Computer Science and Information Systems 21(1):167–202 https://doi.org/10.2298/CSIS221210006N

An Approach for Supporting Transparent ACID Transactions over Heterogeneous Data Stores in Microservice Architectures

Lazar Nikolić, Vladimir Dimitrieski, and Milan Čeliković

University of Novi Sad, Faculty of Technical Sciences
Trg Dositeja Obradovića 6, 21000 Novi Sad, Serbia
{lazar.nikolic,dimitrieski,milancel}@uns.ac.rs

Abstract. Microservice architectures (MSA) are becoming a preferred architectural style for data-driven applications. A transaction within MSA can include remote calls to multiple services, turning it into a distributed transaction. Participating services may have their own data stores running local transactions with varying levels of transactional support and consistency guarantees. Coordinating distributed transactions in such an environment is a key challenge for MSA. The existing approaches are either highly consistent at the expense of scalability or scalable at the expense of consistency. Furthermore, implementing any of them requires architectural and code adaptation. In this article, we present the Service Proxy Transaction Management (SPTM) approach, which offers scalable reads and ACID transactions in MSA. The novelty of this approach is that it is based on intercepting inbound messages to services, rather than having services directly communicate with a transaction manager. As a result, transaction management is completely transparent to services and has little-to-no impact on code or architecture. We provide experimental results showing that SPTM can outperform lock-based approaches by up to a factor of 2, while still providing high consistency without the scaling bottleneck associated with locking.

Keywords: distributed transaction management, consistency, microservice, saga, 2pc, acid, base.

1. Introduction

In the past decade, there has been a shift from traditional, monolithic services to MSA utilizing multiple small services for data-driven applications. The trend has spread to many areas of the software industry, such as smart cities [55,46,10], e-commerce and financial systems [57,40], driving assistance [53], and edge computing [27]. The general idea is to split a complex domain into subdomains and assign each to a service. Ideally, services are small, mostly self-contained, and communicate with other services only when their domain boundary is crossed. This leads to improvements in scalability, robustness, and development flexibility enabled by loose coupling between services [48,45,66]. But MSA also brings new challenges to the table, with data management being one of the major categories [48,45,66]. One notable challenge in this category is facilitating transactions, which arises from the way they are executed in MSA. With a monolithic service, a transaction is comprised of a set of direct function calls to different submodules. The

transaction and the data it handles are under the control of the monolithic service. In contrast, transactions in MSA include inter-service remote calls via messaging protocols such as Hypertext Transfer Protocol (HTTP) or, more commonly, HTTP Secure (HTTPS). The communication in this manner happens at the *service communication level*.

Each service may have its own data store in a database-per-service fashion, with which it communicates at the *database communication level*. A particular data store used by a particular service might be chosen to address the specific needs of the service. For example, a service responsible for connections between people in a social network application would benefit the most from a graph database. This is known as the polyglot persistence paradigm [30], and applying it creates a heterogeneous environment of data stores. Providing a uniform level of consistency and transactional support is difficult even in homogeneous environments; heterogeneity takes it to another level of complexity. Transactional support and consistency levels offered by data stores in a system following the polyglot persistence principle are hardly ever uniform. For example, while relational databases are highly consistent with full transactional support, NoSQL data stores come with low consistency levels and transactional limitations. For instance, popular NoSQL data stores, such as Elasticsearch [16] and Cassandra [17], ensure atomicity only at the level of individual documents.

1.1. Motivation

Even if all the data stores in a system were identical, the challenge of coordinating a distributed transaction is still present. A distributed transaction in MSA is composed of multiple local transactions, each running on a different data store. Data stores have no inherent mechanism to connect local transactions to a larger context of a distributed transaction. Thus, distributed transaction coordination must be built on top of the data stores. Furthermore, transactional properties, such as Atomicity, Consistency, Isolation, and Durability (ACID) guarantees, do not extend beyond local transactions. This means that even if a distributed transaction is composed entirely of local ACID transactions, it is not ACID by default.

Workloads in MSA generally fall under two broad categories: Online Analytical Processing (OLAP) and Online Transactional Processing (OLTP). OLAP is characterized by complex, long-running ad hoc queries over many items. In contrast, OLTP is characterized by short-lived transactions on a small number of items, usually involving write operations. OLTP appears in data-driven applications very often [48]. An example of OLTP is a typical e-commerce scenario: a user checks out a product, product stock is updated, and payment is made. Within MSA, each step may be carried out by a different service in a distributed transaction. ACID guarantees are natural requirements for OLTP workloads due to the high consistency demands of data-driven applications handling them.

The level of consistency of a transaction can be tied to its isolation level [69]. Isolation level is defined by how many *isolation anomalies* [6] it allows to happen: fewer anomalies indicate a higher isolation level. An isolation anomaly is an unwanted effect on data caused by concurrent execution of transactions. An example of an isolation anomaly is *Lost Update* [1], in which a transaction can overwrite the result from another active transaction. Consider an example with transactions *T1* and *T2* that attempt to increment value i=1. Both *T1* and *T2* read the value i=1 and write the incremented value, which is i=2. The

expected outcome would be i=3, but a result from one of the transactions is effectively lost.

Isolation levels are defined in the order from the least to the most restrictive, as follows: *READ UNCOMMITED*, *READ COMMITED*, *CURSOR STABILITY*, *REPEATABLE READ*, *SNAPSHOT*, and *SERIALIZABLE* [6]. The higher the level, the more checks a system must run. This ultimately means that a higher isolation level leads to higher consistency, but at the cost of lower performance. A high consistency level is also commonly referred to as strong or strict consistency, whereas lower consistency levels are referred to as weak consistency.

Existing approaches to handling distributed transactions are synchronous and strongly consistent, or asynchronous and weakly consistent. The shortcomings of synchronous approaches are attributed to the locking of involved items [67,28], which is absent in asynchronous approaches. The asynchronous approaches in turn bring challenges related to weak consistency, such as transactions reading uncommitted values. There is a space of solutions to be explored that avoids locking, while still offering a high consistency level. Existing approaches also incur changes to both new and ongoing projects. Code and sometimes architectural changes are necessary to accommodate the chosen distributed transaction handling approach. Developers must be familiar with how distributed transactions work within an approach to effectively apply it. This is a notoriously difficult topic, so minimizing this requirement can be a huge boon to the development process. Unfortunately, there is a lack of good and intuitive abstractions to help developers tackle this issue, particularly for strongly consistent approaches, which can be a large barrier to adoption [41,42].

1.2. Contribution

In this article, we describe Service Proxy Transaction Management (SPTM), an approach to distributed transaction management in MSA that provides ACID guarantees for OLTP workloads. ACID is provided even in heterogeneous data store environments, in which not all data stores have ACID capabilities. SPTM combines the lock-free mechanism of asynchronous approaches with the high consistency of synchronous approaches by masking intermediate results of ongoing transactions. SPTM also aims to be as non-invasive to the existing codebase as possible with a low-performance overhead. It does so by intercepting, evaluating, and modifying messages at the service communication level. From the developer's perspective, a distributed transaction within SPTM is a collection of remote service calls with no additional libraries or frameworks, making the approach easy to understand and use. We believe that offering transparent ACID capabilities with the flexibility to combine it with both synchronous and asynchronous messaging makes SPTM very valuable for microservice developers. We also explore the possibilities and limits of the approach in which transactions are managed at the service communication level, using only the information available in the messages.

1.3. Article structure

In addition to the *Introduction* and *Conclusion*, the article comprises five sections. In the *Related work* section, we present and compare existing approaches to SPTM. The SPTM

approach is described in the *Service Proxy Transaction Management — SPTM* section. In the *Evaluation* section, we run benchmarks on an SPTM implementation and compare it to an implementation of another commonly used approach. We also discuss and explain the results of benchmarks. In the *Threats to validity* section, we briefly present factors that could affect the results of this article, primarily the ones presented in the *Evaluation* section. In the *Limitations and future work* section, we describe what the current limitations are and how we plan to overcome them.

2. Related Work

There are several research threads that work on addressing the problem of facilitating distributed OLTP transactions in MSA from various angles. While they are very distinct from each other, every thread exhibits three main characteristics: (i) support for synchronous and asynchronous execution, (ii) consistency guarantees and ACID support, and (iii) impact on the codebase and the software design. In this section, we compare current approaches through the prism of these characteristics. We have found that none fill the niche of both synchronous and asynchronous execution support, high consistency and ACID support, and low impact on the codebase — all of which SPTM strives to offer.

Two-phase commit protocol (2PC) can be considered a good match for OLTP workloads due to its synchronous nature and high consistency. As the name suggests, 2PC carries out transaction execution in two phases: the prepare phase and the commit phase. In the prepare state, participants check if conditions for applying changes. If the check passes, the changes are applied in the commit phase. The phases are coordinated by a transaction coordinator component that sends messages to participants. During the execution of both phases, locks are kept on the items involved in the transaction.

The locking mechanism of 2PC is a bottleneck for scalability and its main downside [67,28]. Furthermore, the 2PC coordinator acts as a single point of failure in the system: new transactions cannot be accepted nor executed if the coordinator is unavailable. 2PC also heavily relies on all data stores providing commands for controlling a transaction flow like *BEGIN*, *COMMIT*, and *ABORT* in SQL. Meeting this requirement is not always possible in heterogeneous data store environments. For these reasons, 2PC is now rarely used in practice within MSA [48]. There are also several 2PC variants, such as Distributed Strong Strict Two-Phase Locking (SS2PL) [8] and 2PC* [28] which scale better, but still fall behind approaches that eschew locking. 2PC only supports synchronous execution with high consistency across all variants. It also comes with code changes and design limitations, as it requires the use of client libraries and narrows the selection of data stores to transactional ones.

Despite falling out of focus of literature and open-source MSA projects in the recent years, 2PC is still present in industry and academia. It is still being used in industry with at least 8-16% practitioners claiming they use it [48], while also being present in research, with some more recent papers still referencing it or using it for comparison [28,73].

Saga is a pattern used for coordinating distributed transactions in Event-Driven Architecture (EDA). It consists of a series of subtransactions carried out by multiple services. Transactions are coordinated by creating an event signaling that the next subtransaction should start. This can be done in two ways [63]:

- **Orchestrated Saga** utilizes a central coordinator that coordinates subtransactions. When a service finishes a subtransaction, it notifies the coordinator. The coordinator in turn creates an event for the next service to start its subtransaction. Orchestrated Sagas help implement complex transactions by having them defined in the coordinator, with all the steps clearly laid out. The main disadvantage is that the coordinator is a potential point of failure. It is also a complex component to develop and maintain [52].
- **Choreographed Saga** has no central coordinator. Instead, each service creates an event that will trigger the next subtransaction. More complex transactions can be harder to implement and understand when compared to *Orchestrated Saga*, but the availability is better due to not having a coordinator that is a potential point of failure.

Saga relaxes ACID in favor of Basically-Available, Soft-state, Eventually-Consistent (BASE) [61]. Given a proper application of Saga and BASE, some of the ACID guarantees can be supported to an extent:

- Relaxed atomicity is achieved by undoing the results of a failed transaction with compensating operations. For example, a compensating operation for creating a user would be deleting the user.
- Strong consistency is replaced by eventual consistency. A properly implemented Saga pattern guarantees that the system will eventually reach a consistent state.
- Isolation is not supported to any extent. Active transactions can see intermediate results of other active transactions.
- Durability can be supported if events are persisted. Replaying an event will execute the corresponding transaction once again. This can be useful when, for example, the effects of a transaction are lost.

Issues arising from favoring BASE over ACID are some of the major pain points for developers [48,45,66]. These include non-atomic message processing, feral ordering, and isolation anomalies. Still, despite the issues and the expressed importance of ACID by developers [45], Saga remains the most popular approach in MSA because the loose coupling of transaction participants is perceived as a good fit for MSA [48,45]. Despite claims within both white and grey literature that Saga scales better than 2PC, we have found only one research paper that directly compares the two approaches [26].

Saga only supports asynchronous, eventually consistent transactions, which is the exact opposite of what 2PC has to offer. It also has a large impact on software design by requiring the application to apply the EDA.

