Abstract—One way to optimize a microservices application is to apply a speculative logic to the workflows it will execute. Some tasks, or entire sub-sequences of them, are executed in parallel if they have no data dependencies, even though they might have logical ones. This dissertation proposes a framework to help reduce the effort required for running speculative workflows in microservice based systems. It executes these workflows, providing atomic rollback mechanisms and crash recovery guarantees in order to ensure the system’s state remains consistent in critical scenarios. It achieves this by using state of the art modules, standards and technologies to handle the problems surrounding speculative executions. The proposed framework is shown to meet its operational requirements, maintaining consistency for different scenarios, including task rollback situations and data concurrency conflicts.

I. INTRODUCTION

In today’s increasingly popular microservice-based enterprise solutions, a single request to a system can implicate the coordinated execution of dozens of individual, fine-grained tasks. One way to optimize this process is to apply a speculative logic to the workflows it will execute. Figure 1 exemplifies a workflow’s transformation from its sequential model into its speculative version.

![Figure 1: Sequential workflow model transformation to speculative model.](image)

This dissertation proposes a framework that will free programmers from the burden of dealing with data consistency and concurrency concerns when using a speculative approach to workflow modeling. It will greatly reduce development efforts by allowing for multiple tasks to be executed in parallel flows in a speculative manner, and applying the proper mechanisms for rolling back the effects of such flows if they are determined to not be needed later on the execution process. The framework will deal with all of these concerns in a way that is transparent to programmers. In summary, it offers the following properties in one package, abstracting them from the developer:

1) Use of Zeebe as the implementation for a microservice coordination layer that allows for the execution of parallel microservice flows;
2) Ensure data consistency and atomicity of speculative microservice workflows, composed by distributed transactions;
3) Transparent rollback of speculative flows in case their execution is not needed.
4) Recovery of the system’s state and resume of active workflows in case the microservice coordination layer, or the data layer crashes, through the use of event sourcing;
5) Provide visibility into the history of the system’s state as a consequence of the implementation of an event stream;
6) Easy and intuitive workflow design (BPMN).

By using this framework, microservice developers can focus on the creation and development of better and optimized business workflows and business logic for each service. Programmers will be able to take a speculative approach to workflow modeling without concerning themselves with the implementation of rollback mechanisms or with guaranteeing the consistency of the system’s state.

II. BACKGROUND

In order to implement a framework that can run speculative workflows, one must analyze some of the problems arising from this type of execution and what possible technologies and methodologies can be used to solve them. The following is a list of important themes that must be studied and analyzed to achieve this dissertation’s goal, and solve the challenges mentioned in chapter I.

- Understand state-of-the-art consistency protocols for distributed systems and how they can apply to microservice architectures. This will help determine which ones are better suited for allowing the parallel execution of different microservice flows, characteristic of speculative workflow models.
- Study about already existing technologies that could be used to deal with cases where entire workflows, or parts
of it need to be atomically rolled back. The rollback process must ensure the system’s consistency after it’s completion.

- Analyse concurrency control mechanisms in order to determine which one better suits microservice environments and ensures consistency in situations of contentious data accesses by parallel running workflows.
- Study transaction recovery mechanisms and what methodologies can be adapted to the Framework’s layers in order to guarantee that it can resume workflow execution after a crash, without leading to illogical relationships and inconsistencies in the system’s state.

After this chapter, it is hoped that the reader will have a more serious understanding of the existing body of work relating to the various components and application layers targeted by the framework proposed in this dissertation.

A. Microservice Architectures

The microservices pattern is a service based pattern and derives from service oriented architectures (SOA). The core software components in service-oriented architecture, are services. Communication between services can be achieved in various ways, using different messaging protocols and lightweight message broker services. A service can be anything from a single function, to a group of classes each working over different data domains.

Microservice and service-oriented architectures differ in some key areas from the service and architecture perspective [1]. One of the main areas of difference from a services perspective between microservices architectures and service-oriented architectures is the granularity a service provides regarding data scope and function. Another major difference from the architecture perspective is component sharing [1]. SOA is built on the concept of a share-as-much-as-possible architecture style, whereas microservices are built on the concept of a share-as-little-as-possible architecture style.

