency

Eventual consistency is a model [4] that guarantees that replicas of the same entity can be inconsistent for a period of time but will eventually converge to the same value. One way to achieve convergence is by issuing events. These events represent a change that has occurred to the state of the system at a given node. Other nodes that wish to remain updated with information from other nodes must subscribe to the respective events. In this model the system may be inconsistent for some time, corresponding to the interval between a change in the system state and the processing of the generated event by other nodes.

Transactional Causal Consistency (TCC) [13] has been proposed as a transactional model that handles some of the problems of eventual consistency, by providing to the transaction a consistent causal snapshot. The entities in the causal snapshot respect the *happens-before* relation and the writes performed when the transactional are atomically visible, regardless of node in which read or write operation occurs. This transactional model handles dirty reads, because the reads are consistent, but continues to allow lost updates, which occur when two concurrent transactions write the same entity, the last to commit overwrites the first one. On the other hand, transactional causal consistency can be implemented using non-blocking algorithms, overcoming the limitations stated in the CAP theorem.

As far as we know, there are two implementations of TCC for serverless computing [7, 8], but these implementations use a key-value store and/or offer a low level API that does not facilitate to experiment the support of complex business logic with TCC. Additionally, they only use toy cases to experiment the application of transactional causal consistency.

On the other hand, the design of microservices is based on the domain-driven design concept of aggregate [4, 9], which denotes the transactional unit of consistency in microservices. This concept is not considered by the existing implementations of TCC, which focus on the management of replicas, instead of the different perspective of the same model in different bounded contexts [9]. However, there are some synergies between both concepts that are worth exploring, but, as far as we know, are not addressed by the literature.

2.3 Advanced Domain Driven Design

There is some research on the extension of aggregates to ensure in data-intensive distributed systems, such as microservices, consistency between replicas of entities [14].

When replicas of a certain bounded context diverge in values it becomes necessary to propagate the updates and reconcile them. That's done by creating a reconciliation branch on each replica, copying all concurrent operations to it, sorting them in a deterministic way that's the same on all replicas, execute them from the last state before the replicas diverged and persist the result. Further operations will be applied to the reconciliation branch which becomes the main branch.

One of the most important factors of the reconciliation process is the sorting of operations. Operations need to be sorted in a way that their sequential execution results in system state that's intended. The authors propose the following classification for operations:

- *Incremental Update*: updates an object by taking into account its state before the update. Ex: counter increments and list appends.
- *True Blind Update*: an update that is completely agnostic of the previous state of the object.

2.4 Conflict-Free Replicated Data Types

Other relevant work that deals with the replica consistency is on conflict-free replicated data types [15, 16] that are another way of ensuring consistency of replicas of objects by guaranteeing every replica converges independently and deterministically to the same value. Convergence of replicas is achieved by semantics: for the same set of events, two replicas will reach the same state by processing the events both according to the same set of rules. For set Conflict-free Replicated Data Type (CRDT)'s can have the following semantics:

- *add-wins*: in the case of a concurrent add and remove of the same object to a set, the add operation wins.
- *remove-wins*: in the case of a concurrent add and remove of the same object to set, the remove operation wins

- *last-writer-wins*: in the case of a concurrent add and remove of the same object to a set, the update that was occurred later, in terms of physical time, wins.

It is easy to conclude that adding and removing an element from a set are not commutative operations, i.e. the order in which they are applied matters. If two operations are commutative they can both be applied in any order ad the final result will be the same on both replicas.

3

Semantics

Contents

3.1 Aggregate Specification

Aggregates are considered the basic building blocks of microservices applications [4]. The concept was imported from domain-driven design [9] and defines a unit of consistency between the aggregate entities, which is defined as the aggregate invariants. An aggregate has a root entity that controls its accesses to guarantee atomicity and aggregate internal entities are not visible from outside. In the context of microservices, aggregate accesses occur in the context of ACID transactions.

When splitting a domain model into aggregates it is necessary to consider the consistency between aggregates. In domain-driven design the consistency between aggregates can be relaxed. To distinguish these two types of consistency, we define intra-invariants and inter-invariants. The former defines a rule on the aggregate entities while the latter between different aggregates. We also consider a upstream-downstream relation between aggregates, which is similar to the same relation between bounded contexts in domain-driven design. An inter-variant is defined in the downstream aggregate, because it is aware of the upstream aggregate model, while the opposite does not occur.

Listing 3.1: Tournament Aggregate

```
Aggregate Tournament
    Integer id
    LocalDateTime startTime, endTime
    List<Participant> participants
    Creator creator
    ...
    Participant is s: CourseExecution.students
        id from s.id
        name from s.name
    Creator is c: CourseExectuion.students
        id from s.id
        name from s.name
    ...
    Intra-Invariants
        START_BEFORE_END
            this.startTime < this.endTime
        CREATOR_PARTICIPANT_CONSISTENCY
            forall p : this.participants | p.id == this.creator.id =>
                p.name == this.creator.name
    ...
    Inter-Invariants
        PARTICIPANT_EXISTS
        forall p : this.participants | p.state != INACTIVE =>
                exists s = CourseExecution.students(p.id)
                p.name == s.name
        CREATOR_EXISTS
            this.creator.state != INACTIVE =>
                exists s = CourseExecution.students(this.creator.id)
                this.creator.name = s.name

    ...
```

Listing 3.1 contains the representation of an excerpt of a tournament aggregate. The root entity is the tournament, which has a unique identifier. Attributes *startTime* and *endTime* represent the period the tournament is open. There are two internal entities: *participants* which represents users participating in the tournament and *creator* which represents a user which created the tournament. Both the *Participant* and Creator entities are associated with a student enrolled in a course exectuion, which is an element of the upstream aggregate (*CourseExecution*), and contains the student identification and name. Therefore, a consistency issue may occur between these two aggregates, which is declared by the *PARTICIPANT_EXISTS* inter-invariant and the *CREATOR_EXISTS* inter-invariant. Finally, *START_BEFORE_END* and *CREATOR_PARTICIPANT_CONSISTENCY* declare intra-aggregate invariants, which should be preserved whenever the aggregate is changed.

Listing 3.2: Tournament Functionalities

```
Functionality updateTimes
    update this.startTime
    update this.endTime

Functionality anonymizeExecutionStudent(p)
    update this.participants(p.id).name
    p.id == this.creator.id => update this.creator.name
```

Aggregates have functionalities. In listing 3.2 declares two tournaments functionalities: *update-Dates* that updates the tournament open period; and *anonymizeExecutionStudent* that anonymizes a participant or a creator. The latter is related with the *PARTICIPANT_EXISTS* and *CREATOR_EXISTS* inter-invariants.

3.2 Functionalities

To define the aggregates semantics in a microservices architecture, when the functionalities are executed using transactional causal consistency, it is necessary to classify the types of functionalities that exist. A functionality is associated with an aggregate, referred as the functionality aggregate or the main aggregate, where it preforms reads and/or writes. A functionality can perform reads and writes on other aggregates, besides the main aggregate, they are referred as the functionality secondary aggregates. Finally, due to architectural upstream-downstream relationship between aggregates, a functionality can directly perform reads and writes on the upstream aggregates of its main aggregate, but cannot do any type of access to the downstream aggregates. Therefore, the secondary aggregates of a functionality have to be upstream aggregates of its main aggregate. Nevertheless, a functionality can publish events that may be subscribed by downstream aggregates, which correspond to a kind of indirect write, if the downstream aggregate uses the event to change its state, but the initiative to perform the change is on the downstream aggregate, which is aware of the semantics of its upstream aggregates. Note that, a functionality cannot perform a read on a downstream aggregate, because publishing events is asynchronous, and, from an architectural perspective, the upstream aggregate is unaware of downstream aggregates.

Therefore, considering these concepts, the functionalities can be classified in the following types:

- *Query*: this type of functionality only contains read operations to aggregates, and is distinguished by the number of aggregates it reads:
  - *Single Aggregate* - all reads belong to the same aggregate, the main aggregate;
  - *Multiple Aggregates* - the reads are done in several aggregates, the main aggregate and one or more secondary aggregates.
- *Simple Functionality*: characterizes the functionalities that write in a single aggregate, the main aggregate, though they may read different aggregates, and so, it is also distinguishes by the number of aggregates it reads:
  - *Single Aggregate* - all reads belong to the same aggregate, the main aggregate, where the write also occurs;
  - *Multiple Aggregates* - the reads are done in several aggregates, the main aggregate, and one or more secondary aggregates.

  The write on the main aggregate may trigger events to be subscribed by downstream aggregates;
- *Complex Functionality*: writes in multiple aggregates, the main aggregate and one or more secondary upstream aggregates. Like in the previous type, it can read aggregates, either a single one or multiple. Additionally, the writes on the aggregates may trigger events to be subscribed by downstream aggregates.

3.3 Transactional Causal Consistency

To define functionalities execution semantics in the context of transactional causal consistency, we introduce the concept of version number. Each aggregate has several versions, where A is the set of all aggregate version, and each version has a unique number, denoted by $a.version$, where $a \in A$. The version numbers form a total order, i.e. it is possible to compare any two version numbers, $\forall_{a_i,a_j \in A} a_i.version \leq a_j.version \vee a_j.version < a_i.version$. Given an aggregate version, $a \in A$, $a.aggregate$ denotes its aggregate, and $a.versions$ denotes all the aggregates versions of aggregate $a.aggregate$. Additionally, there is a version number for each functionality f, denoted by $f.version$, that is assigned when the functionality starts with the number of the last successfully finished functionality incremented by one. This number may be subsequently changed, as it will be described, and assigned as the version number of all the aggregates written by the functionality. F is the set of all executed functionalities, and $F.success \subseteq F$ is the subset of functionality executions that finished successfully. A causal snapshot of an executing functionality is a set of aggregates which are causality consistent given the functionality version number. Therefore, a functionality f causal snapshot, denoted by $f.snapshot$, is a set of aggregate versions, $f.snapshot \subseteq A$, such that there is a single version for each aggregate, $\nexists_{a_i,a_j \in f.snapshot} a_i.aggregate = a_j.aggregate$, and the following condition holds:

$$\forall_{a_i \in f.snapshot} : a_i.version < f.version \wedge$$
$$\forall_{a_j \in a_i.versions} : a_j.version < a_i.version$$

The first condition guarantees that the version was not created by a functionality that finished after f started, and the second condition guarantees that it is the most recent version, considering the first condition. As an example, consider an aggregate version a which as version number 5, it was written by functionality with version number 5. Also consider two functionalities, f_i and f_j, that start concurrently, when the last successfully finished functionality have 7 as its version number. So, the version number of f_i and f_j is 8. Suppose the f_i finishes first and writes a version of a, which will have version number 8. If f_j reads a to its snapshot after f_i finishes, it finds versions 5 and 8, but it will add version 5 because it is the most recent version smaller than 8, which is the f_j executing version number. The execution of a functionality f in a transactional causal consistent context follows the following steps:

1. When the functionality f starts, $f.version = max(F.success.version) + 1$, where $F.success.version$ is the set of version numbers of all the functionalities executions that finished successfully;

2. Whenever an aggregate is read, it is selected a version according to do the functionality snapshot conditions, and added to it, if not already there;

3. Whenever an aggregate is written, it is selected a version according to do the functionality snapshot conditions, and added to it, if not already there. In case the aggregate is being created, it is

assigned the functionality version number. $f.written$ denotes the subset of $f.snapshot$ of the written aggregates;

4. When the functionality finishes execution it proceeds to commit, and if the process succeeds the functionality finishes successfully, otherwise it aborts. It does the following actions:

   (a) The written aggregates preserve the intra-invariants, i.e. $\forall_{a \in f.written,\ irv \in a.intraInv} a.irv$, where $a.intraInv$ denotes the aggregate intra invariants, and $a.irv$ denotes evaluation of the invariant in the a version;

   (b) For each aggregate versions to be written get, if exists, the most recent version of the same aggregate that was created by a concurrent functionality, i.e. $\forall_{a_i \in f.written}\ a_i.toMerge = max_{a.version}\{a_j \in a_i.versions : a_j.version \geq a_i.version\}$;

   (c) For each aggregate that has a version to merge, merge it. $merge(a_i, a_i.toMerge)$ denotes the merged version;

   (d) Update the functionality version number to the most recent successfully finished functionality version number plus one ($f.version = max(F.success.version) + 1$), and commit the written aggregates using the new functionality version number, $f.version$.