Transaction coordinators are a loose group of approaches for distributed transaction handling in MSA applications. Although this landscape is quite varied, all of the approaches either fall into synchronous, highly consistent and asynchronous, eventually consistent categories. Granola [13] and CloudTPS [70] use the classic locking mechanism of 2PC facilitated by a transaction coordinator to ensure high consistency across multiple heterogeneous data stores. Cherry Garcia [14] works on the same principle, but has no central transaction manager. Instead, the transaction state is embedded in the data store objects as a part of the metadata. This comes at the cost of Cherry Garcia being limited to only data stores that provide *Test-and-Set* operators, which enables atomic writes on an object in a single instruction. ReTSO [43] works by combining data-store meta

fields and centralized transaction manager to provide *SNAPSHOT_ISOLATION* consistency level. The transaction state is managed by the Timestamp Oracle component, which is a bottleneck and the single point of failure of the solution. Typhon [3] uses a separate layer for transactional metadata and instead relies on Vector Clocks [56] to avoid centralized timestamp generation, but only comes with causal consistency guarantees. GRIT [68] executes transactions optimistically, by first capturing read and write sets, before logically committing and physically materializing the changes in data stores. It comes with a large architectural requirement: not counting microservices and data stores, GRIT brings six additional components into the application. Furthermore, data stores must support multi-versioning and snapshot reads, further narrowing down the available options. All of the discussed approaches come with client libraries or large design limitations, like the aforementioned *Test-and-set* requirement of Cherry Garcia, or the specific technology stack imposed by Narayana. Ultimately, none fulfills the criteria of supporting highly consistent synchronous and asynchronous transactions with low impact on the codebase.

Federated databases (FDS) [65] and **Polystores** [31] tackle the data management issue by acting as a query execution layer on top of all data stores in a system. This way, a heterogeneous data store environment appears to be homogeneous to a service. A single language is used for querying and manipulating all the data in a system. A service working with data is unaware from which data store the data originates. This is a major upside of Polystores, which keeps services unaffected by the choice of data store technologies. However, there is an overhead added by underlying data migrations involved in query execution. The overhead is acceptable for OLAP workloads but is not favorable for short-lived transactions of OLTP workloads. Transactions on Polystores are asynchronous and eventually consistent, due to how data is replicated between data stores. The impact on the code is minimal as long as services are already using a querying language offered by the polystore. Polystores also bring a limitation to the choice of data stores, as only the ones supported by the polystore can be used.

Database transaction middlewares follow the principle of separation of concerns to bring transactional properties to heterogeneous data store environments. They are similar to FDS and Polystores from the perspective of layered data store approach. The general approach of this thread of work is to add a transactional layer to separate transaction execution from data storage, which is delegated to data stores. MIDDLE-R [59], Calvin [67], Bolt-On Consistency [5], Deuteronomy [50] follow this principle by adding a layer responsible for transaction execution, fault tolerance, and logical replication across different data stores, regardless of their transactional or replication capabilities. As a result, transactional properties and safety guarantees are unified across all data stores. This way strong consistency can be achieved for both synchronous and asynchronous transactions in MSA applications. SPTM takes this thread of work and applies it to microservices instead of data stores, but leaves the microservice source code intact.

Transaction middlewares for Function-as-a-Service (FaaS) are one of the more recent developments in the area of transaction management. Their primary objective is twofold: to act as a cache for objects used in ongoing transactions, and to provide a certain level of consistency guarantees. HydroCache [71] and FaaSTTC [54] are prime examples in this category that offer Transactional Casual Consistency level (TCC) [2], while also improving the overall performance of transactions in a FaaS context. While TCC is weaker than Snapshot Isolation, it is currently noted as being the highest con-

sistency level achievable without a consensus [4]. Transaction middlewares offer client libraries for reading and writing objects within a transactional context that need to be explicitly invoked in the client code. Strong consistency is supported for both synchronous and asynchronous transactions, but code and architecture need to be adapted to use the middleware.

In Table 1 we present an overview of related work compared by the characteristics defined at the beginning of this section. For this comparison, we consider the consistency level as high if it offers stronger guarantees than causal consistency [69]. TCC and *SNAPSHOT_ISOLATION* are examples of such levels. We show that the existing approaches do not cover the desired characteristics of synchronous and asynchronous support, high consistency, and low impact on the codebase and the software design. At best, some approaches, such as transactional FaaS middlewares, cover the first two characteristics. The latter is never considered, which is a gap that SPTM aims to address.

Table 1. Overview of related work

Approach	Sync/Async	Consistency	Code and design impact
2PC	Sync	High	- Client libraries - Data stores must all be transactional
Saga	Async	Eventual	- Must apply the EDA defensive coding due to BASE (e.g. message replay, atomic processing)
CloudTPS [70] Granola [13]	Sync	High	- Client libraries
Cherry Garcia [14]	Sync	High	- Client libraries data store *Test-and-set* operation
ReTSO [43] Typhon [3]	Both	High	- Client libraries
GRIT	Async	Eventual	- Architecture must be adapted to implement GRIT
FDS and Polystores	Async	Eventual	- Can only use data stores supported by the Polystore/FDS
Database transaction middlewares	Both	High	- Client libraries - Can only use data stores supported by the Polystore/FDS
Transaction middlewares for FaaS	Both	High	- Client libraries

3. Service Proxy Transaction Management — SPTM

In this section, we present the SPTM approach. First, we provide a high-level overview of the SPTM approach and provide a list of requirements that a system must fulfill in order to utilize SPTM. Then, we delve into more details on how SPTM assigns and identifies transactions and involved objects from messages. Afterward, we describe how SPTM can achieve fault tolerance and scalability. Next, we present a reference architecture for

the client-server messaging model. Finally, we highlight how SPTM differs from related work.

3.1. Overview of the SPTM Approach

Microservices provide a set of functions that can be invoked remotely, called endpoints. Similarly to regular functions, an endpoint defines a name, a set of parameters, and a return value. The name is usually given as a path or a Uniform Resource Locator (URL). The parameters and the results of an endpoint are sent via the network within messages. The format of these messages is defined in a *message schema*. Endpoints and rules on how to use them form an Application Programming Interface (API) of a service.

SPTM is an approach for transaction management with a focus on transparency to reduce the impact on microservice source code. It does so by fulfilling the following responsibilities:

- **R1 Message metadata retrieval:** SPTM retrieves the information on which messages can be involved in transactions and how data objects (DO) can be extracted and identified from them.
- **R2 Message interception:** Inbound and outbound messages are intercepted and their contents are inspected and modified.
- **R3 Transaction context and data object detection:** This includes transaction identifiers, transaction state transition commands, and DOs involved in the transaction.
- **R4 Transaction lifecycle management:** Transaction state is transitioned based on transaction commands or message success status.
- **R5 Data object version control:** DOs modified by a message are stored within a version control storage. This version control storage is then used to replace uncommitted DOs with their committed versions.
- **R6 Distributed state management:** The updated transaction and version control storage states are distributed to multiple nodes for scaling and fault tolerance.

The SPTM approach is designed to work with any type of messaging protocol, as long as it meets the following four criteria. First, messages must contain a metadata (header) section that can carry additional data, such as the HTTP header section. Second, messages must contain information identifying the name of the endpoint of a service. Third, messages must be able to carry a payload that can be parsed and inspected. Fourth, the messaging protocol allows for building a layered system, in which a participant cannot tell if it is directly communicating with another participant or not. Specifically, a proxy can be installed between any two participants.

The SPTM approach can be applied to both synchronous and asynchronous messaging patterns. Transactions in both synchronous and asynchronous usually begin with a synchronous message that confirms that the operation is accepted. This is sufficient for SPTM to build and detect the transactional context, as long as the followup messages belong to the same transaction.

SPTM is not an all-or-nothing choice and can be applied to a selection of endpoints in a system. Those endpoints need to satisfy the following three criteria. First, the endpoints are using a messaging protocol that meets the requirements of SPTM. Second, only the DOs identified from the messages involved in the endpoint are used in a transaction.

This includes all the messages generated during the endpoint execution, not just the request and the response exchanged in client-server communication. Third, the values in the metadata/header section generated and used by SPTM must not be modified after a transaction starts. They should also be propagated in all of the messages generated by an endpoint that are part of a single transaction.

3.2. Message Metadata Retrieval, Message Interception, and Data Object Detection

In this subsection, we delve into details about how SPTM deals with message metadata retrieval, message interception, and data object detection.

A DO is a key-value structure containing data about an entity handled by an application. Each key is associated with a field of the corresponding entity, with at least one field acting as an identifier (ID). The ID field can be used to uniquely identify the instance of the entity which is represented by the DO. DOs are carried within messages at the service communication level in a data serialization format. A DO representing a user in JavaScript Object Notation (JSON) format is presented in Listing 1. The user is uniquely identified by the *"id"* field and carries information about a person named John Doe.

Listing 1. An example of a user DO in JSON format

```
{
    "id": 123,
    "email": "johndoe@email.com",
    "firstName": "John",
    "lastName": "Doe"
}
```

SPTM works on the premise that messages exchanged at the service communication level contain enough information to determine what DOs are involved and the operation applied to them. This is particularly true for Representational State Transfer (REST) [29]. Consider a typical REST request *PUT /user/123*. One can conclude that this is an update on a user type, judging from the */user* prefix. The */123* suffix indicates that it is a user with identifier 123. *PUT* indicates that this is an update operation; a new version of the object will be created.

Operation type on a DO can be *CREATE*, *READ*, *UPDATE*, or *DELETE* (CRUD). Complex operations on multiple DOs commonly used in MSA can be defined as a composition of CRUD operations on individual DOs. To ensure atomicity, compensating operations can be defined as operations with the opposite effect. *CREATE* is compensated with *DELETE*, and vice-versa. *UPDATE* can be compensated with another *UPDATE* carrying the previous value of a DO. The limitation is that SPTM must be able to create a compensating operation out of the information available in the original operation. For example, a *CREATE* operation must contain an identifier that would be used by the compensating *DELETE* operation.

To our knowledge, there are no strict rules or standards on how a message schema is defined. Still, no matter the schema, messages can contain enough information to detect DOs and determine the operation type. A developer may, for example, choose the Remote Procedure Call style (RPC) over REST for user updates. RPC user update can be defined as *POST /updateUser*. The URL */updateUser* indicates that this is an update on the user object, while the message body carries a user DO that contains the identifier 123.

Since there are no certain assumptions that we can make about message schema at the service communication level, developers must provide the message metadata. SPTM does not have a prescribed way of gathering message metadata but has the requirement of low impact on the source code. We see several ways of achieving this. For example, analysis of a microservice's source code could reveal what endpoints are used and what DOs are involved in transactions. The same goes for API specifications, such as OpenAPI for REST [21], or Protocol Buffer definitions for gRPC [23]. A dedicated configuration file can also be provided to SPTM, which could be hand-written or generated from the source code. Finally, messages can be extended with metadata that can be evaluated on-the-fly. The on-the-fly message schema inspection is the most flexible as it can adapt to message schema changes, in contrast to other approaches presented here, which operate offline. On the other hand, the offline approaches do not bring as much runtime overhead, as most of the computation is already done before runtime.

SPTM has no information about the semantics of a model nor its constraints. Instead, it relies on microservice logic to check the correctness of the detected DOs. This is achieved by optimistically passing them to the microservice, which will determine the outcome of the operation. Subsequent read requests will have their responses replaced by either committed DOs, or DOs written by the transaction in which the request belongs. Allowing writes to finish lets SPTM avoid locking which is a commonly cited problem with 2PC.

In summary, DOs are detected within requests and responses based on a provided configuration. This configuration contains a list of involved services, their endpoints, what DOs they carry, and operation type. Operation types are the standard CRUD. Each endpoint definition also carries information about how to evaluate and identify DOs. For example, a *PUT /users/123* request applies an update based on the JSON message body to the user 123. Endpoints can optionally define a rollback (compensating) endpoint. For example, user creation can be compensated by user deletion.

3.3. Transaction Context, Transaction Lifecycle Management and Data Object Version Control

In this subsection, we delve into details about how SPTM deals with transaction lifecycle management and data object version control.

Since distributed transactions within MSA are spread across multiple remote calls, there is a need to correlate messages to a transaction. This information is carried within a *transaction context* via message headers. Specifically, transaction context is an identifier carried via *Begin-Txn*, *Commit-Txn*, *Abort-Txn*, and *Txn-Id* message headers. A Universally Unique Identifier (UUID) value is chosen because generating such an identifier does not require coordination. A transaction is created by an SPTM node in the *STARTED* state as soon as a *Begin-Txn* value is encountered for the first time, with the value used as the transaction ID. If a transaction with the detected ID already exists, the transaction is instead rejected and the client can choose to restart the transaction with a new *Txn-Id* value. This is the start transaction lifecycle as illustrated in Fig. 1. Subsequent messages carrying the same *Txn-Id* value belong to the same transaction. If any of the messages fail, the transaction transitions to the *FAILED* state and is aborted. Compensating messages are then sent where necessary and the transaction transitions into the *ROLLED_BACK* state if they all succeed. Otherwise, it transitions to the *ROLLBACK_FAIL* state. Transactions can

be committed or aborted by sending the transaction ID in the *Commit-Txn* or the *Abort-Txn* header, transitioning them to the *COMPLETED* or *FAILED* states, respectively.