1) Distributed vs Monolithic Systems: Today’s businesses are making a shift from one-size-fits-all solutions to providing more personalized content to their costumers, creating a great demand for systems that can take into account users and their associated profiles and provide targeted customization. A case can be made for service-based systems as a solution to this shift in relation to e-learning platforms [2]. This provides a good example of how SOA meets the high flexibility requirements modern platforms demand, supporting the fast-expanding range of coarse and finer grained services to provide real custom and flexible solutions.

Of great value, would be to compare between the infrastructure cost of running a web application using a monolithic architecture, versus a microservice-oriented one. Test results show that microservices can significantly reduce infrastructure costs in comparison to standard monolithic architectures by as much as 70% when using services specifically designed to deploy and scale microservices [3].

2) Microservice Architectures Guiding Principles: Guiding principles for microservice architectural models can be established based off of several other studies and papers that report on the advantages, disadvantages and lessons learned of different microservice-oriented system implementations [4]. The following is summary of some of the most commonly reported advantages of microservices architectures.

- Reuse: The maintenance of a single microservice will reflect on any connected project, reducing the effort overhead by applying the same changes to the same component used in different projects.
- Ease of Deployment: Each microservice can be deployed independently, without the need to recompile and redeploy the whole application.
- Observability: Microservice architectures help to visualize the health status of every service in the system in order to quickly locate and respond to any problem that occurs, and to better monitor their state.

From this dissertation’s point of view, isolation and independent deployability of services, their reusability and visibility into their state are some of the most important design principles and should be encouraged by the proposed framework. There are also some potential disadvantages identified by several papers that need to be taken into account [4].

- Implementation Effort: It is reported that implementing microservices requires more effort than monolithic systems.
- Network related issues: Since endpoints are connected via a network, the network should be reliable.
- Automation Requirement: The explosion of the number of services and relationships among them requires a way to automate things. DevOps solutions are meant to tackle this issue.

B. Distributed Transactions and Concurrency Control

Transactions represent a sequence of computational processes that should leave the system in a consistent state, which can be the same as the state the system was in before the transaction. In microservice based systems, transactions become distributed due the nature of such architectures, requiring coordinated action from different service components. Transactions become distributed as they are decomposed into different commands, each triggering local transactions in the respective service component.

Speculative workflows lead to a higher probability of concurrency conflicts due to the higher parallelism degree between microservices they demand for their execution. As such, the Framework should be able to execute multiple workflows and detected any conflicts before they lead to illogical relationships in data synchronizing different operations sufficiently to ensure the isolation requirement of the ACID model is met.

1) 2-Phase Locking: One of the most simple and logical solutions for concurrency control is the use of data locks. Locking is commonly used to synchronize accesses to shared data items. In a 2-phase locking concurrency control policy, locks are applied by transactions to the data objects they are going to access and subsequently released, in two phases. The protocol can be divided into two phases: growing phase and shrinking phase. Other algorithms using scheduling policies which are based on dynamically adjusting the priorities of blocked transactions based on some information provided by the transaction, have been proposed and shown to outperform algorithms that use static scheduling priority [5]. Many of the shortcomings of 2-phase locking procedures come from the possibility of deadlocks and deadlock treatment [6].
2) Optimistic Concurrency Control: An optimistic concurrency control scheme allows all transactions in the system to be active concurrently. Each transaction passes through three phases, or two for read-only transactions, that run sequentially: a read phase, a validation phase, and a final commit phase for read-write transactions.

For the same hardware and testing scenarios, other mechanisms, along with optimistic concurrency control were shown to outperform 2-phase locking mechanisms starting from relatively low concurrency levels, even with abundant hardware capacity [7]. In high-performance computational environments, hybrid OCC schemes help significantly improve the throughput of a system since the overhead required for re-executing failed transactions becomes less significant when compared to the under-utilization of processing power when using pure locking schemes [8].

C. Consistency in Distributed Workflows

Consistency requirements depend on the application being developed, thus the framework should allow for some flexibility in that regard. The framework must also know which mechanism it should implement to properly deliver on it’s ability to run speculative workflows and properly coordinate them in microservice applications.

