Design and Implementation of a Protocol for the SagaMAS Framework to Manage Distributed Transactions in Microservices Architecture

Samuel Lungu* and Mayumbo Nyirenda†
*Department of Computer Science University of Zambia
Lusaka, Zambia
Email: samuel.lungu@unza.zm

†Department of Computer Science
University of Zambia
Lusaka, Zambia
Email: mnyirenda@unza.zm

I. Abstract

SagaMAS is a framework that integrates multi-agent sys- tems (MAS) with microservices to coordinate long-lived dis- tributed transactions. While its original formulation laid out the architectural vision, it deferred the formal specification of its transaction protocol. This paper addresses that gap by defining the SagaMAS protocol through six core rules: initiation, commit-then-report, dependency management, time- outs and retries, compensation, and termination without global commit. The protocol is formally specified in PROMELA and verified using the SPIN model checker against safety, liveness, and responsiveness properties. Verification results confirm that SagaMAS avoids deadlocks, maintains safety, and guarantees eventual termination. By contributing a formally verified transaction protocol, this work reinforces the Saga- MAS framework and lays the foundation for future empirical benchmarking against established orchestration platforms such as Axon Framework, Eventuate Tram, and Netflix Conductor.

***Index Terms*—Distributed Transactions, Microservices, Multi-Agent Systems, Protocol**

II. Introduction

In a systematic literature review conducted by Lungu, S, and Nyirenda, M [1], the authors sought to explore the current state of distributed transaction management within Microservices Architecture(MSA), focusing on the unique challenges, strategies, and technologies utilized to overcome the identified challenges. The study considered a total of sixteen primary studies and brought out several challenges. The foremost challenge highlighted by 56.52% of the primary studies was the complexity of ensuring data consistency and integrity across multiple microservices. One creative solution to the identified challenge was the study by Limon et al entitled ***SagaMAS: a software framework for distributed transactions in the microservice architecture***, which proposed introducing a framework called SagaMAS[2] that utilizes a multi-agent system to manage distributed transactions.

Multi-agent systems(MAS) have had many successful appli- cation areas, including but not limited to modeling complex systems, smart grids, computer networks, civil engineering, and distributed artificial intelligence[3]. One of the main goals of the SagaMAS framework is to enhance user experience, making it easier for software developers to implement and manage transactions in a polyglot database environment. The central concept for the SagaMAS framework is that an in- telligent agent represents each microservice. These agents, corresponding to their respective microservices, form a multi- agent system, an independent layer as depicted in Figure 2.

The authors [2] proposed that the SagaMAS framework would be developed using the Prometheus methodology[4], a software development methodology for Multi-agent systems. The framework is still under development, with ongoing efforts to refine its design, implementation, scalability assess- ments, and performance testing. The Prometheus methodology comprises three distinct phases as depicted in Figure 1. The first phase, *system specification*, is dedicated to pinpointing the system's fundamental functions, including its inputs (per- cepts), outputs (actions), and any vital shared data sources. The second, *architectural design phase* leverages the outcomes from the system specification phase to ascertain the system's agents and their interactions. Lastly, *the detailed design phase* delves into the specifics of each agent, outlining how they will achieve their tasks within the system.

The SagaMAS framework has thus far, worked on the first phase of the Prometheus methodology, *system specification* leaving the other two phases for future work. This study seeks to contribute to the

 ISBN: 978-9982-95-500-3

SagaMAS framework by focusing on the *Architectural design phase* of the Prometheus methodology, particularly the design and implementation of a protocol for agent interaction in realizing data consistency in the manage- ment of distributed transactions.

The article is organized as follows: Section 3 furnishes the background, Section 4 delves into the implementation details, Section 6 focuses on results and discussion, and Section 9 provides the conclusion.

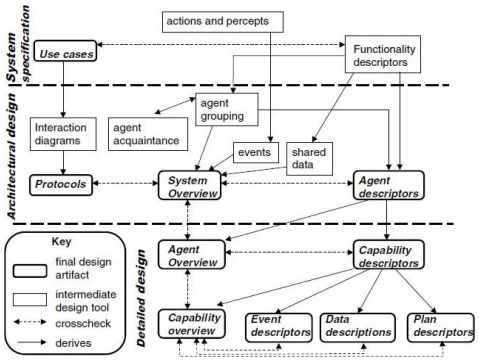

Fig. 1: Phases of Prometheus Methodology

III. Background

To effectively implement a protocol for distributed transac- tion management in the SagaMAS framework, it is crucial to revisit its foundational concepts. This section explores these concepts by referencing the original article[2], which outlines the design choices and key characteristics upon which the SagaMAS framework is built.