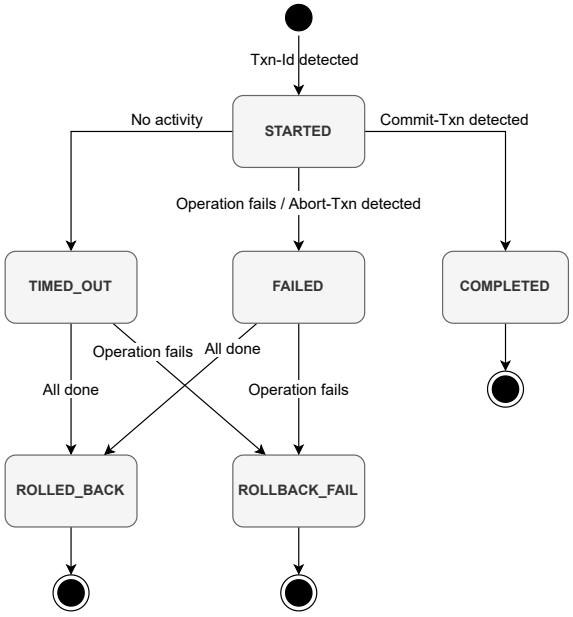


Fig. 1. SPTM transaction lifecycle

SPTM maintains a version chain with multiple versions of a single DO. When a transaction modifies a DO, a new version of the DO is added to the version chain. Transactions can access adequate DO versions using transaction timestamps, which can be physical or logical. Using physical timestamps over logical timestamps could be considered more practical, but runs the risk of clock skew if timestamp generation is not centralized [47]. A timestamp is generated when a transaction starts, which is then used for subsequent operations, regardless of the physical time of the operation. Transactions can only access committed DO versions with starting time before and ending time after the timestamp, or written by themselves.

Conflict detection is initiated on a transaction commit. SPTM goes through versions of DOs involved in a transaction and looks for write-write dependencies in the following manner. Consider a scenario with a transaction *T1*, that writes a new version *v1* on *DO1* at time *t1*. A write-write dependency is detected for transaction *T1* if there is a transaction *T2* at time *t2* that: (i) is in the *COMMITTED* state, (ii) has written a new version *v2* on *DO1*, and (iii) $t2 > t1$. Read-write and write-read dependencies are necessary for *SERIALIZABLE* isolation level [60], but would require a consensus for reads as well, which would adversely affect performance. Therefore the maximum supported isolation level offered by SPTM is *SNAPSHOT_ISOLATION*.

Conflict resolution is done in the first-writer-wins or last-writer-wins approach. Normally, the last-writer-wins approach carries the risk of messages being ordered differently

than what SPTM had observed, potentially violating the happens-before relation between transactions. Consider a scenario in which SPTM receives a message from *T1* writing a version *v1* of DO at *t1*, and *T2* writing a version *v2* at *t2*. SPTM observes that $t1 < t2$, but once messages are sent out to the microservice, they can be processed in any order. The risk is increased with the number of components involved in the processing, which is commonly at least two: microservice and its data store. Furthermore, even if the order of processing on all of the involved components stays identical, the responses could arrive in a changed order. Consider that t_w1 and t_w2 are the times of response for write operations and $t_{db}1$ and $t_{db}2$ are data store write times of *T1* and *T2* respectively. If $t_{db}1 < t_{db}2$, but $t_w2 < t_w1$, SPTM would conclude that $t2 < t1$ and abort *T2*. This is not a problem as long as all read and write operations are served through SPTM, as it possesses a consistent snapshot in which *T1* is committed and *T2* is aborted. First-writer-wins approach might be a better choice if retrying a transaction is less expensive than compensating it, for example when compensations are needed for aborted transactions. However, first-writer-wins effectively behaves pessimistically: all newly started transactions modifying a DO that is modified by an active transaction will be aborted.

Traditional database transactions are directly tied to database connections: when a database connection is closed, the transaction ends. With SPTM, transactions can span many independent messages. Services can fail while processing messages, leading to abandoned transactions. Abandoned transactions take up resources indefinitely and can cause all future transactions to be aborted due to conflicts, should the SPTM implementation use first-writer-win conflict resolution. Transactions that have not seen activity for an extended period will transition into the *TIMED_OUT* state and are treated identically to *FAILED* transactions.

Fig. 2 gives an overview of the concurrency control scheme described by SPTM. For simplicity, we use integers both for transaction identifiers and timestamps. The blue value is the latest committed value, while the green value is the value written by the ongoing transaction started by the green writer. Both readers access the value at timestamp *T=6*, which falls within the timespan of both versions. The green reader can see uncommitted values written by the green writer due to them being in the same transaction. The blue reader cannot see the uncommitted value because it is a part of another transaction.

3.4. Distributed State Management

In this subsection, we delve into details about how SPTM can deal with distributed state management.

SPTM is able to operate on multiple nodes by replicating transaction state changes and DO version chain updates. There are several potential points at which updates can be sent to the cluster:

1. **Option 1 — After each message:** SPTM sends an update containing the new transaction state and newly extracted DOs whenever a new message is processed.
2. **Option 2 — On transaction state change:** SPTM sends the new transaction state and DO deltas only when a transaction is transitioned to a new state.
3. **Option 3 — Periodically:** SPTM sends transaction state and DO deltas at fixed time periods, when a certain number of operations has been executed, or using a combination of both.

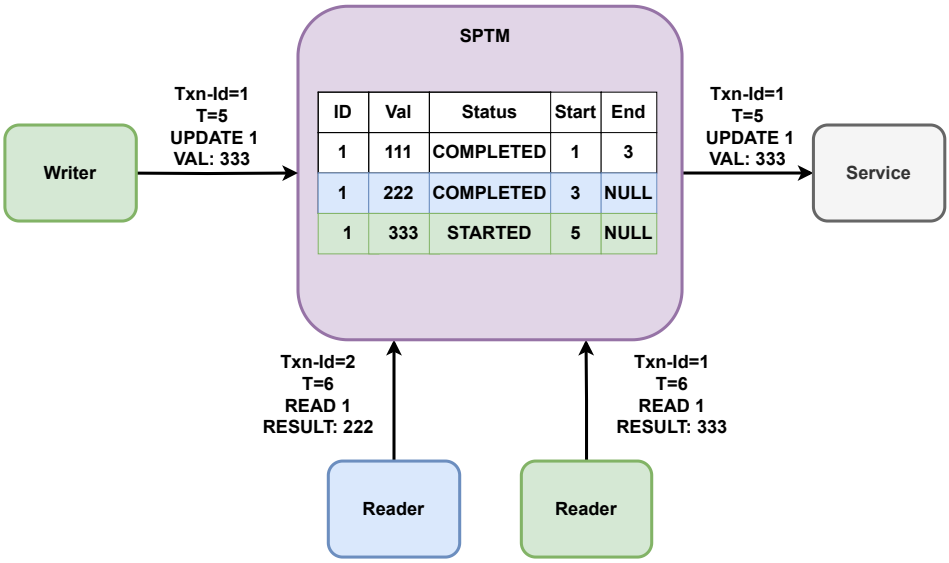


Fig. 2. Concurrency control overview of SPTM

Options should be chosen carefully based on architecture and workload profiles. For example, Option 1 offers the lowest replication lag, but also incurs the highest overhead because updates are sent out on each update. It is suitable for architectures in which multiple nodes can work on a single transaction with high consistency. On the other hand, shared-nothing architectures would benefit more from Option 2 and Option 3. The key design choice here is whether to partition the system on DOs or transactions. Partitioning on DOs is desirable for traditional OLTP workloads in which there are groups of interdependent DOs that are frequently processed together [44]. Such DOs can be collocated on a single node for highly efficient in-memory operations. Distributed state change is necessary only when a transaction operates on multiple unrelated DOs, or to replicate state for fault tolerance. On the other hand, partitioning on transactions could be more efficient when processing multiple unrelated DOs, but has a risk of conflicting writes on DOs.

3.5. Reference Architecture

In this subsection, we present a reference architecture that can be used to implement an SPTM transaction manager. In this particular case, we focus on the client-server communication model. Fig. 3 illustrates the reference architecture and its six components: one for each of the responsibilities defined in Section 3.1.

Metadata retriever is the component that fulfills R1. It reads message metadata from a configuration file, which is either generated from the API specification of a microservice or manually created by its developers. Metadata information is then passed on startup to an SPTM node to be used by other components. We use denoted lines in Fig. 3 for this activity to emphasize that it does not happen during runtime.

Message interceptor is the component that fulfills R2. It is a proxy that can interpret, load, and modify messages of the chosen protocol. Once it detects a request message

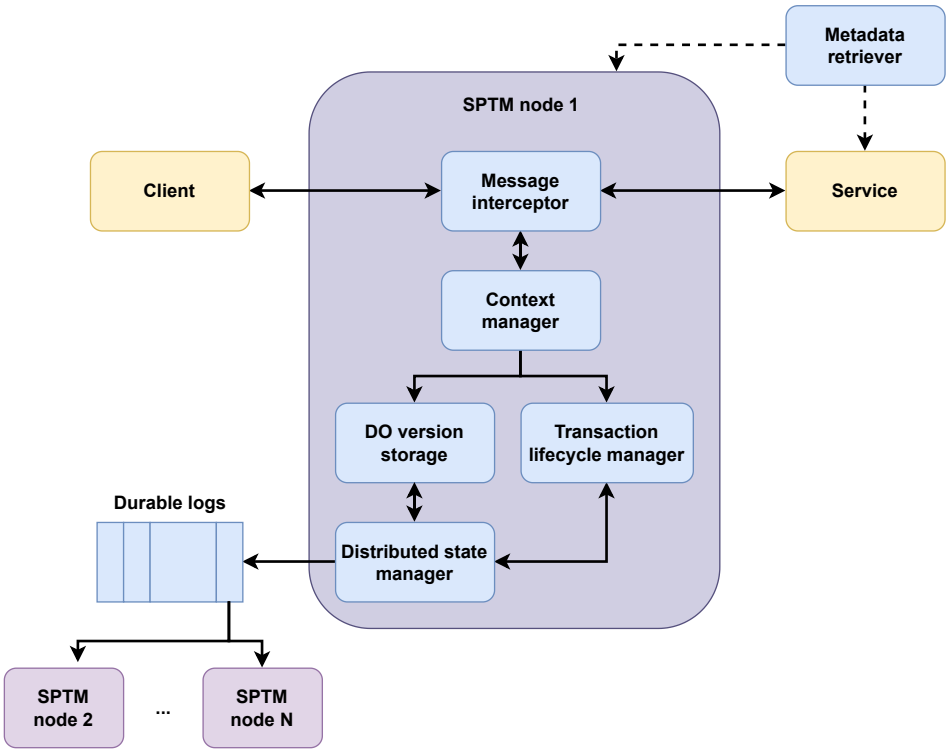


Fig. 3. SPTM reference architecture

matching a rule in the configuration file, it loads the payload and additional data, such as headers, into memory to be processed based on the transaction context. Once the processing is finished, a new connection to the intended microservice is created and the content modified by other components is sent through it. After receiving the response message, the payload, headers, and status are sent to the context manager. If the transaction fails, the response message is modified to reflect transaction failure, usually by setting the appropriate status code. Otherwise, the response message is loaded and processed identically to the request message. The only exception is that the modified content is sent back to the connection that carried the request message instead of creating a new one.

Context manager is the component that fulfills R3. It builds a transaction context, which consists of transaction information and DOs involved in the message. Transaction information carries the transaction timestamp, which the context manager uses to fetch DO versions available to the transaction. These DO versions are then used by the message interceptor to replace the original DOs found in the message payload.

Transaction lifecycle manager is the component that fulfills R4 and is responsible for keeping track of transaction information. This includes their status, timestamp, and dependencies. It is also responsible for transitioning the transaction state and triggering the corresponding actions, such as sending out compensating messages when specified.

DO version storage is the component that fulfills R5 and is responsible for managing DO versions and providing DO versions for a given timestamp. It is also responsible for detecting conflicting DO operations done by different transactions.

Distributed state manager is the component that fulfills R6. It manages the distributed state through the use of durable logs. A durable log contains basic information about a state update and is first stored on a local disk to ensure durability. At first, the log is uncommitted and its effects are not applied. It is then distributed across the network to other SPTM nodes: nodes two through N in Fig. 3. These nodes persist the durable log on their local disks and apply the changes, before sending back a confirmation message. Once the number of confirmations reaches a certain threshold, e.g., the majority of nodes have sent a confirmation, the durable log is committed and its effects are applied.

3.6. Differences to Related Work

In this subsection, we present the rationale behind SPTM and how it differs from existing approaches.

