Making Long-Lived Transactions Easier to Develop

João Pedro Garcia Franco Carvalho

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Examination Committee

Chairperson: Prof. Luís Eduardo Teixeira Rodrigues
Supervisor: Prof. João Manuel Pinheiro Cachopo
Member of the Committee: Prof. João Ricardo Viegas da Costa Seco

October 2013

Acknowledgements

There are many people to thank for the development of this work, many of them without whom it would not have been possible to complete. I first would like to thank everyone at the ESW Sofware Engineering Group at INESC-ID, especially my advisor, Professor João Cachopo. His guidance, knowledge and support were a great contribution to the development of this work, and its quality is greatly due to him.

I would also like to thank my colleages at IST's DSI, in the FenixEdu project, whose ideas and feedback were critical to the making of this work. A special thanks to my current colleages: Luis Cruz, Susana Fernandes, Ricardo Rodrigues, Pedro Santos, Sérgio Silva, Artur Ventura and David Martinho; as well as my former colleages: Diogo Simões, João Neves and João Antunes.

A special thanks to my family and friends who ran this journey by my side and always believed in what I was trying to accomplish.

I dedicate this work to my grandmother Veneranda, whose strength and perserverance were crutial in completing this work.

Lisboa, November 21, 2013
João Pedro Garcia Franco Carvalho

Being the richest man in the cemetery
doesn't matter to me. Going to bed at night
saying we've done something wonderful,
that's what matters to me.

- Steve Jobs

Resumo

Ao longo dos últimos anos, a Memória Transacional tornou-se um tópico bastante popular, crescendo para além de um mero tópico de investigação. Mais recentemente, o conceito foi extendido para suportar persistência, e como tal, o conceito de Memória Transacional Persistente (PSTM) foi criado. Nesta dissertação irei propor uma extensão às PSTMs para suportar Transacções de Longa Duração. Uma Transacção de Longa Duração é uma Transação com um tempo de vida superior a uma Transacção simples, executada em vários passos disjuntos. Os sistemas de suporte transacional existentes hoje em dia não estão preparados para lidar com Transacções de Longa Duração, e como tal, os programadores de aplicações empresariais vêm-se forçados a utilizar *workarounds* para as implementar.

A minha tese é que o suporte a Transações de Longa Duração deveria ser fornecido a nivel infra-estrutural, usando uma Memória Transacional Persistente. Irei descrever os desafios que tornam as Transacções de Longa Duração difíceis de implementar, e irei propor uma solução para facilitar o seu desenvolvimento. Irei mostrar como os programadores podem tirar partido de Transacções de Longa Duração que sobrevivem a *restarts* do sistema, requerem modificações de código mínimas, permitem vários utilizadores concorrentemente e com resultados de performance comparáveis às transacções simples.

Abstract

Over the past years, Software Transactional Memories have become more and more popular, growing to be something more than simply a research topic. On top of that, the concept has been extended to encompass persistence, so the concept of Persistent Software Transactional Memories (PSTM) was born. In this dissertation, I propose an extension to PSTMs to support Long Lived Transactions. Long Lived Transactions are transactions with a lifespan larger than a typical transaction, executed in multiple disjoint steps. Current Transaction Support Systems do not cope well with Long Lived Transactions, forcing programmers to devise clever ways to implement them.

My thesis is that supporting Long Lived Transactions should be done at the infrastructural level on top of a Persistent STM. I will describe the challenges that make Long Lived Transactions hard to implement, and propose a solution to address them. I show how programmers can take advantage of Long Lived Transactions that can survive application restarts, require minimal code modifications, allow multiple concurrent users and show minimal overhead in relation to regular transactions.

Palavras Chave
Keywords

Palavras Chave

Memória Transaccional
Transacções
Transacções de Longa Duração
Consistência de dados

Keywords

Transactional Memory
Transactions
Long Lived Transactions
Data Consistency

Contents

List of Figures

Chapter 1

Introduction

For many years, enterprise applications were developed using two-tiered architectures. In such architectures, there was typically a mainframe with great computational power, which served requests from thin clients. As hardware evolved over the years, so did the development of enterprise applications. Nowadays, most applications are developed using a three-tier architecture: Data Tier, Application Tier and Presentation Tier. Despite this separation, most applications still rely on the Data Tier for transactional support.

With the adoption of multicore architectures over the past few years, *Software Transactional Memory* (STM) has seen many advancements. Because data persistency is a critical requirement in enterprise applications, STMs have been extended to collaborate with persistent storage systems, giving birth to the concept of *Persistent Software Transactional Memory* [11]. Thus, several enterprise applications, such as the FenixEdu[1] web application, are now using PSTMs for transactional support.

Long-Lived Transactions (LLTs) were first described in 1981 as "[..] transactions with lifetimes of a few days or weeks"[15], and can be found in many enterprise applications. Due to their duration, Long-Lived Transactions pose some challenges not encountered in short transactions, and thus, many attempts have been made to support them. Despite such attempts, support is either non existing or lackluster.

In this thesis, I aim at adding support for Long-Lived Transactions in applications with Rich Domain Models. A Rich Domain Model is "An object model of the domain that incorporates both behaviour and data.", as described by [12], meaning that domain objects hold both their data and the business logic that manipulates them, as opposed to an Anemic Domain Model, in which the domain objects simply contain data, and the business operations are handled by a separate Service Layer. Also, in a Rich Domain Model, each domain object typically has an arbitrarily complex web of associations, multivalued attributes, inheritance, and is typically a part of some Object-Oriented design patterns.

The main targets of this work are enterprise applications in which the domain objects are persistent, transactionally updated, and handled transparently at an infrastructural level (meaning that the programmer should be mostly unaware of the persistence/data tier).

1.1 Thesis Statement

My thesis statement is that it is possible to simplify the development of Long Lived Transactions, by providing infrastructural-level support on top of a Persistent STM.

I claim that it is possible to provide a way for programmers to support transparently Long Lived Transactions without the need for significant modifications to existing code, and with performance results comparable to those of regular transactions.

[1] See `http://www.fenixedu.org`

1.2 Contributions

The contribution of this dissertation is a solution that will allow programmers to develop Long Lived Transactions with minimal effort using the Fenix Framework. The major highlights of this contribution are:

- Infrastructural-level support for Long Lived Transactions
- Add Long Lived Transaction support to existing applications with minimal code modifications
- A simple API to manage the life-cycle of Long Lived Transactions
- Support for multiple concurrent users working on a Long Lived Transaction
- Small overhead on the execution of the Long Lived Transaction's steps

1.3 Document Structure

The remainder of this document is organized as follows:

- **Chapter 2** describes what Long Lived Transactions are, why they are difficult to implement and the requirements for the solution.
- **Chapter 3** presents existing work on Long Lived Transactions, and discusses why they are not well suited to solve the problem at hand.
- **Chapter 4** describes the components of the Fenix Framework, which is the Framework used to implement the approach that I propose in this work.
- **Chapter 5** proposes an extension to the Fenix Framework to support Long Lived Transactions.
- **Chapter 6** presents some benchmarks for the proposed solution and some optimizations.
- **Chapter 7** describes a few ideas for further enhancement of the proposed solution.
- **Chapter 8** draws some conclusions from this work.

Chapter 2

Long-Lived Transactions

In this Chapter I will describe what Long-Lived Transactions are and why they are difficult to implement using the currently available tools. I will also describe the objectives of this work, and lay down the requirements that must be fulfilled by the implementation.

2.1 What are Long-Lived Transactions?

Informally, Long-Lived Transactions are transactions with a lifetime larger than a typical database transaction. To better understand this concept, consider the example shown in Figure 2.1, corresponding to a simplified fragment of the domain model for an application in the higher education domain.

In this simplified domain model, a course belongs to a department, has a name, its objectives, the credits granted upon completion, and the recommended bibliography. The domain is deemed to be consistent only if all attributes of course have a defined value, and each course must have a department. Each department is responsible for managing its courses, meaning that it is up to someone who works in the department to start the process of creating new courses.

Note that the creation of a new course should be executed transactionally, because other users of the system should not be able to see a course in an inconsistent state (either not connected to any departement and/or without all attributes properly defined).

The pseudo-code in Listing 2.1 implements the business operation of creating a new course. Assume that `department` is inferred from the user performing the operation.

There are several ways to implement this operation. I will now describe two common scenarios for said implementation.

2.1.1 Business Transaction in a single interaction

A possible implementation (most likely, the simplest) of the course creation operation in a web application is to have a single page in which the user provides all the required information. Once the information is submitted, a new `Course` is created, associated with the department of the person performing the

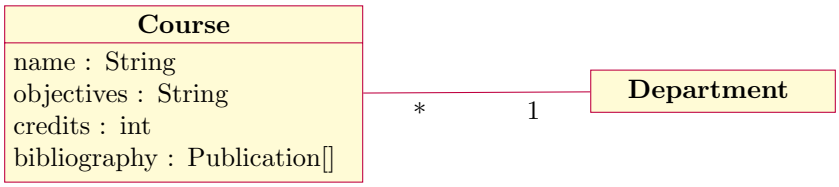

Figure 2.1: Sample Domain Model in UML

```
Course course = new Course();
department.addCourse(course);
course.setName(name);
course.setObjectives(objectives);
course.setCredits(credits);
course.setBibliography(bibliography);
```

Listing 2.1: Pseudo-Code for the creation of a new course. A new Course is created, associated with its department, and its attributes are filled.

operation, and all its attributes are filled according to the information submitted by the user. The new object is then stored in the database, making it persistent and available for other users to view.

In this scenario, in which all the information can be provided in a single user interaction, the transactional guarantees of the operation are ensured by the underlying database (which is assumed to provide the classic transactional semantics [15]), because the whole operation can be performed within the scope of a single database transaction.

An important consequence of implementing the operation in a single database transaction is that the programmer can manipulate all the domain objects involved directly. The order in which the modifications to the objects are performed is irrelevant, as long as the domain is consistent when the transaction is committed. This is the semantics typically expected by a programmer of such applications: There may be instants in which some domain objects are in an inconsistent state (e.g. before defining the course's bibliography), but this inconsistent state will never be seen by the other users of the application. Those users will see the fully created object only when the operation is committed.

2.1.2 Business Transaction across multiple interactions

The model described in the previous scenario, whereas simple and easy to implement, may not be suited for every situation. Imagine that instead of four attributes, `Course` had 50 attributes. It would then be unfeasible to ask the user to fill everything out in a single web page. So, one possible solution would be to split the various attributes in multiple pages, accounting for multiple user interactions.

Let us now assume that the creation of a course is made throughout three interactions. In the first interaction, the user selects the course's name; in the second interaction, the user introduces the objectives and credits; and in the final interaction, the user selects the bibliography, thus creating the course.

At first glance, it would seem quite easy for a programmer to change the logic programmed in the first scenario to meet the new requirements: The programmer would simply have to split the code performed in a single request into three smaller parts, each to be executed in one request.

Yet, having three separate requests implies having three different database transactions, breaking the atomicity and isolation of the operation. After handling the first request, the persisted domain would be in an inconsistent state (a course with no attributes other than its name).

The implementation of the business logic must then take this issue into account, because the programmer cannot write the updates directly on the domain.

This scenario represents what was defined as a Long-Lived Transaction, in which the Business Operation has a larger lifetime that a single database transaction (in this particular case, three database transactions).

2.2 Why are they difficult to implement?

In this multiple interaction scenario, the programmer must take special care when implementing the operation. As mentioned above, we cannot simply split the code used in the first scenario.

The programmer has several choices for the implementation:

1. Keep a database transaction open during all the steps of the operation.
2. Create a parallel representation of the objects that are being manipulated, store them outside the domain and apply the changes only in the last interaction.
3. Change the domain model, to represent the consistency state of the objects being manipulated. This affects the code that operates on that portion of the domain, because it must filter out objects that are still in an inconsistent state.

It should be clear that any of these solutions is far from trivial to implement, has some serious consequences (presented below), and, as I will demonstrate, is an unnecessary burden to the programmer.

2.2.1 Keeping a database transaction open

Given that atomicity and isolation are broken due to the fact that each interaction with the user is done within its own database transaction, we could think that the solution would be to keep the database transaction open during the whole business transaction.

However, most modern Relational Database Management Systems (RDBMSs) do not cope well with transactions that are open for arbitrarily large periods of time, because a long lived database transaction may limit concurrency, cause timeouts and deadlocks or starve the database's connection pool. All these factors contribute to making this approach highly undesirable, making the programmer seek alternative approaches.

2.2.2 Parallel Representation of the domain

In this approach, a series of objects similar to the domain objects are created, and must be managed manually, kept in the user's session context, outside the domain.

As the complexity of the domain and the number of objects manipulated by the operation grows, it becomes harder for the programmer to manage manually all of the data that must be kept. Ultimately, there is a copy of the whole domain stored in the user's session, waiting for the last transaction to update the domain with all the information entered by the user.

This is the opposite of what would be desirable for the programmer: She should be able to operate directly on the domain.

2.2.3 Changing the domain model

Now imagine that an additional requirement is added to the system: The user may be allowed to log off in the middle of the process, and, upon logging back in, continue the work where she left off. Additionally, anyone else in the user's department should be able to pick up where the original user left off. With these requirements, it becomes clear that simply storing a copy of the domain in session-local storage is not enough.

Web application containers already provide an application context, which could potentially solve this issue. Unfortunately, there are no real guarantees that the data stored in that context will be kept for an arbitrarily large period of time (recall that our transaction may span several days or weeks), meaning

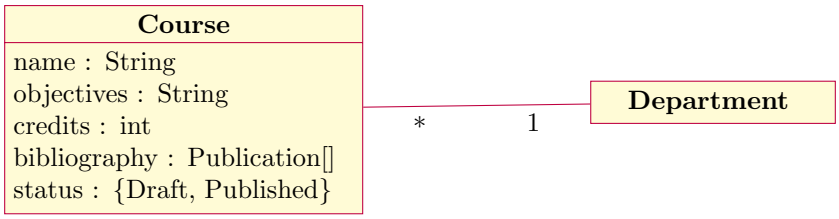

Figure 2.2: Domain Model with state representation

that the intermediate data must be kept persistently, forcing the programmer to be concerned with this issue.

A possible solution for this requirement is to change our domain model, by adding a new attribute to the objects being modified (in this case, `Course`, as shown in Figure 2.2). The `status` attribute indicates whether the course is in a consistent state (`Published`) or not (`Draft`). Adding this attribute has a cost: Not only the domain becomes polluted with information that is not relevant to the object being modeled, but this solution affects other functional code across the application (e.g., course listings must filter out courses that are still in the `Draft` status), scattering the filtering code throughout the application.

2.2.4 Other approaches

There are other approaches to Long-Lived Transactions, such as using the Database. There are some DBMSs that announce support for Long-Lived Transactions, such as Oracle's Workspace Manager. However, such support is not standardised, their API's are proprietary, rendering Web frameworks unusable in those cases (they generally only support the SQL standard), forcing the application to be written against the proprietary API, making it very hard to change and maintain. All these factors contribute to make this approach inadmissible for most cases.

2.3 Objectives

This Chapter shows that implementing Long Lived Transactions using current transaction support systems is a rather difficult task that promotes poor software engineering practices. As such, the work presented in this dissertation aims at providing a solution that:

- Survives system restarts, ensuring that the intermediate data is always available.
- Provides the same correctness guarantees as regular transactions.
- Allows multiple users to collaborate concurrently on the execution of a Long Lived Transaction.
- Does not impose an overhead on the execution of regular transactions.
- Shows performance results comparable to those of regular transactions.

Chapter 3

Related Work

In this section I describe the various areas in which an attempt to solve the problem of Long Lived Transactions has been made, namely Database Management Systems, Workflow Systems, and Object-Relational Mapping Systems.

Also, due to its relevance regarding the solution described in section 4, I briefly present Software Transactional Memories (STM), Nested Transactions, Persistent STMs, and how they cope with short lived transactions.

3.1 Transactions in Database Management Systems

Transactions are an age-old notion in the database world. They comprise a *Unit of Work* performed within a Database Management System, and ensure that concurrent access to shared mutable data is done in a consistent way.

By definition, a database transaction must provide the four ACID properties:

- **Atomicity** Requires that in a transaction, all database operations either *all* occur, or *none* occurs. This property guarantees that the transaction is indivisible and irreductible.
- **Consistency** Requires that every transaction will bring the database from one valid state to another (according to a series of consistency predicates defined for the database).
- **Isolation** Requires that individual record updates within a transaction are not visible outside of that transaction. When the transaction commits, all record updates are visible to the rest of the system.
- **Durability** Requires that once a transaction is committed, it will survive permanently. This means that once a transaction is committed, its effects on the data will be permanent.

3.1.1 Isolation Levels

To better understand the desired isolation level of a Long-Lived Transaction support system, an analysis of the most common isolation levels is in order.

Attaining full isolation between transactions, while seemingly necessary, can oftentimes be a burden to the DBMS, due to the required overhead. Thus, isolation is, out of the four ACID properties, the one most often relaxed. Isolation is typically implemented using Locks or Multiversion Concurrency Control, which may result in a loss of concurrency. To cope with that fact, several standard isolation levels have been defined.