Transactional causal consistency has the lost update anomaly, which occurs when there are several concurrently executing functionalities updating the same aggregates. The merge tries to handle this anomaly by verifying if it is possible to merge concurrent aggregate versions, while preserving the aggregate semantics and the intentions of the functionalities that changed them. The merge occurs in two versions of an aggregate, $merge(a_i, a_j)$, where a_i is the version of the functionality to commit and a_j the version already committed. To do the merge, it is necessary to find the version that is the common ancestor of both versions, in order to identify the differences. Since the version to commit evolved from a committed version, this version is the common ancestor and is denoted by $a_i.prev$. Note that between $a_i.prev$ and a_j can be several other committed versions, if several concurrent functionalities have committed. Nevertheless, it is possible to identify which attributes were changed when compared with the common ancestor, which is denoted, respectively, by $diff(a_i.prev, a_i) \subseteq a.attributes$ and $diff(a_i.prev, a_j) \subseteq a.attributes$, where $a.attributes$ denotes the attributes of $a.aggregate$. Consider the example above, where, instead of two, there are three concurrent functionalities executing with version number 8, f_i, f_j and f_k, and all writing aggregate a. Suppose that f_i finishes first with version number 8, and then f_j with version number 9, when f_k tries to finish will find aggregate a with version 9. The common ancestor will be version 5, and so the differences will be between version 5 and 9 and version 5 and the version $f_k$is trying to write.

The aggregate designer has to define the semantics of consistent merges, which depends on the aggregate semantics. The idea is that the merge should preserve the intention of each one of the functionalities. For instance, suppose a functionality that changes the start and end dates of a tournament, and one user invokes the functionality to change the start date while keeping the end date, and another user, concurrently change the end date while preserving the start date. In this case it does not make sense to merge the two versions, because it would dismiss the intention of each one of the users, they change one of the dates while observing the value of the other one. So, the merge can only occur if the merge does violate the intention of each one of the functionalities. Note that the intention semantics is not an intra-invariant, it has to do with TCC semantics. Additionally, if both functionalities changes all the attributes of an intention, we cannot say that a functionality violates the intention of the other.

Therefore, given an aggregate a, the subsets of attributes that cannot be simultaneously changed in different versions are denoted by $a.intentions \subseteq \mathcal{P}(a.attributes)$. The following condition defines the cases where the merge cannot occur, the lost update anomaly cannot be handled, and the functionality has to abort:

$$\begin{aligned}&\exists_{at_i \in diff(a_i.prev, a_i), at_j \in diff(a_i.prev, a_j), at_i \neq at_j} \exists_{i \in a_i.intentions} :\\ &\{at_i, at_j\} \subseteq i \land\\ &\neg(i \subseteq diff(a_i.prev, a_i) \land i \subseteq diff(a_i.prev, a_j))\end{aligned}$$

The second step of the merge process is to merge any changed attributes, which is done using pre-defined merge methods defined by the developer. Note that, it may not be possible to merge attributes, in which case the merge fails, which is similar to the of non-incremental operations concept defined in the conflict-free data types literature. Therefore, the merge is defined using developer defined $merge$ methods per attribute:

$$\begin{aligned}at \in diff(a_i.prev, a_i) \cap diff(a_i.prev, a_j)&\\ \implies merge(a_i, a_j, at)&\end{aligned}$$

Overall, the process of merging two aggregate versions follows the steps:

1. Verify intention conditions;

2. Merge changed attributes;

3. Run intra-invariants in the merged version.

Listing 3.3: Extend Aggregate with Merge Semantics

```
Tournament Causal Consistency
    Merge
        Intentions Tournament t1, t2:
            (t1.startTime, t2.endTime)
            ...
        Methods Tournament tN, tC:
            startTime
                this.startTime = tN.startTime
    ...
```

The specification in Listing 3.3 illustrates the extension of the *Tournament* aggregate for transactional causal consistency, where the merge method for the start date uses the most recent update (*tN* is the new version whereas *tC* is the committed). If the intention is not violated, overwrite is allowed, the last functionality to commit overwrites the start date.

3.4 Eventual Consistency

The transactional model for aggregates has an additional level of complexity due to the upstream-downstream relation between aggregates. A functionality cannot directly interact with aggregates that are downstream of its main aggregate. So, if the execution of the functionality has some impact on downstream aggregates it cannot occur in the context of the causal consistent transaction executing the functionality. Instead, an event is published and eventually processed by the interested subscribing downstream aggregates. These events are inferred from the downstream inter-invariants.

Listing 3.4: Extend Aggregates with Events

```
CourseExecution Causal Consistency
    Events
        Publish
            ANONYMIZE_STUDENT

Tournament Causal Consistency
    Events
        Subscribe
            ANONYMIZE_STUDENT: PARTICIPANT_EXISTS, CREATOR_EXISTS, ...
```

Therefore, it is necessary to extend the aggregate specification to accomodate this situation. Listing 3.4 shows the transactional causal consistency extension due to the tournament *PARTICIPANT_EXISTS* inter-invariant. The processing of the event triggers the execution of functionality *anonymizeParticipant* in Listing 3.2. As a consequence, the set of events an aggregate version subscribes to is dynamically calculated. In the example, a version of tournament subscribe to the *ANONYMIZE_STUDENT* events emitted by the course execution version inferred from the *Participant is* declaration, where participant *id from s.id*, see Listing 3.1. The set of events an aggregate version subscribes to is dynamically calculated, because it depends on the aggregate version state. In the example, a tournament subscribes the anonymize events that refer to its participants, which can change dynamically through, for instance, the add participant functionality.

Due to the occurrence of these events, it is necessary to reassess the causal consistency associated to the functionality execution. The causal snapshot associated with functionality execution should also guarantee that all the aggregates subscribing to an event are consistent when involved in a causal transaction, i.e. they have processed all the events in common.

As an example, consider three different aggregate versions a_i, a_j and a_k, where the last two subscribe events of the first one. In a causal snapshot two cases can occur: (1) a_j and a_k belong to the snapshot but a_i does not; (2) a_i and a_j belong to the snapshot. Other cases are similar or combinations of these two cases. In the first case, considering the previous condition for causal snapshot consistency, the versions of the aggregates can have different values, smaller than functionality version number, but it may be the case that one of the versions has processed an event from a_i while the other does not. So they are not consistent. Therefore, it is necessary to guarantee that both versions process the same events they are subscribed to. In the second case, it is necessary to guarantee that all the events emitted by a_i have been processed by a_j, if it subscribe them.

Given an event $e \in E$, where E is the set of all events, where $e.type$ denotes the type of event, $e.aggregate$ the aggregate that emits the event, and $e.version$ denotes the version of the aggregate emitting the event. Given an aggregate version a, $a.subsEvs \subseteq E$ denotes the events it subscribes to, and $a.emitEvs \subseteq E$ the set of events it has emitted.

An aggregate event subscription is a condition that uses the event sender aggregate version and the event type to identify the subscribed events. The sender aggregate version indicates the last version of the aggregate sender the aggregate subscriber depends on. Therefore, after an aggregate processes an event it updates the event sender aggregate version. For instance, tournament subscribes the anonymize event of course execution, it contains the course execution version it subscribes to. Note that, it is not necessarily the most recent version of course execution, because the course execution may have evolved without sending events relevant for the tournament. Of course, it can also be the case that the course execution emitted an event the course execution subscribed to but has not processed it yet.

Therefore, the subscriber can process events sent by aggregates that have a higher version number than the version it subscribes to. This is evaluated by the subscription condition. We are also considering that the publish-subscribe channel is causal order.

Therefore, to handle the impact of eventual consistency in the system, two additional conditions have to be added to to the functionality causal snapshot.

For the first condition, if two aggregates subscribe to an event of an upstream aggregate and they are in causal snapshot, they should be consistent according to the processing of the event, even if the upstream aggregate is not part of the causal snapshot.

$$\begin{aligned}
&\forall_{a_j, a_k \in f.snapshot, a_i \in A} \forall_{e \in a_i.emitEvs} \\
&e \in a_j.subsEvs \cap a_k.subsEvs \vee \\
&e \notin a_j.subsEvs \cup a_k.subsEvs
\end{aligned}$$

Note that a_j and a_k can subscribe to different versions of the aggregate $a_i.aggregate$, because the aggregate can evolve in aspects that they do not subscribe. Therefore, the condition guarantees that it only applies for the changes that matter. In the example, the condition forbids a_i to emit an event that is processed by a_k but not for a_j, or, a_i emits the event, a_j does not process it and a_k is created already using a_i, which corresponds to the $\leq$ case.

The second condition specifies that if upstream and downstream aggregates are in the snapshot, all the events emitted by the former have to be processed by the latter.

$$\begin{aligned}
&\forall_{a_i, a_j \in f.snapshot} \forall_{e \in a_i.emitEvs} \\
&e \notin a_j.subsEvs
\end{aligned}$$

In this case it is necessary to guarantee that there is no version of the aggregate, with version number between the version a_j subscribes to and the version a_i added to the causal snapshot, that emits and event subscribed by a_j.

Events are emitted at functionality commit time, to guarantee atomicity of the creation of the new aggregate version and the emission of the event. The event version is equal to the functionality version that did the change in the aggregate, which is the aggregate version as well. Similarly, an event is considered processed when the functionality it triggered commits. An event may be associated with more than one inter-invariant, if it depends on the same change of the upstream aggregate, which corresponds, actually, to the first case above.

4

Simulator

Contents

To experiment the design and behaviour of functionalities business logic execution, using transactional causal consistency, on a microservice system implemented as a set of upstream-downstream aggregates, we developed a simulator that replicates the relevant aspects of aggregate executing under TCC.

The simulator is implemented as a Java Spring Boot[1] web application, where the aggregate functionalities are published as web services, and aggregates are accessed through Spring transactional services. The aggregates are stored in a PostgreSQL[2] database accessed using Hibernate[3]. The emitted events are also stored in the database.

4.1 Architecture

The simulator uses an architecture where each functionality is available as a web-service, the interactions with the main and upstream aggregates is done using transactional invocations and the interactions with downstream aggregates is done writing events in a database that is periodically queried by a scheduler. When the event is detected by the scheduler it is also processed using transactional causal consistency. The *Unit of Work* pattern [17, Chapter 11] is used for implementation of transactional causal consistency.

Figure 4.1 presents a component-and-connector view [18] of the simulator architecture. It has three component types, *Aggregate*, *Event Manager* and *Data Store*, and four connector types. The request/reply connector implements the upstream-downstream relations between aggregate components, where the invoker is the downstream aggregate. The publish-subscribe connector implements the event channels, which are managed by the *Event Manager* component, and have publish and subscribe roles, for instance aggregate *b* is both a publisher and subscriber because it emits events that are consumed by its downstream aggregate component *c* and subcribes events from its upstream aggregate *a*. Aggregate components use data access connectors to manage their aggregates persistence, whereas the event manager component used the data access connector to persist the events. Finally, the aggregate components have a HTTP interface connector that it is used to start its functionalities, the functionalities that have the component aggregate as main aggregate.

Figure 4.2 depicts the interactions associated with the execution of a simple functionality of aggregate *b*, which reads aggregate *c*. The execution, which occurs in the component of the main aggregate, starts by creating a unit of work, that is responsible to manage the causal transaction. The read of aggregates is intercepted by the unit of work, that adds the version to the causal snapshot. The changes done on the *b* aggregate are registered in the unit of work. On commit, the unit of work does the verifications

[1] `https://spring.io/projects/spring-boot`
[2] `https://www.postgresql.org/`
[3] `https://hibernate.org/`

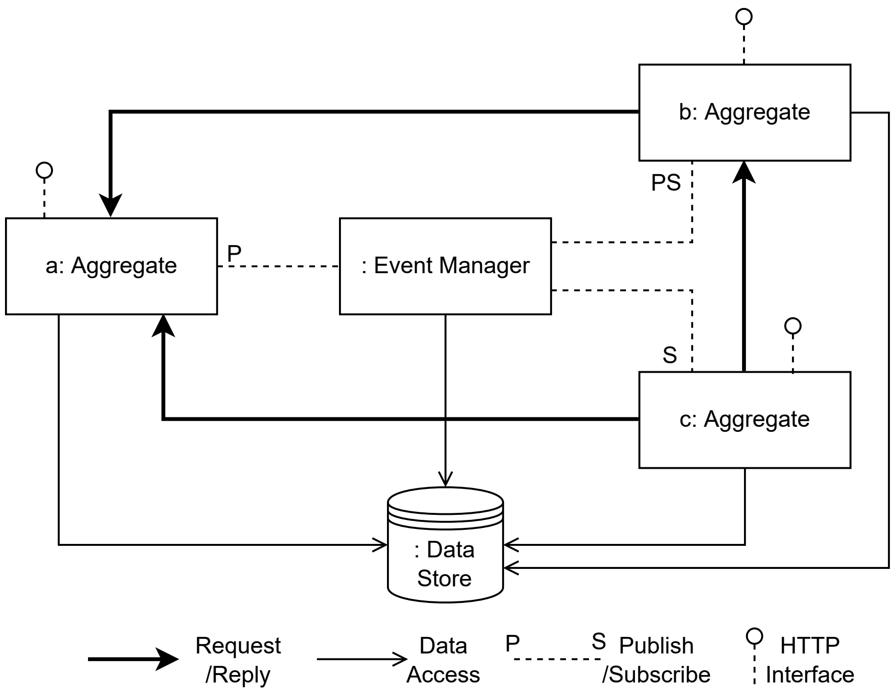

Figure 4.1: Simulator: Component-and-Connector View

and required merges, finishing by writing the new version of the aggregate and the events to emit, and processed, in the data store.