As outlined in the article[2], certain design and imple- mentation choices have already been made, for example, the JaCaMo[5] development framework for the Multi-Agent development stack was selected as the foundation for develop- ing the SagaMAS multi-agent architecture. JaCaMo integrates three technologies: Jason, CArtAgO, and MOISE, which are essential for building multi-agent systems. Jason is an exten- sion of the agent-oriented programming *AgentSpeak*, which itself is based on the Belief-Desire-Intention architecture lan- guage and an underlying abstract language. CArtAgO[6] is a framework and infrastructure for environment programming and execution in multi-agent systems. Finally, MOISE[7] is a framework for the organizational aspects of MAS, guiding and managing agent interactions, coordination, and shared objectives.

The article[2] further, provided key features on which the SagaMAS general model is based, these are indicated below:

i. **Agent-Based Approach:** Utilizes multi-agent system (MAS) concepts, including communication through speech acts and coordination mechanisms.
ii. **Eventual Consistency:** Agents participating in dis- tributed transactions do not require immediate global state agreement. Instead, they rely on the eventual consistency
[8] [9] to propagate state changes across services. This data persistence model is designed for greater scalability and heterogeneity.
iii. **Saga Pattern Implementation:** Aligns with the Saga pattern, which inspired the name SagaMAS.
iv. **Semi-Orchestrated Asynchronous Model:** Operates without a central control unit, as agents independently coordinate actions by requesting subsequent steps. Com- munication is asynchronous, enabling agents to send requests and respond to events reactively without waiting for immediate replies.
v. **Workflow and Transaction Modeling:** Defines a flexible transaction model to describe workflows and transaction configurations.
vi. **Error Handling Strategies:** Provides robust mechanisms for error detection, tolerance, and recovery, leveraging MAS's abstraction capabilities for efficient error resolu- tion.

Additionally, the authors proposed a transaction model based on graph theory that allows transactions to be broken down into manageable sections, each with its associated informa- tion. This model supports the configuration and representation of transactions at the agent level. The framework includes strategies for error handling and compensation workflows, ensuring that transactions can be effectively managed even during failures.

Transactions are depicted as graphs and divided into sub- transactions, comprising sequences of sections representing in- dividual steps. Each section aligns with a specific microservice action and

 ISBN: 978-9982-95-500-3

includes input and output data, compensation ac- tions for rollbacks in case of failure, and information about the next agent responsible for continuing the transaction workflow.

IV. Design and Implementation

The book *Developing intelligent agent systems: A practical guide* [10] provides a detailed guide on the Prometheus software methodology for intelligent multi-agent systems. In describing the *Architectural design phase*, the authors high- light three(3) main functions of this phase:

i. Deciding what agent types will be implemented and developing the agent descriptors.
ii. Describing the dynamic behavior of the system using interaction diagrams and interaction protocols
iii. Capturing the system's overall (static) structure using the system overview diagram.

In deciding on the agent types to be implemented as per func- tion (i) above, our abstraction specifies two agent types, i.e. an agent can be an initiator(originator) or a mere participant in a particular given transaction. The original article [2] further guides that the framework would connect each microservice to a specific agent to handle transactions, allowing the two(agent- and-microservice) to communicate directly or indirectly. A separate Multi-Agent layer manages errors, Figure 2.

By definition, a SagaMAS is a protocol where smart agents work together to complete a multi-step transaction, and if anything goes wrong, they undo the steps that have already succeeded by performing compensating transactions.