In the ANSI/ISO SQL standard[22], four isolation levels have been defined (from most to least relaxed):

- **Read Uncommitted** This is the lowest isolation level provided by any transaction support system. At this level *Dirty reads* may occur. A *Dirty read* is defined as a read operation that reads data updated by a live (not yet committed) transaction.
- **Read Committed** In this level, *Dirty reads* cannot happen (meaning they only read data written by already committed transactions). However, multiple reads to the same data location may read different values (from transactions that committed in between), even within a single transaction.
- With **Repeatable Read**, all reads within the same transaction read the snapshot established by the first read.
- **Serializable** is the *highest* isolation level defined in SQL. Serializability requires repeatable reads, as well as guaranteeing that *Phantom reads* do not occur. A *Phantom read* occurs when, in the course of a transaction, two identical queries (that act upon a range of values) return a different result collection. This can occur when running queries like "SELECT * FROM users WHERE age BETWEEN 10 AND 30;". If an interleaving transaction creates a new user with an age between 10 and 30 years, the second run of the query will show a result which contains the new user.

In [25], an extension to Serializability is described: Strict Serializability. In addition to the requirements for Serializability, Strict Serializability requires that transaction histories are real-time ordered, meaning that all transactions histories must be ordered in a way consistent with the precedence order of the operations. In practice, this isolation level is not implemented by DBMSs, due to its performance impact.

Another common isolation level (not described in SQL-92) is **Snapshot Isolation**. In snapshot isolation, all reads made within a transaction are guaranteed to see a consistent snapshot of the database, and the transaction itself will only commit if no updates it has made conflict with any concurrent updates since the snapshot. Snapshot isolation arose from work on Multiversion Concurrency Control, and thus did not make it into the lock-centered mindset of the SQL-92 standard. This gap was heavily criticised in [2].

3.1.2 Concurrency Control

Concurrency Control is the mechanism that ensures that "Concurrent execution should not cause application programs to malfunction" [26]. This property was coined in 1993 as the first law of Concurrency Control, by Jim Gray. In fact, concurrency control mechanisms are what ensure that the ACID properties of transactions are kept, in an environment with concurrent access to shared mutable data. Research on the topic dates back to the 1970s [28] and 1980s [15], and is still a hot topic nowadays.

There are two main categories of Concurrency Control:

- **Optimistic** Delaying integrity checking until the end of the transaction, without blocking any of its reads and writes. Optimistic concurrency control allows for greater concurrency, because concurrent operations can proceed without interfering with each other. On the other hand, given that integrity is only checked at the end of the transaction, conflicts are not detected until all the work is done and the transaction has to be restarted. It is typically implemented by Multiversion Concurrency Control and Timestamp Ordering mechanisms [3].
- **Pessimistic** Every data access in the transaction acquires a lock before proceeding. During the acquisition process, if an integrity violation is detected, the transaction is aborted, rolling back every

write and releasing every lock held. Once the transaction is finished, it is marked as committed, and all the locks are released. The main method in this category is Two-Phase Locking [3].

Historically, pessimistic concurrency control has been the dominant category, and even nowadays, most Relational Databases implement it.

Due to this fact, most of the work regarding Long-Lived Transaction support in DBMSs is related with locking approaches. In fact, one of the main reasons why databases do not cope well with long transactions, is that the transaction holds a lock for each accessed record. In lock-based DBMSs, the frequency of deadlock goes up with the square of the concurrency level and the fourth power of the transaction size [15]. Thus, several strategies have been devised to minimize the probability of deadlocks on Long-Lived Transactions.

3.1.3 Relaxation of Transactional Properties

Given that holding the locks for the duration of the transaction was not viable, the first proposed solution for this problem was to accept a lower degree of consistency, by allowing transactions to release their locks before commit.

In [15], the author proposed a model in which only *active* transactions (the ones currently updating the database) hold locks. *Sleeping* transactions (not currently updating the database - also known as *User Think-Time*) do not hold any locks. This means that the isolation property is broken, due to the fact that other transactions will be able to see an uncommitted value.

In [29], the authors propose an extension to Two-Phase locking, called *altruistic locking*. This extension provides the concurrency control manager with a set of rules that allow it to release locks early in the transaction. The altruistic protocol takes advantage of the transaction's data access pattern. In particular, it takes advantage of the knowledge that a transaction no longer needs access to a data object that it has locked. Transactions can then issue *release* operations, indicating that a certain piece of data will no longer be accessed. Other transactions will be allowed to access this data concurrently, while agreeing to abide by certain restrictions that were put in place to ensure that all transactions see a consistent database state.

3.1.4 Transaction Logs

Another problem with Long-Lived Transactions in DBMSs results from the use of a Transaction Log, which contains a record of every operation performed in the database, so that the DBMS can perform crash recovery. The problem is that the size of that log is finite and, over time, it fills up, thus forcing Long-Lived Transactions to abort, as some of their log records will be overwritten by more recent transactions.

In [17], the authors propose "Log Record Forwarding", as a means to relocate log entries belonging to Long-Lived Transactions, so that they will not be aborted for that reason. In this proposal, active records in the log area are copied (forwarded) to the end of the log, thus surviving another log-reclaiming cycle.

3.1.5 Sagas

Garcia-Molina and K. Salem proposed, in their 1987 paper [14], the notion of sagas. The basic idea of the saga model is to allow a transaction to release resources prior to committing. A Long-Lived Transaction is a saga if it can be seen as a sequence of sub-transactions that can be interleaved in any way with other transactions. Each sub-transaction in the saga guarantees that the ACID properties on the database are preserved.

Partial executions of the saga are undesirable, so the DBMS is responsible for guaranteeing that either all transactions in a saga are successfully completed, or compensation actions are run to amend the partial execution. Thus, each sub-transaction is associated with a compensating transaction, which undoes the changes performed by the original transaction. Note that this action does not necessarily return the database to its original state, but instead acts upon the *business state* of the application.

More formally, consider a saga T, consisting of sub-transactions $T_1, T_2, \ldots, T_n$, and compensating actions $CT_1, CT_2, \ldots, CT_n$. The system guarantees that either the sequence $T_1, T_2, \ldots, T_n$ is executed (meaning the whole saga is executed), or $T_1, T_2, \ldots, T_j, CT_j, \ldots, CT_2, CT_1$ for some $0 \leq j \leq n$, will be executed (meaning that a failure occurred at T_{j+1}, causing the previously executed sub-transactions to be compensated).

3.1.6 Offline Concurrency Patterns

Martin Fowler, in his book *Patterns of Enterprise Application Architecture* [12], introduces a series of *Offline Concurrency Patterns*, as the building blocks for Long-Lived Transaction support at the application level. In his patterns, Fowler assumes that a Long-Lived Transaction is run as a series of short-lived system transactions, whose transactional support is provided by the underlying DBMS.

The patterns provide a workaround for the fact that the ACID properties are broken when using several system transactions.

- **Optimistic Offline Lock** This pattern solves the problem by validating that the changes about to be committed by one session (or business transaction) do not conflict with the changes of another session. This is achieved by, once the session is being committed, validating that the changes about to be committed are consistent, making sure that the value previously read by the current session was not changed by another session (i.e. obtaining an Optimistic Offline Lock). The most common implementation of this pattern consists on associating a version number with the record in the system. When a record is loaded, the record's version is kept in the session, and that version is used when acquiring the Optimistic Offline Lock (by comparing the two versions). Once the lock is successfully acquired, the record's version is incremented, and the system transaction can be committed. If the lock cannot be acquired, a conflict is detected, the current system transaction is rolled back, and the business transaction must either abort or attempt to resolve the conflict and retry.

- **Pessimistic Offline Lock** This pattern prevents conflicts by avoiding them altogether. It forces a business transaction to acquire a lock in each record before using it, so once you begin the business transaction, you are mostly sure that it will complete without concurrency control issues. These locks however, are not database-level locks, but are instead high-level, typically managed by a lock manager in the application, and can have various granularities (which will have different impacts on system performance). In this pattern, whenever a business transaction wishes to access a record (for either read or write), it must acquire a lock with the lock manager, ensuring that a business transaction that wishes to write for that particular record is the only one doing so.

Typically **Optimistic Offline Lock**s are used in an environment where conflicts are expected to be low. If conflicts are likely, it is not good user experience to report a conflict after the user finished his work and is ready to commit. In that case, a **Pessimistic Offline Lock** is in order, avoiding throwing away work, because conflicts can be detected early in the transaction. Also, if the cost of a conflict is too high, no matter its likelihood, this is the appropriate pattern.

Although with the use of these patterns the development of Long-Lived Transactions is facilitated, they require the programmer to pollute the domain with the addition of code that should be at an

infrastructural level.

3.2 Workflow Management Systems

A Workflow Process is described as a sequence of connected steps (or activities), with special emphasis on the flow paradigm, where each step of the process follows the precedent, and ends just before the next step can begin.

A Workflow Process typically may have a long duration, involve many users (in fact, each step is typically executed by a different user), and range across a variety of heterogeneous systems. Individual activities range from computer programs and applications to human activities such as meetings and decision making.

In this document I shall use an abstract model to describe Workflow Processes, using the following concepts:

- **Process** is a sequence of steps necessary to achieve a goal. A process consists of activities and data.
- **Activity** is each step within a process. Activities have a name, pre and post-conditions, and a series of constraints.
- **Control Flow** is the order in which activities are executed.
- **Data Flow** is the mapping between the inputs/outputs of the data required by the activities.
- **Conditions** specify the circumstances in which certain events will happen. *Transition Conditions* specify whether a certain activity may or may not become *startable*. *Start Conditions* specify when an activity is actually started, either by all its triggering events (AND) or by just one (OR). *Exit Conditions* specify whether an activity has finished successfully.

In this model, an activity with no incoming activities is considered a *Start Activity*. A process is considered finished once all its activities are terminated.

Over the years, many Workflow Modelling Languages have been devised, and in fact, it is still a very active area of research. Examples of such languages are YAWL [33], AWGL [9] and UEML [34], among many others.

In [31], the term *Transactional Workflow* was introduced, recognising the relevance of transactions to workflows. Transactional Workflows provide functionality required by each workflow process (such as task collaboration), which is usually not available in typical DBMSs. This, however, does not imply that workflows are similar to database transactions, nor that they support all the ACID properties (namely concurrency control, backward recovery, and data consistency).

Workflow Management Systems (WFMSs) are the systems that manage user-defined Workflow processes for different types of jobs and processes. It is the WFMS that provides the transactional properties to the transactional workflow. These include transaction management techniques such as logging, compensation, etc.

Given that their support for recovery and failure handling is lacking, unlike regular transaction systems, they typically provide the means for the user to determine the actions necessary to recover from failure and consistency issues.

3.2.1 Implementing Transactional Workflows

In [5], a workflow is seen as a *Long-Running Activity*, and is modeled as a set of execution units that may consist recursively of other activities, or top-level transactions. Control and data flow is specified

either statically in the activity's *script*, or dynamically using Event-Condition-Action (ECA) rules. This model contemplates compensation, execution unit communication, activity status queries and exception handling.

In [36], the author suggests that semantic transaction concepts be merged with workflow concepts to promote consistent and reliable workflow systems. The author defines a transactional workflow to be a control sphere that binds these transactions, using dependencies to enforce as much behavioral consistency as possible.

For a more comprehensive review on the topic, refer to [37].

3.2.2 Sagas

The concept of Sagas was previously presented in Section 3.1.5. In this section, I show how sagas can be implemented on top of Workflow Management Systems.

All the sub-transactions of the saga are grouped into a block (Forward Block). The control flow within the block represents that of the saga, with each sub-transaction represented as an activity. Every activity has an incoming transition condition, which is that the previous activity has finished successfully (i.e. the corresponding transaction has committed).

If a transaction aborts, the corresponding outgoing condition will evaluate to false, and no other activity in the block will execute. When execution of the block terminates, its data container will contain a list of the status of each activity. In this case, control is passed to a second block (Compensation Block), containing the compensating activities in reverse order.

When the compensation block is executed, all the conditions that correspond to activities that executed in the Forward Block are activated, causing the compensating activities to be executed in order.

3.2.3 Implementing Long-Lived Transactions using WFMSs

As described in the previous sections, several attempts have been made to implement Advanced Transaction Models on top of Workflow Systems, as a means of supporting Long-Lived Transactions.

Several other methods have been proposed in the literature (e.g., [1, 35]). However, such implementations are not suitable for the requirements defined in Section 2.1, due to the fact that most WFMSs do not provide full ACID properties.

3.3 Object-Relational Mapping

Due to the increase in popularity of object-oriented programming, a large number of Object-Relational Mapping (ORM) tools have arisen over the past few years [32]. These tools range from simple data access layer abstractions (which simply abstract the necessary instructions to perform the desired operation), to fully fledged tools that handle every aspect of the Object-Relational Mapping, making the underlying database completely transparent to the programmer.

Object-Relational Mapping tools are available for a great number of languages. For Java, there are many competing persistence standards, including: Java Persistence API[1] (JPA), Java Data Objects[2] (JDO), Spring Data[3], among others. Both specifications allow a programmer to create domain entities by decorating Plain Old Java Objects (POJOs) with annotations, making the persistence layer transparent to the programmer. Whereas JDO provides a larger number of features than JPA, the latter has seen greater adoption by the community, both in terms of usage and support. The most well-known implementations

[1] `http://jcp.org/en/jsr/detail?id=317`
[2] `http://www.jcp.org/en/jsr/detail?id=243`
[3] `http://projects.spring.io/spring-data/`

of the JPA are Hibernate (`http://www.hibernate.org`), OpenJPA (`http://openjpa.apache.org`) and EclipseLink (`http://www.eclipse.org/eclipselink/`).

To provide transactional support for domain objects, ORMs typically delegate transaction management to the underlying Database (which is assumed to provide full ACID support). In fact, as of this writing, all the examples shown above operate this way. Thus, Long-Lived Transaction support in ORM tools is largely dependent on the underlying DBMS, presenting the same challenges as described in Section 3.1.

Given the popularity of Java, I will now consider the JPA, and one of its most popular implementations, Hibernate, to understand how they cope with Long Lived Transactions. Note that despite the concreteness of the examples presented, their concepts are generic enough to be applied to any other ORM tool.

3.3.1 JPA Optimistic Concurrency Control

The specification of the Java Persistence API assumes the use of optimistic concurrency control (or optimistic locking). In this context, optimistic locking is used to prevent the database from holding on to critical resources, potentially causing high degrees of contention. This is achieved by operating directly on the domain objects (in memory), delaying the propagation of the changes to the database as much as possible.

As part of the JPA, there is the concept of Version field. This field allows *disconnected operation*, meaning that the reads/writes from the database are deferred until a checkpoint or the end of transaction. When using this optimistic approach, all the versions of the objects used in the transaction are checked against the version present in the database, thus ensuring that no dirty reads will occur.

3.3.2 Hibernate Long Conversations

Hibernate supports the concept of *Long Conversations* by allowing a session[4] to remain open as long as the user interaction lasts, potentially executing multiple database transactions. However, the programmer is responsible for ensuring the atomicity and isolation of the business transaction, by following a very strict pattern: All transactions but the last must only read data, and, thus only the last database transaction can update data. Hibernate assists the programmer in verifying that the data read across the multiple transactions is consistent, using an object versioning mechanism. However, this verification only works if all the data required for the application is read within the first database transaction.

This support is not good enough for several reasons:

- It imposes a pattern that may not be suited for many applications.
- All the state of the transaction is kept in memory, which may cause the application to run out of memory in very large transactions.
- The session is not kept persistent. If the application server restarts in the middle of the transaction, all data related to the ongoing Long Conversation is lost.
- Despite the versioning support, the business transaction may still suffer from inconsistent reads, as it spans multiple isolated database transactions. Suppose that in a database transaction (T2) other than the first one (T1), the application reads an object for the first time. If the object was concurrently modified by yet another transaction (T3) that occurred between T1 and T2, T2 would see the value written by T3, which would not be consistent. As such, these transactions are not *Serializable* (See Section 3.1.1).

All of these reasons violate the desired properties described in 2.2, making this system undesirable.

[4] A Hibernate session implements the Unit of Work design pattern [12].

3.4 Software Transactional Memories

Given that the solution I propose in Section 5.2 is built upon a Software Transactional Memory (STM), I provide here a brief introduction to the topic.

The original idea behind Transactional Memories (TM) [19] is to provide efficient lock-free data structures for highly concurrent systems using hardware. Its main goal was to free programmers from using locks and monitors to manage concurrent accesses to shared data. Managing lock granularity and composability is a cumbersome and error-prone task, and may lead to unwanted situations such as deadlocks. Over the years, the original concept of Hardware Transactional Memories has evolved, giving birth to Software Transactional Memories [30].

Software Transactional Memories bring into the realm of programming languages, the age-old notion of transactions, well-known in the area of DBMSs. However, unlike those, STMs are not concerned with the Durability property of the ACID model, and thus, many STM implementations have little in common with their database counterparts.