The simulator exercises the causal transactional behavior of the functionality. Each aggregate invocation is done in a transaction, which allows the interleaving with other executing functionalities. The commit is a serializable transaction, which guarantees the atomic write of all versions, where each version is written with the unit of work version number. On the other hand, the events are also written in the commit, which guarantees that their emission is atomic with the aggregates changes.

The processing of events is periodically triggered by the event manager, that checks the events in the data store and start their processing in their own functionality, which executes in a causal transaction. Therefore, during a functionality execution it is possible that one of its aggregates to update is changed in the consequence of the processing of an event, which will be detected during the commit and trigger a merge.

4.2 Domain Model

Figure 4.3 presents the simulator domain model, which is the core of the simulator and it is encapsulated by services. *Version* is a persistent singleton entity that contains the number of most recent committed causal transaction. *Event* contains the id and version of the aggregate that emits the event. Events

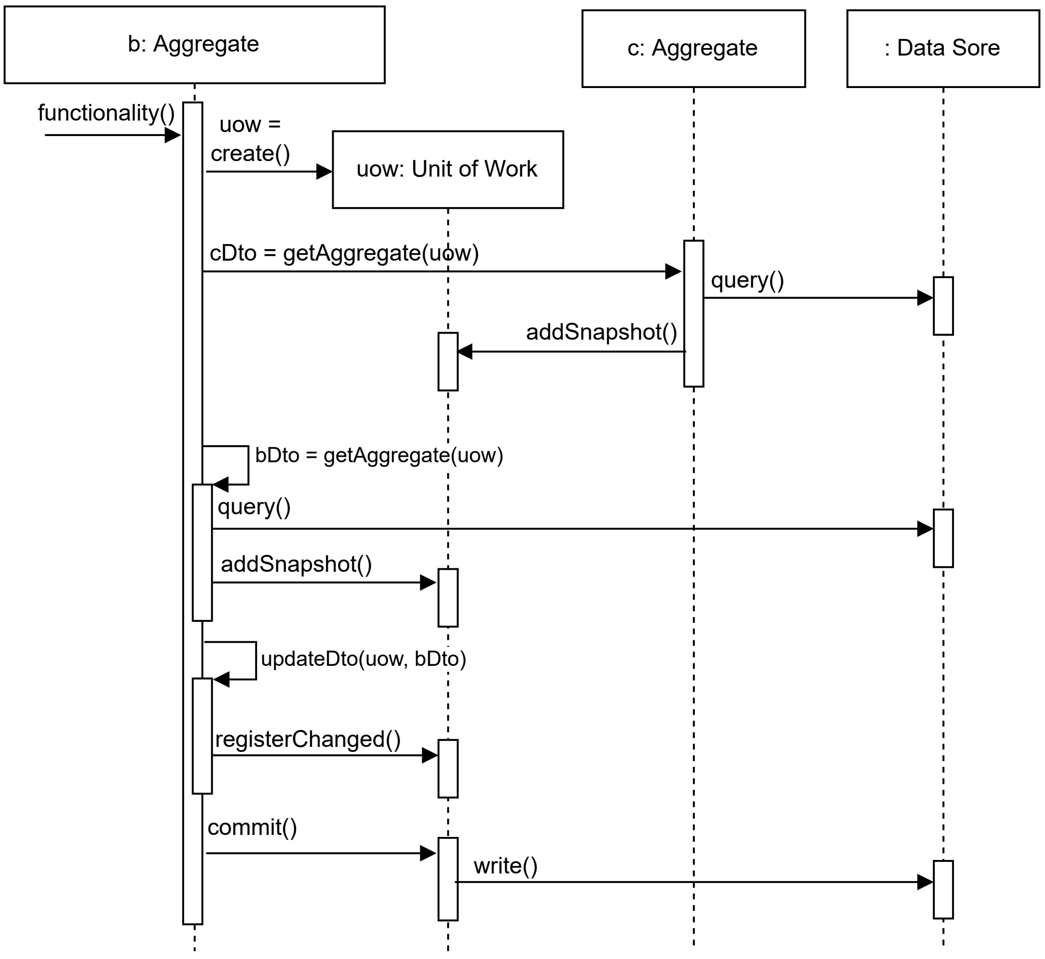

Figure 4.2: Simulator: Component-and-Connector View

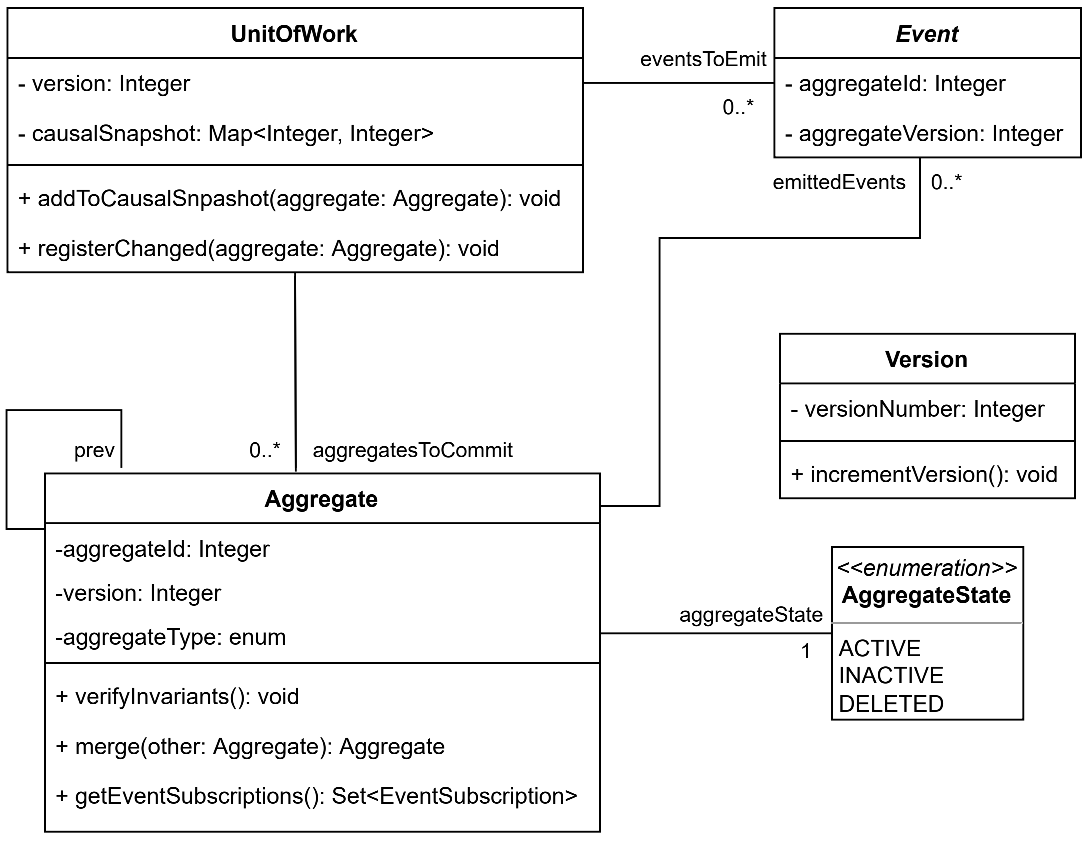
UnitOfWork
- version: Integer
- causalSnapshot: Map<Integer, Integer>
+ addToCausalSnpashot(aggregate: Aggregate): void
+ registerChanged(aggregate: Aggregate): void
eventsToEmit
0..*
Event
- aggregateId: Integer
- aggregateVersion: Integer
emittedEvents
0..*
Version
- versionNumber: Integer
+ incrementVersion(): void
prev
0..*
aggregatesToCommit
Aggregate
-aggregateId: Integer
-version: Integer
-aggregateType: enum
+ verifyInvariants(): void
+ merge(other: Aggregate): Aggregate
+ getEventSubscriptions(): Set<EventSubscription>
aggregateState
1
<<enumeration>>
AggregateState
ACTIVE
INACTIVE
DELETED

Figure 4.3: Simulator: Domain Model View

are stored in a database and are periodically checked by event handlers, which trigger the handling. The *Aggregate* contains its *id*, *version*, and *aggregateType*. Aggregate type refers to the type of aggregate, which is set in subclasses. Aggregates can be in one of three states: *ACTIVE*, *INACTIVE*, and *DELETED*. *ACTIVE* is the normal state, where the aggregate behaves to support the business logic. *INACTIVE* is a state where the aggregate reached an inconsistent state and cannot change anymore, although end users can observe it. Finally, an aggregate is in a *DELETED* state after being removed, which is a relevant information for internal system management, but not visible from outside. Aggregates are implemented as a group of embedded elements such that can be read and written to the database as a whole. Additionally, although all aggregates are implemented in the same database, they are completely independent, not having any referential dependency. Having all aggregates in the same database simplifies the unit of work atomic commit. The aggregate refers to its previous version, that is used by the merge method. On the other hand, each aggregate version has the set of events it emitted. The three aggregate methods in Figure 4.3 are extended in the subclasses with aggregate-specific business logic. The *Unit of Work* is a non persistent entity that contains the version of the causal transaction executing the functionality and the causal snapshot. While being changed in unit of work, attribute *aggegatesToCommit*, the aggregates are detached from the database, which creates a version whose changes are local to the functionality execution. When committing, the *Unit Of Work* creates new aggregate versions with its version number and store them in the database. Additionally, the commit stores the *emittedEvents* in the database, which corresponds to its emission and processing. The *Unit Of Work commit* is implemented in the service, not in the domain entity, because Spring provides transactional behavior at the service level.

4.3 Extension

To simulate a microservice system using transactional causal consistency, the developer has to extend from the implementation abstract classes.

Listing 4.1 shows the implementation of the tournament aggregate excerpt previously specified. Attribute *creator*, line 7, is an embedded object which represents the tournament creator and *participants*, line 10, is an embedded collection, which is loaded with the aggregate. The following elements in the listing are elements which are strictly necessary to be implemented in order for this aggregate to be used in *TCC* functionalities.

Lines 13-69 implement the inter-invariants through event subscriptions. The *getEventSubscriptions* method is an implementation of an abstract method in the *Aggregate* superclass, returns all the event subscriptions, as a set if the aggregate is in *ACTIVE* state. Method *interinvariantCreatorExists*, line 23, implements the inter-invariant *CREATOR_EXISTS* by subscribing to events coming from the course execution aggregate, which allows to establish an equality between the creator and the corresponding execution student fields. The same process is followed for the tournament participants in method *interinvariantParticipantExists*, line 47, which verifies the inter-invariant *PARTICIPANT_EXISTS*. The *getEventSubscriptions* method is used by the *Unit Of Work* to make verifications for the causal snapshot and, in the event detection, to determine which events have been processed by the aggregate and which must still have to be processed.

Lines 72-95 implement the intra-invariant verifications that maintain the consistency within the aggregate. The *invariantStartTimeBeforeEndtime* verifies the *START_BEFORE_END* invariant, by checking that the tournament start time is before the tournament end time. The *invariantCreatorParicipantConsistency* verifies the *CREATOR_PARTICIPANT_CONSISTENCY* by verifying that if the creator and some participant reference the same execution student they must have the same information about it. The *verifyInvariants* is an implementation of an abstract method in the *Aggregate* superclass, that compiles all intra-invariants for the current aggregate. It is used several times during the commit, one as standard verification on the aggregate and the remaining after every merge.

Lines 99-180 implement extensions that allows the generic merge method, in the *Aggregate* superclass to run. The *getFieldsChangedByFunctionalities* method, line 99, provides the merge with the name of the fields that it should look for when computing the changed fields on two concurrent versions. The *getIntentions* method, line 105, provides the intentions of the aggregate, i.e. pairs of fields which cannot be updated independently on two different versions. This specific implementation instructs the merge that the start time and end time fields cannot be updated individually on two different concurrent versions. The *mergeFields* method, line 112, returns a merged version of the current aggregate, *this* with another, *committedVersion*, requested by the merge. The following methods represent the merges of each field. The method mergeStartTime, line 134, merges the start time by returning the lastest updated start time. That corresponds to the start time of the current aggregate, *this*, if it was updated. Otherwise, it corresponds to star time of the *committedVersion*. The *mergeParticipants* method, line 143, is slightly more complex. It combines the initial participants before both updated, the removed and added participants on each update into a set of participants. The method call in line 153 is used to synchronize the participant version in case there are participants with different versions on the aggregate versions.