There are two key differences between the existing approaches and the SPTM approach. First, SPTM strives to be non-invasive, with little-to-no impact on the code. This lifts the burden of distributed transactions from developers, allowing them to focus on application logic. Second, SPTM intercepts, evaluates, and modifies messages at the service communication level. Other approaches operate at the database communication layer: as a proxy to a data store, by embedding metadata in the data store, or as a standalone component that the service must use.

We chose SPTM to operate at the service communication level because REST style with HTTP appears to be the *de facto* standard in the industry. In Table 2 we show the technology choices of eight open-source MSA projects. The table, alongside industry studies [9], reveals that HTTP/REST is the only constant across all services, unlike data stores that are more varied. Relational databases are most common but are too different even among themselves. Supporting all of them with a single solution is not a simple task: it remained an unresolved challenge of FDS [11].

The results of industry studies cannot be extended to the entire industry due to sample size. However, we believe they provide a good case that there is much less variance in service level protocols than in data stores. Operating at the service communication level allows SPTM to cover more ground with a single implementation. The SPTM approach is applicable to any protocol and style beyond HTTP/REST, such as gRPC [20] or GraphQL [19]. Furthermore, it allows SPTM to support both synchronous and asynchronous messaging, as long as the chosen messaging protocol satisfies the requirements of the SPTM, as defined in Section 3.1.

4. Implementation

As a part of our research, we implemented a transaction manager called *fed-agent* that follows the reference architecture defined in Section 3.5. It is written in the Go programming language [18], with a focus on the REST architecture and JSON DOs.

Message metadata is provided in a configuration file and is built using the same building blocks used to define a microservice endpoint. These are the URL, the message body,

Table 2. Overview of technologies in open source MSA projects

Project	Languages	Messaging	Data stores
eShopOnContainers [32]	C#, .NET	HTTP REST, RabbitMQ	SQL Server, Redis
ESPM [33]	Java, Spring	HTTP REST, RabbitMQ	MySQL, Redis
LakesideMutual [35]	Java, Spring	HTTP REST, RabbitMQ	H2
FTGO [34]	Java, Spring	HTTP REST, Kafka	MySQL, DynamoDB
Vert.x Blueprint [39]	Java, Spring	HTTP REST, Kafka	MySQL, MongoDB, Redis
Sentilo [36]	Java, Spring	HTTP REST, Kafka	MongoDB, Elasticsearch, Redis
Spring Pet Clinic [38]	Java, Spring	HTTP REST	HSQLDB, MySQL
Sock Shop [37]	Java, Spring, Go	HTTP REST, RabbitMQ	MySQL, MongoDB

the headers, and the method. In Listing 2 we show a configuration example for a single HTTP/REST endpoint. *PUT /user/*$\{id\}$ request is an *UPDATE* operation on a user identified from the path parameter *id*. The *request* field describes the content of the request. It is a JSON object, as described by the *content_type* field. It contains a single user DO, that can be identified by reading the *id* property of the JSON object within the message body. The configuration also describes a compensating operation in the *rollback* field. The *target* field contains the name of the endpoint that will undo the effects of this operation. It is the same endpoint, but the *data* field has a *data_source* subfield specifying how to form a compensating operation. In this instance, the last committed version of the DO should be used in the request body of the compensating operation.

Not all messages need to be intercepted by *fed-agent* though and it should be positioned within the architecture in a manner that only adds overhead where necessary. More specifically, only messages involved in OLTP workloads should be considered. When an HTTP request is received, *fed-agent* first checks if the URL is specified in the configuration. If not, the request is passed through to the microservice and is ignored by *fed-agent*. Otherwise, *fed-agent* copies the payload and deserializes its JSON payload for inspection before passing the unmodified request to the microservice. After receiving the response, *fed-agent* does a series of operations depending on the operation type. In the case of writes, *fed-agent* first checks the response code. If the code is not in the success range, i.e. 2XX, the transaction state is set to *FAILED*, and the update is sent via consensus. Otherwise, DOs are extracted from the request payload based on the configuration rules and added to the DO version store. The new versions are then sent out via consensus and the state is reconstructed by the followers. For reads, *fed-agent* deserializes the JSON payload and extracts all the DOs from it. Then it finds the last committed version of each DO and replaces their occurrences in the payload before sending the modified response payload to the client.

Listing 2. *Fed-agent* configuration example for a user update endpoint

```
{
    name = "update-user-profile"
    idempotent = true
    method = "PUT"
    path = "/user/{id}"
    type = "UPDATE"
    request {
        content_type = "json"
        entities {
            user {
                id_source = "body"
                id_path = "id"
            }
        }
    }
    response {
    }
    rollback {
        target = "update-user-profile"
        data {
            content_type = "json"
            entities {
                user {
                    data_source = "version"
                    data_target = "body"
                }
            }
        }
    }
}
```

To provide a high level of isolation and consistency, *fed-agent* needs to ensure that intermediate results of active transactions are not visible to other transactions. For this purpose, a concurrency control scheme needs to be in place. We chose Multi-version concurrency control (MVCC) [7] using timestamp ordering (MVTO) [72] as the concurrency control scheme in *fed-agent*. There are also two other MVCC variants: two-phase locking (MV2PL) and optimistic (MVOCC). We find these less suitable due to the following reasons. MV2PL acquires locks on DOs, which can lead to deadlocks and connection exhaustion. MVOCC works by holding on to changes until the end of the transaction, at which point all of the changes are sent to storage, or in this case, services. This is not desirable for two reasons. First, *fed-agent* has no information about data constraints and validation: they are delegated to the services. Therefore, *fed-agent* cannot determine the validity of DOs in a message until it gets a response from a service. Second, applying all changes at the end of a transaction can generate a burst of messages in a short period. The pressure created by a large burst of messages can disrupt microservices, leading to failed messages. Failed requests can trigger compensating messages, which amplifies the issue. However, as an SPTM implementation, *fed-agent* keeps a consistent snapshot of DOs and compensating operations are not necessary if all reads and writes go through *fed-agent*. We leave this as an option to allow for eventual consistency of DOs, in case they need to be accessed directly in data stores.

Fed-agent periodically runs a background routine that deletes data from finished transactions and progresses active transactions toward their finished state. A transaction is in a finished state if its status is *COMPLETED*, *ROLLED_BACK*, or *ROLLBACK_FAIL*. *Fed-*

agent first goes through finished transactions and deletes their DO versions from the DO version store. The only exceptions are DO versions from *COMMITTED* transactions that are the last committed version for their respective DOs. Then, *fed-agent* tries to progress currently active transactions. A transaction is moved from *STARTED* to *TIMED_OUT* state if there have been no operations within a configured period. Compensating operations for *TIMED_OUT* and *FAILED* transactions are attempted for a configured number of retries or until a configured period is passed before transitioning to *ROLLED_BACK* or *ROLLBACK_FAIL*.

With *fed-agent*, we chose Option 1 from Section 3.4 for distributed state management. *Fed-agent* uses the Raft [58] consensus protocol for transaction tracking. A *fed-agent* cluster has one leader and multiple followers that can serve reads. Updates to transaction statuses and the version chain are accepted by the leader and replicated to followers on each intercepted message. Having more followers increases availability, fault tolerance, and read parallelism at the cost of write performance. Should the leader become unavailable, one of the followers can become the new leader. The new leader then triggers the recovery process for in-flight messages: messages that were detected and passed to a service, but a response was not registered. Since no response is received, the transaction state cannot be determined and it must be either replayed or aborted. If a message is marked as idempotent in the configuration, it can be replayed safely. Otherwise, the corresponding transaction must be aborted to maintain consistency.

Transaction updates are made durable by writing a log to persistent storage. A log contains information about a non-read transaction operation, namely its type, timestamp, and data. After accepting a non-read operation, the leader writes a log to the local disk and sends it over the network to followers. Upon receiving the log, the followers also write the log to disk and send a confirmation back to the leader. Once the majority of nodes have confirmed a log write, the transaction state is successfully updated. The more nodes there are in a cluster, the longer this process takes. However, having a larger cluster increases availability, as a cluster will continue to operate as long as the majority of nodes are online. Larger clusters also offer improved read parallelism, due to the higher number of followers that can serve reads. This trade-off between write performance and availability plus read parallelism can be tuned by adjusting the size of the cluster.

In Fig. 4, we depict a scenario in which the results of a write operation served by the leader are replicated to followers to be provided to readers. Dashed lines are used to denote replication via consensus protocol, while solid lines are used to denote direct communication from transaction participants. After the green writer creates a new DO version, it is replicated to both followers. The green reader can then read the new value without accessing the leader.

Fed-agent has two distinct modes of operations: single-node and multi-node. In multi-node mode, *fed-agent* creates a Raft consensus group with a single leader and multiple followers. All write operations are handled by the leader since they can affect *fed-agent's* state by modifying a transaction's status and the DO version store. Physical timestamps are used to order these updates and are only generated by the leader, which eliminates the risk of clock skew at the expense of write throughput. Operations are sent out to followers, which then execute them in the ascending timestamp order to construct the global state. In contrast, read operations do not generate any further network communication as they do

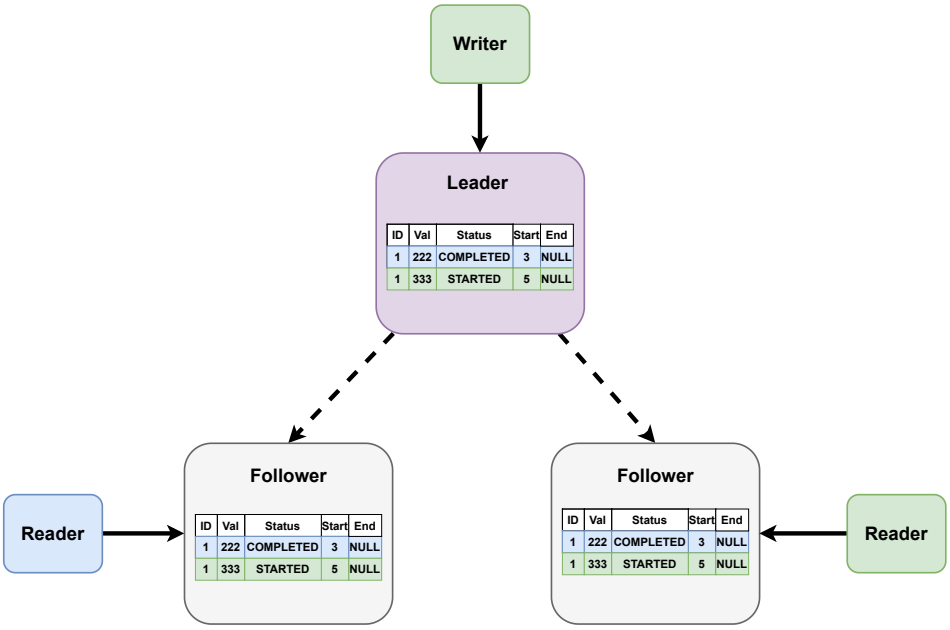


Fig. 4. A *fed-agent* cluster consisting of a leader serving writes and followers serving reads

not modify the state of *fed-agent*. Single-node mode avoids all consensus-related modules and directly writes updates to the local storage for persistence.

Fed-agent mostly operates in-memory to minimize its overhead. We believe that the in-memory storage architecture is well-suited for this case, as the memory footprint of active transaction data is usually small enough to fit even into the memory of programmable network switches [51]. DOs from active transactions are stored in a DO version store in the memory of the *fed-agent* process, which offers *SNAPSHOT* isolation level [1]. The minimum information necessary for crash recovery is stored within logs on the persistent storage on each transaction operation. These logs can be replayed to restore the pre-crash state of *fed-agent*. Periodic snapshots of the state are also stored to reduce the number of logs that need to be replayed during recovery.

5. Evaluation

In this section, we evaluate SPTM by comparing *fed-agent* to 2PC and Saga as main representatives of their categories, due to how ubiquitous they are in MSA applications. For both comparisons, we used mostly identical setups, with only a message queue component added in for Saga. Transactions used in the benchmarks are implemented to result in the same state in both synchronous and asynchronous versions, used by 2PC and Saga comparisons respectively.

5.1. Setup

We ran benchmarks on an e-commerce system for an online video game. In this system, a user can buy appearances for their in-game characters, called *skins*. There are four services involved in buying a skin: *store-service*, *payment-service*, *game-service*, and *gateway-service*. The responsibilities of services are as follows: *store-service* keeps track of user credits and skin prices, *payment-service* manages payments, *game-service* stores a list of skins available to a user, and *gateway-service* coordinates transactions made by a user. Each service is a REST service implemented in Go that uses PostgreSQL 13 [22] as the underlying database in a database-per-service fashion. Only the standard Go libraries and low-level database drivers are used for the implementation.