There have been several recent proposals for STM implementations, such as those described in [4, 18, 20, 6, 27, 21].

STMs use transactions to isolate memory operations in atomic Units of Work. A transaction typically contains a Read Set and a Write Set, which are used to register its read and write operations, respectively. The STM provides a series of mutable memory locations that can be read from or written to. In many STMs, it is up to the programmer to confine all of the application's mutable state to these memory locations, whereas some STMs provide fully transactional memory. It is the responsibility of the STM to guarantee that concurrent accesses to such memory locations are correct according to the specified correctness criteria.

3.4.1 Correctness Criteria

In Section 3.1.1, I presented the typical isolation levels (or correctness criteria) that can be found in DBMS. In [16], the author claims that the *traditional* correctness criteria are unfit for TM implementations, and proposes the *opacity* criterion. Opacity is described as a safety property that ensures that "[...] (1) all operations performed by every *committed* transaction appear as if they happened at some single, indivisible point during the transaction lifetime, (2) no operation performed by any *aborted* transaction is ever visible to other transactions (including live ones), and (3) every transaction always observes a *consistent* state of the system.", meaning that unlike Serializability, even *non-committed* transactions are prevented from accessing inconsistent states. This correctness criteria is ensured by many TM systems.

3.4.2 Nested Transactions

The best practices in software development encourage programmers to modularize their applications and have them abide by well-defined interfaces. Lock-based approaches do not cope well with this principle, because the programmer must be aware of module's internal locking conventions. Transactions, on the other hand, do not have this limitation, allowing for better composability.

In the transaction paradigm, it would then be common for application code to make a call to a library that uses transactions to protect its shared internal state. In this scenario, the transaction created by the library would be *nested* in the outer (application-owned) transaction. Nested Transactions are an extension to the basic transaction structure, supporting multiple levels of transactions: Transactions that may contain other sub-transactions, forming a transaction tree. Nested Transactions have long been used in the database world, and many of its concepts have been adapted to transactional memories.

Consider the kind of monolithic transactions commonly found in enterprise applications. In such transactions, the volume of work that must be done if the transaction needs to be rolled-back is increased. Using nested transactions, smaller portions of the work can be done in isolation, each within a nested transaction. If one of those portions fails, only that portion needs to be rolled-back, and an attempt can be made to compensate for that failure. Similarly, clients of opaque libraries may suffer performance hits if the transaction fails because of the work performed in said libraries.

In [23], Moss and Hosking defined two types of nesting: Closed and Open.

Closed Nesting

A transaction is either *top-level*, which is similar to a non-nested transaction, or is nested within a *parent* transaction. In this model, only transactions with no current children can access data. Transactions accumulate Read and Write sets, which will determine the outcome of the transaction. When a transaction reads a value, it sees either the value in its own read or write set, or the value seen by its parent (a top-level transaction sees the latest value).

When a nested transaction commits, its read and write sets are merged with the ones of its parent. It is only when the top-level transaction commits that its writes become permanent. If the nested transaction aborts, its read and write sets are discarded.

Open Nesting

Open Nested Transactions relax the isolation requirements by making the results of committed sub-transactions visible to other concurrently running transactions. When a sub-transaction commits, instead of merging the write set with its parent's, the writes are committed at the top-level. This way, a higher degree of concurrency is achieved, because resources may be released earlier and conflict detection can be applied at a higher level. It is up to the implementations to provide the mechanisms to handle recording of necessary compensating actions in case an ancestor transaction aborts.

3.4.3 Nested and Long-Lived Transactions

Nested Transactions present many similarities with Long-Lived Transactions. In both cases there is a top-level transaction, which can be divided in several smaller steps. Open nesting uses the same basic principle as the techniques described in previous sections, based on the relaxation of transactional properties, and compensating actions. Closed Nesting is similar to the desired properties presented in Section 2.1.

3.5 Persistent STMs

Over the years, Enterprise Application architecture has evolved at a very fast pace. However, one aspect has remained constant: They still rely on a relational database to handle both data persistence and transactional support. In [11], the authors argue that such design, while once justified by hardware limitations, is no longer suited for today modern applications.

A new architecture was proposed, using a Software Transactional Memory for transaction support at the application server tier, shifting the responsibility from the DBMS to the application server. The database still plays an important role in this architecture, due to the fact that the STM must be extended to support persistence (PSTMs).

By shifting the responsibility for transaction handling to the STM, enterprise applications can benefit from the many advancements in the STM area, such as multi-core machine scaling, stronger correctness guarantees and nonblocking progress conditions such as lock freedom [10].

Using a PSTM, enterprise application developers no longer have to trade correctness for performance. Modern PSTM implementations allow for Strict Serializability, while providing better performance than regular DBMSs, yielding a throughput increase of up to 23 times [11].

Another advantage of PSTMs is that, like in STMs, the programmer benefits from a much simpler and transparent programming model, allowing for much cleaner and maintainable code, better composability by using lock-free data structures, making the process of translating business requirements to code much simpler and bug-free.

In this project, I aim at extending the PSTM model, by adding the necessary building blocks to support Long-Lived Transactions (described in Section 5.2).

3.6 Why those solutions are unfit

Among all the solutions presented above, one common aspect stands out: They all require the programmer to deal with Long-Lived Transactions at the application level, which goes against my claim that support should be transparent, and at an infrastructural level.

Some of the presented solutions are based on relaxation of transactional properties, such as allowing breaches in isolation. Long-Lived Transactions should run with the same transactional properties as regular transactions, and thus, relaxation is not acceptable.

Despite the similarities between Workflow Processes and Long-Lived Transactions, WFMSs are not suited for our requirements. Typical WFMSs are not concerned with the ACID properties of transactions, and rely heavily on high-level compensatory actions for data consistency.

The solution proposed in Section 3.3 (Hibernate) imposes a very strict programming model, which may not be adequate for some applications, and does not guarantee that Long-Lived Transactions will have the same transactional properties as regular transactions.

None of the proposed solutions fit the requirements specified in Chapter 2, and thus, a new approach is needed.

Chapter 4

Fenix Framework

"Fenix Framework allows the development of Java-based applications that need a *transactional and persistent domain model.*"[1]

This chapter describes in detail the major components of the Fenix Framework, which is the Framework used to implement the solution proposed throughout this document. Section 4.1 describes the Domain Modelling Language, used to describe the application's domain model. Section 4.2 describes the high-level architecture of the Framework, briefly describing its major components and their interaction. Section 4.3 describes the process of Code Generation. Section 4.4 presents the Java Versioned STM (JVSTM), and its integration with the Fenix Framework. The information presented in this chapter is critical to understanding the proposed solution, as well as its challenges.

4.1 Domain Modelling Language

The Fenix Framework is aimed at entreprise-class applications with a rich domain model in an object-oriented paradigm. Such applications typically consist of class hierarchies representing entities with relationships among them, forming an interconnected graph.

The Domain Modelling Language (DML) is a Domain-Specific Language designed to represent such domain models, separating the domain's structure from its behaviour. The DML is designed with modularity as a core concern, allowing for incremental and modular domain definition.

In a DML file, programmers write their domain definition in a Java-like language. A class definition consists of the class name, the entity slots (either primitive or value types), and the super class. Listing 4.1 shows how the `Course` and `Department` classes from Figure 2.1 could be described in the DML. Note that as arrays are not natively supported, a Value Type must be created, describing an array of publications. Value Types are described in more detail below.

Relations in DML are named, first-class citizens that represent relationships between two classes. Relations are always bi-directional, meaning that updating one side of the relation will automatically update the other side. Relations can be concealed in one of the sides (meaning that it will not be possible to access it), however their state is still kept.

It is possible to define *one-to-one*, *one-to-many* and *many-to-many relationships*, and it is possible to define boundaries on the multiplicity of each relation (for example, a `Department` can have between 0 and 30 `Courses`). Any violation to these constraints would put the relation in an inconsistent state and is discarded.

To-many relations in the Fenix Framework have Set semantics, meaning that an object can only be present in a relation once. Also, there are no ordering guarantees when accessing the relation.

[1] `http://fenix-framework.github.io`

```
class Course {
  String name;
  String objectives;
  int credits;
  PublicationList bibliography;
}

class Department {
  String name;
}
```

Listing 4.1: DML for the `Course` and `Department` classes

```
relation DepartmentHasCourses {
  Department playsRole department {
    multiplicity 1..1;
  }
  Course playsRole course {
    multiplicity *;
  }
}
```

Listing 4.2: DML for the relation between `Course` and `Department`

Listing 4.2 shows how the relation presented in Figure 2.1 could be described in the DML. The relation is given a name that describes the relationship between the two classes, as well as names to describe the role each class takes in the relation. The multiplicity is defined for each of the roles, in this case, one `Department` has zero or more `Courses`, and one `Course` has between one and one (exactly one) `Department`.

From the domain definition, the Fenix Framework generates Java getters and setters for the properties and relations. For each class described in the DML, two Java classes are created: the domain class, in which programmers can include business logic, and a `Base` class (which the domain class extends) containing generated methods to access the persistent entities of the object.

Consider the DML on Listing 4.1. Listing 4.3 shows the Code generated for the `Course` class and its corresponding base class, as well as the `Department` base class, containing the API generated from the domain definition. For each slot declared in the DML a pair of Getter and Setter is generated, providing access to the persistent field. For relations there are two types of generated methods, depending on the multiplicity of the relation in the class. For instance, as one `Course` has one `Department`, the generated methods are simple getters/setters for the `Department`, as if it were a simple slot. On the other side of the relation, as a `Department` has multiple `Course` objects, the generated methods return a Set containing all the elements in the relation. The Framework also generates two methods to add and remove an element from the Set.

Whereas in the code shown in this document the body of such methods is `/* Generated */`, the actual code will depend on the backend used in runtime. More details about the Code Generation process are

```
public class Course extends Course_Base {
  public Course() {
    super();
  }
}

public abstract class Course_Base extends
                                        AbstractDomainObject {

  public String getName() { /* Generated */ }
  public void setName(String value) { /* Generated */ }

  public String getObjectives() { /* Generated */ }
  public void setObjectives(String value) { /* Generated */ }

  (...)

  public Department getDepartment() { /* Generated */ }
  public void setDepartment(Department value) {/* Generated */}

}

public abstract class Department_Base extends
                                        AbstractDomainObject {

  public Set<Course> getCourseSet() { /* Generated */ }

  public void addCourse(Course couse) { /* Generated */ }

  public void removeCourse(Course course) { /* Generated */ }

}
```

Listing 4.3: Generated Course class

given in Section 4.3.

4.1.1 Value Types

In the DML there is a distinction between entities and value objects. Whereas an entity is transactional and persistent, a value object is immutable and not persistent. Value objects are used as the values for slots of DML classes, and must be of well-known types, known as Value Types.

A value type contains information regarding the Java Type (such as `java.math.BigDecimal`), an alias (such as `BigDecimal`), and information regarding how the object will be externalised/internalized.

There are two categories of Value Types: Builtin (Java Primitives and their wrappers, Enums, Joda-Time types, JsonElement, and byte arrays) and user-defined. User-defined types allow the programmer to use any type they wish as a slot, provided the type is immutable (explanation as to why is provided ahead) and can be expressed in terms of other Value Types.

The Framework knows how to handle Builtin Value Types (i.e. how to store/retrieve from persistent support). User-defined, on the other hand, require that the programmer specify how the type is exter-

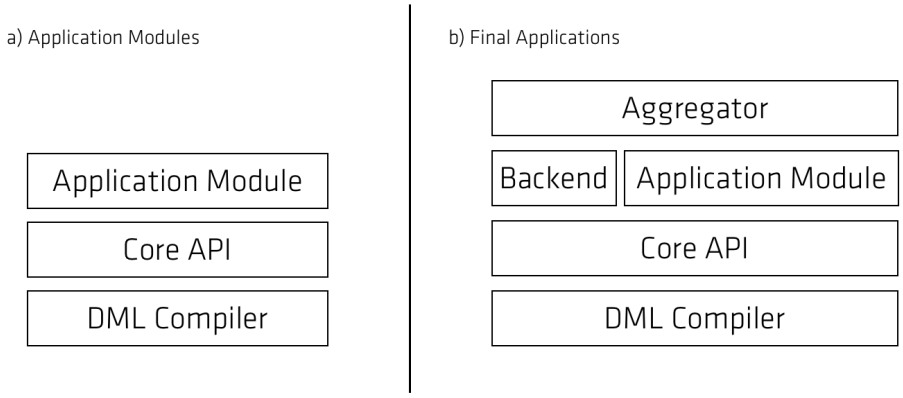

Figure 4.1: Fenix Framework's Layered Architecture

nalised/internalized. The externalised type must be either a Builtin Value Type, or a used-defined type that ultimately is externalised to a builtin type.

4.1.2 JSON

JSON (JavaScript Object Notation) is described as being "[...] a lightweight data-interchange format". JSON is quickly becoming the de-facto standard to interchange data across heterogeneous systems, replacing XML in many cases.

As such, in the context of this work, the Framework was extended to provide native JSON support (by allowing it as a Builtin Value Type), using Google's GSON[2]. Using JSON allows the programmer to define arbitrarily complex Value Types with little effort, as well as simplifying externalisation code.

The Framework also provides support to transform any Value Type to/from JSON, meaning that a JsonElement slot is enough to keep any value the Fenix Framework is able to handle. As shall be presented in Chapter 5, this proves to be a crucial feature.

4.2 Architecture

The first versions of the Fenix Framework (as presented in [11]) had a rather monolithic architecture. Transactional support was provided by the JVSTM (see Section 4.4), and persistence was implemented on top of MySQL, leaving the programmer with no other choice of technology.

With the second major version of the Framework (released earlier this year), a great architectural shift has occurred. There is now a clear separation between the Framework's public API and the transactional/persistence backends, allowing for a "write once, run everywhere" paradigm. The programmer simply writes his application against a public API and is able to run it on multiple backends.

This way, not only are applications portable across several technologies (MySQL, Neo4j, Hibernate, etc), but also testing support is greatly enhanced, as there is no need to mock the persistence API, as tests can be run using an in-memory backend.

Figure 4.1 shows the major modules that constitute the Fenix Framework.

The DML Compiler module contains the parser responsible for reading DML files and creating an in-memory description of the Domain Model, as well as all the necessary classes to represent it. It also contains the base `DomainObject` interface, which all objects of the Domain Model implement. Also present in this module are the base Code Generators used to create the Base classes for all domain objects.

[2] `http://code.google.com/p/google-gson/`

The core of the Framework is in the `Core API` module. Transaction management APIs, configuration, entry points, backend interfaces, are all defined in this module.

Many backend-independent modules are provided with the Fenix Framework bundle. These include persistent Abstract Data Types (B+Trees, Linked Lists, etc), support for Consistency Predicates [24], statistics collection tools, indexing and transaction introspection. These modules are used internally by the various backends, but can (and in most cases should) also be used by the programmers.

Backends provide the concrete implementations of transactional and persistence support, and are required for applications to work.

4.2.1 Public API

One of the major advantages of the Fenix Framework is that applications built on top of it are independent of the underlying persistence and transactional backend. However, to guarantee this property, the modules that form such applications must be compiled against the Public API of the Framework.

The public API is split in two major components:

- **DML Compiler** Allows applications to have runtime access to the structure of the domain and allows registering relation listeners that will be invoked each time a relation is modified. This module also defines the API that is generated based on the DML, which must be supported by all backends.
- **Core API** Provides the entry point for the domain's object graph through the `DomainRoot` class, the mechanism to retrieve a Domain Object from its unique identifier, transaction APIs that can be used to either mark a piece of code as transactional (@Atomic annotation) or to manually manage the lifecycle of transactions, and utilities to configure the Framework.

With this separation, the classpath of the application modules is not polluted with implementation details, allowing for a faster development and test cycle.

4.2.2 Backends

Backends are a crucial part of the Fenix Framework. They provide concrete implementations of the transactional and persistence support. Application modules should not depend directly on the backends, as their API is private and as such subject to change even among minor versions, and having a dependency on a specific backend means that portability must be sacrificed.

The fact that backends have a clear separation from the Public API allows for a much faster evolution of the backend's implementation. Major internal changes can be done without affecting the end-users directly, even in a revision release, whereas changes to the public API require either a major or minor release.

4.3 Code Generation

As previously described, access to persistent fields of domain objects is done using generated methods in Base classes.

Code generation is closely tied to the specific backends, as it is typically used to support the process of persisting an object to a database. An example of an operation performed in generated code is binding a `PreparedStatement` with the values of the object's slots, or externalizing the object to JSON.

There are two major components in code generation:

- The default code generator, from which every other generator inherits, defines the public generated API for domain classes. Its major use-case is to compile backend agnostic application modules that only need base classes because of their API (modules aren't bundled with base classes, those are generated on-demand on depending modules or applications).
- Backend-provided code generators. These generators extend the base ones, thus providing the same API, using backend-specific artifacts to fullfil the API. Backends can also use code generators to optimise runtime performance, by injecting in generated code values that otherwise would have to be computed at runtime.