Listing 4.1: Tournament Aggregate Implementation

```
1 @Entity
2 public class Tournament extends Aggregate {
3     private LocalDateTime startTime;
4     private LocalDateTime endTime;
5 
6     @Embedded
```

```java
    private final TournamentCreator creator;

    @ElementCollection
    private Set<TournamentParticipant> participants;

    @Override
    public Set<EventSubscription> getEventSubscriptions() {
        Set<EventSubscription> eventSubscriptions = new HashSet<>();
        if (this.getState() == ACTIVE) {
            interInvariantCreatorExists(eventSubscriptions);
            interInvariantParticipantExists(eventSubscriptions);
            ...
        }
        return eventSubscriptions;
    }

    private void interInvariantCreatorExists(Set<EventSubscription> eventSubscriptions) {
        eventSubscriptions.add(
            new EventSubscription(
                this.courseExecution.getAggregateId(),
                this.courseExecution.getVersion(),
                UNENROLL_STUDENT, this
            )
        );
        eventSubscriptions.add(
            new EventSubscription(
                this.courseExecution.getAggregateId(),
                this.courseExecution.getVersion(),
                ANONYMIZE_EXECUTION_STUDENT, this
            )
        );
        eventSubscriptions.add(
            new EventSubscription(
                this.courseExecution.getAggregateId(),
                this.courseExecution.getVersion(),
                UPDATE_EXECUTION_STUDENT_NAME, this
            )
        );
    }

    private void interInvariantParticipantExists(Set<EventSubscription> eventSubscriptions) {
        eventSubscriptions.add(
            new EventSubscription(
                this.courseExecution.getAggregateId(),
                this.courseExecution.getVersion(),
                UNENROLL_STUDENT, this
            )
        );
        eventSubscriptions.add(
            new EventSubscription(
                this.courseExecution.getAggregateId(),
                this.courseExecution.getVersion(),
                ANONYMIZE_EXECUTION_STUDENT, this
            )
        );
        eventSubscriptions.add(
            new EventSubscription(
                this.courseExecution.getAggregateId(),
                this.courseExecution.getVersion(),
                UPDATE_EXECUTION_STUDENT_NAME, this
            )
        );
    }

    @Override
    public void verifyInvariants() {
        if (!(invariantAnswerBeforeStart() &&
            invariantCreatorParticipantConsistency && ...)) {

            throw new TutorException(INVARIANT_BREAK, getAggregateId());
        }
    }

    public boolean invariantStartTimeBeforeEndTime() {
        return this.startTime.isBefore(this.endTime);
    }

    private boolean invariantCreatorParticipantConsistency() {
        for(TournamentParticipant participant : this.participants) {
            if(participant.getAggregateId().equals(this.creator.getAggregateId())) {
                if(!participant.getVersion().equals(this.creator.getVersion())
                    || !participant.getName().equals(this.creator.getName())
                    || !participant.getUsername().equals(this.creator.getUsername())) {

                    return false;
                }
            }
        }
        return true;
    }

    @Override
    public Set<String> getFieldsChangedByFunctionalities()  {
        return Set.of("startTime", "endTime",
                      "participants", ...);
    }

    @Override
    public Set<String[]> getIntentions() {
        return Set.of(
            new String[]{"startTime", "endTime"},
        ...);
    }

    @Override
    public Aggregate mergeFields(Set<String> toCommitVersionChangedFields,
        Aggregate committedVersion, Set<String> committedVersionChangedFields ){
        if (!(committedVersion instanceof Tournament)) {
            throw new TutorException(AGGREGATE_MERGE_FAILURE, getAggregateId());
        }

        Tournament committedTournament = (Tournament) committedVersion;
        Tournament mergedTournament = new Tournament(this);

        mergeCreator(committedTournament, mergedTournament);

        mergeStartTime(toCommitVersionChangedFields, committedTournament,
            mergedTournament);
        ...
        mergeParticipants((Tournament) getPrev(), this, committedTournament,
            mergedTournament);

        ...

        return mergedTournament;
    }

    private void mergeStartTime(Set<String> toCommitVersionChangedFields,
        Tournament committedTournament, Tournament mergedTournament) {
        if (toCommitVersionChangedFields.contains("startTime")) {
            mergedTournament.setStartTime(getStartTime());
        } else {
            mergedTournament.setStartTime(committedTournament.getStartTime());
        }
    }
    ...
    private void mergeParticipants(Tournament prev, Tournament v1, Tournament v2,
        Tournament mergedTournament) {

        Set<TournamentParticipant> prevParticipantsPre =
            new HashSet<>(prev.getParticipants());
        Set<TournamentParticipant> v1ParticipantsPre =
            new HashSet<>(v1.getParticipants());
        Set<TournamentParticipant> v2ParticipantsPre =
            new HashSet<>(v2.getParticipants());

        TournamentParticipant.syncParticipantsVersions(prevParticipantsPre,
            v1ParticipantsPre, v2ParticipantsPre);

        Set<TournamentParticipant> prevParticipants =
            new HashSet<>(prevParticipantsPre);
        Set<TournamentParticipant> v1Participants =
            new HashSet<>(v1ParticipantsPre);
        Set<TournamentParticipant> v2Participants =
            new HashSet<>(v2ParticipantsPre);


        Set<TournamentParticipant> addedParticipants =  SetUtils.union(
                SetUtils.difference(v1Participants, prevParticipants),
                SetUtils.difference(v2Participants, prevParticipants)
        );

        Set<TournamentParticipant> removedParticipants = SetUtils.union(
                SetUtils.difference(prevParticipants, v1Participants),
                SetUtils.difference(prevParticipants, v2Participants)
        );

        Set<TournamentParticipant> mergedParticipants = SetUtils.union(
            SetUtils.difference(prevParticipants, removedParticipants), addedParticipants);
        mergedTournament.setParticipants(mergedParticipants);

    }
    ...
}
```

Listing 4.2 illustrates the functionality for updating a tournament. How it creates the unit of work, pass it in the invocations of transactional services, and finishes committing it. The update service uses the unit of work to create the causal snapshot and register the updated aggregates. The method *get-CausalTournamentLocal* does the query to obtain the correct tournament version and add it to causal snapshot.

Listing 4.2: Tournament Functionalities Implementation

```java
public class TournamentFunctionalities {
    public void updateTournament(TournamentDto tournamentDto, ...) {
        UnitOfWork unitOfWork = unitOfWorkService.createUnitOfWork();
        ...
        TournamentDto newTournamentDto =
        tournamentService.updateTournament(tournamentDto, ..., unitOfWork);
        ...
        unitOfWorkService.commit(unitOfWork);
    }
    ...
    public void addParticipant(Integer tournamentAggregateId,
        Integer userAggregateId) {

        UnitOfWork unitOfWork = unitOfWorkService.createUnitOfWork();

        TournamentDto tournamentDto = tournamentService
            .getCausalTournamentRemote(tournamentAggregateId,  unitOfWork);

        UserDto userDto =
            courseExecutionService.getStudentByExecutionIdAndUserId(
                tournamentDto.getCourseExecution().getAggregateId(),
                userAggregateId, unitOfWork);

        TournamentParticipant participant = new TournamentParticipant(userDto);

        tournamentService.addParticipant(tournamentAggregateId, participant,
            userDto.getRole(), unitOfWork);

        unitOfWorkService.commit(unitOfWork);
    }
}

@Service
public class TournamentService {
    @Transactional
    public TournamentDto updateTournament(TournamentDto tournamentDto,
                                            ... , UnitOfWork unitOfWork) {
        Tournament oldTournament =
            getCausalTournamentLocal(tournamentDto.getAggregateId(), unitOfWork);
        Tournament newTournament = new Tournament(oldTournament);

        if (tournamentDto.getStartTime() != null ) {
            newTournament.setStartTime(
                LocalDateTime.parse(tournamentDto.getStartTime()));
            unitOfWork.registerChanged(newTournament);
        }

        if (tournamentDto.getEndTime() != null ) {
            newTournament.setEndTime(
                LocalDateTime.parse(tournamentDto.getEndTime()));
            unitOfWork.registerChanged(newTournament);
        }
        ...
        return new TournamentDto(newTournament);
    }
    ...

    @Transactional
    public void addParticipant(Integer tournamentAggregateId,
        TournamentParticipant tournamentParticipant, String userRole,
        UnitOfWork unitOfWork) {

        if(tournamentParticipant.getName().equals("ANONYMOUS") ||
            tournamentParticipant.getUsername().equals("ANONYMOUS")) {

            throw new TutorException(ErrorMessage.USER_IS_ANONYMOUS,
                tournamentParticipant.getAggregateId());
        }

        Tournament oldTournament =
            getCausalTournamentLocal(tournamentAggregateId, unitOfWork);

        if(LocalDateTime.now().isAfter(oldTournament.getStartTime())) {
            throw new TutorException(CANNOT_ADD_PARTICIPANT,
                tournamentAggregateId);
        }

        Tournament newTournament = new Tournament(oldTournament);
        newTournament.addParticipant(tournamentParticipant);
        unitOfWork.registerChanged(newTournament);
    }
    ...
}
```

Finally, Listing 4.3 illustrates the event detection, where for each subscribed event it triggers the transactional causal execution of the event handler functionality, *processAnonymousStudentEvent*. The detector is launched every second, after the previous detection, and, for each tournament aggregate, detects if any event for the subscribed type, *ANONYMIZE_STUDENT*, was emitted, *eventsToProcess*.

Listing 4.3: Tournament Event Detection

```java
public class TournamentEventDetection {
@Scheduled(fixedDelay = 1000)
    public void detectAnonymizeStudentEvents() {
        Set<Integer> tournamentAggregateIds = getActiveAggregateIds("Tournament");
        for (Integer aggregateId : tournamentAggregateIds) {
            Tournament tournament = tournamentRepository.
                                    findLastTournamentVersion(aggregateId).get();
            Set<EventSubscription> eventSubs =
                tournament.getEventSubscriptionsByEventType(ANONYMIZE_STUDENT);
            for (EventSubscription eventSub : eventSubs) {
                List<Event> eventsToProcess = eventRepository.findAll()
                    .stream()
                    .filter(eventSubscription::subscribesEvent)
                    .sorted(Comparator.comparing(Event::getTs).reversed())
                    .collect(Collectors.toList());
                for (Event eventToProcess : eventsToProcess) {
                    tournamentFunctionalities.processAnonymizeStudentEvent(
                        aggregateId, eventToProcess);
                }
            }
        }
    }
...
}
```

4.4 Implementation

The aggregate *id* is what identifies an aggregate. The system holds several versions of an aggregate, where each version corresponds to a tuple in the database. Therefore, the identification cannot be supported by unique *id* automatically generated by the RDBMS. The identification of an aggregate version is done by the aggregate *id* plus the version *id*. The aggregate *id* is manually generated and attributed to the aggregate when the first version is created. The version *id* is given by the functionality version number. However, we have decided to also use the RDBMS generated *id* as the declared primary because, because it's easier to maintain a single numerical value as a primary key than a composed primary key. The aggregate *id* is generated by a class called *AggregateIdGenerator*. Every time a new aggregate *id* is requested, a service creates new instance of the *AggregateIdGenerator*, persists it with Java Persistance API (JPA) and returns the generated *id* which is unique.

The version number is part of the identification of an aggregate and is used by functionalities to determine which aggregate versions they are allowed to read. Therefore it is very important that aggregates written in different functionalities have different version numbers. The version number is global across all aggregate types and it is maintained by a counter. The counter is managed by two services: one that just returns the current counter value and other which increments the counter value and returns the incremented value. The former service is used when the functionality starts, whereas the later is used when the service commits. Both service run in *SERIALIZABLE* isolation so that two functionalities cannot commit using the same version number.

In the simulator, an aggregate is stored in the database as a set of tuples in a set of tables. Each aggregate type uses the same set of tables. For instance, even though every tournament aggregate shares the *tournament_creator* table, each aggregate references a different tuple in the table and no two aggregates reference the same tuple, even if the creator of two aggregates represents the same student. Therefore, there are no foreign keys among tuples belonging to different aggregates, there are not constraints between them at database level. Aggregates can only interact with each other through API invocations done in the context of their business logic, to comply with the concept of aggregate.

There are two aspects which allow the simulation of TCC semantics: version queries and causal snapshot building. For each aggregate type there is a query that retrieves the most recent version of an aggregate, given its aggregate and a version number provided by the functionality which calls it. This is important because it prevents functionalities from reading writes performed by other concurrent functionalities. Listing 4.4 shows the query used to retrieve the most recent version of tournament *:aggregateId*, whose version is inferior to *:maxVersion*. For the remaining aggregate types, the query is the same except it accesses the aggregate type's respective table. Note that deleted aggregates are not retrieved.

Listing 4.4: Find Causal Tournament

```sql
select *
from tournaments t
where
    t.aggregate_id = :aggregateId AND
    t.version < :maxVersion AND
    t.state != 'DELETED' AND
    t.version >= (
        select max(version)
```

```
            from tournaments
            where aggregate_id = :aggregateId AND
            version < :maxVersion
        )
```

However, not all queries respect the causal snapshot, because they are part of management mechanisms instead of being a part of standard functionality execution. There are two cases where this occurs: in event detection and search of concurrent versions during the commit. During event detection, when it is necessary to know which events need to be processed for an aggregate, a query is issued to get the most recent version of aggregate *:aggregateId*. An example of this query is showed in listing 4.5.