1) 2-Phase Commit: One the most early and widely-used solutions for the distributed atomic commitment problem is the Two-Phase Commit (2PC) protocol. In 2PC, the participating components can be aggregated within two main groups: local resource managers and transactions managers. 2PC requires a coordinator, a dedicated transaction manager where a global transaction is triggered and that is responsible for arriving at the decision to commit, or abort a transaction and to propagate that decision to the other protocol participants. This protocol achieves ACID guarantees.

2) Saga Pattern: The Saga pattern is one the most successful attempts at solving the global distributed transactions consistency problem without the use of 2PC, as this approach does not provide the necessary scalability for today’s distributed environments. The case for Sagas comes from the concept of long-lived transactions, and it was first described in 1987 [9]. A saga comprises a sequence of local transactions. This sequence might involve calling multiple services in parallel.

There are two major mechanisms for coordinating a saga-based global transaction: choreography and orchestration. Some relevant conclusions can be reached by making a direct comparison between these two main coordination methods [10].

D. Transaction Recovery for Workflow Execution

According to the ACID model, a mechanism must be implemented that guarantees a transaction fails atomically and durably. This is especially relevant in distributed transactions since they can span multiple services, data contexts and databases.

1) Event Sourcing: The Event Sourcing pattern defines an approach to handling operations on data that’s driven by a sequence of events, each of which is recorded in an append-only store. The concept behind this pattern is that instead of storing the system’s state, one should store the actions that led to said state. The originating historic event log can be used to gain visibility into the system’s operations and to reconstruct previous states.

The event sourcing pattern was used in the implementation of an Enterprise Resource Planning system [11]. Debugging procedures were made simpler and problems easier to identify and resolve. Adding to it’s advertised advantages, the unexpectedly low adaptation curve for this pattern allowed non-technical stakeholders to better understand more technical scenarios and participate in the development process.

III. Architecture

In this chapter, a framework is proposed that provides a crash resistant environment for running speculative workflow models. The Framework can be divided in three major sections: workflow modeling, the microservice coordination layer and the data layer. These parts working in conjunction should provide developers with the necessary tools to easily deploy and execute instances of complex workflows. This chapter will describe both the microservice orchestration layer and the data layer as defined by the Framework. Their mechanisms will be analysed with regards to their functionality and what properties they provide to the framework. Figure 2 provides an overview of the entire framework and of where it fits in a microservice architecture.

A. Application Layer

Designing and implementing business workflows, requires a big communication and coordination effort between many
different teams in the project. Since this dissertation’s proposed framework aims to reduce the workflow implementation burden overall, it makes sense that it should promote a more intuitive way to plan and design these workflows.

To achieve this goal, the proposed framework makes use of the Business Process Model And Notation 2.0 standard (BPMN). Being that BPMN is a time-tested graphical notation model, it facilitates communication and collaboration between the different teams developing an application system. The microservice orchestration layer provided by the Framework has the ability to interpret and run BPMN designed workflows. Workflow designers can easily map BPMN business tasks to microservices and use it to accurately represent the flow of data and service calls. Designing workflows using BPMN does not just ease the workflow communication and interpretation effort, but also allows for designing workflows in a way that makes it easy to uphold the loose coupling principles of microservice architectures.

B. Microservice Coordination Layer

From the information gathered in chapter II, it can be concluded that the 2-Phase Commit protocol does not scale well for environments where long-lived, distributed transactions can run concurrently. One of the reasons for this, is that 2PC and its variations maintain locks on data objects for the duration of a transaction which can be a big performance bottleneck. The Saga pattern provides a better solution for distributed environments, since a saga splits a work transaction into smaller steps that run as local transactions in dedicated services whose effects can be reversed after an operation is performed. This was the pattern chosen for the microservice coordination layer of the proposed framework. The Zeebe workflow engine is used to implement this layer due to its almost perfect fit into the Framework’s requirements.

A central orchestrator allows to better decouple workflow awareness and state handling from microservices. By using an orchestrator, services can be developed with minimal, or no consideration for the workflows they will participate in and can focus solely on business logic and processing optimizations.