Whereas this Code Generation architecture allows for great optimizations (it allows backends to perform complex operations without resorting to reflection), it comes with a tradeoff: Forcing the domain classes to inherit from Base classes hinders the readability of the code (as the user is required to either check the DML or the base class to find out the super class), makes it impossible to invoke constructors of the super class (as the Base class only generates the no-arg constructor), does not allow the programmer to choose the visibility of the generated methods (as they are always generated public) among other issues.

The Code Generation step also provides a mechanism to transfer compile-time information to the runtime. Information such as the App Name (which is used at runtime to generate the graph of DML dependencies), the name of the Backend used to compile the final application (to allow for automatic initialization), as well as pass user-defined parameters to runtime.

4.4 JVSTM

The Java Versioned Software Transactional Memory (JVSTM) [4] is a pure-Java implementation of a Software Transactional Memory (see Section 3.4).

The JVSTM uses the concept of Versioned Boxes (VBoxes) to make a memory location transactional, keeping the history of values for that position, one for each version of the box. Reads and writes to VBoxes are tracked by the JVSTM in a per-transaction basis.

Each transaction begins at a given moment, acquiring the version number at that moment. The version number is used during the transaction to ensure that all reads get the correct value at the time of the transaction's start, thus providing Opacity guarantees [16]. This allows for conflict-free read-only transactions, as concurrent transactions writing to the read boxes will write a new version instead of overwriting its previous value.

4.4.1 Integration

The JVSTM is integrated with the Fenix Framework, as one of the multiple available backends. This document focuses on the backend named `jvstm-common`. This backend uses the JVSTM for the transactional support, while abstracting the persistence details. Despite being meant to be extended, `jvstm-common` provides an in-memory implementation of the persistence API.

The implementation of the solution proposed in Chapter 5 rests on top of this abstract backend, meaning that it will work on top of any persistent support, as long as the JVSTM is used.

In this backend, Base classes use VBoxes to store the value objects transactionally, thus taking advantage of the JVSTM. The generated getters and setters are backed by `VBox.get()` and `VBox.set()`, and can be invoked only from within a transaction.

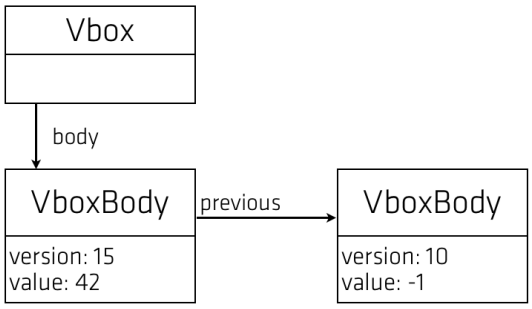

Figure 4.2: VBox Structure

4.4.2 VBoxes

A plain JVSTM VBox is simply a wrapper to a Linked-List of pairs [Version, Value], containing the history of values for that box, as seen in Figure 4.2.

A Fenix Framework VBox however, also contains a back-pointer to its owner, as well as the name of the slot it represents. This allows for the persistence support to know where to store the value of the Box.

Those specialised VBoxes can have their previous values Garbage-Collected and reloaded from persistent support on-demand.

VBoxes for slots

There are two layouts for Domain Object's slots: *(1)* Using one VBox to keep the entire state of the object (One-Box), and *(2)* Using one VBox per slot (Multi-Box).

The first layout suffers from a higher number of conflicts, as reading one slot will conflict with writing another slot on the same object (as they are mapped to the same VBox), however the memory usage is much lower, as each VBox has a cost for its underlying data structures. This aspect is critical in applications with a dense domain model, with many objects and many slots.

Smaller applications can use the Multi-Box layout to greatly reduce conflicts. In this layout, each domain slot is given its own VBox.

In both layouts, a specialised VBox called `PrimitiveBox` is used to store either the object's state or the slot's value. There is no added behaviour or data in a PrimitiveBox, however persistence support uses this information to determine whether to reload a box containing the state (or part of it) of an object or a box for a relation.

VBoxes for relations

In the One-Box layout, to-one relations are kept inside the object's state, and as such require no special handling.

On the other hand, for a Multi-Box layout, the reference to the related object is kept in a `ReferenceBox`. Just like a `PrimitiveBox`, it contains no extra behaviour or data, and serves only as a marker for persistence support to know it has to load an object reference.

To-many relations however, are handled in a very different manner. The preferred approach is to use a B+Tree [8] to store the objects on the to-many side of the relation. For each to-many relation, a *ReferenceBox* is generated, containing a reference to the domain object representing the B+Tree.

The Fenix Framework provides an implementation of persistent B+Trees that does not use to-many relations (by design, so it can be used to implement them). In this implementation, each node of the tree

is a domain object containing only a reference to its parent and its sibling. Inner nodes use an immutable TreeMap wrapper to keep indexed references to the nodes they point to. Leaf nodes follow the same strategy, keeping the mapping between keys and the objects they point to (in reality they can be used to point at anything, however for this document we are only interested in B+Trees containing domain objects).

As a B+Tree is generically a key-value map, its usage can be twofold: provide a simple way to implement to-many relations and provide support for relations indexed by a slot of the related object. Consider the previously presented *Course* example. In this model, one Course has many Students. As such, there is a B+Tree slot in the Course Base class, which contain the references to all the students enrolled in that particular course.

In the usual scenario, the B+Tree contains a mapping between OIDs and the respective target object (which is rather useless, as the Fenix Framework provides a lightweight API to read an object by its OID). Now consider a scenario where the set of students must be indexed by student name. The B+Tree will now contain a mapping between the student's name and the target student, allowing for indexed lookups of students by name.

Chapter 5

Solution

With a proper understanding of the Fenix Framework, this chapter describes the solution proposed to simplify the development of Long Lived Transactions. Section 5.1 describes the challenges faced when implementing Long Lived Transactions. Section 5.2 describes the architecture of the proposed solution, with the rationales for each design decision. Section 5.3 describes how the proposed architecture was implemented on top of the Fenix Framework using the JVSTM. Finally, Section 5.4 shows that both the architecture and implementation fullfil all the requirements, and attempts to measure the effort required to use the implementation.

5.1 Challenges

In Section 2.2 I presented several approaches for implementing Long Lived Transactions using the already existing mechanisms. Two of those approaches required the programmer to rework his code to support LLTs (keeping a parallel representation of the domain and changing the domain model), which does not fit in the requirements for this solution.

The only approach that did not require the programmer to modify his domain code was keeping the database transaction open during the multiple steps of the LLT. The reason this approach was not desirable is that most DBMSs are not prepared to handle transactions open for long periods of time.

A naive approach when implementing LLTs on top of the JVSTM could simply be running each step of the LLT on the context of a regular transaction, suspending it in between steps, only committing it in the end of the LLT. This approach presents several similarities with the approach of using a single database transaction, including some of the limitations:

- As the data is kept in transient memory, not only would LLTs not survive system restarts, but the memory usage of the system would also increase rather quickly.
- As JVSTM transactions are not designed to be shared across multiple threads, supporting multiple concurrent users would require the usage of Parallel Nesting [7].

Despite this limitation, this approach is a step in the right direction. With a few modifications, it is possible to overcome the limitations and obtain a fully functional solution. To do so, there are some challenges that need to be overcome:

- How is information persistently carried between the various steps of the transaction?
- How to use a unique representation for every possible value any domain slot can take?
- How to ensure that multiple users can concurrently access a Long Lived Transaction?

Throughout the rest of this chapter I will present solutions for these challenges and create a solution that fulfils all the requirements.

5.2 Architecture

The main goal of this work is to relieve programmers of the burden of dealing with Long Lived Transactions, making the effort needed to program an LLT similar to the effort of programming a regular transaction.

So, what does the single interaction scenario has that makes it so easy to program? It has a single transactional context that spans the whole operation (provided by the STM transaction). In the multiple interaction scenario the system transaction was shorter than the business transaction, so in each step the context was lost.

Looking at the information that is kept during the lifespan of a regular transaction, we can identify three major pieces:

- The version in which the transaction is running. This version number corresponds to the logical point in time at which the transaction started.
- A list of all the items written throughout the transaction (and the respective written values). This is the critical piece, as it contains the updated data that will be written to the global state of the application on transaction's commit.
- A list of all elements read throughout the transaction. This piece of information is critical to ensure the correctness of the operation, as the outcome of the transaction may depend on the values of all the read data.

These pieces are crucial to ensure the correct operation of an STM-based transactional system. STM libraries provide them for regular (short-lived) transactions. The solution presented below aims to provide them for Long Lived Transactions, using the short-lived transactions as its building blocks.

Short lived transactions keep all the necessary information in transient transaction-local storage (typically in memory), until the time they commit, merging the write set with the global state of the application. With Long Lived Transactions, merging this write set in each step is not possible.

Recall the Course creation example. Consider that in the first step a Course is created with only a name and the department it is associated to. Merging the Write Set for this step with the global state would leave the system in an inconsistent state in which the departement is associated to an unfinished course, as this transitory state would be visible to the outside world.

There needs to be a way to persistently store the Write Set of each step, outside the global state of the application so that the changes are not visible to the outside world. Only upon committing the Long Lived Transaction these changes would be merged to the global state, meaning that the writes performed by each step are effectively delayed until the end of the transaction. Note that for correctness purposes, the Read Set of the transaction must also be collected, so that at commit time both Read Set and Write Set can be replicated, taking advantage of the already existing JVSTM support.

As the main feature of the Fenix Framework is providing mechanisms to manipulate transactionally and to persist Domain Objects, perhaps it would be a good approach to use regular Domain Objects to store the required information.

5.2.1 Data Structures

Consider the domain model presented in Figure 5.1 that represents the reification of the necessary data necessary for a transaction. A `TransactionalContext` is the centerpiece of the domain, it represents a

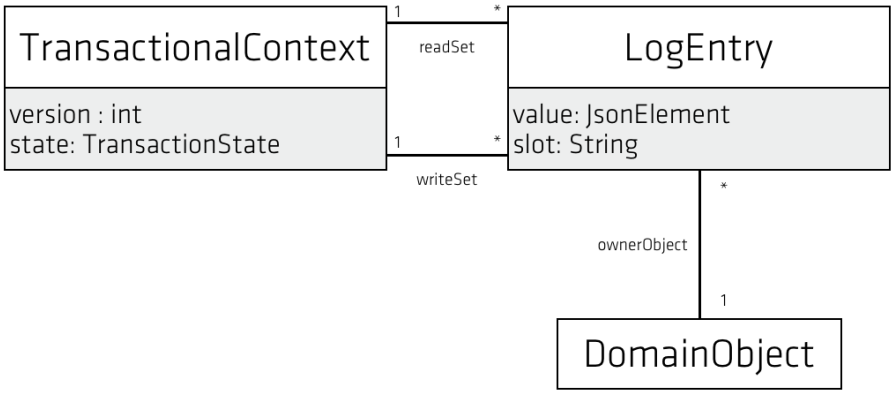

Figure 5.1: Transactional Context's Domain Model

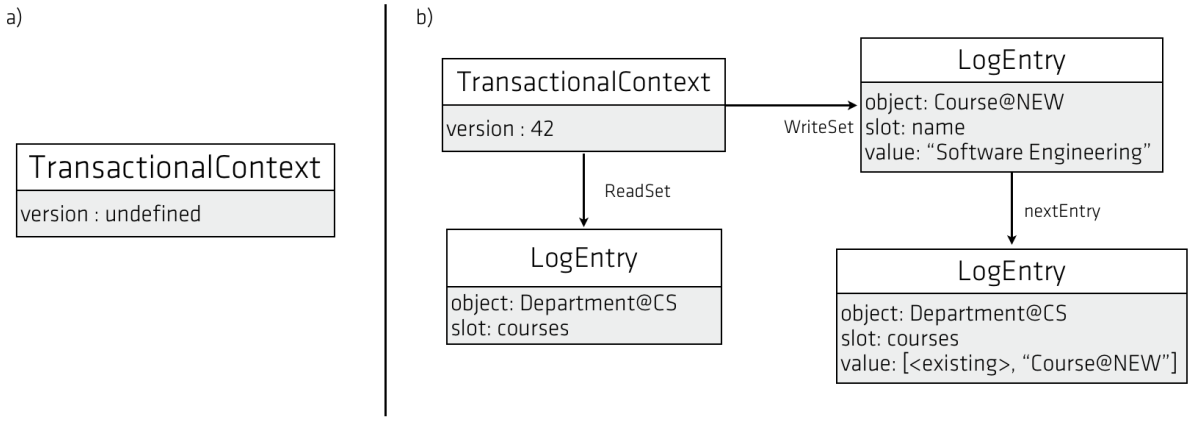

Figure 5.2: Transactional Context instances in a Course creation transaction

Long Lived Transaction, holding together the entire state of the transaction. By keeping the state of the LLT in regular domain objects, we are ensuring that it is stored persistently (or at least as persistent as the rest of the application) as well as transactionally safe. Updates to the context are performed using a regular transaction, allowing for multiple users to concurrently run LLT steps (which will simply perform reads and writes to the context).

In this model, the `TransactionalContext` keeps the state of the transaction (whether it is started, committed, aborted or in conflict) and the version marker (corresponding to the "current" version when the first step of the transaction executed).

A `TransactionalContext` has two associated sets of `LogEntries`, one for the Read Set and one for the Write Set. A `LogEntry` represents one read or written slot throughout the transaction, by keeping a reference to the object, slot name, as well as the value that was written (note that the value is only kept if the LogEntry belongs to the Write Set).

Once again recall the Course creation example, in which a Long Lived Transaction is used to perform a multi-step creation of a new Course for a given Department. To perform such operation, a new `TransactionalContext` is created, initially empty (Figure 5.2a), with an undefined version.

In the first step, a new Course named "Software Engineering" is created, and it is added to the CS department. As this step performed a single read and two writes, three `LogEntries` are created: *(1)* In the Read Set representing the reading of the department's Course list, *(2)* In the Write Set representing

```
public void runStep(TransactionalContext context) {
  // IllegalStateException if already within a context
  LongTransaction.setContextForThread(context);
  try {
    transactionalOperation();
  } finally {
    LongTransaction.removeContextFromThread();
  }
}

@Atomic
public void transactionalOperation() {
  (...)
}
```

Listing 5.1: Example of TransactionalContext usage

the updaded Course list for the CS department, *(3)* and in the Write Set representing the name slot for the newly created course. Note that as this step was the first, the version of the `TransactionalContext` is now defined as the "current" version. Figure 5.2b shows the contents of the context after the execution of the first step.

5.2.2 Transaction Isolation

Having the necessary data structures laid out is crucial for a proper implementation of Long Lived Transactions, but it is just the beginning. Whereas these structures are agnostic to the specific backend, the backend must be able to recognise when a transaction is executing within the context of a Long Lived Transaction, so that reads and writes are isolated from the global state of the application (otherwise the whole world could see the intermediate values).

In the Fenix Framework, transactions are bound to a specific thread, allowing for multi-threaded applications to execute multiple concurrent transactions, each one in its own thread. As such, to run a transaction in the context of a Long Lived Transaction, one must first bind the context to the current thread. This way, when beginning a new Transaction, the backend will check for the presence of a `TransactionalContext`, to determine whether to start a regular transaction or a LLT step. Listing 5.1 shows the programmer API for binding a `TransactionalContext` to a given thread. This way, the programmer is free to run any piece of transactional code as a step of a Long Lived Transaction.

Transactions occurring within the context of a Long Lived Transaction must be aware of that fact, as this means that the semantics of domain getters and setters is changed.

When reading the value of a slot within a `TransactionalContext`, it is the responsibility of the backend to check whether the slot is in the Write Set of transaction (so that written values can be later read) and if it is not, read the value of the slot in the correct version (the version recorded in the context). Slot reads are recorded as a `LogEntry` in the Read Set.

When writing the value of a slot, the written value is stored as a `LogEntry` in the Write Set, so that it can later be retrieved by read operations and used to update the state of other domain objects when committing the transaction.

Backends will typically intercept reads and writes to the domain, and use their regular methods for accessing the underlying transactional context. As such, in the end of a step, only slots belonging to the

`TransactionalContext` and its `LogEntries` are written to persistent support.

5.2.3 Committing the Long Lived Transaction

So far we have seen what data is stored in a `TransactionalContext` and how it is populated. Now we shall look at what happens when the Long Lived Transaction finishes, and the context is committed.

Just like in a regular transaction, a Long Lived Transaction must be atomic and consistent, meaning that its effects must appear to have occurred at a single well-defined point in time. To accomplish this, all elements of the Read Set must be validated to be in the same version, thus ensuring that all writes were performed based on fresh data (more details on how this is accomplished in Section 5.3.2). If the validation step is successful, all the written data must be merged into the global state of the application, by iterating over all `LogEntries` in the Write Set, and writing the recorded value to the correct slot.