Listing 4.5: Find Most Recent Tournament Version

```
select *
from tournaments t
where
    t.aggregate_id = :aggregateId AND
    state = 'ACTIVE' AND
    t.version >= (
        select max(version)
        from tournaments
    )
```

The reason to select the last version of the aggregate is due to the fact that event detection occurs outside the context of a functionality. It makes sense to choose the most recent version, because it is most likely the version that is going to be read in the event handling functionality, which means events are more likely to be successfully processed. Note that events not processed by previous versions can be processed, if not yet, in the most recent version.

During commit we query for concurrent versions, if any, because it is not possible to use causal snapshot queries. We want to obtain the most recent committed version of the aggregate we want to commit. It is enough to get the most recent, because it integrates the changes done by all previous ones. The implementation, actually, uses the *prev* aggregate attribute. Listing 4.6 shows the query that searches all aggregate versions, of aggregate *:aggregateId*, higher *:prevVersion*, and selects the one that have the highest version number.

Listing 4.6: Find Concurrent Versions

```
select *
from tournaments
where id =
    (
        select max(id)
        from tournaments
        where
            aggregate_id = :aggregateId AND
            version > :prevVersion
    )
```

Each microservice implements the detection of each one of the event types it subscribes. When the simulator starts, the periodic event detection occurs simultaneously for all events, but, as shown in line 2 of listing 4.3, after the first processing the periodic detection will be desynchronized for each event. This occurs because the next detection for each particular detection will happen 1 second (1000 ms) after the previous one finishes.

During event detection it can occur that a version that subscribes the event is updated before event handling. Suppose that in line 17 of listing 4.3, before the event handling starts, another functionality creates a new version of the tournament where the participant, which the event is going to anonymize, is removed. In this case the event handling is going to find a new version of the tournament which, actually, does not subscribe the event anymore. This is not a problem because a redundant verification is done inside the event handling as can be seen in lines 9 and 22 of the event handling in listing 4.7.

Listing 4.7: Tournament Event Handling

```
1  @Transactional(isolation = Isolation.READ_COMMITTED)
2  public Tournament anonymizeUser(Integer tournamentAggregateId, Integer executionAggregateId,
3      Integer userAggregateId, String name, String username, Integer eventVersion,
4      UnitOfWork unitOfWork) {
5
6      Tournament oldTournament = getCausalTournamentLocal(tournamentAggregateId, unitOfWork);
7      Tournament newTournament = new Tournament(oldTournament);
8
9      if (!newTournament.getCourseExecution().getAggregateId().equals(executionAggregateId)) {
10         return null;
11     }
12
13     if (newTournament.getCreator().getAggregateId().equals(userAggregateId)) {
14         newTournament.getCreator().setName(name);
15         newTournament.getCreator().setUsername(username);
16         newTournament.getCourseExecution().setVersion(eventVersion);
17         newTournament.setState(INACTIVE);
18         unitOfWork.registerChanged(newTournament);
19     }
20
```

```
    for (TournamentParticipant tp : newTournament.getParticipants()) {
        if (tp.getAggregateId().equals(userAggregateId)) {
            tp.setName(name);
            tp.setUsername(username);
            newTournament.getCourseExecution().setVersion(eventVersion);
            unitOfWork.registerChanged(newTournament);
        }
    }

    return newTournament;
}
```

We have decided to run the commit code with *SERIALIZABLE* isolation, as shown in listing 4.8, so no other aggregate versions are committed concurrently, which could influence this commit. This ensures that the set of aggregate versions that can be concurrent with one that is trying to be committed remains the same throughout the entire commit execution. It prevents a given commit from starving and being stuck always finding new concurrent versions, and as soon as the concurrent aggregate event detection ends the commit can safely persist the aggregate versions without risk.

Listing 4.8: Commit Serialization

```
 @Retryable(
    value = { SQLException.class },
    backoff = @Backoff(delay = 5000))
@Transactional(isolation = Isolation.SERIALIZABLE)
public void commit(UnitOfWork unitOfWork) {
    ...
}
```

5 Evaluation

Contents

5.1 Case Study

QuizzesTutor[1] is a large monolith application for teachers to prepare different types of questions and propose quizzes to students. Students can answers quizzes, generate quizzes for self-assessment, selecting the questions topics, and organized quizzes tournaments, among other features. It is a business logic rich system implemented as a web application.

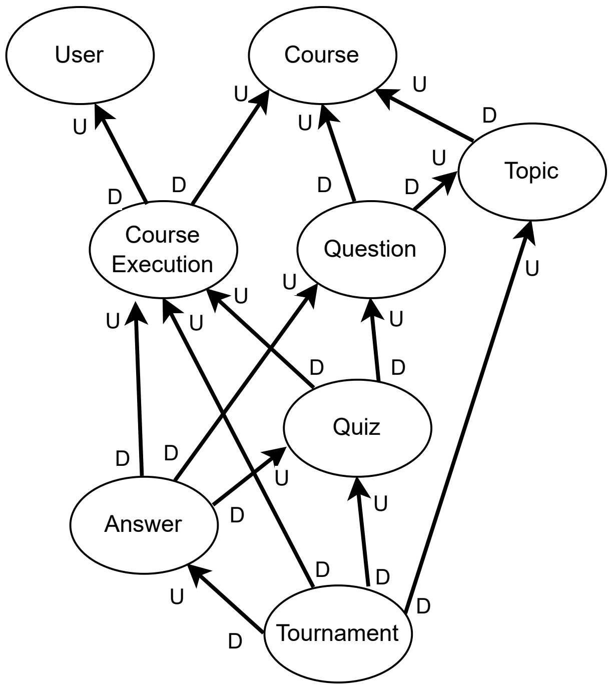

Figure 5.1: Quizzes Tutor Inter-Aggregate Context Map

To experiment the design of business logic using transactional causal consistent aggregates, a set of aggregates and their upstream-downstream relations where identified in a subset of the Quizzes Tutor system, see Figure 5.1. We now describe each aggregate in the context of the domain logic.

A *User* represents a person using the system and can be of one of the following 3 types: *STUDENT*, *TEACHER* and *ADMIN*. It it composed of a name and a username. An *STUDENT* can be enrolled in several course executions which allows him to answer quizzes and participate in tournaments. A *TEACHER* is responsible for managing the quizzes and questions. An *ADMIN* manages the creation and deletion of courses and course executions. Additionally, if our implementation of this domain model had authentication, the user can also contain other information, such as email or password.

A *Course* is a course in a teaching institution. It is composed of a name and a type. Type can be either *TECNICO* or *EXTERNAL*. The reason of this distinction is that the application was designed at first for courses only and was later extended to accommodate courses from other institutions. The

[1] https://quizzes-tutor.tecnico.ulisboa.pt/

Course functions as a blueprint for course executions.

A *Course Execution* is an instance of a course in a specific year. It's composed by an acronym an academic term and an end date. It contains a list of execution students, which represents the enrolled users in the current course execution and has a reference to a course.

A *Topic* is an information topic within a course. For instance, in the context of the Software Engineering course, we can have topics such as "*Agile*" or "*Test Driven Development*". A topic is composed of a name and has a reference to a course.

A *Question* represents a question about certain topics. It contains a title a content and a list of options. We implemented only multiple choice questions although there are more types of questions. A question also contains a list of topics which are related to the question. The question exists within a course, meaning all course executions of that course can use the question for their quizzes.

A *Quiz* is set of questions that can be answered within a certain time frame by students of the course execution the quiz belongs to. It's a common evaluation method used by teachers of course executions. It contains a title and creation, available, conclusion and results dates. It also contains the course execution it belongs to, the questions it has and it's type, which in the current implementation can *IN_CLASS*, when it created directly or *GENERATED* when it is generated with random questions.

An *Answer* is the answer of student to a quiz. It contains information about whether it has been answered and when it was answered. It references also the quiz this answers refers to, the student which answered it and the course execution of the quiz. Additionally it contains a reference to the questions that compose the quiz and the specific answers the user gave to that question.

A *Tournament* is a quiz created by students for students to answer in the context of a competition. It exists within a certain course execution, has a set of topics, a student as a creator and set of students as participants. Contains a set of dates, start time and end time and the number of questions. It also references a quiz, which is generated automatically with questions associated to the tournament topics and the tournament number of questions. Additionally it references the answer of each participant to the quiz.

Table 5.1 shows the implemented aggregates, the number of upstream-downstream relations between them, and the number of implemented intra and inter invariants.

The upstream-downstream relations show the dependencies between aggregates. In particular, a downstream aggregate can have the identification of the root elements of its upstream aggregates. For instance, the *Course Execution* aggregate contains the identification of the *Course* root element, and the the identification of *User* that is a student of the course execution.

Overall, the upstream-downstream relations reduce the dependencies in the system, increasing its modularity. For instance, the software team developing the *Course* aggregate do not need to be aware of its downstream aggregates. On the other hand, the upstream-downstream relations define a layered

Table 5.1: Aggregate invariants and references count

Aggregates	**Intra-Invariants**	**Inter-Invariants**	**Upstream References**
Course	2	0	0
User	2	0	0
Topic	0	1	1
Question	0	3	2
CourseExecution	4	2	2
Quiz	7	3	2
Answer	5	8	3
Tournament	13	19	4

architectural style, where the upstream aggregates are in lower layers in relation to their downstream aggregates.

Therefore, the *Tournament* aggregate is in the top layer, since it is built on the services provided by other aggregates. As usual of the layered architectural style, the communication from the lower layers to their upper layers should be done using callbacks, which in our case are events.

5.2 Simulator Correctness

To verify the correctness of the solution and the simulator correct implementation, we designed scenarios that exercise the interleaved execution of functionalities and event handling. For each of the scenarios, JMeter[2] tests were also written.

In the scenarios, R, W, P and E, denote, respectively, the read of a version, write in a version, write a new version (persistent), and emit an event associated with a new version. The functionality and event processing versions correspond to their initial version number. Commits and merges are operations that can span several aggregates and are represented with a vertical line to represent their atomicity.

JMeter provides a set of constructs that are used to implement the tests:

- *Thread Group*: a construct that aggregates a set of web requests. Within it, the web requests are executed sequentially in a specified order. The thread groups of a test can either execute sequentially in a specified order or concurrently;
- *Setup Thread Group*: a special thread group which executes before all other thread groups in the test.
- *Assertion*: construct used to verify the information in request response, be it in the response header or response body;
- *Constant Timer*: stops the current thread execution for a specified amount of time in milliseconds;

[2] https://jmeter.apache.org/

- *Database Access API*: allows to do queries and updates on database tables, besides the web requests.

Using these constructs the simulator tests follow a similar structure:

- *Setup*: a setup thread group which sets the system to a desired state before performing the actual test. It has the following steps:
  - Creation of a course and a course execution associated to the course;
  - Creation of a user, respective activation and enrollment in the course execution. This step can be repeated twice if more than one student is needed;
  - Creation of topic associated with the course. This step can be repeated several times;
  - Creation of a question associated with a topic. There are as many questions as topics, and each one is associated to a different topic. This allows to more easily perform verifications on quizzes and tournaments;
  - Creation of a tournament associated to the course execution, with a creator and participant as execution students, and associated to the tournament topics.
- *Scenario*: the core part of the test. This part is different for every test because each one tests a specific behaviour;
- *Assertions*: validates a certain criteria. This can be either an assertion on a value of an entity by performing web request to the entity and analysing the JSON response or an assertion of a response code. Most of the times the test expects the request to succeed which corresponds to the JMeter default behaviour. However, there are special cases in which the test expects a certain request to fail and an additional assertion must be added. Assertions can be used in the middle of the scenario or at the end of the test or both, depending on the test.

JMeter allows for the execution of thread groups concurrently. However it is not possible to control how they interleave. In order to have better control over concurrent execution of functionalities, we have decided to run all threads groups sequentially and simulate concurrency by manipulating the version number of the system. A functionality which writes an aggregate finishes and commits with version number n. When another functionality starts, it has version number $n+1$, which means if it reads the aggregate version created by first functionality. This is a standard sequential execution. If we want to simulate a concurrent execution of these two functionalities, we can decrement the system version number by 1, between their executions. This results in the second functionality also starting with a version number of n like the first, instead of $n+1$. This ensures that the second functionality reads the same aggregate version read by the first functionality. If the second functionality also performs a write on the same aggregate, upon committing it detects the version written by the first functionality as concurrent, initiating a merge operation between the two. We can decide which functionality initiates the merge by running it after the version decrement. After the concurrent functionalities are executed, the system version number is incremented by the previously decremented amount to guarantee that further actions observe the state set my the first functionality. The system version number increment and decrement is done by a SQL update on the respective table, through the JMeter API.