1) Orchestration Broker: In the proposed framework, workflows are deployed to the orchestration layer usually composed by multiple Zeebe brokers, stateful processes that control the flow of workflow instances as they make their way through the system. Once deployed, instances of those workflows, representing sagas, can be started at will. The broker, as a typical saga orchestrator, is responsible for triggering the appropriate services in response to state change events originating from the workflow’s execution.

Zeebe brokers persist their state in immutable event logs that record the entire history of workflow execution in the system. These logs can be partitioned and replicated in order to meet scalability requirement and provide fault tolerance in the case a broker crashes mid-operation.

2) Orchestration Broker’s Client Interface: The orchestration layer functions as a client/server model. As such, to connect to the orchestration layer clusters, the participating actors need to implement an appropriate client interface. Client applications can be scaled completely independently from the brokers.

Clients interact with the orchestration layer through a stateless gateway. This gateway exposes the layer’s functionalities and works as an interface between the clients and the actual brokers. It provides the ability to for example, create new workflow instances, poll brokers for any available work, publish relevant messages, or deal with workflow, or service related incidents. Usually, a client embedded in a microservice will register the type of work it can do with the broker. Several microservices can be registered as consumers of the same type of work, to provide load-balancing.

3) Transaction Recovery and Visibility: Following the discussion in section II-D1, the event sourcing pattern was deemed to be the most appropriate to provide the base for the Framework’s necessary transaction recovery and visibility mechanisms. In this section’s context, transaction recovery refers to the recovery of a workflow’s state in the microservice coordination layer. The event log produced by this pattern allows for the recreation of past transactions in case the microservice coordination layer crashes unexpectedly. This pattern also facilitates another major requirement of the Framework which is to provide visibility into the current and past states of workflows in the system. The event log can be exported to any components outside the orchestration layer and consumed for debugging, monitoring, or additional processing. In the event an orchestration broker crashes, the transaction recovery process is applied upon restart, enabling it to resume operations from where it left off by recreating the state of in-flight workflows at the time the process crashed.

C. Data Layer

The data layer is a logical representation of the data access infrastructure provided by the proposed framework. It is through this layer that microservices will interact with the application’s database systems, allowing them to perform all relevant data operations, such as reading from existing documents, inserting new ones, or modifying existing data objects.

1) Atomicity and Isolation Challenges: Referring to microservice-based architectures, usually implies the use of some design patterns, namely the database-per-service pattern. This pattern introduces some challenges to atomicity in saga-based environments.

Denormalized Data Models: One widely used approach to guarantee atomicity, is to denormalize data models by using embedded documents in conjunction with non-relational databases. The embedded document approach to achieve atomicity, while very adaptable, goes against proper microservice design due to the fact it will, more often than not, cross multiple data contexts.

Compensating Transactions: Compensating transactions come as the default answer for how to abort a failed saga while not leaving the system in an inconsistent state. The need for compensating logic requires microservices to maintain some sort of “abort” state, where the information required to rollback each step in a saga is contained. Since to create good compensating logic requires at least some awareness of the end-to-end workflow, the reusability of the components in the system for other application is severely reduced. Implementing compensating transactions requires a large development and coordination effort for the teams involved in the project.
2) Data Broker: The main component in this data abstraction layer is the data broker. The purpose of this component is to equip developers using the Framework with a module that transparently handles all data accesses within a workflow, guaranteeing they are atomic and isolated from other concurrently running workflows. The broker can be divided into four main modules: the database module, the services module, the interface module and operation logger module.

Database Module: The database module is responsible for executing database operations. This module connects to the database management system through specific libraries, and calls the appropriate methods in response to data operation requests.

Services Module: The services module is how the microservice orchestration layer interacts with the data layer to make it aware of the changing state of active workflows in the system. It provides three distinct services: "StartTransaction", "CommitTransaction" and "AbortTransaction". This module is responsible for managing a data transaction state table where all active workflows are indexed by a data transaction identifier that is mapped to a database session object.

Interface Module: The interface module is the module through which the data broker communicates with clients. This module provides microservices with access to all relevant database operations.