To ensure the correctness of the commit operation, both validation and merge are performed within a regular transaction, in a backend-specific manner (as only the backend knows how to write to an arbitrary slot and to check if the read value is still valid). Any conflicts on this operation, such as multiple concurrent commit operations, or writes to validated slots will cause the commit transaction itself to abort and restart.

Programmers can also manually rollback the Long Lived Transaction. In this situation, all the information stored in the corresponding `TransactionalContext` is deleted. As the Reads and Writes performed by the transaction are stored exclusively in the context, no further action is required.

5.3 Implementation

The previous section described a solution to ease the development of Long Lived Transaction. This section describes how that solution was implemented on the Fenix Framework, using the JVSTM as the transactional support provider.

The implementation is divided in two parts:

1. The **long-tx-api** module, which contains the domain specification of the `TransactionalContext` and `LogEntries`, as well as the API available to the programmer.

2. A **JVSTM-based** implementation of Long Lived Transactions.

The goal of this division is twofold: to allow alternative implementations on top of non-JVSTM backends, and to hide internal implementation details from the programmer (who should not depend on backend code).

5.3.1 API

The **long-tx-api** module is pretty straightforward. It contains the domain definition described in Figure 5.1. The domain is public API, so that Long Lived Transactions can be associated with any programmer-defined object (e.g., to a user, to a group, a process etc). This design decision allows for a simple solution, as cross-cutting concerns such as security and sharing are abstract, and also gives the programmer more flexibility.

It is the programmer's responsibility to instantiate a new TransactionalContext every time a new Long Lived Transaction is to be started. With the context in hand, the programmer simply needs to bind it to the thread running the step. Listing 5.1 shows the code necessary to bind the context to a thread.

Using only this simple API, the programmer is able to easily code features that benefit from Long Lived Transactions with little effort.

```
TransactionalContext context =
    LongTransactionSupport.getContextForThread();
if (context != null) {
  // Begin a new Top Level Write-Transaction
  JvstmInFenixTransaction underlying =
      Transaction.begin(false);

  LLTStepTransaction longTxStep =
      new LLTStepTransaction (context, underlying);

  longTxStep.start();
}

(...)

// In the beginning of the new transaction
if (context.getVersion() == null) {
  context.setVersion(Transaction.current().getNumber());
}
```

Listing 5.2: Beginning a new Long Lived Transaction step

5.3.2 JVSTM implementation

In the proposed architecture, several features are required to be provided by the backend:

- **Context Detection** The backend detects the presence of a `TransactionalContext` in the current thread, and begins a context-aware transaction.
- **Intercepting reads/writes** Ensure that reads and writes performed in LLT steps are not performed to the global state of the application.
- **Context commit** Once the Long Lived Transaction is finished, merge its changes with the global state.

Context Detection

When beginning a transaction in `jvstm-common`, the backend checks for the presence of a `Transactional Context` bound to the current thread, and if one is present, the following happens:

1. A regular transaction is started. This transaction will be used to access the context and previous versions of VBoxes (for when a read is made and the context cannot provide a value).
2. A nested `LLTStepTransaction` is started. This will be the active transaction, and route reads and writes to the corresponding `TransactionalContext`.
3. In case this is the first step of the LLT, it is necessary to define its version. To ensure that every other step of the LLT has the same view of the world as this one, mark the LLT's version as the version of the current transaction.

This algorithm is shown in Listing 5.2.

The main reason to use a Nested transaction is to allow portability across concrete persistence implementations, as the `LLTStepTransaction` will be agnostic to the specific underlying transaction (provided it fulfils the required API). A `LLTStepTransaction` holds a reference to the `TransactionalContext` backing it, so that it can be used to aid in reading and writing.

Intercepting Reads and Writes

As in the JVSTM backend each domain object slot is backed by a VBox, intercepting reads and writes to slots can be easily done. The implementation of the get/set methods in a VBox simply delegates the read/write to the current transaction.

As such, the `LLTStepTransaction` was introduced. Its goal is to intercept the reading and writing of boxes, ensuring that such reads and writes are both isolated from the outside world and collected into the `TransactionalContext`.

Just like in regular transactions, writing a VBox will simply keep the written value in the transaction's Write Set. This Write Set will be processed when committing the step.

The algoritm for reading a VBox within a `LLTStepTransaction` is shown in Listing 5.3. It consists of the following steps:

1. If the VBox was previously written in this step, return the written value. This is accomplished by looking up the value in the step's own transient Write Set.

2. In case the VBox being read is owned by either a `TransactionalContext` or a `LogEntry`, the read is delegated to the parent transaction in the current version (so that always the latest version is read). Such reads will typically be nested within reads of other boxes, when the context's Write Set is being searched.

3. If the VBox was written in a previous step of the Long Lived Transaction, return the previously written value, obtained from a `LogEntry` in the Write Set of the LLT.

4. Else, delegate the read to the underlying transaction, in the same version as the context, and add the VBox to the current transaction's Read Set.

As domain slots can hold virtually any value, it is not possible to store the value directly in a single statically typed Fenix Framework slot. To solve this issue, the values are stored in JSON format, in a single slot on the `LogEntry` class. Recall from Section 4.1.2 that the Fenix Framework provides native support for converting any ValueType to/from JSON. With a little help from the backend-specific Code Generation step, domain objects can now be asked to convert the value of a given slot to/from JSON, making it possible for Long Lived Transactions to get a JSON value from any VBox.

When the transaction finishes, the `LLTStepTransaction` has in its Read and Write Sets the actual VBoxes that were read/written during the transaction. Recall that in each step the Read Set and Write Set of the step must contain only the changes to the context that reflect what has been read and written. As such, when committing the step, the Read Set and Write Set are processed in the following manner (Listing 5.4 shows the detailed algorithm):

1. The parent transaction is set as the current transaction.

2. For each item in the original Write Set, the written value is converted to JSON and it is added to the `TransactionalContext`. As this operation runs within the parent (backend-specific) transaction, the newly created and updated `LogEntries` are added to the Read and Write sets of the parent transaction.

```java
// In the LLTStepTransaction class
public <T> T getBoxValue(VBox<T> vbox) {
  // Check the local Write Set
  T result = getLocalValue(vbox);
  if (result == null) {
    if (vbox.getOwnerObject().
          getClass().isAnnotationPresent(NoLogEntries.class)) {
      // For classes annotated with @NoLogEntries
      // check the parent transaction in the current version
      return parent.getBoxValue(vbox);
    }

    // Check the TransactionalContext
    result = lookupBoxValueInContext(vbox);
    if (result == null) {
      // Read from the parent
      result = parent.getPreviousBoxValue(vbox, version);
      bodiesRead.put(vbox, null);
    }
  }
  return result == NULL_VALUE ? null : result;
}

private <T> T lookupBoxValueInContext(VBox<T> vbox) {
  JVSTMDomainObject owner = vbox.getOwnerObject();
  String slotName = vbox.getSlotName();

  LogEntry writeSetEntry = context.getWriteSet();

  while (writeSetEntry != null) {
    if (writeSetEntry.getDomainObject().equals(owner) &&
        writeSetEntry.getSlotName().equals(slotName)) {
      String contents = writeSetEntry.getContents();
      return owner.getValueFromJSON(slotName, contents);
    }
    writeSetEntry = writeSetEntry.getNextEntry();
  }
  return null;
}
```

Listing 5.3: Reading a VBox within a LLTStepTransaction

```java
protected void commit() {
  // Go through the ReadSet and the WriteSet
  // No consistency validations are performed here, as the
  // purpose of this transaction is to manipulate the contents
  // of the read/write set.

  // Ensure the LogEntry creation is performed within
  // the parent transaction
  Transaction.setCurrent(parent);

  for (Entry<VBox, Object> entry : boxesWritten.entrySet())
  {
    VBox<?> vbox = (VBox<?>) entry.getKey();

    JVSTMDomainObject owner = vbox.getOwnerObject();

    // Get the JSON value with a little help
    // from Generated Code
    String json =
          owner.getJSONStringForSlot(vbox.getSlotName(),
                                     entry.getValue());

    // Create or update the LogEntry
    context.addWriteSetEntry(owner, vbox.getSlotName(), json);
  }

  for (VBox vbox : bodiesRead.keySet()) {
    // Create a new LogEntry if this slot wasn't
    // read in a previous step
    context.addReadSetEntry(vbox.getOwnerObject(),
                            vbox.getSlotName());
  }

  // At this point, the parent transaction has the
  // TransactionalContext and the updated LogEntries
  // in its Read and Write Set
}
```

Listing 5.4: Algorithm for committing a Long Lived Transaction's step

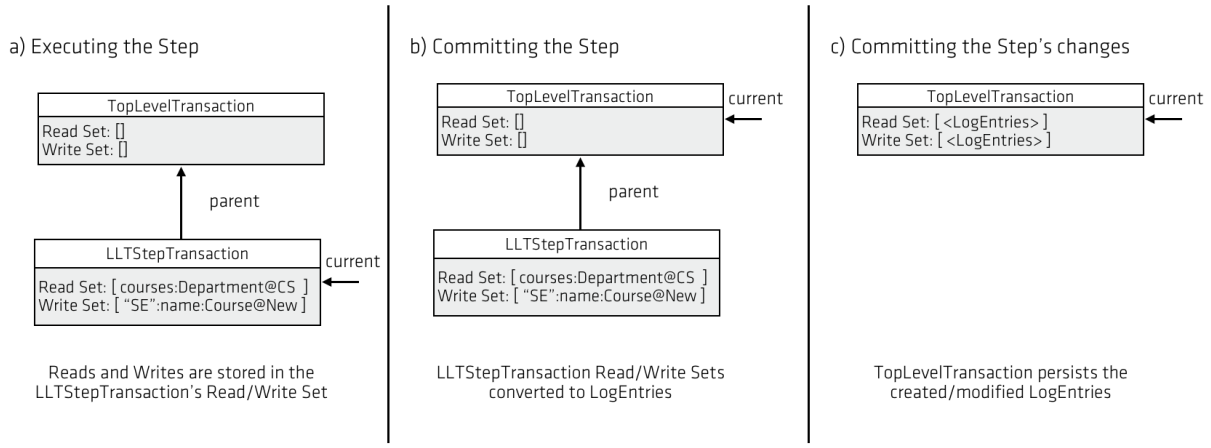

Figure 5.3: LLTTransactionStep lifecycle

3. The same is done to the Read Set.

Once the nested transaction is committed, the parent (backend-specific) transaction will ensure that the updated `TransactionalContext` is stored in persistent support. Note that the only objects written in the parent transaction will be the ones representing the context and its `LogEntries`.

Figure 5.3 recaps the lifecycle of a `LLTStepTransaction`. Figure 5.3a shows that as the step is running, the *current* transaction is the `LLTStepTransaction`, as such, reads and writes are kept in its Read/Write Sets. During the commit of the step (5.3b), the *current* transaction is the parent (backend-specific), and the `LLTStepTransaction`'s Read/Write Sets are converted into `LogEntries`, written to the parent's Write Set. Finally, once all the `LogEntries` are created, the parent transaction is committed, persisting the step's changes (5.3c).

Finishing the Long Lived Transaction

Once all steps of the Long Lived Transaction are finished, the LLT must be committed, ensuring that the changes performed in it are visible to the outside world. Much of this process is heavily dependent on the backend as it involves direct access to the underlying data structures.

The commit process of a Long Lived Transaction occurs within a regular transaction. In this process, the data collected throughout the multiple steps is validated and replayed. Once this transaction commits, all written data will be merged with the global state of the application. The commit algorithm is shown in Listing 5.5.

The first step in committing the LLT is verifying whether all the read data is still valid. As every slot is mapped in a JVSTM VBox, all the boxes corresponding to the read slots must be verified. The process iterates over all read slots, locating the VBox that represents the slot. It then reads the VBox, so that it is added to the Read Set of the current transaction. Then, the latest version of the VBox is compared to context's version, thus ensuring that the read value was the latest. If the box's current version if posterior to the read version, the Long Lived Transaction is aborted.

There is a critical subtlety in this verification process. Modifications to the VBox's underlying data structures are performed only at commit time, inside a commit lock. As the verification process is not run within the commit lock, it is possible for another transaction to concurrently update a VBox after the validation. Preventing incorrect behaviour is quite simple: Just read the box. By doing this, the box will be validated when the transaction commits, ensuring that the verification was correct (i.e. the version read during the verification is still valid).

Consider the following scenario: VBox A was read in a Long Lived Transaction in version 1 and not changed afterwards. When the Long Lived Transaction commits (in a transaction X), A will be validated, its current version (1) compared to the version of the transaction (1). It passes the test and validation succeeds, proceeding with the commit. Concurrently (after the validation, before X commits), another transaction writes to A and commits, increasing its version to (X+1). When X attempts to commit, its read set (which mirrors the Read Set of the Long Lived Transaction) is validated. As A was concurrently written, X will be restarted, and the version verification will fail, marking the Long Lived Transaction as conflicting.

Once the Read Set of the Long Lived Transaction is validated, the Write Set must be merged into the global state of the application. This process iterates over the written slots, and for each slot: (1) Locates the VBox representing it, (2) Converts the JSON value to the concrete value, and (c) Writes the value to the VBox.

After the merge process is finished, the committing transaction has a full mirror of the LLT's Read and Write Sets, and once it commits, every change in the Write Set is available to the outside world.

5.4 Validation

This chapter proposed a solution that allows programmers to take advantage of Long Lived Transactions with little effort and no code modifications. I shall now validate the proposed solution, both in terms of correctness and ease of use.

5.4.1 Correctness

One of the greatest challenges when implementing Long Lived Transactions is ensuring that the solution provides the same correctness guarantees as regular transactions. I will now demonstrate that the proposed solution provides the same correctness guarantees as regular transactions.

Transactions in the JVSTM satisfy the Opacity correctness properties (refer to Section 3.4.1). When integrated with the Fenix Framework, transactions may also provide the Durability property, depending on whether the concrete backend supports persistence.

The proposed solution also all the ACID properties for Long Lived Transactions, as well as the Opacity correctness property.

- **Atomicity** is provided as all the writes performed during the LLT are only written to the global state of the application when the it commits, which is performed by a regular transaction.
- **Consistency** is provided in the same way as a regular transaction, by ensuring that every read is still valid at commit time.
- **Isolation** is provided as no data is ever written to the global context until the LLT is finished, and data reads are isolated.
- **Durability** is provided if the underlying backend supports data persistence.

Throughout the execution of the Long Lived Transaction, written data is collected and stored inside the `TransactionalContext`. Once the transaction is finished, written data is **atomically** written to the global context, using a regular transaction which simply reads data from one domain object (the context) to be written in another (the actual objects written during the transaction).

Long Lived Transactions are **consistent**. Similarly to regular transactions, a LLT only commits if all the data read during the transaction is still valid at commit time. This is done by comparing the version of the LLT against the version of every box in the Read Set, just like a regular transaction would do.

```
void commitContext(TransactionalContext context) {
  // Don't do anything if nothing was written
  if (context.getWriteSet() != null) {
    if (!validateContext(context)) {
      throw new TransactionError();
    }
    mergeContext(context);
  }
}

boolean validateContext(TransactionalContext context) {
  int version = context.getVersion();
  LogEntry readSetEntry = context.getReadSet();
  while (readSetEntry != null) {
    JVSTMDomainObject owner = readSetEntry.getDomainObject();

    VBox box = owner.getBoxForSlot(readSetEntry.getSlotName());

    // Read the box to ensure it goes in the read set of
    // the current transaction. This way, if a concurrent
    // transaction writes to this box between this version
    // check and the commit of this transaction, it will
    // be restarted
    box.get();

    if (box.body.version > version) {
      // This means that the box was written after the LLT
      // started, meaning it has to be aborted.
      return false;
    }
    readSetEntry = readSetEntry.getNextEntry();
  }
  return true;
}

void mergeContext(TransactionalContext context) {
  LogEntry entry = context.getWriteSet();
  while (entry != null) {
    JVSTMDomainObject owner = entry.getDomainObject();
    VBox box = owner.getBoxForSlot(entry.getSlotName());
    box.put(owner.
                getValueFromJSON(writeSetEntry.getSlotName(),
                                 writeSetEntry.getContents()));
    entry = entry.getNextEntry();
  }
}
```

Listing 5.5: Implementation of the Long Lived Transaction commit operation

Isolation is perhaps the hardest property to demonstrate. When performing a read operation within a Long Lived Transaction, the system ensures that the returned value will be the one at the moment the first step of the LLT executed (i.e. the value consistent with the transaction's version), just like it happens with regular transactions. As for writes, similarly to what happens with regular transactions, written values are kept in transaction-local storage (i.e. in `LogEntries`) and as such, can only be accessed by the transaction itself.

Durability is an "optional" property, as not all JVSTM-based backends provide persistent support. Those that do however, ensure that the effects of Long Lived Transactions are durable, as it works as if writes occurred in a single regular transaction (which already provides the Durability property).

I have shown that Long Lived Transactions provide the same correctness guarantees as regular transactions. There is, however, one key piece missing. Correctness is only guaranteed provided that all the information about reads and writes is collected throughout the transaction, and is available at commit time.