Another technique was used to handle event detection. Event detection and processing runs periodically on a predetermined interval. Sometimes it is useful to trigger the event detection at a precise moment in the test, for instance when trying to test the concurrent execution with a functionality. To achieve it, we created web services that trigger the detection and processing of a specific event type for an aggregate. On the other hand, to also test the periodic event detection, if the test behavior allows it, we add a constant timer that stops the test for a few milliseconds, waiting for the event processing to occur.

There are situations where the periodic event detection, which occurs every second, can compromise the expected test behavior. Therefore, a request is implemented, to be used by the JMeter tests, that disable and enable the period event detection.

Figures from 5.2 to 5.11 show the different interleaving associated with the execution of update student and add participant functionalities, where the former emits an event that can be subscribed by the latter.

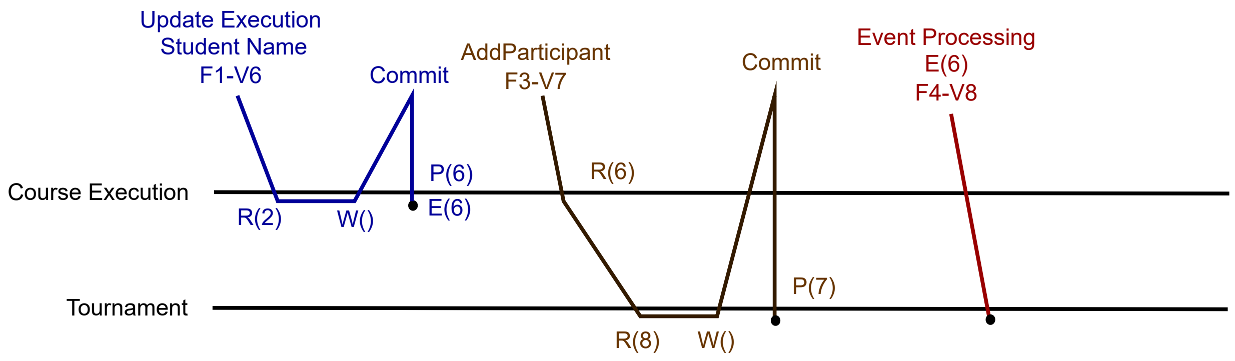

Figure 5.2: Sequential: Update, Add, Event

In Figure 5.2 add participant succeeds when it adds the tournament to the causal snapshot, because the updated student in the course execution is not, yet, a tournament participant. Then, afterwards, the event is not detected by the tournament aggregate, because it already contains the updated participant.

A JMeter test[3] simulates the described behaviour. Two sequential group threads execute, respectively, update execution student name and add participant functionality. Before functionalities execution,

[3] 5a-updateStudentName-addParticipant-processUpdateNameEvent.jmx

the periodic event detection is disabled to avoid events to be processed before add participant finishes. After the add participant finishes the event periodic event detection is enabled. Then, a after a few miliseconds wait, the assertions are executed to verify that the tournament has the participant with the updated name.

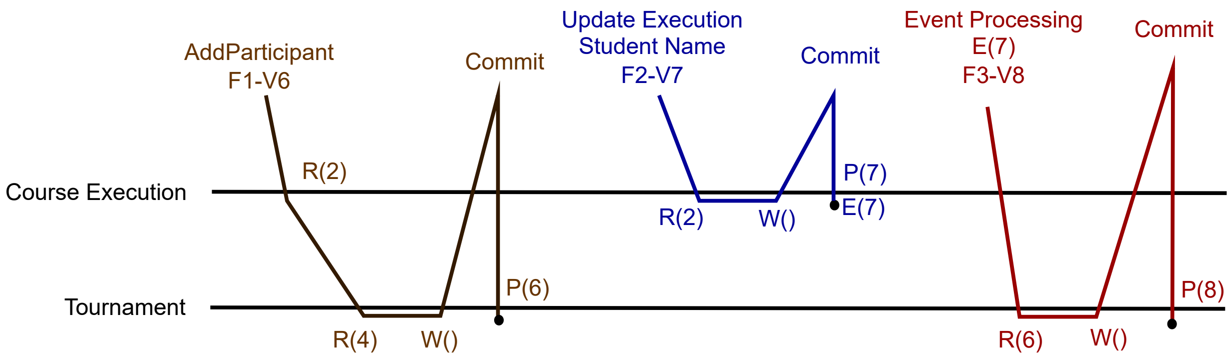

Figure 5.3: Sequential: Add, Update, Event

In figure 5.3 the add participant commits first and the event processing after both functionalities commit. The event processing changes the participant with the new information, because the add participant committed first.

A JMeter test[4] simulates the described behaviour. Two sequential thread groups execute the functionalities. Then, the test waits for the event to be processed and does a request to verify that the tournament participant has been updated.

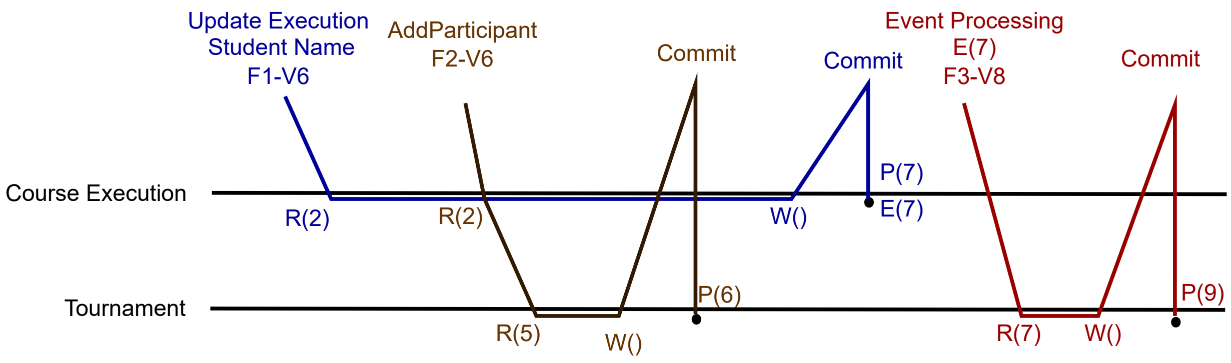

Figure 5.4: Concurrent: Update(1), Add, Update(2), Event

Figure 5.4 illustrates the case when both functionalities execute concurrently and add participant finishes first. In this case the event processing updates the participant with the new information.

A JMeter test[5] simulates the described behaviour. A thread group executes the add participant. Then

[4] 5b-addParticipant-updateStudentName-processUpdateNameEvent.jmx
[5] 5c-updateStudentName1-addParticipant-updateStudentName2-processUpdateNameEvent.jmx

it decrements the system version number by 1 and starts update student. This represents a concurrent execution of both functionalities in which the add participant finishes first. Finally, the test waits for the periodic event processing, before performing a request to the tournament to assert that it contains the updated name on the participant.

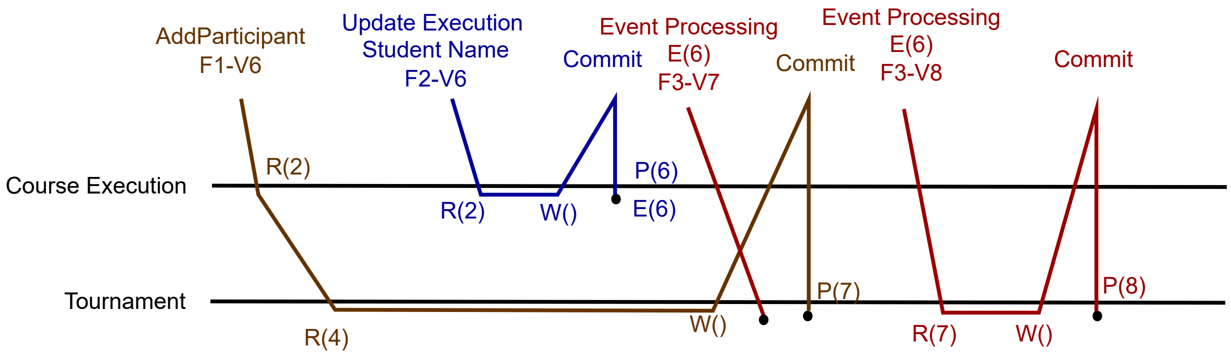

Figure 5.5: Concurrent: Add(1), Update, Event(1), Add(2), Event(2)

Figure 5.5 illustrates the other concurrent case, but the update student finishes first. In the scenario the event detection occurs twice. First, before add participant commits, where there is no version subscribing it, and a second time after add participant commits, where the event is processed and updates the participant information.

A JMeter test[6] simulates the described behaviour. The test executes the update execution student functionality first, followed by a triggering of the event processing in the tournament. The event processing doesn't perform any changes because when the event detection the tournament is not yet subscribed to the event. Then a get request is made on the tournament and it is asserted that no change has been made to its participants, i.e. it is still empty. Then the periodic event detection is disabled, the system version number is decremented by 1 and the add participant functionality is executed. This simulates a concurrent execution between the update execution student and the event processing with the add participant. The decrement by 1 is sufficient because the event processing doesn't write any aggregate. Finally the test enables the periodic event detection, and waits for the event handling, after which makes a get request to the tournament and asserts that it contains the updated name on the participant.

Figure 5.6 represents the concurrent execution of two complex functionalities, the update of tournament. The first commits without any problems. When the second tries to commit it encounters a concurrent tournament and a concurrent quiz. Both have to be merged. In this case, the merge methods override the previous versions with the last versions, such that topics and quiz are consistent.

A JMeter test[7] simulates the described behaviour. The test executes an update to the tournament

[6] 5d-addParticipant1-updateStudentName-processUpdateNameEvent1-addParticipant2-processUpdateNameEvent2.jmx

[7] 8-5-update-tournament-concurrent-intention-pass.jmx

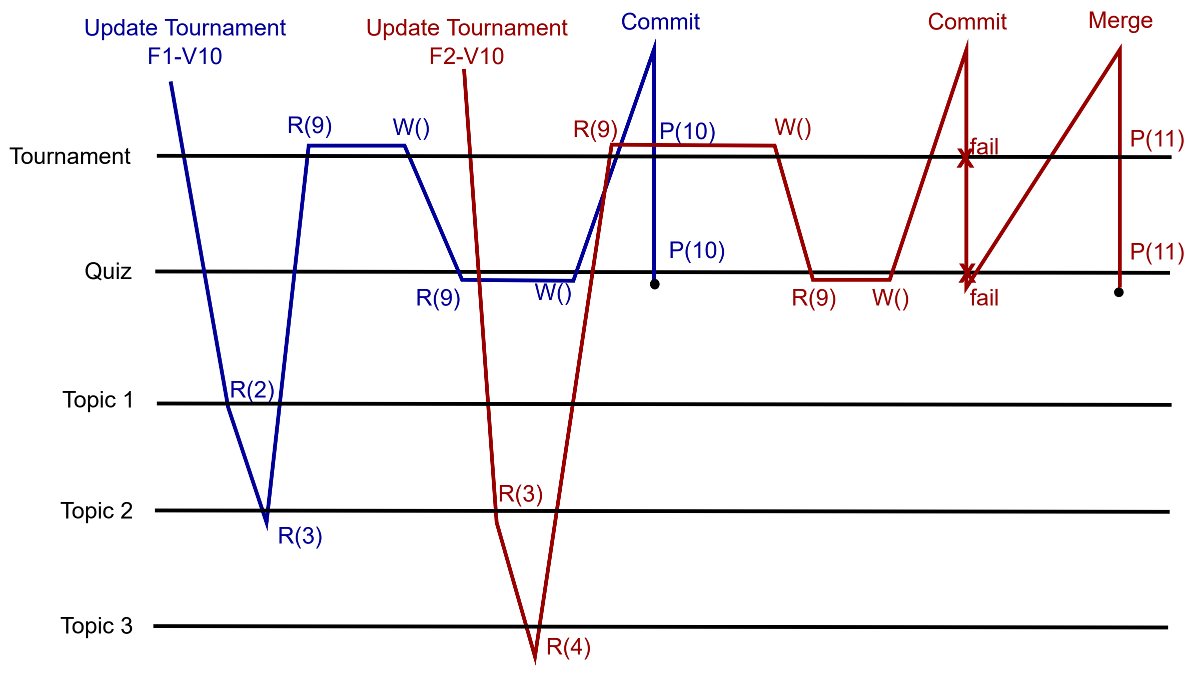

Figure 5.6: Concurrent complex functionalities

which changes the topics, decreases the system version number and executes another update to the tournament, that does a different change of the topics. This simulates a concurrent execution of two tournament update functionalities. The second update, when trying to commit detects both written aggregates, the tournament version and quiz version, updated by the other functionality, and performs the merge. Finally the test does a get request to the tournament, retrieving the merged version, and it is asserted that it has the correct set of topics. A get request is also done to the quiz, retrieving the merged version, and it is asserted that it contains the correct set of questions. This last assertion is possible due to the one-to-one question topic relation, we defined in the setup.