Operation Logger Module: The operation logger module has a very specific function relating to the internal processes of the data broker. It is responsible for intercepting all calls to the interface and services modules and persist to the database an historic log of all data operations executed by any given workflow instance.

3) Data Broker’s Client Interface: As in the microservice orchestration layer, the data layer also functions with a client/server model. The client acts as the data broker’s interface from the perspective of microservices and is used to interact with data brokres in the system, making available the functions provided by the broker’s own interface module. Data clients are also the way participating services in the system can communicate their intent to abort an active workflow, due to errors arising from processing failures, or business rules violations.

4) Commit/Abort Mechanism: The framework proposed in this dissertation avoids the use of compensating transactions as mechanism for rolling back failed workflows, by following the argument that commit and abort operations only concern the data layer and do not need to be communicated to the microservice components that form the system. Data brokers are used to decouple these operations from the actual services performing the work.

Concurrency issues arising from multiple accesses to the same data object, are handled through an optimistic concurrency control mechanism made available by MongoDB’s Transaction API. This was the chosen mechanism to work with this framework since it allows for better scalability, provides higher transactional throughputs and more efficient use of processing power in high data contention situations, when compared to 2-phase locking schemes. One important aspect to highlight with regard to the optimistic concurrency control mechanism used by this framework, is the fact that concurrency conflicts are detected immediately upon happening.

Commit: If a saga, representing a workflow, executes all it’s steps successfully, then it will execute a final step, calling the Commit Transaction service made available by data brokers. This call will trigger the validation phase of the optimistic concurrency control algorithm used by the session managing the data transaction associated with the workflow instance. If the commit succeeds, then the data transaction session is closed and removed from the data broker’s state table. If it fails, the transaction is aborted and rolled back.

Abort: In the case there is an error in a step of a saga workflow deemed fatal, meaning the workflow should no longer proceed, then the "Abort Transaction" service should be called. This service will abort the data transaction session associated with the calling workflow instance by dropping the read/write sets maintained by the optimistic concurrency control mechanism. In this way, all data changes made by the aborted workflow are completely discarded without ever affecting the system’s state.

D. Framework Integration with Workflow Models

Figure 4 provides two examples of how workflows can be modeled to integrate with the proposed framework. The framework specifies three data transaction processes mapping to the three services made available by the data broker. Ignoring the greyed out area, it can be seen that the workflow begins by telling the broker to start a new data transaction and only after that is the actual business workflow triggered. If the business workflow completes successfully, the associated data transaction is committed through the "CommitDataTransaction" process. If it fails, an "AbortWorkflow" message is published, triggering the "AbortDataTransaction" process. The flow’s grey area represents another possible execution path that allows for a pre-defined number of retries, if it is aborted.

1) Operation Logger and Transaction Recovery Mechanism: One important aspect to take into account, is the fact that the data layer maintains and monitors the state of the distributed data transactions associated with active workflows. A situation where the data layer crashes implies
that the state of active data transactions is lost and thus, upon restarting the data brokers, any pending workflows that tried to resume their execution and issue a data operation would cause an error to be thrown. To tackle this issue, the data broker implements a transaction recovery mechanism based off of the event sourcing pattern, similar in functionality to the one implemented by the orchestration brokers. The operation logger module is responsible for providing the needed functions and processes for this mechanism’s implementation.

2) Data Operations Consistency Mechanism: One of the assumptions made by this framework, is that most production applications will be using replicated databases to account for database failures. According to the CAP theorem, applications with replicated database systems need to trade-off some functionalities. This motivated the Framework to provide some liberty in determining the isolation and consistency requirements of data operations.

When a data broker process is first started, it loads the information pertaining to the acknowledgment level of the data operations execution. Two configurable parameters, “readAck” and “writeAck” allow developers to pass this information to the broker. Different combinations of the levels set by these parameters yield different consistency and isolation results. The data operations acknowledgment level is related to the consistency of the system in the sense that, depending on the set levels, the consistency restraints of the system are altered. This means consistency requirements can have a bigger resemblance to the ACID model, or to an eventual consistency model.