Recall from the previous sections that in each step of the Long Lived Transaction, every read and write operation is intercepted, and for each operation a record is created. Such records are kept in `LogEntries`, which in turn are writting using regular transactions. As such, once the LLT is committing, it is able to read all the recorded data, and use it for all the necessary verifications and data updates.

5.4.2 Ease of Use

The primary goal of this work is to ease the development of Long Lived Transactions, and as such, it is rather important to provide a simple and concise API.

The proposed solution fares well in that regard, as it allows existing code to be adapted to use Long Lived Transactions with no modifications. It is possible to program the business logic of your whole application using regular transactions, and with a simple wrapper add Long Lived Transaction support.

Consider a Web Application wishing to share with its users the benefits of Long Lived Transactions, by allowing each individual user to keep a series of Long Operations. Support for this feature could be added at an infrastructural level, by providing the user with a UI to manage his Operations (creating, committing, enabling, etc). Creating and committing the operation would imply simple domain object manipulation (in particular creating and committing a `TransactionalContext`).

Making every action performed by the user as a step of the Long Lived Transaction would be as simple as keeping the context in session-local storage, and binding it before every transaction start (using a Web Filter or similar). Listing 5.7 shows a possible implementation of a Web Filter to accomplish the desired behaviour, and Listing 5.6 shows how the session binding can be performed, using a industry-standard web framework.

With this architecture, it is possible to make every operation in the application part of a Long Lived Transaction, without the need to change existing code, or changing the methodology used to develop new functionalities.

```java
@RequestMapping("/activate/{oid}")
public void activate(@PathParam String oid) {
  // Get the TransactionalContext from the given OID
  TransactionalContext ctx = getAndCheckContext(oid);
  request.getSession().setAttribute(LONG_TX_SESSION_PARAM,ctx);
}

@RequestMapping("/deactivate")
public void deactivate() {
  request.getSession().removeAttribute(LONG_TX_SESSION_PARAM);
}
```

Listing 5.6: Controller methods to associate a TransactionalContext with a user's session

```java
public void doFilter(ServletRequest request,
                     ServletResponse response,
                     FilterChain chain) {
  HttpSession session = request.getSession();
  TransactionalContext context =
                  session.getAttribute(LONG_TX_SESSION_PARAM);
  if (context == null) {
    chain.doFilter(request, response);
  } else {
    LongTransaction.setContextForThread(context);
    try {
      chain.doFilter(request, response);
    } finally {
      LongTransaction.removeContextFromThread();
    }
  }
}
```

Listing 5.7: Web Filter to wrap every transaction in a TransactionalContext

Chapter 6

Optimization

The solution presented in Chapter 5, despite providing correctness guarantees and ease of use, performs poorly even in trivial test cases. This chapter describes a series of optimizations made to keep the overhead as low as possible, without sacrificing correctness.

6.1 Example Application

To perform accurate measurements, I developed a sample banking application. In this application's domain (see Listing 6.1) there is a central Bank with Customers, which in turn have their accounts. For each account the application keeps its balance, and there is a method to show a customer's balance (by summing the balance of all his accounts). Every time a banking transaction (transfer, deposit, etc.) is performed, a new `TransactionRecord` is created, containing a timestamp, the origin and destination accounts, as well as the amount transfered.

In this scenario, a Business Transaction will consist of several banking transactions, spread throughout a series of Business Operations. This is the perfect candidate for a Long Lived Transaction.

Using this banking domain, a sample application was developed. In this fictitious scenario, the bank starts with a certain number of customers (this number is configurable so that different parameters may be measured) and a configurable number of operations. Within each operation, new customers and accounts are created, money is shuffled between all the accounts of every customer (creating a new TransactionRecord for each transaction), and the total balance is calculated. This means that in each step: (1) every object in the system is read, (2) many objects are written and (3) many new objects are created.

Listing 6.3 shows the implementation of the described Business Operation. Note the `@Atomic` annotation in the `doOperation` method. This ensures that each operation runs in its own transaction.

In the *regular* scenario, each business operation will run within a regular JVSTM transaction (and thus every intermediate state is visible to the outside world), whereas in the *Long Lived* scenario the Business Transaction is mapped into a Long Lived Transaction, and each individual operation runs within an LLT step (meaning that intermediate state will not be visible to the outside).

With the support presented in previous sections, the programmer simply writes the application's code as he would without considering Long Lived Transactions. No changes to the data structures and business logic are required. Listing 6.2 shows the two methods used to invoke both versions under test. Notice that they both look very similar as they both invoke the `doScript` method, the only difference is that in the LLT version, a new `TransactionalContext` is created and bound to the current thread.

```
class Bank;
class Customer;

class Account {
  Money balance;
  DateTime opened;
}

class TransactionRecord {
  DateTime when;
  Money amount;
}

relation BankHasCustomers {
  Customer playsRole customer {
    multiplicity *;
  }
  Bank playsRole bank;
}

relation CustomerHasAccounts {
  Customer playsRole owner {
    multiplicity *;
  }
  Account playsRole account {
    multiplicity *;
  }
}

relation TransactionRecordHasFrom {
  Account playsRole from;
  TransactionRecord playsRole outgoingTransaction {
    multiplicity *;
  }
}

relation TransactionRecordHasTo {
  Account playsRole to;
  TransactionRecord playsRole incomingTransaction {
    multiplicity *;
  }
}
```

Listing 6.1: Domain Model for the Banking Application

```
public void doRegular() {
  doScript("regular");
}

public void doLongTx() {
  TransactionalContext context = createContext();
  LongTransaction.setContextForThread(context);
  doScript("long");
  LongTransaction.removeContextFromThread();
}

public void doScript(String scriptName) {
  for(int i = 0; i < NUMBER_OF_OPERATIONS; i++) {
    doOperation();
  }
}
```

Listing 6.2: Invoking the business operation

6.2 Initial Performance Analysis

Figure 6.1 shows the running time for a varying number of operations of the application presented above. As expected, the running time grows linearly as the number of operations increases.

When looking at the Long Lived version (where every operation is a step of one large Long Lived Transaction), performance quickly degrades as the number of operations grows. The main reason for this performance hit is that every box read will cause many other boxes to be read (instead of just reading the box directly, all LogEntries representing the write set must be traversed, meaning that many more boxes are read along the way, and many more boxes will be read as the transaction grows in size). Recall from Chapter 4 that in the JVSTM using a multi-box layout, each slot of an object is kept in a separate box, and as such reading a single Log Entry will cause at least 3 boxes to be read: the slot name, the target object, and the next box in the list.

Figure 6.3 shows that in a regular transaction, the number of boxes read grows slowly as with the number of operations. A similar behaviour is observed with Long Lived Transactions: Figure 6.4 shows that the number of boxes read grows following the same pattern as the running time.

As reading boxes is one of the major sources of the poor performance results of this implementation, I made several improvements regarding the number of boxes read and boxes written. The next sections present those optimizations.

6.3 Read-Set differentiation

In the initial implementation, `LogEntries` were used to reify both the Read Set and the Write Set.

Recalling Figure 5.1, `LogEntries` store a reference to the DomainObject they refer to, the name of the slot, and the slot's value. Using the same objects to represent both sets proved to be quite expensive, as the Read Set only cares about which slots were read, completely ignoring its value.

By analysing the nature of the Read Set, we may conclude that it is only necessary to store the pairs [DomainObject, Slot] read by the transaction (the actual version read is not relevant, as it will always be coherent with the transaction's version).

```java
// Bank class
public Money getTotalMoney() {
  Money money = Money.zero();
  for (Customer customer : getCustomerSet())
    money = money.add(customer.getTotalMoney());
  return money;
}

// Customer class
public Money getTotalMoney() {
  Money money = Money.zero();
  for (Account account : getAccountSet())
    money = money.add(account.getBalance());
  return money;
}

public void shuffle() {
  Account firstAccount = getRandomAccount();
  for (Account account : getAccountSet()) {
    if (!firstAccount.equals(account)) {
      account.transfer(account.getBalance(), firstAccount);
    }
  }
}

// Account class
public void transfer(Money amount, Account to) {
  this.withdraw(amount);
  to.deposit(amount);
  new TransactionRecord(from, to, amount);
}

// Benchmark class
@Atomic(mode = TxMode.WRITE)
public void doOperation() {
  createCustomers();
  logger.info("Bank has {} money.", bank.getTotalMoney());
  shuffleMoney();
  shuffleMoney();
  createCustomers();
  shuffleMoney();
  logger.info("Bank has {} money.", bank.getTotalMoney());
}

private void createCustomers() {
  for (int i = 0; i < 5; i++) {
    Customer customer = new Customer(generateRandomName());
    customer.addAccount(new Account(200d));
    customer.addAccount(new Account(100d));
  }
}

private void shuffleMoney() {
  for (Customer customer : bank.getCustomerSet())
    customer.shuffle();
}
```

Listing 6.3: Code for the Business Operation

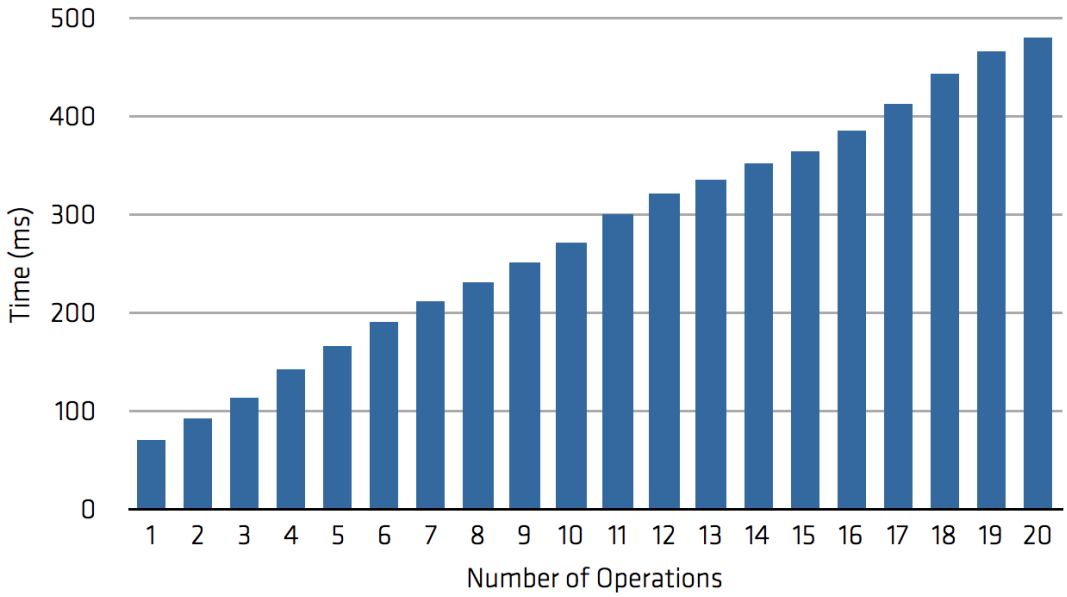

Figure 6.1: Running time with regular transactions

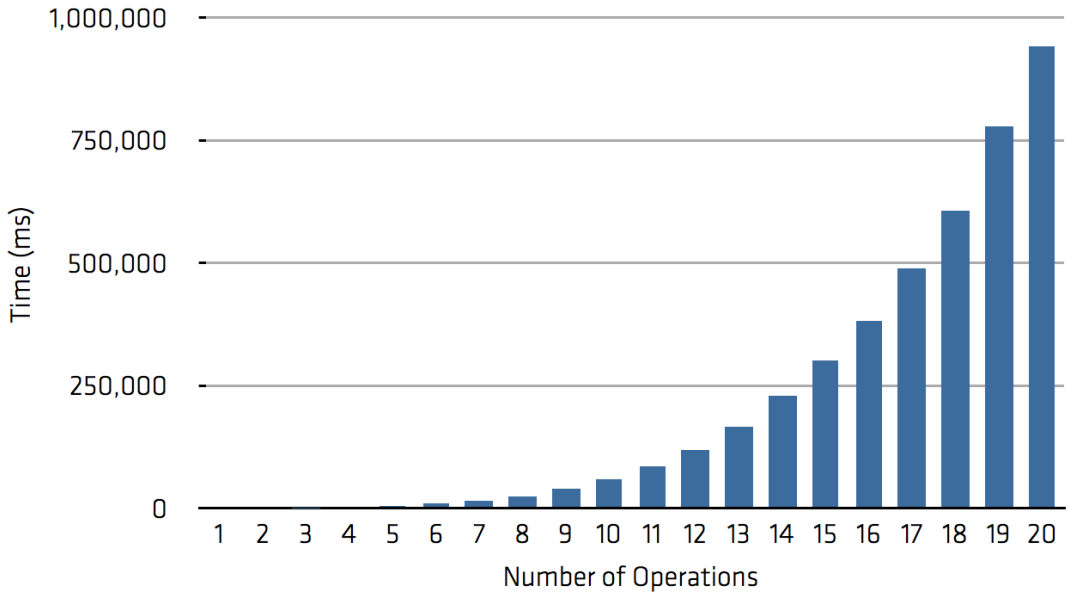

Figure 6.2: Running time with Long Lived Transaction steps

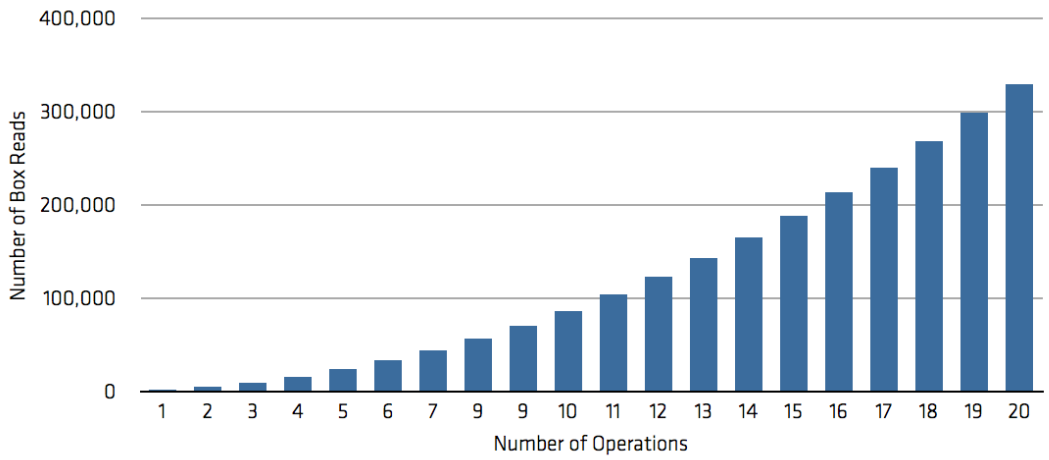

Figure 6.3: Number of boxes read on a regular transaction

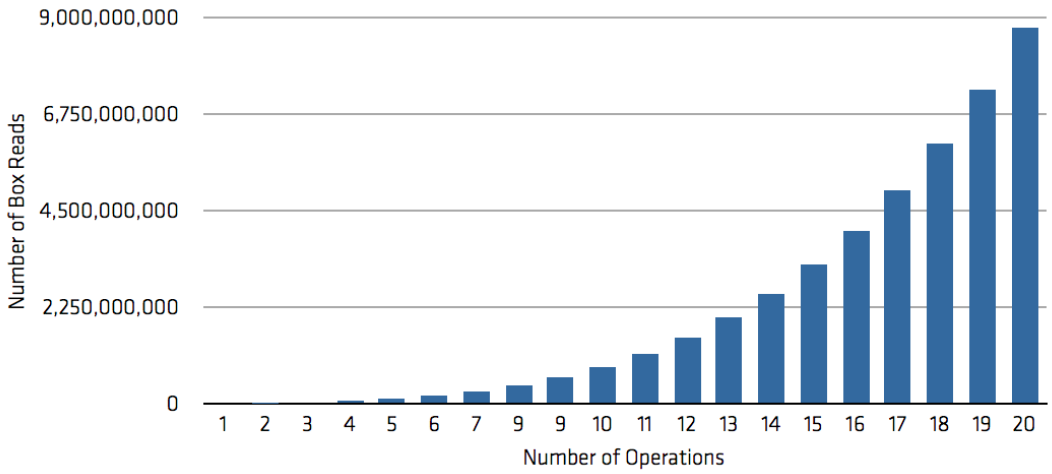

Figure 6.4: Number of boxes read using Long Lived Transaction steps

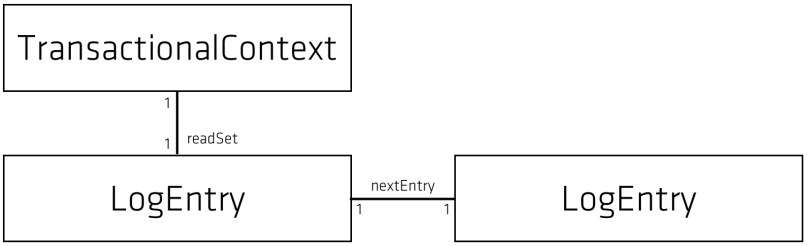

Figure 6.5: LogEntry linked lists

As such, the Read Set has been replaced by an immutable ValueType,[1] containing a set of DomainSlotKey's (an immutable, lightweight object representing the pair [DomainObject, Slot]), stored directly into the `TransactionalContext`.