We define *BuySkin* transaction offered by *gateway-service*. The workflow is shown in Fig. 5:

1. *GetUserBuyInfo*: Sends a request to the *store-service* to check if the user has the credit to buy the skin.
2. *MakePayment*: Create a payment with a value equal to the skin's price and send it to *payment-service*.
3. *AddUserSkin*: Make the skin available to the user by updating the user profile in *game-service*.
4. *UpdateUserCredit*: Redact price from user credits and send an update to *store-service*.

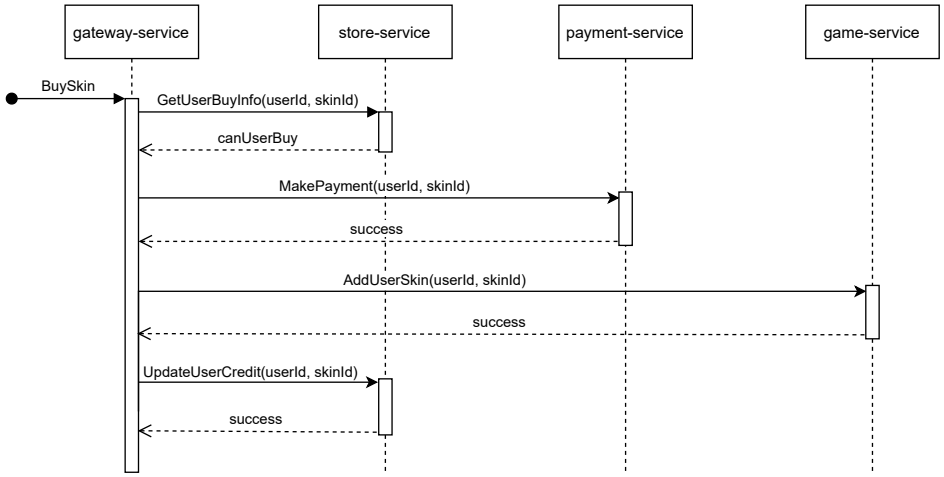


Fig. 5. Sequence diagram of *BuySkin* transaction

The setup was deployed on the Amazon EC2 service with each service, database, and client deployed on a separate *t3.2xlarge* instance (8vCPU, 32GB RAM) [15]. The isolation level of PostgreSQL transactions was set to *REPEATABLE READ* so that 2PC matches the isolation level of *fed-agent*.

Benchmarks are executed by starting an HTTP client calling the *BuySkin* endpoint with a random skin ID. Since transactions can abort due to a conflict, the client retries the

transaction until it succeeds with a two-minute timeout. Time until transaction success is measured and referred to as execution time. We also track the number of aborted transactions to help understand the execution time better. We compare execution time for 2PC, Saga, and *fed-agent* on the following dimensions:

- **Concurrency**: number of parallel clients, i.e., threads calling the *BuySkin* endpoint.
- **Contention**: number of concurrent clients attempting to modify the same DO, controlled via distribution. Distribution defines the probability for a DO to be accessed by a transaction. It can be uniform, hotspot, or Zipfian. Uniform distribution chooses a DO uniformly at random, while Zipfian chooses a DO according to the Zipfian distribution, resulting in a small number of popular DOs. Hotspot distribution defines two parameters: the fraction of operations accessing "hot" DOs and the size of the "hot" dataset. Uniform and Zipfian distributions are the most common distributions in web services [12], while hotspot gives us better control over contention scaling.

We formulate the following scenarios for benchmarks:

- **Scenario 1 — Uniform distribution**: the number of parallel clients is scaled from 0 to 200 using a uniform distribution. The purpose is to examine the behavior of systems under high concurrency and low-medium contention.
- **Scenario 2 — Zipfian / extreme contention**: The number of parallel clients is scaled from 0 to 60 using Zipfian distribution. The purpose is to examine the behavior of the systems under extreme contention.
- **Scenario 3 — "Hot" dataset**: The number of parallel clients is 100 using hotspot distribution. We define a "hot" dataset of DOs as 2.5% to 10% of the total dataset, which is accessed by 20% of all operations. The purpose is to examine the impact of scaling contention level on the behavior of the systems.

Each run of each scenario executes 30,000 transactions on the dataset of 1,000 skins and one user. Since a user-skin reference is considered a standalone object, having exactly one user is sufficient to test concurrency and contention. It also allows us to better control those two parameters by keeping the user fixed and changing the distribution of skins.

5.2. Comparison with 2PC

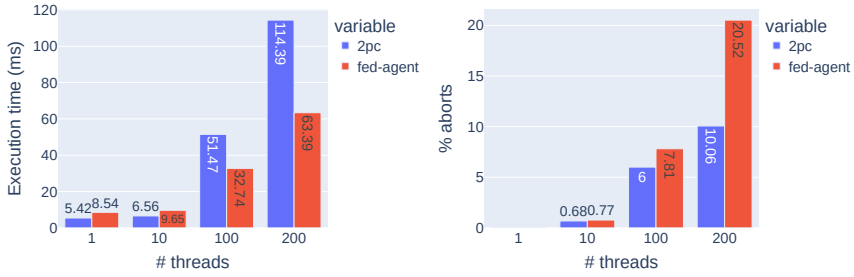
In this subsection, we compare *fed-agent* with a 2PC implementation by comparing the transaction execution times of both. A 2PC implementation is close to the *fed-agent*: a distributed transaction is a sequence of HTTP requests. This makes direct comparison with 2PC implementation rather straightforward.

We only compare a single-node *fed-agent* cluster to 2PC for one main reason. During our testing, we noticed that having multiple 2PC coordinators does not improve transaction execution time. This is because the main work is done by databases, not the coordinators. The coordinators only facilitate the transactions by opening and maintaining connections to databases that have a finite connection pool. Multiple coordinators enable the system to open connections faster, practically letting the system exhaust the connection pool earlier. Since the goal of this benchmark is to measure transaction execution time, developing a highly complex, multi-node 2PC implementation will not meaningfully contribute to the comparison.

This comparison employs two separate *BuySkin* versions in *gateway-service*: one using 2PC (*BuySkin2PC*) and one using *fed-agent* (*BuySkinFedAgent*). Both versions bring minor tweaks to the original *BuySkin* transaction. For *BuySkin2PC*, *gateway-service* generates and sends a *Txn-Id* header containing a transaction ID with each request. Services correlate this value to a local database transaction that is active until the global transaction finishes. For *BuySkinFedAgent*, *gateway-service* generates and sends a *Txn-Id* header containing a transaction ID with each request. *Fed-agent* takes care of the transaction management with no additional changes to services.

It should be noted that each run had to be split into 15 iterations of 2,000 transactions due to 2PC implementation blocking indefinitely after a certain time under high loads. *Fed-agent* was more resilient in this regard and would continue working, but would block for almost a minute before continuing due to high IO activity on the *fed-agent* node. We attribute this to benchmarks hitting the Amazon AWS I/O throughput limit for the instance types used [15].

Fig. 6a and Fig. 6b contain benchmark results for low contention (under 20% aborts) and increasing concurrency. *Fed-agent* is slightly slower than 2PC when the number of concurrent users is low. *Fed-agent*'s overhead is relatively constant compared to execution time, making it more noticeable at low loads. As the load increases, *fed-agent* execution becomes almost twice as fast as that of 2PC. *Fed-agent* exhibits a 20% aborted transaction rate which, while twice as high as that of 2PC, is still low enough that it does not lead to performance degradation.



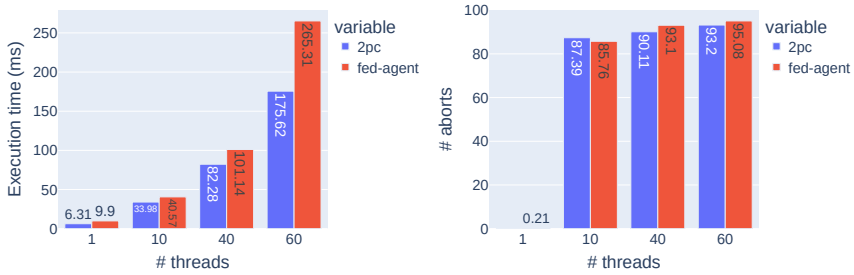
(a) Median execution time for scenario 1 **(b)** Aborts for scenario 1

Fig. 6. The effects of the number of concurrent clients for scenario 1 (uniform distribution)

Fig. 7a and Fig. 7b contain benchmark results for extreme contention and moderate concurrency (scenario 2). The results show that 2PC and *fed-agent* behave comparably for abort rates up to 90%. Beyond that point, *fed-agent* starts to fall behind. We attribute this primarily to *fed-agent*'s usage of MVOCC, which is inherently worse for high abort rates than MV2PL used by the combination of PostgreSQL and 2PC.

We have observed that roughly 10% of 2PC transactions were dropped and never completed in the timeout period for 40 and 60 thread benchmarks. That likely happened because services maintain open database connections throughout the execution of a trans-

action, exhausting the I/O resources of the EC2 instance [15]. *Fed-agent* did not have this problem because database connections are short-lived in this case. We believe that this is an indication of the bottleneck of 2PC that prevents it from scaling.



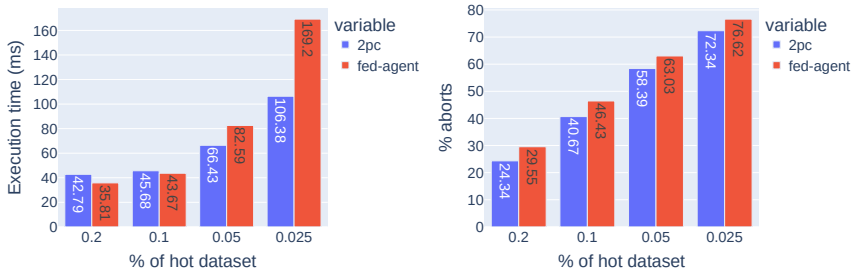
(a) Median execution time for scenario 2 **(b)** Aborts for scenario 2

Fig. 7. The effects of the number of concurrent clients for scenario 2 (Zipfian / extreme contention)

Fig. 8a and Fig. 8b show the effects of contention on execution time (scenario 3). Contention is scaled by keeping the number of concurrent users at 100, while scaling the "hot" dataset from 20% to 2.5% of the entire dataset. Each operation has a 20% chance of accessing a DO from the "hot" dataset in all runs. We see that *fed-agent* performs slightly better than *2PC* until roughly 60% abort rate. *Fed-agent* starts to quickly fall behind beyond 70% abort rate for 100 concurrent users.

Scenario 2 and scenario 3 benchmarks reveal that rollbacks have the highest impact on *Fed-agent*'s performance. Fig. 9a and Fig 9b show the total number of transaction attempts during the execution of scenario 2 and scenario 3 respectively. Despite a similar abort percentage, the total number of requests is disproportionately higher for *fed-agent*. For example, for 60 concurrent users using Zipfian distribution, for a 2% difference in abort rate, there is a 30% difference in total attempts. We suspect this discrepancy was caused by two things. The first is the first-write-wins conflict resolution of the *fed-agent*: once a transaction has written a new value for a DO, all transactions attempting to update the same DO are aborted. The second is *fed-agent*'s eager conflict detection: transaction conflict detection is done on each transaction status update, not just before committing. This is an implementational choice not imposed by the SPTM approach. Both properties can lead to a high number of short-lived aborted transactions, which is potentially detrimental to performance for high abort rates. Future research will focus on lowering the number of aborts or sending fewer compensating messages whenever possible.

In conclusion, *fed-agent* performs better than 2PC in high concurrency, low-medium contention scenarios while offering similar isolation and consistency levels. 2PC outperforms *fed-agent* in scenarios with high abort rates. It should be noted that *fed-agent* rollback mechanism always sends compensating operations when a transaction fails. 2PC is not safe from this either: if a database transaction fails to commit in the commit phase, other database transactions within the distributed transaction need to be compensated. The



(a) Median execution time for scenario 3 **(b)** Aborts for scenario 3)

Fig. 8. The effects of contention for scenario 3 (”hot” dataset)

(a) Total attempts for scenario 2 **(b)** Total attempts for scenario 3

Fig. 9. Total number of requests scenario 2 (Zipfian / extreme contention) and scenario 3 (”hot” dataset)

2PC implementation used in benchmarks does not support this. The absence of rollback messages could have a noticeable impact on the performance for high contention and bring 2PC execution time closer to that of *fed-agent*. The biggest caveat to using a 2PC is the fact that it is only applicable if all underlying databases are transactional. *Fed-agent* does not have this requirement. Furthermore, 2PC was observed to stop accepting transactions after hitting a certain amount of requests, most likely caused by the connection pool exhaustion and locking. The benchmarks on *fed-agent* show that an implementation of SPTM can offer high consistency while being scalable and non-intrusive to the application code.

5.3. Comparison with Saga

In this subsection, we compare *fed-agent* with a Saga implementation by comparing the total execution time of all transactions, rather than individual execution times. We take both finished and compensated transactions into account.