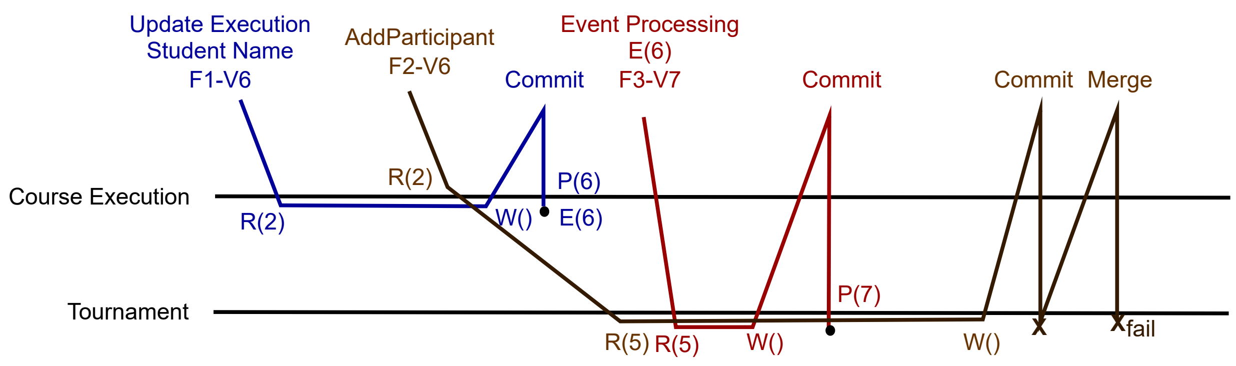

Figure 5.7: Concurrent: Update(1), Add(1), Update(2), Event, Add(2)

Figure 5.7 represents the concurrent execution of add participant and update student, where the event is processed before add participant commits. An added complexity is that the participant being added is the tournament creator, which means that the event is actually processed in the tournament before the participant is added. When the add participant tries to commit it finds a concurrent version of the tournament. It then tries to merge but it is not able to and the functionality aborts. The abort occurs because in the version 7 of the tournament the creator has the name updated, but in the added participant has the old name. Upon the intra-invariants verification, done at the end of the merge, the *CREATOR_PARTICIPANT_CONSISTENCY* intra-invariant does not hold and the functionality aborts.

A JMeter test[8] simulates the described behaviour. The test executes the update student functionality, and then waits for the event detection and its processing on the tournament. Afterwards, the version counter is decremented by 2, and the add participant execution started. The version number decrement is 2, because besides simulating concurrency between the add participant and the event processing it is also necessary to simulate it between the add participant and the update student. If the version number had been decremented only by 1, the add participant would read the updated version of the course execution and not the older one, which would eliminate the need for event processing. When the add participant functionality tries to commit it will find the tournament version written by the event processing as concurrent, triggering the merge process which will lead to the invariant break. Therefore, we assert that the response code of the add participant functionality is not 200, which means it was not successful. Then the version number is incremented by 1, because of the add participant fails and does not increment the version number after commit. If the test does not increment the version number the next read won't read the most recent version of the tournament. Finally, is done a get request to the tournament to assert that it does not contain the participant added.

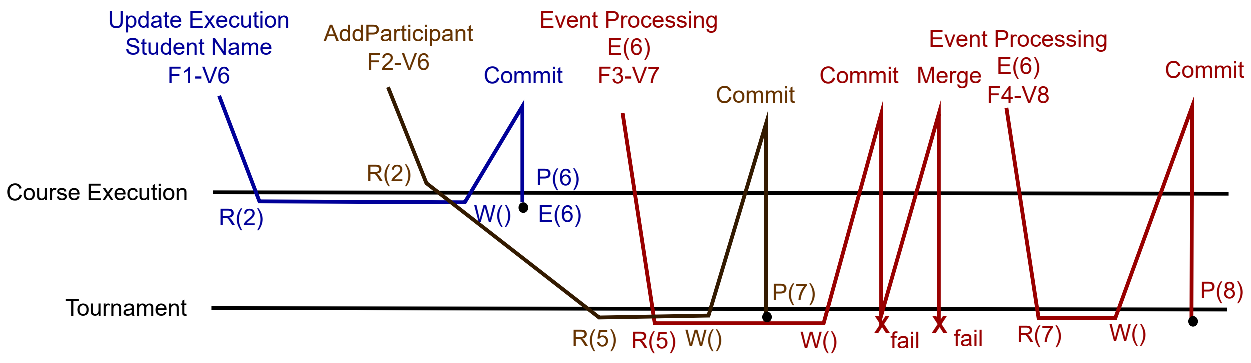

Figure 5.8: Concurrent: Update(1), Add(1), Update(2), Event(1), Add(2), Event(2), Event(3)

Figure 5.8 represents a similar example to the one in figure 5.7 in which the event processing finishes after the add participant functionality. Here, there is also an abort because of the *CRE-*

[8] 8-6-add-participant-concurrent-update-execution-student-name-processing-ends-first.jmx

ATOR_PARTICIPANT_CONSISTENCY intra-invariant: the event processing when it started was only able to read a version of the tournament which only contained the creator and therefore it can only update the creator name. When the merge is attempted it is verified that the participant, for the same execution student as the creator, has the old name which causes the invariant to break. The event can still be processed later, after the participant is added, updating both the participant and creator together.

A JMeter test[9] simulates the behaviour in figure 5.8. This test starts by disabling the periodic event detection and then update student executes. Next, the system version number is decremented by one and add participant executed. Then, the system version number is decremented by one, and the event detection triggered, to simulate the concurrent event handling with the add participant, where the add participant finishes first. Finally, the system version number is incremented by one and the event detection triggered again. Several required verifications are done, like that the first event processing returns a code different from 200.

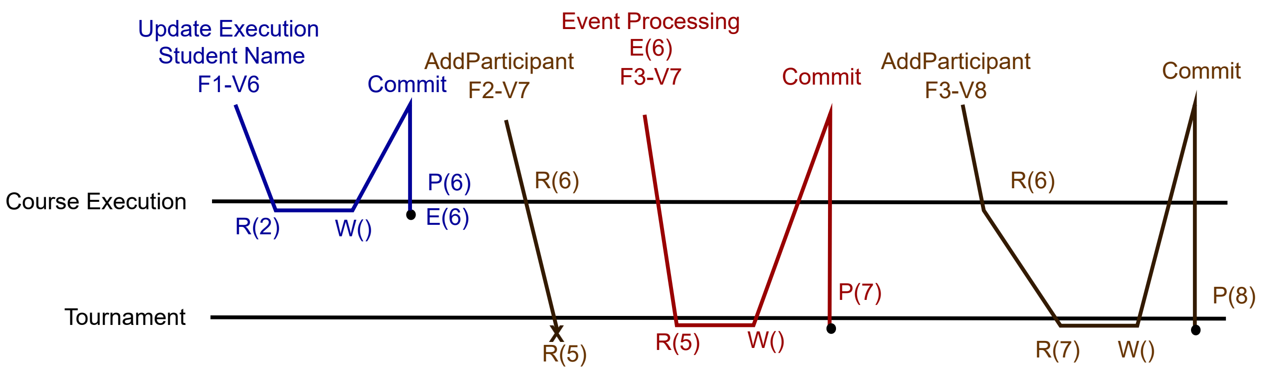

Figure 5.9: Concurrent: Update, Add, Event, Add

Figure 5.9 represents a scenario where the creator is being added as participant. The add participant fails because the tournament aggregate did not processed the update student event, and it cannot have the new version of course execution and the current version of tournament in the same causal snapshot. Afterwards, the event is processed and when the add participant execution is tried a second time, it finally succeeds.

A JMeter test[10] simulates this behaviour. The test implementation follows the same rules. In this case, it is necessary to disable the periodic event detection, and execute the functionalities and event processing in the scenario order.

Figure 5.10 represents a situation where a student is anonymized in a course execution, which is the tournament creator, whereas, concurrently, the creator is added as a participant. Additionally, the anonymize event is processed before the add participant finishes. Therefore, the event processing

[9] 8-7-add-participant-concurrent-anonymize-event-processing-processing-ends-last.jmx
[10] 8-8-update-execution-student-add-participant-process-event-add-participant.jmx

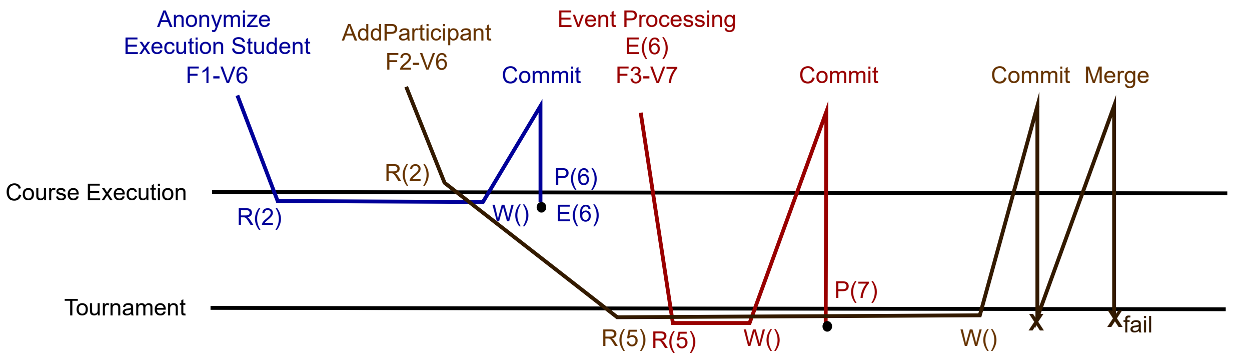

Figure 5.10: Concurrent: Event

makes the tournament INACTIVE, because its creator has been anonymized. Meanwhile, when the addition of the participant tries to commit, it requires a merge and it fails because it is done with an INACTIVE version of the tournament.

A JMeter test[11] simulates the described behaviour. The anonymize occurs first, then the event processing. The add participant is executed after the decrement of the system version number by 2. After the add participant fails, the system version number is incremented by 2, and get request to the tournament is issued, to verify that the tournament is INACTIVE.

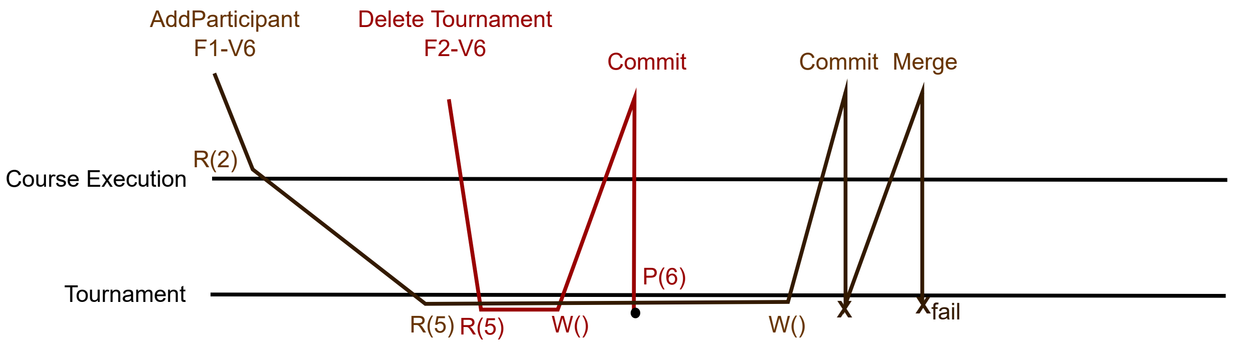

Figure 5.11: Concurrent: Event

Figure 5.11 represents a scenario where a tournament is deleted while a participant is being added. The participant addition will fail when it tries to commit because it tries to perform merge with a *DELETED* aggregate.

A JMeter test[12] simulates the described behaviour. The same techniques are applied. The add participant is executed after delete tournament, and with a decrement of the system version number by one. At the end, the system version number is incremented by one, and the test asserts that the add

[11] 8-9-add-participant-concurrent-anonymize-event-processing-processing-ends-first.jmx
[12] 8-10-concurrent-delete-tournament-add-participant.jmx

Table 5.2: Quizzes Tutor Experiment

Aggregates	**Entities**	**Queries**	**Simple**	**Complex**	**Intentions**	**Merge Methods**
Course	1	0	0	0	0	0
User	1	3	3	0	0	0
Topic	2	1	3	0	0	1
Question	3	2	4	0	3	4
CourseExecution	3	2	4	1	0	1
Quiz	3	2	2	0	3	5
Answer	4	1	3	0	0	4
Tournament	5	3	5	3	4	9

participant functionality fails, by verifying that the response code is not 200 because during the merge it finds the deleted tournament as a concurrent version and tries a merge with it. A get request to the tournament is also made, whose response code should be different than 200 because the tournament is no longer accessible.