This optimization greatly reduced the space required by the Transaction (both in-memory and persistently), as the representation of the Read-Set became more compact (a single slot in an object vs several objects).

The commit time for the various steps of the Transaction also improved, as the lookups/insertions of entries in the Read Set are done entirely in memory, without the need to traverse (and potentially reload a large object graph).

6.4 Using BPlusTrees to hold LogEntries

As described in Chapter 4, the Fenix Framework uses BPlusTrees and other collections to implement to-many relations. These collections are transparently handled by the Framework, and are implemented using regular Domain Objects (such as Leaf Nodes and Inner Nodes). As keeping track of changes in relations is a requirement for our implementation, it is critical that changes to BPlusTrees are correctly tracked.

This posed a great issue, as conceptually one `TransactionalContext` has many `LogEntries` in both its Read and Write Sets. If theses relations were to be implemented using regular one-to-many relations, a BPlusTree would be generated by the Framework. But as BPlusTrees must be tracked by the `TransactionalContext`, they could not be used to implement this relation.

The initial approach to this problem consisted on implementing the relation using a Linked List, in which a LogEntry would be directly connected to the next one in the list, keeping the list sorted by insertion order. Figure 6.5 shows how this was designed. This approach proved to be quite inefficient, as lookups in the Write Set were *O(n)* in the number of written objects, making it impractical, as every `getBoxValue` operation required potentially traversing the whole list.

To solve this issue, a specialised `WriteSetBPlusTree` was developed. The major difference between a `WriteSetBPlusTree` and a regular BPlusTree is that the former is designed to be kept out of the scope of the TransactionalContext, making it possible to use it to implement the one-to-many relation between a TransactionalContext and its LogEntries.

Another advantage of directly using a BPlusTree is that it is possible to take full advantage of all its features. Generically speaking, a BPlusTree is a map between keys and values. When the Fenix Framework generates a BPlusTree to represent a to-many relation, the map contains the pointed objects as values, and the object's identifiers (OID) as keys. In this case however, lookups are not performed by OID, but by DomainSlotKey (DomainObject+slot). As such, a WriteSetBPlusTree will map DomainSlotKeys

[1] ValueTypes are explained in detail in Chapter 4.

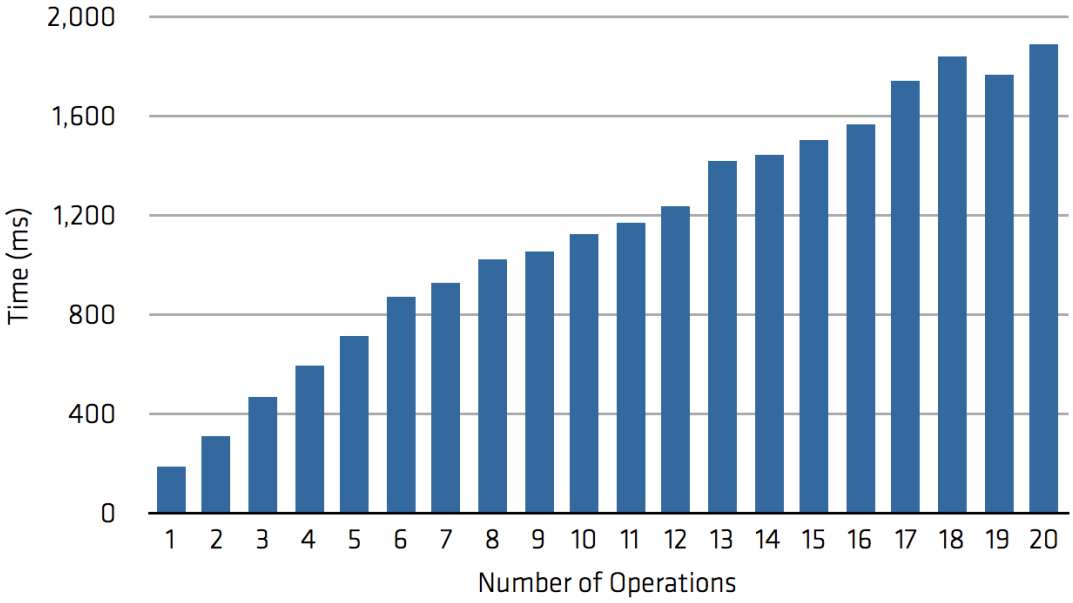

Figure 6.6: Running time using BPlusTrees

to their corresponding LogEntries.

With this approach, looking up a DomainSlotKey in a TransactionalContext means that the BPlusTree is indexed using the DomainSlotKey, and as such, lookup times are now *O(log(n))*.

In Figures 6.6 and 6.7 we can see that the running time using BPlusTrees is closer to the regular version. However, the growth rate (while linear) is still bigger than the original, with the benchmark taking on average 4.4x to complete. One of the big contributors for this speed improvement seems to be the number of Box Reads that happens throughout the transaction. Whereas in the original implementation this number reached over 8.7B, it is now reduced to about 2.1M.

6.5 Removing LogEntries

With the relation between the `TransactionalContext` and the `LogEntries` implemented using a BPlus-Tree, another issue arisen.

Despite having great lookup times, the commit of a Long Transactions's step was greatly affected. Whereas inserting elements in a Linked List is *O(1)*, the insertion in the BPlusTree was still painfully slow.

This is because the Fenix Framework requires that ValueTypes (such as the ones used to back the BPlusTree) are immutable combined with the fact that the BPlusTree provides no API for batch insertion, meaning that for each of the elements written within a given step, a new insert was performed, and the backing TreeMaps were duplicated over and over.

The approach to solve this issue, was to use a solution similar to the one used for the Read-Set: create an immutable ValueType, containing the mapping between all written slots and their respective values.

With this change, `LogEntries` were completely taken out of the picture, as the only extra piece of information they provided was the JSON contents of the slot, which could be embedded directly into the WriteSet object.

The issue with this approach is that, due to the immutability requirement, every time a batch of entries was inserted, the whole Map had to be duplicated, which was rather wasteful both in terms of

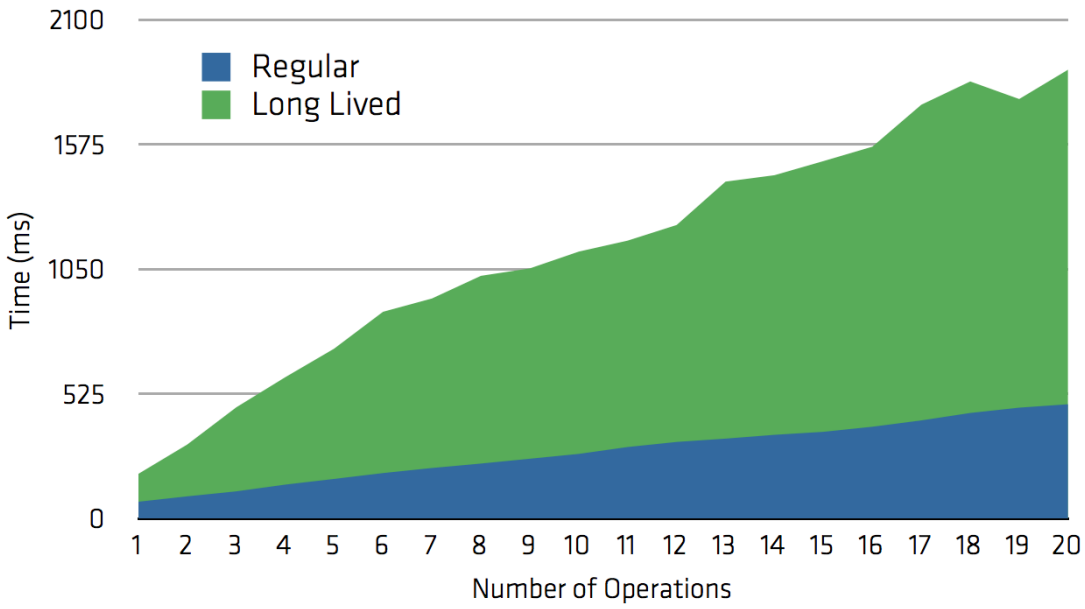

Figure 6.7: Running time comparison using BPlusTrees

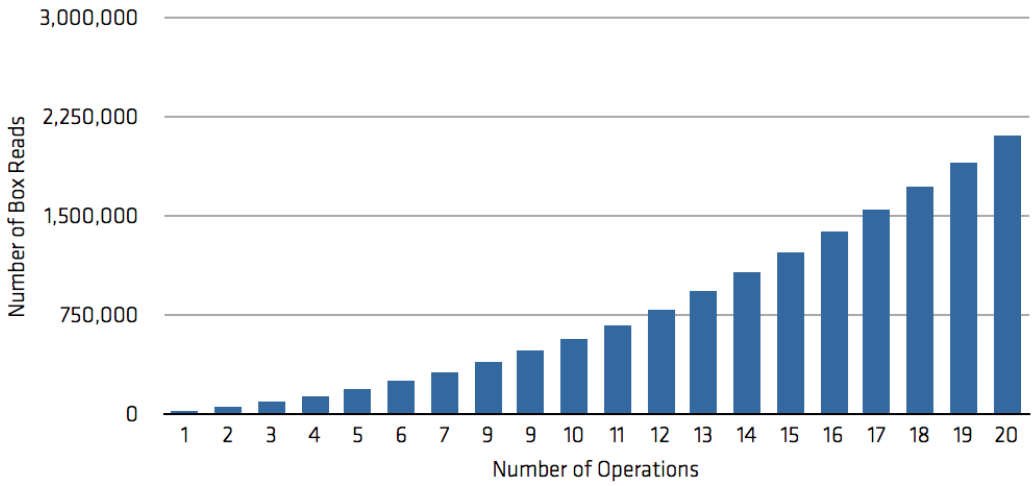

Figure 6.8: Box reads using BPlusTrees

time and allocated memory. So, instead of duplicating the whole Map, the WriteSet is actually a Linked-List of Maps, containing one node per transaction step. This means that the performance of lookups is now *O(s*log(n))*, n being the average size of each step, and s the number of steps (in which something is written) of the transaction.

There is, however, a tradeoff. Transactions with a small number of large steps (which is the typical use case) are greatly improved (as lookups in an in-memory hash map are fast), however transactions with a large number of small steps take a major performance hit, as a new node is created in every step with small amounts of information.

As such, a new node is only created if the current node has size above a certain (user-defined) threshold. This threshold defines whether it is more cost-effective to create a new node (which is practically instant on insertion but makes lookups more expensive) or replace the current one (thus duplicating the map). The user can set the threshold to be lower or higher according to the usage patterns of the application.

Applications with a large number of small steps benefit from a large threshold, as insertions will be quite cheap and have little to no impact on lookups. On the other hand, applications with a small number of large steps should set the threshold lower than the average size of a step, so that there is no need to duplicate a potentially large map.

6.6 Final Performance Analysis

After applying all the optimizations described throughout this chapter, the solution provides quite competitive results. Figure 6.9 shows the running time on this implementation. The growth rate is now very similar to the implementation using regular transactions. Figure 6.10 shows a runtime comparison between using regular transactions and the final implementation of Long Lived Transactions: The slow down is now under 40%.

The number of Box Reads is also very close to the regular transaction version, adding very little overhead. As the Write and Read Sets are now represented in specialised data structures, it is no longer necessary to read vast amounts of VBoxes just to determine whether a given element is in one of the sets. Figure 6.11 shows the total number of Box Reads, and Figure 6.12 shows the number of Box Reads of both the Regular and Long Lived Transaction implementations.

There is also one very important measurement that has not been mentioned thus far: The performance impact of the solution in regular transactions. A solution in which Long Lived Transactions are performant at the cost of having expensive regular transactions is not an acceptable one. For the proposed solution, there is close to zero overhead on starting regular transactions. The only performance hit is the added verification upon starting a new transaction, to check whether the current thread if bound to a `TransactionalContext`. Other than that, no changes were made that would affect the performance of regular transactions.