IV. EVALUATION

The framework put together and presented during this dissertation has the stated goal of reducing programmers’ implementation efforts when developing microservice application systems that allow for the execution of parallel flows within a workflow. Reducing programmers’ efforts in this sense means that the proposed Framework should be able to handle any processes that relate to the proper execution of speculative workflows in microservice systems.

A. Test Environment

In order to test the framework’s functionalities, a dummy web retailer application, built with Node.js, was created to integrate with it. Since this system was built for testing purposes, only a small set of functionalities were implemented in order to run a test workflow. The test workflow used, is comprised of a sequence of tasks that need to be performed to place a new order for an item in the dummy application system.

1) Metrics Framework: In order to perform some of the tests in this section, a very simple metrics framework is used. Similarly to the dummy application system, this framework was also implemented using Node.js. It is used to time workflow executions and update the retry counter of aborted workflows when appropriate. Figure 5 illustrates the test workflow embedded with the Framework in a retry setup, and integrated with the metrics framework.

B. Sequential vs Speculative Workflow Execution

To provide some insight into the motivation for the proposed framework, and to show why programmers would want to model their workflows in a speculative manner, a comparison is made between the execution times of the sequential workflow model of the test workflow and its speculative iteration.

The final average execution duration of each workflow model is stated in table I. The test shows they reduce the workflow’s execution time by an average of around 170 milliseconds, corresponding to a 21.3% reduction in the workflow’s execution time. This result helps to assert this dissertation’s motivating principle: the performance benefits of speculative logic as the execution model for microservice based workflows.

<table>
<thead>
<tr>
<th></th>
<th>Sequential Model [ms]</th>
<th>Speculative Model [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution Time</td>
<td>797.44</td>
<td>627.41</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Difference [ms]</th>
<th>Optimization %</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>170.57</td>
<td>21.3</td>
</tr>
</tbody>
</table>

Table I: Final duration, in milliseconds, of the test workflow’s sequential and speculative models with time and percentage differences between them.
C. Workflow Rollback

The first functionality to be analysed in this section, is the Framework’s ability to roll back tasks performed by speculative workflows in case these tasks are determined to not be needed. Next is a list providing a summary description of the three testing scenarios that were executed in this section.

Baseline (test 0): run the test workflow in an optimal condition scenario, meaning all business rules should be met along the workflows’ execution. The workflow makes it’s effects durable on the dummy application’s state by committing the data transaction at the end.

Authorization Failure (test 1): run the test workflow with an unauthorized user. The authentication task should triggers the workflow to abort due to a lack of user permissions.

Billing Failure (test 2): run the test workflow with the user’s account set to zero. The account billing task triggers the workflow to abort due to insufficient funds in the user’s account.

D. Concurrency Control

It is important to understand if the Framework can accurately detect data concurrency conflicts between different workflow instances. To test this, multiple test workflow instances will be running concurrently while trying to access and modify the same data object. Next, is a summary description of the concurrency control tests the Framework was put through, and the conclusions drawn from them.

No Concurrency Control (test 1): issue two test workflow instances simultaneously, representing two different users trying to order the same, unique item, with the data layer’s concurrency control mechanism disabled. The application’s state reflects the inconsistencies generated by the conflict, showing the item’s stock amount to be negative one, and that two orders were placed for the same item.

Same Item Orders (test 2): issue two test workflow instances simultaneously, representing two different users trying to order the same, unique item, with concurrency control mechanism enabled. Only one instance commits its results, while the other is aborted due to the detection of a data concurrency conflict.

Same Item Orders with Retry (test 3): similar to the previous test but using a Framework integration setup with retry logic, more similar to what one would find in a production environment. The execution times of both sequential and speculative test workflow models using this setup, are compared. Results from table II show the speculative model outperforms its sequential counterpart by a factor (around 37%) greater than the one obtained in the section IV-B.

<table>
<thead>
<tr>
<th>Sequential Model (_{\text{m,s}})</th>
<th>Speculative Model (_{\text{m,s}})</th>
</tr>
</thead>
<tbody>
<tr>
<td>1712.44</td>
<td>1080.07</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Difference (_{\text{m,s}})</th>
<th>Optimization(_{\text{m,s}})</th>
</tr>
</thead>
<tbody>
<tr>
<td>632.21</td>
<td>36.9</td>
</tr>
</tbody>
</table>

Table II: Final duration, in milliseconds, of the Same Item Order with Retry test using both sequential and speculative models with time and percentage differences between them.