5.3 Experiment Analysis

In the experiment we implemented the business logic of 42 functionalities: 14 queries; 24 simple functionalities; and 4 complex functionalities. Table 5.2 details the number of involved elements. The experiment allowed us to identify common rules for the business logic implementation and some particular cases, which provide useful insights.

Overall, the implementation of the functionalities business logic is quiet uniform over all types of functionalities because of the use of transactional causal consistency. In what concerns simple and complex functionalities, it is necessary to be more careful, in order to handle the lost update anomaly, which requires the definition of intentions and operation merge methods. Anyway, and as it will be discussed in the complexity analysis subsection, the definition of the consistency can be done at the aggregate granularity level.

However, three aggregate states, not explicitly associated to the semantics aggregates would have in a ACID transactional system, have to be added: *ACTIVE*, *INACTIVE*, *DELETED*. The normal state of an aggregate is ACTIVE. Aggregates can be *INACTIVE* due to the upstream-downstream relationship between aggregates. This occurs when the execution of the functionality in the upstream aggregate emits an event that informs the downstream aggregate of a fact that makes the aggregate inconsistent. For instance, anonymizing the tournament creator in the course execution makes the tournament *INACTIVE*, but does not delete it. Note that, in an implementation using *ACID* transactional behavior, it would be possible to write the business logic such that if the student is the creator of a tournament, they cannot be anonymized. This is not possible in this case, because the course execution business logic is not aware of tournaments, and so an event is emitted and has to be handled. *INACTIVE* aggregates can be part of the causal snapshot and used in queries to show users the impact on business logic resulting that only intra-invariants have *ACID* transactional behavior. For instance, students can see that a tournament they have enrolled in is *INACTIVE*. However, a functionality should abort if it tries to write in an *INACTIVE* aggregate, *INACTIVE* is a final version. On the other hand, aggregates can be *DELETED*, a delete request is issue during a functionality execution. Functionalities cannot access *DELETED* aggregates, so it would make sense to just remove the aggregate entries from the database. However, a concurrent execution of a functionality that deletes an aggregate with another that updates one of its field can result in anomalous behaviour. If the update finishes later, when it tries to find concurrent version to merge, it will not find any, and will simply commit with no restrictions. With the *DELETED* state, in this situation the update would be able to find a concurrent aggregate version and verify that it is in *DELETED* state and abort its commit, thus preserving the consistency of the system. Overall, the *INACTIVE* state is used to reflect a violation of an inter-invariant, and may be visible to the end user, while the *DELETE* state is an internal state to manage versions, and it is not visible to the end user. It is up to the business logic designer to decide when to apply each one of the states.

Although most of the implementations of inter-invariants result in business logic where the downstream aggregate has to handle the upstream aggregate events, this is not always the case. We have identified a situation where the downstream aggregates *owns* the upstream aggregate. The upstream aggregate is mostly updated by downstream aggregate functionalities. Therefore, the change in the upstream aggregate does not need to be propagated to the downstream aggregate. It is up to the downstream functionality to guarantee the consistency, and this occurs in the context of transactional causal consistency functionality. In particular, the tournament creates a quiz, and, even though, the quiz is a upstream aggregate it is mostly changed by tournament functionalities. Therefore, the tournament does not have to subscribe to the quiz events, which simplifies the business logic. The only case where the tournament have to subscribe to its quiz events, occurs when a question of the quiz is deleted, the quiz is set to the invalid state *INVALID* and informs the tournament by sending an event. Then, the tournament can try to generate a new quiz. Overall, the owns relation between aggregates may simplify the business logic, and the weight of the relation is business logic dependent. In the case of the relation between tournament and its quiz, a single event was identified where the tournament does not control the quiz behavior.

Merges are a mechanism used to enforce system consistency by preventing the lost update anomaly while making an effort for the functionality not to abort. Programmer defined intentions are what makes the merge capable of achieving the described behaviour. Intentions provide some semantic insight, by specifying which pairs of fields cause a lost update when updated independently by two different functionalities. Intentions are crucial aspect of merges because, if merges were implemented blindly the system would still be vulnerable to these anomalies. The merge mechanism offers a compromise between system consistency and functionality execution success rate. On the other hand, the fields merge operations allow additional levels of freedom for successfully merge concurrent versions. For instance, when two participants are concurrently added to a tournament it is possible to have a participants merge operation that accepts both adds. This behavior is similar to the incremental operations describe in the literature for conflict free data types.

The merge between aggregates during commit is specified at the aggregate granularity. However, if the merge occurs in a complex functionality it is necessary to guarantee that all merges are consistent. Therefore, it may be necessary to consistently define the intentions and merge operations of the different aggregates. The functionality that updates a tournament may also update the quiz. This is the case for the *updateTournament* functionality in which the quiz may also need to be updated, according to the tournament state. If two *updateTournament* execution occur concurrently, both the tournament and quiz versions need to be consistently merged, where the merged tournament version is consistent with the quiz merged version. For instance quiz and tournament must start at the same time. Therefore, the intentions of the tournament and quiz must be consistently designed, as should their merge operations. For instance, the tournament cannot allow the *startTime* and *endTime* to be updated separately in the same way the quiz does not allow the *availableDate* and *conclusionDate* to be updated separately, and their merge operations should be similar.

Although the merge operation for list of participants can follow the incremental strategy, this decision is business logic dependent. The tournament has a list of topics, which can be changed, and the questions in the quiz are chosen based on this set of topics. Therefore, when two concurrent versions of a tournament have, both, their topics changed, an incremental merge operation cannot be applied, because it would not be consistent with the merge of the questions in the quizzes versions, due to the lack of information on the quiz side. Therefore, the decision was that the semantics for the merge operations for topics and quiz questions, was an overwrite, where the last to commit prevails.

Usually aggregates only have one element of an upstream aggregate replicated once. However, there are cases where the element can be replicated more than once. For instance, the tournament creator and participants are students and it is possible for the creator to participate in the tournament. In the context of a tournament, these are two different elements referring the same student. This case need the definition of an intra-invariant, *CREATOR_PARTICIPANT_CONSISTENCY*, as already described by some of the test scenarios.

5.4 Complexity Analysis

In terms of the solution complexity, we compare with a microservices system implemented with eventual consistency and the Saga pattern [4]. While the complexity of implementing the business logic using sagas depends on the number of distributed transactions and semantic locks [5], the complexity using transactional causal consistency depends on the number of aggregate elements, intentions, and merge operations. The number of aggregates is usually smaller than the number of functionalities, and, in TCC, the propagation impact of adding a new aggregate is confined to the aggregate and its inter-invariants, while, in sagas, the impact of adding a new functionality may impact all other functionalities. However, more research and experimentation is required.

Field merges and intentions are static, they don't depend on the functionality which called or the context in which they are executed or defined. For instance, when a quiz needs to be merged as a result of two concurrent quiz updates, it follows a set of defined intentions and it's field merges are performed in a certain way. If a quiz needs to be merged, in the context of a tournament update, it will follow the same exact intentions and will merge the fields in the exact same way. The same occurs for the tournament. But the tournament's intentions and field merge operations need to comply with those of the quiz. And they are always compliant even if the functionality executed on the tournament does not affect the quiz. This is another instance on which we can that a downstream aggregate, the tournament, depends on the upstream aggregate, the quiz.

Anyway, further research needs to be done on the complexity analysis.

5.5 Threats to Validity

The model does not consider that two aggregates can have a bidirectional relationship, but it may be considered that in this case each aggregate is downstream/upstream of the other, and the functionalities of any of the aggregates use the other as a downstream aggregate.

To simplify the explanation of transactional causal consistency, we used version numbers to select the aggregates that belong to a causal snapshot. However, in a distributed implementation the generation of a total order of version numbers is not possible, or at least it does not scale. Nevertheless, existing TCC [7, 8] implementations can support a similar semantics in the construction of causal snapshots using timestamp intervals. Additionally, the causal snapshot we defined is a best effort snapshot, which in the distributed context may be harder to achieve. Anyway, the transactional semantics and how it impacts on the business logic design is the same.

The simulator does not implement distributed communication in the invocations between the different services implementing a functionality. This simplifies the simulator and it does not impact the business logic, because each service invocation occurs in a different transaction. Even though they are not distributed, the interleaving between service invocations can occur. On the other hand, it does not simulate distributed faults but this is not a concern of the simulation.

We are not addressing the replication of aggregates. This problem is usually understood as a low level data issue, associated with performance. However we look at it as a higher level of abstraction by defining the inter-invariants, which define consistency rules for the information replicated between two aggregates. This is, in our opinion, the correct way to look at it, from a business logic perspective.

Commits in the simulator are divided into two parts: verification and write. During verification it is done the concurrent version detection and the versions merge. The commit only progresses to write verifying that there are no more concurrent versions. However, it is possible that new concurrent aggregate versions are added to the system after verification, because another commit is done concurrently. To avoid this situation the commit should be have serializability isolation. The TCC distributed implementations support commit serializability.

The techniques used to implement the scenarios tests in 5.2 do not correspond to a real concurrent execution of the scenarios. However, they allowed to precisely simulated the relevant interactions.

6 Conclusion

Contents

6.1 Results

Microservices architectures have to handle the burden of eventual consistency. The Saga pattern has been used to implement the microservices functionalities, but it is well known that there is a trade-off between the amount of application business logic and its implementation effort using a microservices architecture.

We leverage on previous work, that proposes the use of transactional causal consistency (TCC) in serverless computing, to define an approach for the use of TCC on the implementation of microservices business logic, which reduces its implementation complexity.

The approach proposes new aggregate constructs, which specify the intra and inter aggregate business logic, and their extension for transactional causal consistency. Additionally, due to the low level of existing TCC implementations, we designed and implemented a TCC simulator, which supports the new aggregate constructs. Finally, the constructs and their implementation using the simulator were experimented in a business logic rich application. Therefore, we have positively answered both research questions.

The simulator code and the case study implementation are publicly available at `https://github.com/socialsoftware/business-logic-consistency-models`.

6.2 Future Work

One possible extension for our work could be the annotating the existing code with Java annotations and implement code generation associated to them to simplify the implementation of functionality and extensions. For instance, annotating methods with the *@TCCFunctionality* annotation, indicating that they implement a TCC functionality, which means that the unit of work would have to be instnatiated and a commit would have to be called at the end. Or annotating certain methods with *@Intrainvariant* to denote that the method represents an invariant definition so the *Aggregate* superclass can know more easily which methods of a specific aggregate type define invariants.

Other possible extension for our work could be creating a formal declarative language for the definition of aggregates. This language would describe the aggregate fields, composing entities and corresponding mappings to upstream aggregates, functionality outlines, intra-invariants, inter-invariants, fields that can be changed by functionalities, intentions and event subscriptions. The objective is to have a way of defining an aggregate abstracted from the programming language.

Then from the declarative definition of the aggregate a processor could be developed that would generate code in Java that implements the described aggregate using the annotations described in the first extension.

A further improvement could also be extending the simulator to work in a distributive environment with a more complete TCC implementation.

Bibliography

[1] C. O'Hanlon, "A conversation with werner vogels," *Queue*, vol. 4, no. 4, p. 14–22, May 2006.

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

[3] M. Fowler, "Microservices," Web page: http://martinfowler.com/articles/microservices.html.

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

[5] 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.

[6] D. Haywood, "In defense of the monolith," *Microservices vs. Monoliths - The Reality Beyond the Hype*, 2017. [Online]. Available: https://www.infoq.com/minibooks/emag-microservices-monoliths/

[7] C. Wu, V. Sreekanti, and J. M. Hellerstein, "Transactional causal consistency for serverless computing," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 83–97. [Online]. Available: https://doi.org/10.1145/3318464.3389710

[8] T. Lykhenko, R. Soares, and L. Rodrigues, "Faastcc: Efficient transactional causal consistency for serverless computing," in *Proceedings of the 22nd International Middleware Conference*, ser. Middleware '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 159–171. [Online]. Available: https://doi.org/10.1145/3464298.3493392

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

[10] P. Bailis and A. Ghodsi, "Eventual consistency today: Limitations, extensions, and beyond," *Communications of the ACM*, vol. 56, no. 5, pp. 55–63, 2013.

[11] 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.

[12] 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.

[13] 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.

[14] 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

[15] N. Preguiça, J. M. Marques, M. Shapiro, and M. Letia, "A commutative replicated data type for cooperative editing," in *29th IEEE International Conference on Distributed Computing Systems*, ser. ICDCS 2009. IEEE, 2009, pp. 395–403.

[16] W. Yu and C.-L. Ignat, "Conflict-Free Replicated Relations for Multi-Synchronous Database Management at Edge," in *IEEE International Conference on Smart Data Services, 2020 IEEE World Congress on Services*, Beijing, China, Oct. 2020. [Online]. Available: https://hal.inria.fr/hal-02983557

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

[18] 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 2nd Edition*. Addison-Wesley, 2011.