The Saga implementation uses RabbitMQ [24] to pass transaction messages between participants. We only compare a single-node *fed-agent* cluster to a single-node RabbitMQ

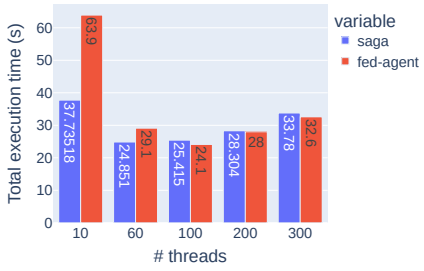
cluster with default configuration and durable message queues. The RabbitMQ instance is hosted on a separate machine. RabbitMQ clients utilize a connection-per-thread approach instead of a channel-per-thread approach to maximize throughput. The number of open connections is equal to the number of client threads used in the benchmark.

This comparison employs two separate *BuySkin* versions in *gateway-service*: one for Saga (*BuySkinSaga*) and one for *fed-agent* (*BuySkinFedAgent*). *BuySkinFedAgent* is identical to the one used in comparison with 2PC. *BuySkinSaga* is a choreographed Saga implementation of *BuySkin* that uses RabbitMQ messages. We choose the choreographed approach to avoid additional message passing between participants and the coordinator that would negatively affect the performance. Each subtransaction of *BuySkinSaga* has a compensatory transaction that undoes its effects, while *BuySkinFedAgent* does not need them. Each subtransaction also sends an acknowledgment back to the RabbitMQ cluster once a message is successfully processed, or a negative acknowledgment if it fails. Both *BuySkin* variants are started by the *gateway-service*, with *BuySkinFedAgent* synchronously returning a response once the transaction finishes and *BuySkinSaga* being asynchronous, returning a success when a transaction is accepted. In both cases, a transaction is started by an HTTP request from the test client. We also modify benchmarks so that Saga executions can detect conflicts by adding a unique constraint on the relationship between user and weapon skins.

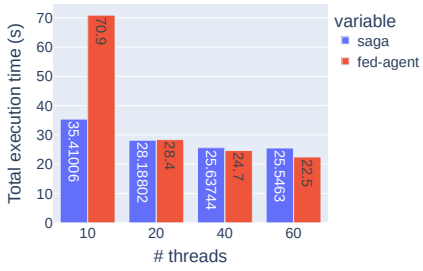
Fig. 10a contains benchmark results for low contention (under 20% aborts) and increasing concurrency, and Fig. 10b contains benchmark results for extreme contention and moderate concurrency (scenario 2). Both benchmarks reveal that Saga vastly outperforms *fed-agent* when the number of threads is less than 20 due to the disparity of synchronous and asynchronous message processing in the test setup. The test client works by sending HTTP requests sequentially in the specified number of threads. Since *fed-agent* transactions appear to be synchronous to the clients, the system's resources will remain underutilized as transactions are started one by one in a low number of threads. On the other hand, starting a Saga transaction is much faster because the *gateway-service* only needs to send a RabbitMQ message and return a positive response. As a result, the test client starts all of its transactions much faster in the lower thread range, allowing the system to better utilize its resources. This difference is gone as soon as the system becomes saturated with more concurrent client threads ($>$20). *Fed-agent* has identical execution times across all of the scenarios as Saga when the number of threads is over 20, while also offering higher consistency.

Fig. 10c show the effects of a small "hot" dataset on execution time (scenario 3). *Fed-agent* has no need for compensations, leading to a 50–60% faster execution than Saga. Saga degraded in performance in this scenario but not in scenario 2 because scenario 2 did not reach the concurrency level needed to cause a high number of compensations.

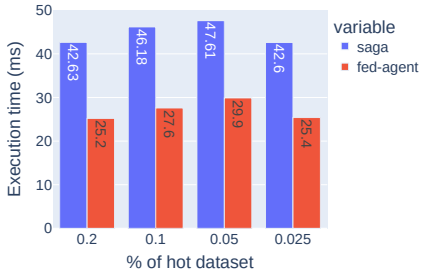
In conclusion, our benchmarks show that Saga vastly outperforms *fed-agent* in scenarios under 20 concurrent users. This happens because transactions are started much faster asynchronously by the test client when the system is not saturated. Both perform equally in low-to-moderate contention scenarios, with *fed-agent* offering high-consistency and ACID properties. In high-contention scenarios, *fed-agent* outperforms Saga by a wide margin, due to Saga needing to compensate failed transactions, whereas *fed-agent* does not. We also report that the implementation of a choreographed Saga is more challenging



(a) Total execution time for scenario 1



(b) Total execution time for scenario 2



(c) Total execution time for scenario 3

Fig. 10. Execution times of *fed-agent* and Saga across all scenarios

than that of *fed-agent* because the developer needs to take care of message acknowledgment, compensations, retries, and tracking of decentralized transaction execution.

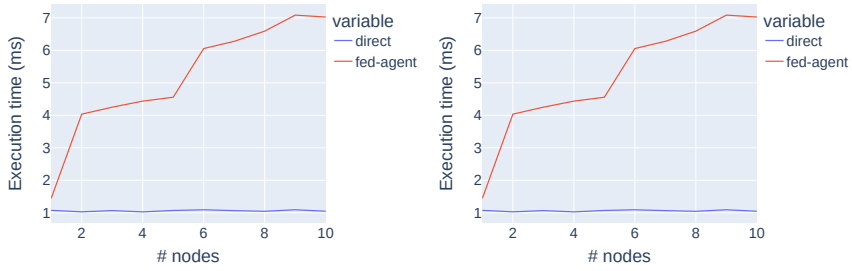
5.4. Overhead Characteristics

In this subsection, we measure the overhead of *fed-agent* as the difference between the execution time of direct microservice calls and calls via *fed-agent*. We consider the impact of cluster size and payload size on read and write overheads. We run all benchmarks in a single thread to better isolate the overhead from other factors, such as concurrency and contention.

In Fig. 11 we show that a single-node *fed-agent* cluster adds 0.5ms to reads and 2ms to writes. Moving to a multi-node cluster shifts *fed-agent* into another operation node, causing a large increase in overhead. Read and write overhead increase to 3ms and 10ms, respectively. Each node beyond the second adds roughly 0.5–1ms overhead for both reads and writes. The number of messages to reach consensus increases with cluster size, increasing the overhead.

The main component of write overhead is the persistence of transaction logs. A single-node *fed-agent* cluster persists logs to the local disk only and does not need to send them via the network for consensus, leading to a much lower overhead.

The main component of read overhead is transaction conflict detection. Reading transaction status in a single-node cluster is done by directly accessing the local storage. How-



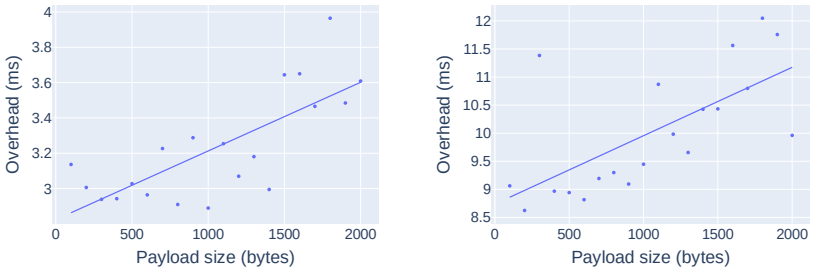
(a) Write overhead, increasing cluster size **(b)** Read overhead, increasing cluster size

Fig. 11. The effect of cluster size on write and read overhead

ever, reads in multi-cluster deployment need to be *linearizable* [69] to ensure that the previously made changes are applied first. This is necessary to provide high consistency, but it adds an overhead that increases with cluster size. We believe that a weaker consistency level could suffice and would noticeably lower the overhead.

In Fig. 12 we show the effects of payload scaling on write operations. Network communication is needed to reach a consensus on the new transaction state. Transaction status update message contains the original payload, hence the potential impact of payload size on performance. Reads do not update the transaction state, so they are left out of the benchmarks. Payload is scaled to 2kB because most HTTP REST JSON API implementations appear to have payload sizes of 1–2kB, with the median of around 1500B [62].

In both single-node and three-node clusters scenarios, we show the trendline due to the stochastic nature of network latency. For a single-node cluster, write overhead increases by 0.7ms. The increase is attributed to proxying a larger request and the extra time needed to store the payload in the local storage. There is no additional network communication in the single-node cluster, hence a low overhead. The three-node cluster needs to reach a consensus for each write, adding up to 3ms of additional overhead.



(a) Write overhead in a single-node cluster **(b)** Write overhead in a three-node cluster

Fig. 12. The effect of payload size on write overhead

In conclusion, the overhead of a single-node *fed-agent* cluster for a typical MSA application is expected to be 0.5ms for reads and 2ms for writes. We believe that the overhead can be negligible in an application with usual response times of tens of milliseconds. A multi-node cluster offers higher availability and read parallelism, but increases the overhead to 3ms for reads and 10ms for writes.

5.5. Isolation Level

Fed-agent implements the MVOCC scheme as specified by the SPTM approach. Proper implementation of said scheme should lead to the *SNAPSHOT* isolation. As we are not aware of any tool for formal validation of isolation levels, we implement a test suite attempting to cause isolation anomalies [6]. If anomalies do not occur, we consider the system to be operating at the corresponding isolation level. *Read Skew* and *Write Skew* anomalies are not considered because *fed-agent* supports a limited set of predicate-based operations. *Phantom Read* is the exception and could still occur, as shown in the tests. *Cursor Lost Update* is not considered because *fed-agent* has no notion of cursors. The tests are defined as follows:

- **P0 Dirty Write**: this phenomenon occurs when a transaction overwrites a value written by another active transaction. The test attempts to manifest the anomaly by starting transactions *T1* and *T2* simultaneously, each updating the same user with a different value. *T1* commits before *T2*, which leads to *T2* getting aborted to maintain consistency.
- **P1 Dirty Read**: this phenomenon occurs when a transaction reads a value written by another active transaction. The test attempts to manifest the anomaly by starting transactions *T1* and *T2* simultaneously, each updating the same user with a different value. Both *T1* and *T2* update the user before reading the new value. Both *T1* and *T2* only see the value they modified, meaning that *Dirty Read* did not occur.
- **P2 Non-Repeatable Read**: consider a transaction *T* that reads a value before and after it is modified by another transaction. If *T* reads a different result both times, *Non-Repeatable Read* has occurred. The test attempts to manifest the anomaly by starting transactions *T1* and *T2* simultaneously. *T1* reads the value of a user before and after *T2* updates it. Since both reads yielded the same result, *Non-Repeatable Read* did not occur.
- **P3 Phantom Read**: consider a scenario in which a transaction does a predicate-based read, and another transaction does a write that adds a DO to the result set. If the predicate-based read is repeated and it captures the new value, Phantom Read has occurred. The test attempts to manifest the anomaly by starting transactions *T1* and *T2* simultaneously. *T1* reads all skins for a user, which is a predicate-based search. *T2* then updates values for one of the skins and adds a new skin to the set. *T1* then repeats the original read. Since the result stayed the same, *Phantom Read* did not occur.
- **P4 Lost Update**: this phenomenon occurs if active transactions read and modify the same value. The test attempts to manifest the anomaly by starting transactions *T1* and *T2* simultaneously. Both *T1* and *T2* read user credit and increase it by a fixed value. *T1* commits before *T2*, which leads to *T2* getting aborted to maintain consistency.

We ensure that the tests are deterministic by artificially pausing a transaction execution when a state in which an isolation anomaly can happen is reached. Then, another

transaction that attempts to cause a data anomaly executes and the state is checked. We also define tests with invariants that would be violated if data anomalies happened. For example, the total sum of funds across all bank accounts should stay constant before and after the test run, or state of a product must not be a negative number.

5.6. Threats to validity

In this subsection, we briefly discuss the threats to the validity of the results of our research. The biggest threats are those related to the implementation of SPTM, 2PC, and Saga in the Evaluation section. We identify the following:

- 2PC and Saga implementations were specifically built for the purpose of evaluation of *fed-agent*. While it was built with great care, there could be another way to implement 2PC or Saga which would lead to a different evaluation outcome.
- The choice of messaging technology is an important one when implementing Saga. We chose RabbitMQ as it is, in our judgment, a good representative of how a Saga would be implemented in practice due to its popularity. Other message queue technologies, such as ZeroMQ [25], could lead to a lower execution time. However, we believe that the characteristics of Saga, such as a high number of compensations under heavy contention, ultimately have a bigger impact the benchmark results than a choice of technology within the same family.
- The evaluation was made on a specific use case: an e-commerce application. We acknowledge that there are different use cases that have different performance profiles, and thus universal claims cannot be extrapolated from a single comparison.
- The evaluation was not made on a real production system, but on a system built to simulate one. Production systems can be expected to be much more complex, and evaluations within them could lead to different outcomes.
- The evaluation was done on a public cloud provider, namely Amazon Web Services (AWS). This takes away control from us over variables such as network availability and resource contention.