E. Transaction Recovery Mechanisms

This test tries to prove the validity and correct functioning of the Framework’s transaction recovery mechanisms. In this testing scenario, multiple workflow instances are issued simultaneously and halted at different stages of their execution. The instances’ payloads were setup in a way so that no concurrency conflicts, or business rules violations should arise during their execution. A crash is simulated by turning off both the microservice coordination and data layers.

When the Framework’s layers are turned back on, the data layer’s transaction recovery procedure is shown to detect the unfinished data transactions and successfully recreate their previous data operations. The microservice coordination layer also recovers the workflow’s states and is able to resume operations from where they left off. All unfinished workflow instances are shown to commit successfully. This test shows the event sourcing pattern meets the Framework’s requirements for its transaction recovery mechanism, providing the expected functionalities.

F. Visibility into Workflow Execution

This test tries to show the validity of the event sourcing pattern as a mechanism to provide visibility into a running system’s state. By implementing this pattern, Zeebe, the microservice coordination layer, is able to export its internal historic event log to other outside components. Some examples of how the microservice coordination layer of the Framework offers visibility through Zeebe, are provided. These examples include the usage of external tools for visualizing, monitoring and interacting with workflow instances executed in the system. Other example use cases of visibility could be exporting the event stream produced by orchestration brokers to external components that would expand on the workflow coordination capabilities of Zeebe.

G. Data Layer Performance Cost

A final aspect to study regarding the Framework’s implementation is how its data layer affects the performance of the system. The data layer introduces another access level before the application can reach its data. Depending on the performance hindrance this layer introduces in the system, it might not be viable to use the Framework when high workflow execution throughputs are required. In order to measure the impact the data layer has in the overall workflow’s execution duration, a similar test to the one described in IV-B is executed. The difference is that the microservices that execute the business logic tasks now access the data layer directly, instead of going through a data broker.

From table III, it is concluded that the test workflow incurs in an execution performance cost of around 9.2% when using the Framework’s data broker client interface, when compared to accessing the data directly from the database. From this dissertation’s point of view, the performance cost incurred by introducing one extra layer between the application and the database is mitigated when considering the functionalities and benefits the full Framework brings to the system.

<table>
<thead>
<tr>
<th>Sequential Model (_{\text{m,s}})</th>
<th>Seq. w/ Direct Access (_{\text{m,s}})</th>
</tr>
</thead>
<tbody>
<tr>
<td>797.44</td>
<td>730.22</td>
</tr>
</tbody>
</table>
Table III: Final duration, in milliseconds, of the Same Item Order with Retry test using both sequential and speculative models with time difference and percentage cost.

<table>
<thead>
<tr>
<th>Difference[ms]</th>
<th>Cost%</th>
</tr>
</thead>
<tbody>
<tr>
<td>67.22</td>
<td>9.21</td>
</tr>
</tbody>
</table>

V. CONCLUSIONS

In the beginning of this dissertation, an argument was made in favor of speculative workflow models as a way to optimize workflow execution in microservice based applications. The reasoning behind this assertion is the same as the motivation for the speculative execution of instructions in computer processors with pipelined architectures. The operational requirements of the Framework proposed in chapter I, are considered to be successfully implemented. This dissertation was able to successfully prove that speculative models do represent an optimization of their sequential counterparts, with a reduction in execution time of around twenty percent. The Framework’s ability to atomically rollback unwanted changes and detect data concurrency conflicts was also successfully achieved. Another achievement worthy of notice is the Framework’s ability to completely recover from fatal system crashes.

From this dissertation’s point of view, the resulting Framework completely met its goal of freeing programmers from the burden of dealing with data consistency and concurrency concerns when using a speculative approach to workflow modeling. The Framework was shown to deal with this concern successfully in a way that is transparent to the programmer. All this was achieved while ensuring the Framework respects and promotes important principles and features of microservice architectures.

REFERENCES