Implementing the proposed protocol within the SagaMAS framework is crucial for consistently managing distributed transactions. This section aims to transform the architectural

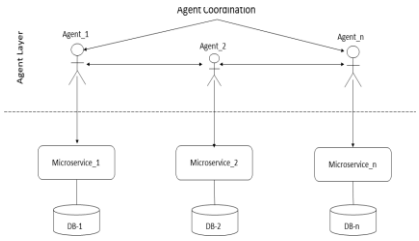

Fig. 2: SagaMAS General Model, each microservice has a database BD associated and an agent. Agents, in the independent agent layer, coordinate among themselves to perform transactions between microservices.

design of the SagaMAS framework into a functional protocol for agent interaction. By leveraging the architectural design phase of the Prometheus methodology, the implementation focuses on two key areas: defining interaction protocols and developing the dynamic behavior of agents to align with the framework's transaction model. This process includes creating interaction diagrams to illustrate agent communication and workflows, integrating error-handling mechanisms, and ensur- ing scalability and robustness. These efforts bridge the gap between theoretical design and practical application, providing a reliable foundation for managing distributed transactions in microservices architectures.

A. Protocol Design

The protocol formalizes the Saga pattern[11] using an agent- oriented approach in which each microservice is paired with an intelligent agent. These agents collaborate asynchronously to execute, monitor, and recover distributed transactions using defined message exchanges and compensation logic.

In alignment with the semi-orchestrated asynchronous model outlined in the original SagaMAS framework [2], the protocol designates a coordinator(initiator) role for each transaction. This role is assumed by the initiating agent and is specific to the transaction without necessarily serving as a centralized controller across all transactions. The Coordinator is responsible for sequencing sub-transactions, monitoring their outcomes, and initiating compensation procedures in the event of failures.

Participant agents carry out their respective local operations independently and report outcomes back to the Coordinator. This model retains the advantages of orchestration—such as structured sequencing and robust error handling—while eliminating the risk of a persistent single point of failure. By enabling loosely coupled, asynchronous interactions among autonomous agents, the protocol

 ISBN: 978-9982-95-500-3

promotes fault tolerance and ensures eventual consistency

B.]Protocol Components

A complete computer protocol specification must consist of five distinct components[12]. Each specification should explicitly define the following:

i. ***Service***: The service or services to be provided by the protocol
ii. ***Assumptions***: The assumptions about the environment in which the protocol is executed
iii. ***Vocabulary***: The vocabulary of messages used to imple- ment the protocol
iv. ***Encoding***: The encoding (format) of each message in the vocabulary
v. ***Procedure***: The procedure rules guarding the consistency of message exchanges

1) Services Provided by Sagamas: The SagaMAS protocol must implement the coordination of distributed transactions across microservices using a Multi-Agent System (MAS). It ensures:

i. Orchestration of microservice transactions via agent com- munication
ii. Eventual consistency across distributed databases
iii. Detection, tolerance, and recovery mechanisms for errors such as connectivity loss or agent shutdown
iv. Asynchronous coordination using agents to decouple mi- croservices
v. Compensation workflows for failed transaction steps

2) Assumptions : The Sagamas protocol will be based on the following assumptions about the environment:

i. Each microservice has an associated agent capable of inter-agent communication
ii. Microservices are heterogeneous, use various persistence technologies (SQL/NoSQL), and reside on distributed nodes
iii. The agent layer handles the entire transaction, while microservices expose business logic as callable actions.
iv. The network may suffer loss of connectivity, message delays, or agent process crashes, which must be tolerated and recovered from. Agents have access to logs and stateful recovery mechanisms.

3) Vocabulary: The vocabulary of a protocol defines the finite set of message types that can be sent and interpreted by the systems involved in the protocol. It defines messages types for agent transaction coordination, control, and recovery. Each message has a symbolic name and purpose:

TABLE 1: Message Types and Descriptions

Message Type	Description
START_TRANSACTION	Initiate a distributed transaction
SUBTRANSACTION_EXEC	Execute a step of a transaction
SUBTRANSACTION_SUCCESS	Notify completion of a step
SUBTRANSACTION_FAILURE	Notify a failed step
ROLLBACK	Trigger compensation flow
ROLLBACK_COMPLETE	Notify compensation is done

ISBN: 978-9982-95-500-3

TABLE 2: Standard Fields in Agent-to-Agent Messages

Field	Type	Description
type	String	The symbolic name of the message (e.g., SUBTRANSACTION_EXEC)
transactionId	String	A unique identifier for the distributed transaction
fromAgent	String	The sending agent's identifier
toAgent	String	The intended receiving agent's identifier
payload	Object	A flexible, nested structure containing domain-specific data