6. Limitations and Future Work

SPTM relies on messages containing enough information to keep track of DO values throughout transaction execution. To our knowledge, there is no rule enforcing the message schema in any API style. What can be defined are the levels of information availability:

1. Level 1 — Messages contain neither DO nor any other identifying information. For example, ”count recently added items”.
2. Level 2 — Messages do not contain DO but contain identifying information. For example, an increment endpoint for a product stock that returns an empty response.
3. Level 3 — Messages contain partial or convertible DO. For example, an update endpoint that takes or returns partial DO information, or a search endpoint that returns a modified DO.
4. Level 4 — Messages contain the full DO. Single DO format is used for all endpoints handling a type.

SPTM can handle level 1 endpoints if they behave as materialized views. These endpoints can, in some cases, provide enough additional information and auxiliary endpoints for SPTM to estimate the state. Consider a “top 10 selling products” endpoint. A service can provide the count for 10+N products, where N is the buffer in case one of the items of lower placement enters the top 10. Additionally, the service provides a rule to increment the appropriate top 10 record on the ”buy” endpoint hit. For more complex rules, snapshotting of endpoints on each DO modification is necessary. This means that in the previous example, the ”buy” endpoint would cause SPTM to re-fetch DOs from ”top 10 selling products”.

Handling level 2 endpoints is possible if a corresponding read endpoint is provided. In that case, SPTM can track reads on DOs and add them to the version chain. Detected DOs can be used to allow subsequent reads to see a consistent state. If no DO version is known, it is pre-fetched before proceeding with the transaction.

This approach can also be applied to level 3. Alternatively, a conversion algorithm is provided to SPTM, either via a Domain Specific Language (DSL) or a plug-in system. Level 4 endpoints are fully compatible with SPTM and require no modifications.

We expect level 3 and level 4 endpoints to be the most common for OLTP workloads, as it is characterized by short-lived operations on clearly identifiable DOs. Level 1 endpoints serve to connect OLTP results with OLAP data stores, so they are also expected to appear quite commonly. We conjecture that level 2 endpoints could be considered bad practice, as they do not align with any of the major API styles, such as REST. Therefore, we believe that supporting level 1 and level 3 endpoints should be prioritized in the future work before adding support for level 2 endpoints.

In this article, we present an implementation that only considers Option 1 from Section 3.4 for distributed state management. Furthermore, it can only scale reads horizontally, and not writes. In the future, we intend to explore Options 2 and 3 and see how writes can be scaled horizontally within SPTM. We intend to do so by using Conflict-free Replicated Data Types (CRDT) [64] and the MVOCC approach of private transaction workspaces [49]. Transactions are assigned to an SPTM node with a hashing function at the start, which will execute all transaction-related operations in-memory. Messages are sent over the network only once a transaction is finished, minimizing the total number of messages.

Because SPTM operates at the service communication layer, it does not prevent services from accessing uncommitted DO values by directly accessing a data store. We see three potential solutions:

- Service endpoints can be defined as microtransactions: each endpoint can read or write only DOs identifiable from the request or the response. A static analysis tool examining the source code and suggesting changes to fit this criteria can be implemented and provided to developers. However, we recognize that this can be quite restrictive for some systems.
- A service can explicitly ask SPTM what DOs are visible to a transaction. This adds additional overhead and has the potential to violate non-invasiveness property of SPTM.
- Libraries for multiple languages can be developed that inject active values into the persistence layer of a service. This approach should be the least invasive out of the three if properly implemented. The main drawback is that libraries need to be implemented for multiple target languages.

Having a low-to-no impact on existing systems is an important goal of SPTM. In the future, we will be looking into reducing the usage overhead by generating configuration files from existing code or documentation. The first step in this direction would be generating configuration files from language-agnostic documentation specifications, namely Swagger and OpenAPI. Implementing language-specific support for a choice of popular programming languages would come after that.

7. Conclusion

In this article, we introduce the SPTM, a transaction handling approach that supports ACID transactions for OLTP workloads in MSA. A transaction manager implemented in the SPTM approach is easy to use and has a very low impact on code or architecture.

We compare an implementation of SPTM, called *fed-agent*, to 2PC and Saga implementations in a lab-grown e-commerce system. The results show that *fed-agent* outperforms the 2PC implementation by a factor of two for low-medium contention levels. It falls short only in extremely high contention scenarios. As the contention increased in our benchmarks, the 2PC implementation ran into locking bottlenecks that stopped the system from accepting new transactions. *Fed-agent* did not run into this issue and would continue to work beyond the bottleneck point of 2PC. The comparison with Saga shows that *fed-agent* achieves similar performance, while providing high consistency. We also measure the overhead of *fed-agent* and show that it can be negligible for MSA applications with typical response times of tens of milliseconds.

We provide empirical evidence showing that *fed-agent* supports ACID transactions at the *SNAPSHOT ISOLATION* level, which is on par with 2PC using a typical relational database. We believe that having transparent ACID transactions improves the development of MSA in two major ways. First, developers do not need in-depth knowledge of how distributed transactions work to effectively implement them. Second, having ACID transactions for both synchronous and asynchronous message processing helps with data consistency, primarily by reducing the amount of data issues caused by concurrent execution.

References

1. Adya, A., Liskov, B., O'Neil, P.: Generalized isolation level definitions. In: Proceedings of 16th International Conference on Data Engineering (Cat. No.00CB37073). pp. 67–78. IEEE Comput. Soc, San Diego, CA, USA (2000), http://ieeexplore.ieee.org/document/839388/
2. Akkoorath, D.D., Tomsic, A.Z., Bravo, M., Li, Z., Crain, T., Bieniusa, A., Preguica, N., Shapiro, M.: Cure: Strong Semantics Meets High Availability and Low Latency. In: 2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS). pp. 405–414. IEEE, Nara, Japan (Jun 2016), http://ieeexplore.ieee.org/document/7536539/
3. Arora, V., Nawab, F., Agrawal, D., Abbadi, A.E.: Typhon: Consistency Semantics for Multi-Representation Data Processing. In: 2017 IEEE 10th International Conference on Cloud Computing (CLOUD). pp. 648–655. IEEE, Honolulu, CA, USA (June 2017), http://ieeexplore.ieee.org/document/8030645/

4. Bailis, P., Davidson, A., Fekete, A., Ghodsi, A., Hellerstein, J.M., Stoica, I.: Highly available transactions: virtues and limitations. Proceedings of the VLDB Endowment 7(3), 181–192 (November 2013), https://dl.acm.org/doi/10.14778/2732232.2732237
5. Bailis, P., Ghodsi, A., Hellerstein, J.M., Stoica, I.: Bolt-on causal consistency. In: Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data. pp. 761–772. ACM, New York New York USA (June 2013), https://dl.acm.org/doi/10.1145/2463676.2465279
6. Berenson, H., Bernstein, P., Gray, J., Melton, J., O'Neil, E., O'Neil, P.: A critique of ANSI SQL isolation levels. In: Proceedings of the 1995 ACM SIGMOD international conference on Management of data - SIGMOD '95. pp. 1–10. ACM Press, San Jose, California, United States (1995), http://portal.acm.org/citation.cfm?doid=223784.223785
7. Bernstein, P.A., Goodman, N.: Multiversion concurrency control—theory and algorithms. ACM Transactions on Database Systems 8(4), 465–483 (December 1983), https://dl.acm.org/doi/10.1145/319996.319998
8. Bernstein, P.A., Hadzilacos, V., Goodman, N.: Concurrency control and recovery in database systems. Addison-Wesley Pub. Co, Reading, Mass (1987)
9. Bogner, J., Fritzsch, J., Wagner, S., Zimmermann, A.: Microservices in Industry: Insights into Technologies, Characteristics, and Software Quality. In: 2019 IEEE International Conference on Software Architecture Companion (ICSA-C). pp. 187–195. IEEE, Hamburg, Germany (March 2019), https://ieeexplore.ieee.org/document/8712375/
10. Braubach, L., Jander, K., Pokahr, A.: A novel distributed registry approach for efficient and resilient service discovery in megascale distributed systems. Computer Science and Information Systems 15(3), 751–774 (2018), http://www.doiserbia.nb.rs/Article.aspx?ID=1820-02141800030B
11. Conrad, S., Eaglestone, B., Hasselbring, W., Roantree, M., Schöhoff, M., Strässler, M., Vermeer, M., Saltor, F.: Research issues in federated database systems: report of EFDBS '97 workshop. ACM SIGMOD Record 26(4), 54–56 (December 1997), https://dl.acm.org/doi/10.1145/271074.271089
12. Cooper, B.F., Silberstein, A., Tam, E., Ramakrishnan, R., Sears, R.: Benchmarking cloud serving systems with YCSB. In: Proceedings of the 1st ACM symposium on Cloud computing. pp. 143–154. ACM, Indianapolis Indiana USA (June 2010), https://dl.acm.org/doi/10.1145/1807128.1807152
13. Cowling, J., Liskov, B.: Granola: Low-Overhead Distributed Transaction Coordination. In: 2012 USENIX Annual Technical Conference (USENIX ATC 12). pp. 223–235. USENIX Association, Boston, MA (June 2012), https://www.usenix.org/conference/atc12/technical-sessions/presentation/cowling
14. Dey, A., Fekete, A., Rohm, U.: Scalable distributed transactions across heterogeneous stores. In: 2015 IEEE 31st International Conference on Data Engineering. pp. 125–136. IEEE, Seoul, South Korea (April 2015), http://ieeexplore.ieee.org/document/7113278/
15. Docs: Amazon ec2 instance types. https://aws.amazon.com/ec2/instance-types/t3/, accessed: 2022-05-09
16. Docs: Cassandra documentation. https://cassandra.apache.org/doc/latest/, accessed: 2022-05-09
17. Docs: Elasticsearch documentation. https://www.elastic.co/guide/en/elasticsearch/reference/current/index.html, accessed: 2022-05-09
18. Docs: The go programming language. https://go.dev/doc/, accessed: 2022-05-09
19. Docs: Graphql documentation. https://graphql.org/, accessed: 2022-05-09
20. Docs: grpc documentation. https://grpc.io/, accessed: 2022-05-09
21. Docs: Open api specification. https://spec.openapis.org/oas/latest.html, accessed: 2022-05-09
22. Docs: Postgresql 13. https://www.postgresql.org/docs/13/index.html, accessed: 2022-05-09

23. Docs: Protocol buffers specification. https://developers.google.com/protocol-buffers/docs/overview, accessed: 2022-05-09
24. Docs: Rabbitmq. https://www.rabbitmq.com/, accessed: 2022-04-16
25. Docs: Zeromq. https://zeromq.org/, accessed: 2022-04-16
26. Dürr, K., Lichtenthäler, R., Wirtz, G.: An Evaluation of Saga Pattern Implementation Technologies. In: CEUR workshop proceedings. pp. 74–82 (2021), https://fis.uni-bamberg.de/handle/uniba/49721, iSSN: 1613-0073 Issue: 2839
27. Fan, G., Chen, L., Yu, H., Qi, W.: Multi-objective optimization of container-based microservice scheduling in edge computing. Computer Science and Information Systems 18(1), 23–42 (2021), http://www.doiserbia.nb.rs/Article.aspx?ID=1820-02142000041F
28. Fan, P., Liu, J., Yin, W., Wang, H., Chen, X., Sun, H.: 2PC*: a distributed transaction concurrency control protocol of multi-microservice based on cloud computing platform. Journal of Cloud Computing 9(1), 40 (December 2020), https://journalofcloudcomputing.springeropen.com/articles/10.1186/s13677-020-00183-w
29. Fielding, R.T.: Architectural Styles and the Design of Network-Based Software Architectures. PhD Thesis, University of California, Irvine (2000), iSBN: 0599871180
30. Fowler, M.: Polyglot persistence. M. Fowler website, [Online]. Available: https://www.martinfowler.com/bliki/PolyglotPersistence.html (accessed May 2022)
31. Gadepally, V., Chen, P., Duggan, J., Elmore, A., Haynes, B., Kepner, J., Madden, S., Mattson, T., Stonebraker, M.: The BigDAWG polystore system and architecture. In: 2016 IEEE High Performance Extreme Computing Conference (HPEC). pp. 1–6. IEEE, Waltham, MA, USA (September 2016), http://ieeexplore.ieee.org/document/7761636/
32. Github: eshoponcontainers github repository. https://github.com/dotnet-architecture/eShopOnContainers, accessed: 2022-05-09
33. Github: Event stream processing microservices github repository. https://github.com/kbastani/event-stream-processing-microservices, accessed: 2022-05-09
34. Github: Ftgo github repository. https://github.com/microservices-patterns/ftgo-application, accessed: 2022-05-09
35. Github: Lakeside mutual github repository. https://github.com/Microservice-API-Patterns/LakesideMutual, accessed: 2022-05-09
36. Github: Sentilo platform github repository. https://github.com/sentilo/sentilo, accessed: 2022-05-09
37. Github: Sock shop microservices demo repository. https://github.com/microservices-demo/microservices-demo, accessed: 2022-05-09
38. Github: Spring petclinic github repository. https://github.com/spring-petclinic/spring-petclinic-microservices, accessed: 2022-05-09
39. Github: Vert.x blueprint project. https://github.com/sczyh30/vertx-blueprint-microservice, accessed: 2022-05-09
40. Hasselbring, W., Steinacker, G.: Microservice Architectures for Scalability, Agility and Reliability in E-Commerce. In: 2017 IEEE International Conference on Software Architecture Workshops (ICSAW). pp. 243–246. IEEE, Gothenburg, Sweden (April 2017), http://ieeexplore.ieee.org/document/7958496/
41. Helland, P.: Life beyond distributed transactions. Communications of the ACM 60(2), 46–54 (January 2017), https://dl.acm.org/doi/10.1145/3009826
42. Helland, P.: Data on the outside versus data on the inside. Communications of the ACM 63(11), 111–118 (October 2020), https://dl.acm.org/doi/10.1145/3410623
43. Junqueira, F., Reed, B., Yabandeh, M.: Lock-free transactional support for large-scale storage systems. In: 2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops (DSN-W). pp. 176–181. IEEE, Hong Kong, China (June 2011), http://ieeexplore.ieee.org/document/5958809/