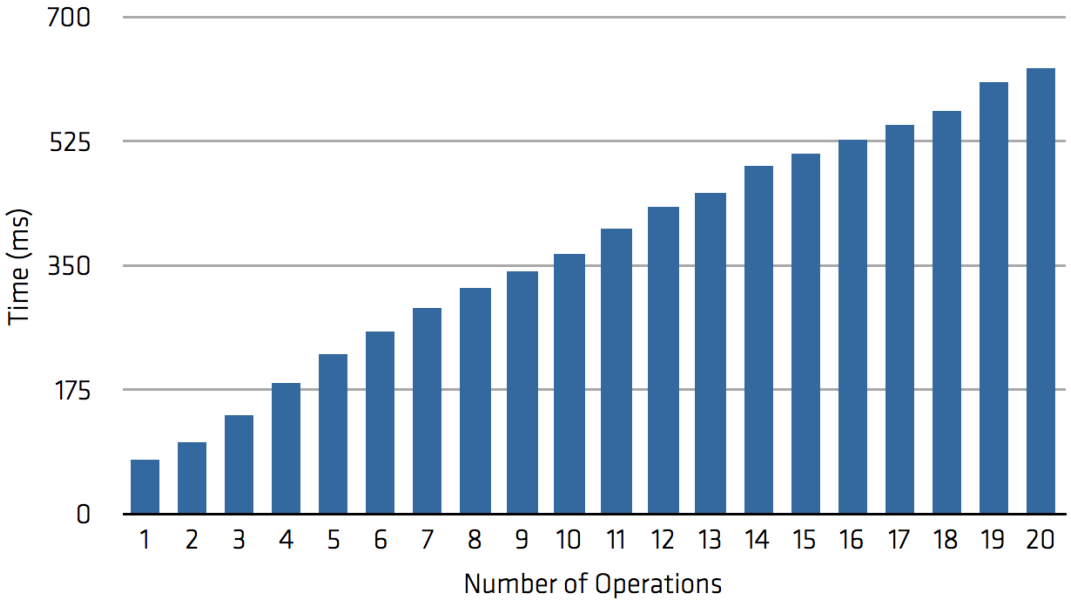

Figure 6.9: Running time in the Final Implementation

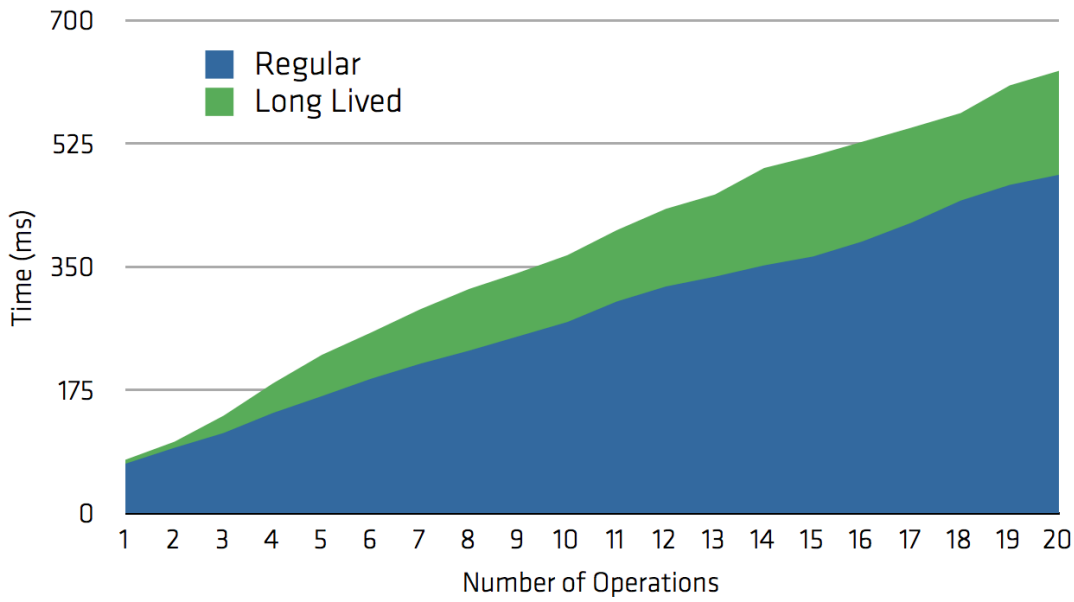

Figure 6.10: Running time comparison for the Final Implementation

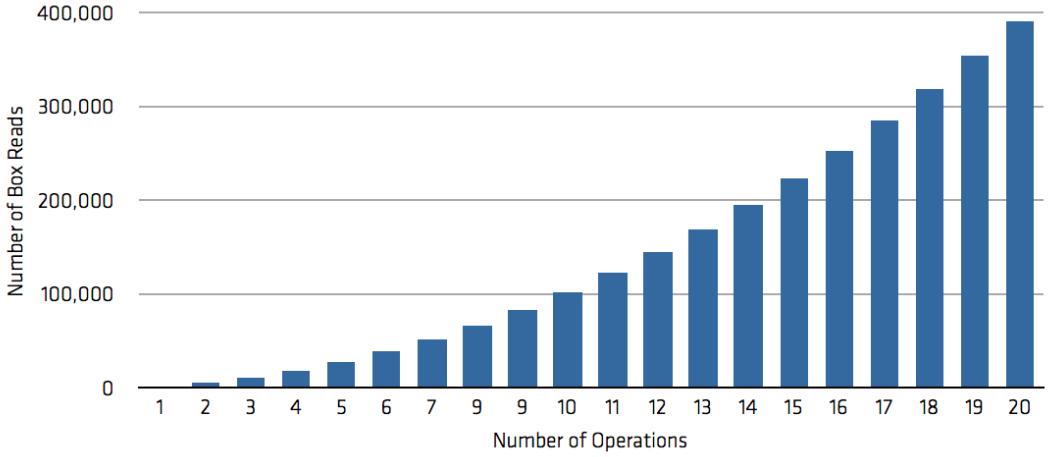

Figure 6.11: Box reads using Long Lived Transactions - Final Version

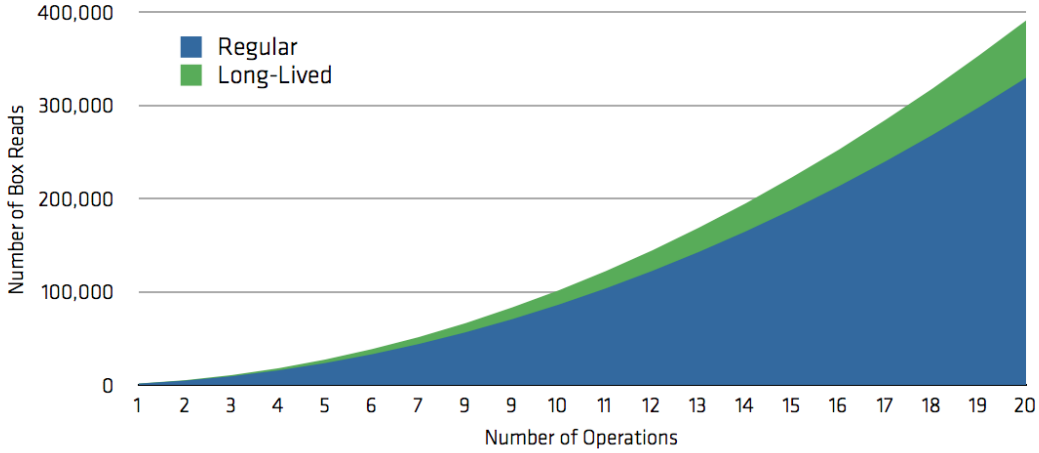

Figure 6.12: Box reads comparison for the Final Implementation

Chapter 7

Future Work

The solution proposed in this document, despite providing great improvements over existing solutions, still presents several challenges. This chapter presents a few modifications which would greatly enhance the solution, making it both more user and developer friendly.

7.1 Extending to other backends

At the time this document was written, only a handful of Fenix Framework backends were able to take advantage of the improvements for Long Lived Transactions.

This is mostly due to the fact that the implementation requires multi-versioning support, meaning that when a new value is written for a given box, its previous value cannot be Garbage Collected.

For in-memory backends this doesn't pose much of a problem, as those backends are typically used for testing and demonstrations, and as such deal with a small data set, making it easy to fit all versions in memory.

Backends with persistence support on the other hand, are greatly affected by this requirement. By storing every version persistently, the size of the database (in whatever technology it may be: SQL, NoSQL, text files, etc) will grow uncontrollably. Consider an application containing a *User* entity, with a slot containing the last time the user logged in. In the extreme case, the whole *User* object would be replicated each time the user logged in to the application.

To cope with this growth rate, a Garbage Collection system would have to be created. Assuming that Long Lived Transactions are the only reason for multi-versioning (which in practice may not be so), such a system could be developed.

The system would have to scan all pending Long Lived Transactions, find the oldest one (i.e. the one with the smallest version number), and scour through the whole database, removing entries older than such a version. A naive approach however may not be sufficient, as stale transactions (i.e. not accessed for a long time) would leave the old versions sitting around for an indefinite amount of time.

An automated approach to take care of stale transactions would then be needed. The solution for this issue is far from trivial, as a Long Lived Transaction represents a user's Unit of Work, and as such cannot be blindly removed. However, as old transactions are likely to be in conflict, it should be safe to discard those, presenting the user with a summary of the changes performed by the Long Lived Transaction. Section 7.2 briefly discusses this in regard to conflict resolution.

However, there is a possible solution that does not require multi-versioning. This solution would restrict the programming model, by forcing the programmer to read every piece of data within the first transaction step, thus ensuring that the read-set is filled within the first step (whose version number is

chosen for the transaction). The Read Set would also have to store the read value (in a way similar to the Write Set), ensuring that all future reads could be satisfied.

7.2 Conflict Management

Perhaps the biggest limitation of this solution is the lack of mechanisms for conflict management. Unlike a regular transaction, a typical Long Lived Transaction has a greater duration and size, thus increasing both the probability of conflits and the amount of lost work in case of a conflict.

The solution proposed so far focuses solely in strict correctness, providing no mechanisms for conflict reduction and handling. This section presents some ideas to solve those issues. It is divided in two main areas: reducing conflicts altogether, and improving the way conflicts are handled.

7.2.1 Reducing Conflicts

Consider the Course creation scenario presented in previous chapters. Now consider that the application shows a dashboard containing relevant information to the user. With the current implementation of Long Lived Transactions, any change to information shown in the dashboard would cause the Course creation to be rolled back, even if such information had nothing to do with the creation of the Course.

This is because rendering the Course creation page would read the domain objects used to render the dashboard, putting them in the read set, despite the fact that they are not directly related to the operation the user is performing.

A possible approach would be to extend the DML to allow the programmer to specify specific classes or slots to have their transactional properties relaxed in Long Lived Transactions. Slot reads for those classes would still be registered in the TransactionalContext, however, having slots of such classes on the Read Set would not forcibly cause the Long Lived Transaction to abort. Using this feature would require special care and consideration from the programmer, as it makes it possible to perform operations based on an old version of a part of the data.

Another possibility is to take advantage of restartable transactions. Restartable transactions were first introduced in the JVSTM in [4], and was explored in more detail in [13].

Restartable transactions are programmer-defined portions of code that are marked as being re-executable within the context of a single transaction. When a conflict is detected due to a piece of data that was only read inside a restartable transaction (i.e. a piece of code marked as restartable), only that smaller transaction is restarted. If the outcome of this transaction (which reads commit-time values) is the same as the original execution, then it is safe to commit the whole transaction.

Consider the naive implementation of a *contains* method in a *Collection* class shown in Listing 7.1.

```
@Restartable
public boolean contains(Object obj) {
  for(Object o : this)
    if (Objects.equals(o, obj)) return true;
  return false;
}
```

Listing 7.1: Restartable *contains* method

A write transaction that at some point during its execution invokes the *contains* method may cause the whole collection to be read. If a concurrent transaction adds a newly created item to the list and commits, that will cause the original transaction to abort, as the read value for collection is no longer the latest value. As the conflicting item was only read inside *contains*, the method is re-executed with the latest data (in which the new element is in the collection). If the outcome of the method is the same, the transaction can now commit safely.

There are many challenges and limitations regarding restartable transactions, and they are addressed in greater detail in [4] and [13].

The concept of Restartable transactions can possibly be extended to support Long Lived Transactions. In a Long Lived Transaction there are clearly defined, individual steps which can be potentially restarted.

Recall the menu items issue presented above. Consider that between two steps of the Long Lived Transaction, one of the menu items has its title changed. This means that at least one step of the Long Lived Transaction has read old data, which had no impact on the outcome of the step.

With a few modifications to the commit algorithm and data structures, it would be possible to restart the conflicting step(s) and check its result against the original state. In this example, as the menu change did not affect the operation, the outcome would be the same, and the Long Lived Transaction could be committed.

7.2.2 Handling Conflicts

In the proposed solution, when a conflict happens the only possible course of action is to rollback, as it is not possible to restart the whole transaction.

If the detected conflict is not recoverable, it is up to the user to determine whether the written data can actually be merged with the global context. As such, handling conflicts in Long Lived Transactions is more of a User Interaction problem than an infrastructural problem.

An interface could be devised, showing the end-user what data was written in the transaction, and the conflicting read data, similar to existing conflict resolution tools (such as Meld and Git's conflict resolution mechanism).

The user could then analyse that data to determine whether the conflicts are relevant according to the business rules, and choose to rollback the Transaction, to merge parts of the written data or to simply commit the operation.

Giving users so much power (as this solution could allow them to deliberately leave the system in an inconsistent state) is dangerous. Should the interface be poorly designed, it would be simple for a user to perform the wrong action. It requires a great deal of analysis, user testing and interaction testing.

7.3 Expanding beyond the Fenix Framework

Many enterprise applications use industry-standard persistence frameworks, such as JPA, Hibernate, Spring Data, iBatis and many others. Despite being possible to use the Fenix Framework to work on top of such frameworks, converting existing applications to use the Fenix Framework core API is a rather difficult and cumbersome task.

The solution proposed in this document is focused solely in the Fenix Framework, both from a design and implementation standpoints. Long Lived Transactions however, are a problem that is agnostic to the technology. As such, it would be interesting to port this solution to other, more popular, technologies.

Most of the design is actually agnostic to the Fenix Framework, and requires features already provided by many other technologies: access to previous versions of the domain objects and the ability to intercept transaction boundaries as well as reads and writes.

Chapter 8

Conclusion

Many enterprise applications have requirements involving operations that may span arbitrarily long periods of time. Yet, most modern data persistence frameworks are lackluster in regard to Long Lived Transactions. This forces programmers to devise clever ways to implement their requirements, promoting bad engineering practices and adding unnecessary complexity to the system.

The Fenix Framework, as a Framework to support enterprise applications that require a persistent and transactional rich domain model, merely provided support for regular (short-lived) transactions, and as such, suffered from the same perils of many other frameworks.

This document has described an extension to the Fenix Framework that allows drop-in support for Long Lived Transactions, without the need to modify existing business code. The presented extension shows many characteristics desirable of such a solution. These include support for system restarts by storing intermediate data persistently, support for concurrent users collaborating on a single Long Lived Transaction, the same correctness guarantees as regular transactions as well as comparable performance on the execution of the transaction's steps.

Throughout this document I presented a description of Long Lived Transactions, explained why they are difficult to implement using the currently available tools and shown several approaches programmers use to implement their requirements on top of short-lived transactions. I presented previous work on Database Transactions, Long Lived Transactions, Workflow Management Systems and Software Transactional Memories, in an effort to understand the current solutions. I also demonstrated that the currently existing solutions are unfit to the problem I was trying to solve.

To familiarize the reader with the Fenix Framework, I provided a brief description of the Framework's architecture, with detailed descriptions of the parts most relevant to the solution.

I then presented my solution to the problem at hand, showing the proposed architecture and implementation details, providing rationales for every design decision.

The initial implementation, while fully functional, present very poor performance results, rendering even the simplest of test cases unusable. I then presented an in-depth analysis of the implementation, and was able to track down the reasons for such poor performance. By doing several improvements on how the intermediate data is stored and searched, I was able to accomplish very good performance results, providing execution times close to those of regular transactions.

There is, however, one major limitation in the proposed solution. To ensure the correctness of the solution, transaction validation is performed in a similar way to regular transactions, by ensuring that the outcome of the transaction is not affected by reading outdated values. Due to the duration of a Long Lived Transaction, the probability of conflicts is much larger than with regular transactions. This can be a problem, as a large amount of work can be lost if a Long Lived Transaction is aborted. Even though solving this limitation is out of the scope of this work, I presented some ideas that could address this issue.

Bibliography

[1] G. Alonso, D. Agrawal, A. El Abbadi, M. Kamath, R. Gunthor, and C. Mohan. Advanced transaction models in workflow contexts. In *Data Engineering, 1996. Proceedings of the Twelfth International Conference on*, pages 574–581. IEEE, 1996.

[2] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ansi sql isolation levels. *ACM SIGMOD Record*, 24(2):1–10, 1995.

[3] P. A. Bernstein and N. Goodman. Concurrency control in distributed database systems. *ACM Comput. Surv.*, 13(2):185–221, June 1981.

[4] J. Cachopo and A. Rito-Silva. Versioned boxes as the basis for memory transactions. *Science of Computer Programming*, 63(2):172–185, 2006.

[5] U. Dayal, M. Hsu, and R. Ladin. Organizing long-running activities with triggers and transactions. In *ACM SIGMOD Record*, volume 19, pages 204–214. ACM, 1990.

[6] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. *Distributed Computing*, pages 194–208, 2006.

[7] N. M. L. Diegues. Practical parallel nesting for software transactional memory, July 2012.

[8] R. Elmasri and S. Navathe. *Fundamentals of database systems.* 2009.

[9] T. Fahringer, J. Qin, and S. Hainzer. Specification of grid workflow applications with AGWL: an abstract grid workflow language. In *Cluster Computing and the Grid, 2005. CCGrid 2005. IEEE International Symposium on*, volume 2, pages 676–685. IEEE, 2005.

[10] S. Fernandes and J. Cachopo. Lock-free and scalable multi-version software transactional memory. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, pages 179–188. ACM, 2011.

[11] S. Fernandes and J. Cachopo. Strict serializability is harmless: a new architecture for enterprise applications. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pages 257–276. ACM, 2011.

[12] M. Fowler. *Patterns of enterprise application architecture.* Addison-Wesley Professional, 2003.

[13] B. J. G. Franco. Reducing conflicts in software transactional memory, June 2013.

[14] H. Garcia-Molina and K. Salem. *Sagas*, volume 16. 1987.

[15] J. Gray et al. The transaction concept: Virtues and limitations. In *Proceedings of the Very Large Database Conference*, pages 144–154, 1981.

[16] R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 175–184. ACM, 2008.

[17] R. Hagmann and H. Garcia-Molina. *Implementing long lived transactions using log record forwarding.* 1991.

[18] M. Herlihy, V. Luchangco, M. Moir, and W. Scherer III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 92–101. ACM, 2003.

[19] M. Herlihy and J. Moss. *Transactional memory: Architectural support for lock-free data structures*, volume 21. ACM, 1993.

[20] V. Marathe, W. Scherer, and M. Scott. Adaptive software transactional memory. *Distributed Computing*, pages 354–368, 2005.

[21] V. Marathe, M. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. Scherer III, and M. Scott. Lowering the overhead of nonblocking software transactional memory. In *Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT)*, 2006.

[22] J. Melton. ANSI x3. 135-1992, American national standard, database language SQL. *American National Standards Institute*, 1992.

[23] J. Moss and A. Hosking. Nested transactional memory: model and architecture sketches. *Science of Computer Programming*, 63(2):186–201, 2006.

[24] J. C. Neves. On preserving domain consistency for an evolving enterprise application, July 2011.

[25] C. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM (JACM)*, 26(4):631–653, 1979.

[26] G. Reuter and J. Gray. Transaction processing: Concepts and techniques. *MorganKaufmann, San Mateo, CA*, 1993.

[27] T. Riegel, P. Felber, and C. Fetzer. A lazy snapshot algorithm with eager validation. *Distributed Computing*, pages 284–298, 2006.

[28] D. Rosenkrantz, R. Stearns, and P. Lewis II. System level concurrency control for distributed database systems. *ACM Transactions on Database Systems (TODS)*, 3(2):178–198, 1978.

[29] K. Salem, H. Garcia-Molina, and R. Alonso. Altruistic locking: A strategy for coping with long lived transactions. *High Performance Transaction Systems*, pages 175–199, 1989.

[30] N. Shavit and D. Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, 1997.

[31] A. Sheth, M. Rusinkiewicz, et al. On transactional workflows. *Data Engineering Bulletin*, 16(2):37–40, 1993.

[32] M. Stonebraker, D. Moore, and P. Brown. *Object-Relational DBMSs: Tracking the Next Great Wave.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 1998.

[33] W. Van Der Aalst and A. Ter Hofstede. YAWL: yet another workflow language. *Information Systems*, 30(4):245–275, 2005.

[34] F. Vernadat. UEML: towards a unified enterprise modelling language. *International Journal of Production Research*, 40(17):4309–4321, 2002.

[35] T. Watanabe. Agent-oriented model for managing long-lived transaction, based on work-flow and task-graph. In *Computational Intelligence and Multimedia Applications, 1999. ICCIMA '99. Proceedings. Third International Conference on*, pages 10 –13, 1999.

[36] G. Weikum. Extending transaction management to capture more consistency with better performance. In *Invited Paper, 9th French Database Conference, Toulouse*, 1993.

[37] D. Worah and A. Sheth. Transactions in transactional workflows. *Advanced Transaction Models and Architectures*, pages 3–34, 1997.