44. Kallman, R., Kimura, H., Natkins, J., Pavlo, A., Rasin, A., Zdonik, S., Jones, E.P., Madden, S., Stonebraker, M., Zhang, Y., others: H-store: a high-performance, distributed main memory transaction processing system. Proceedings of the VLDB Endowment 1(2), 1496–1499 (2008), publisher: VLDB Endowment
45. Knoche, H., Hasselbring, W.: Drivers and Barriers for Microservice Adoption – A Survey among Professionals in Germany. Enterprise Modelling and Information Systems Architectures (EMISAJ) pp. 1:1–35 Pages (January 2019), https://emisa-journal.org/emisa/article/view/164
46. Krylovskiy, A., Jahn, M., Patti, E.: Designing a Smart City Internet of Things Platform with Microservice Architecture. In: 2015 3rd International Conference on Future Internet of Things and Cloud. pp. 25–30. IEEE, Rome, Italy (August 2015), https://ieeexplore.ieee.org/document/7300793/
47. Kulkarni, S.S., Demirbas, M., Madappa, D., Avva, B., Leone, M.: Logical Physical Clocks. In: Aguilera, M.K., Querzoni, L., Shapiro, M. (eds.) Principles of Distributed Systems, vol. 8878, pp. 17–32. Springer International Publishing, Cham (2014), series Title: Lecture Notes in Computer Science
48. Laigner, R., Zhou, Y., Salles, M.A.V., Liu, Y., Kalinowski, M.: Data management in microservices: state of the practice, challenges, and research directions. Proceedings of the VLDB Endowment 14(13), 3348–3361 (September 2021), https://dl.acm.org/doi/10.14778/3484224.3484232
49. Larson, P.A., Blanas, S., Diaconu, C., Freedman, C., Patel, J.M., Zwilling, M.: High-performance concurrency control mechanisms for main-memory databases. Proceedings of the VLDB Endowment 5(4), 298–309 (December 2011), https://dl.acm.org/doi/10.14778/2095686.2095689
50. Levandoski, J.J., Lomet, D.B., Mokbel, M.F., Zhao, K.: Deuteronomy: Transaction Support for Cloud Data. In: Fifth Biennial Conference on Innovative Data Systems Research, CIDR 2011, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings. pp. 123–133. www.cidrdb.org (2011), http://cidrdb.org/cidr2011/Papers/CIDR11_Paper14.pdf
51. Li, J., Lu, Y., Zhang, Y., Wang, Q., Cheng, Z., Huang, K., Shu, J.: SwitchTx: scalable in-network coordination for distributed transaction processing. Proceedings of the VLDB Endowment 15(11), 2881–2894 (July 2022), https://dl.acm.org/doi/10.14778/3551793.3551838
52. Limon, X., Guerra-Hernandez, A., Sanchez-Garcia, A.J., Perez Arriaga, J.C.: SagaMAS: A Software Framework for Distributed Transactions in the Microservice Architecture. In: 2018 6th International Conference in Software Engineering Research and Innovation (CONISOFT). pp. 50–58. IEEE, San Luis Potosí, Mexico (October 2018), https://ieeexplore.ieee.org/document/8645853/
53. Lotz, J., Vogelsang, A., Benderius, O., Berger, C.: Microservice Architectures for Advanced Driver Assistance Systems: A Case-Study. In: 2019 IEEE International Conference on Software Architecture Companion (ICSA-C). pp. 45–52. IEEE, Hamburg, Germany (March 2019), https://ieeexplore.ieee.org/document/8712376/
54. Lykhenko, T., Soares, R., Rodrigues, L.: FaaSTCC: efficient transactional causal consistency for serverless computing. In: Proceedings of the 22nd International Middleware Conference. pp. 159–171. ACM, Québec city Canada (December 2021), https://dl.acm.org/doi/10.1145/3464298.3493392
55. M. Del Esposte, A., Kon, F., M. Costa, F., Lago, N.: InterSCity: A Scalable Microservice-based Open Source Platform for Smart Cities:. In: Proceedings of the 6th International Conference on Smart Cities and Green ICT Systems. pp. 35–46. SCITEPRESS - Science and Technology Publications, Porto, Portugal (2017), http://www.scitepress.org/DigitalLibrary/Link.aspx?doi=10.5220/0006306200350046
56. Mattern, F.: Virtual time and global states of distributed systems. Univ., Department of Computer Science (1988)

57. Mazzara, M., Dragoni, N., Bucchiarone, A., Giaretta, A., Larsen, S.T., Dustdar, S.: Microservices: Migration of a Mission Critical System. IEEE Transactions on Services Computing 14(5), 1464–1477 (September 2021), https://ieeexplore.ieee.org/document/8585089/
58. Ongaro, D., Ousterhout, J.: In Search of an Understandable Consensus Algorithm. In: 2014 USENIX Annual Technical Conference (USENIX ATC 14). pp. 305–319. USENIX Association, Philadelphia, PA (June 2014), https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro
59. Patiño-Martinez, M., Jiménez-Peris, R., Kemme, B., Alonso, G.: MIDDLE-R: Consistent database replication at the middleware level. ACM Transactions on Computer Systems 23(4), 375–423 (November 2005), https://dl.acm.org/doi/10.1145/1113574.1113576
60. Ports, D.R.K., Grittner, K.: Serializable snapshot isolation in PostgreSQL. Proceedings of the VLDB Endowment 5(12), 1850–1861 (August 2012), https://dl.acm.org/doi/10.14778/2367502.2367523
61. Pritchett, D.: BASE: An Acid Alternative: In partitioned databases, trading some consistency for availability can lead to dramatic improvements in scalability. Queue 6(3), 48–55 (May 2008), https://dl.acm.org/doi/10.1145/1394127.1394128
62. Rodríguez, C., Baez, M., Daniel, F., Casati, F., Trabucco, J.C., Canali, L., Percannella, G.: REST APIs: A Large-Scale Analysis of Compliance with Principles and Best Practices. In: Bozzon, A., Cudre-Maroux, P., Pautasso, C. (eds.) Web Engineering, vol. 9671, pp. 21–39. Springer International Publishing, Cham (2016), https://link.springer.com/10.1007/978-3-319-38791-8_2, series Title: Lecture Notes in Computer Science
63. Rudrabhatla, C.K.: Comparison of Event Choreography and Orchestration Techniques in Microservice Architecture. International Journal of Advanced Computer Science and Applications 9(8) (2018), http://thesai.org/Publications/ViewPaper?Volume=9&Issue=8&Code=ijacsa&SerialNo=4
64. Shapiro, M., Preguiça, N., Baquero, C., Zawirski, M.: Conflict-Free Replicated Data Types. In: Défago, X., Petit, F., Villain, V. (eds.) Stabilization, Safety, and Security of Distributed Systems, vol. 6976, pp. 386–400. Springer Berlin Heidelberg, Berlin, Heidelberg (2011), http://link.springer.com/10.1007/978-3-642-24550-3_29, series Title: Lecture Notes in Computer Science
65. Sheth, A.P., Larson, J.A.: Federated database systems for managing distributed, heterogeneous, and autonomous databases. ACM Computing Surveys 22(3), 183–236 (September 1990), https://dl.acm.org/doi/10.1145/96602.96604
66. Soldani, J., Tamburri, D.A., Van Den Heuvel, W.J.: The pains and gains of microservices: A Systematic grey literature review. Journal of Systems and Software 146, 215–232 (December 2018), https://linkinghub.elsevier.com/retrieve/pii/S0164121218302139
67. Thomson, A., Diamond, T., Weng, S.C., Ren, K., Shao, P., Abadi, D.J.: Calvin: fast distributed transactions for partitioned database systems. In: Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data. pp. 1–12. ACM, Scottsdale Arizona USA (May 2012), https://dl.acm.org/doi/10.1145/2213836.2213838
68. Viennot, N., Lécuyer, M., Bell, J., Geambasu, R., Nieh, J.: Synapse: a microservices architecture for heterogeneous-database web applications. In: Proceedings of the Tenth European Conference on Computer Systems. pp. 1–16. ACM, Bordeaux France (April 2015), https://dl.acm.org/doi/10.1145/2741948.2741975
69. Viotti, P., Vukolić, M.: Consistency in Non-Transactional Distributed Storage Systems. ACM Computing Surveys 49(1), 1–34 (March 2017), https://dl.acm.org/doi/10.1145/2926965

70. Wei, Z., Pierre, G., Chi, C.H.: CloudTPS: Scalable Transactions for Web Applications in the Cloud. IEEE Transactions on Services Computing 5(4), 525–539 (2012), http://ieeexplore.ieee.org/document/5740834/
71. Wu, C., Sreekanti, V., Hellerstein, J.M.: Transactional Causal Consistency for Serverless Computing. In: Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data. pp. 83–97. ACM, Portland OR USA (June 2020), https://dl.acm.org/doi/10.1145/3318464.3389710
72. Wu, Y., Arulraj, J., Lin, J., Xian, R., Pavlo, A.: An empirical evaluation of in-memory multi-version concurrency control. Proceedings of the VLDB Endowment 10(7), 781–792 (March 2017), https://dl.acm.org/doi/10.14778/3067421.3067427
73. Zhang, G., Ren, K., Ahn, J.S., Ben-Romdhane, S.: GRIT: Consistent Distributed Transactions Across Polyglot Microservices with Multiple Databases. In: 2019 IEEE 35th International Conference on Data Engineering (ICDE). pp. 2024–2027. IEEE, Macao, Macao (April 2019), https://ieeexplore.ieee.org/document/8731442/

Lazar Nikolić received his M.Sc degree from the Faculty of Technical Sciences, at the University of Novi Sad in 2016. In the same year, he enrolled in the Ph.D. program at the Faculty of Technical Sciences, at the University of Novi Sad. He was a teaching assistant until early 2023 at the Faculty of Technical Sciences, at the University of Novi Sad, where he participated in several courses in the area of Web Programming and Internet Networks. He is currently working on his Ph.D. thesis in the area of Distributed Systems, Microservice Architectures, and Transaction Management Systems.

Vladimir Dimitrieski works as an associate professor at the University of Novi Sad, Faculty of Technical Sciences, Serbia. There, he went through all the academic education levels, receiving a Ph.D. in computer science in 2018. Currently, he is a lecturer in several courses at the Faculty of Technical Sciences that cover domains of (meta-)modeling, domain-specific languages, and data engineering. With a strong background in the domain of Industry 4.0, (meta-)modeling and data engineering, he has been part of multiple national, international, and industrial projects in these domains.

Milan Čeliković received his M.Sc. degree from the Faculty of Technical Sciences, at the University of Novi Sad in 2009. He received his Ph.D. degree in 2018, at the University of Novi Sad, Faculty of Technical Sciences. Currently, he works as an assistant professor at the Faculty of Technical Sciences at the University of Novi Sad, where he lectures several Computer Science and Informatics courses. His main research interests are focused on: Databases, Database management systems, Information Systems, and Software Engineering.

Received: December 10, 2022; Accepted: November 22, 2023.