4) Encoding: SagaMAS uses distinct encodings for inter-

```
{
  "status": "SUCCESS",
  "transactionId": "T123",
  "data": {
    "receiptNo": "R-90811",
    "postedAt": "2025-08-11T09:12:03Z"
  },
  "error": null
}
```

vices. JSON's support for hierarchical data structures makes it well-suited for representing complex interactions, and its broad compatibility allows seamless parsing by agents imple- mented in Java, Python, and other widely used programming languages. Moreover, maintaining a consistent JSON structure streamlines logging, enables message replay, and simplifies the development of test harnesses

Listing 1: Example of JSON-encoded SUBTRANSACTION_ EXEC message

```
{
  "type": "SUBTRANSACTION_EXEC",
  "transactionId": "T123",
  "fromAgent": "agent_deposit",
  "toAgent": "agent_finance",
  "payload": {
    "studentId": 1001,
    "amount": 2000
  }
}
```

Listing 2: Microservice → Agent: standard response

b) Agent–Agent (Jason Speech Acts).: Inter-agent com- munication uses Jason's AgentSpeak messaging primitives (speech acts) rather than REST. Performatives (e.g., tell, achieve, askOne) carry the semantics of the protocol (e.g., subtransaction success/failure, compensation). For schema consistency, the *content* of a message may embed a JSON string, but the transport/envelope is Jason's message layer.

Listing 3: Agent-to-Agent transaction in Jason

```
+!handle_deposit_response("success") <-
    .send(coordinator_agent, tell,
      subtransaction_success(deposit_service)).
```

5) Procedure: As outlined in the protocol components section, a procedure refers to a set of rules that govern the consistency of message exchange. These rules function as a grammar, constraining which interaction sequences are valid based on the defined vocabulary, message formats (syntax), and service semantics [12]. Since these rules are interpreted *concurrently* by interacting processes, they must be expressed *unambiguously* using a formal notation rather than informal time-sequence diagrams.

To ensure this level of precision and enable rigorous validation under concurrency, the SagaMAS procedural rules are specified as a *validation model* using PROMELA (Process Meta Language), a formal language designed for protocol modeling

 ISBN: 978-9982-95-500-3

and verification [12]. The PROMELA model[1] is provided in Appendix A.

The remainder of this section outlines the high-level proce- dural steps and the corresponding algorithms.

a) SagaMAS procedure rules (summary).:
We formalize SagaMAS with the following core rules:

P1. Initiation & roles. The initiating agent assumes the **Coordinator** role for the transaction and notifies all participants with START_TRANSACTION(tr_id).

P2. Commit-then-report. Each participant executes and *commits locally* its sub-transaction (against its microservice) before sending SUBTRANSACTION_SUCCESS to the Coordinator; on error it sends SUBTRANSACTION_FAILURE. (Eventual consistency.)

P3. Dependencies. Dispatch respects the configured depen- dency graph; parallel branches are allowed when de- clared; otherwise sequencing is enforced.

P4. Timeouts & retries. If a participant does not report within Δ, the Coordinator treats it as failure and proceeds to compensation; retries are idempotent.

P5. Compensation. On the first failure (or timeout), the Coordinator triggers COMPENSATE in *reverse order* of completed sub-transactions, collecting COMPENSATION_DONE acknowledgments.

P6. No global commit. If all participants have reported success, the Coordinator records success and publishes results to end users/monitoring—no final commit mes- sage is sent to participants.

[1]GitHub repository: https://github.com/samuellungu/sagamas.

 ISBN: 978-9982-95-500-3

The above six(6) rules are well illustrated by the three(3) algorithms in the subsequent paragraphs.

ALGORITHM 1:

Transaction initialization: bind the initiator as Coordinator, creates a fresh tr_id, informs all participants, and persists the set of sub-transactions $\{t_1, \ldots, t_k\}$ with their actions and dependencies. It does *not* execute work yet—only prepares authoritative metadata used by the Coordinator loop and for recovery.

Algorithm 1: Transaction Initialization and Configuration

Input: Request from Microservice M, set of participating agents $A = \{A_1, A_2, \ldots, A_n\}$
Output: Configured transaction metadata shared among all agents

1 STEP 1: TRANSACTION START
2 $agent_M \leftarrow$ agent associated with microservice M
3 $tr\_id \leftarrow$ generateUniqueTransactionId()
4 $agent_M$ records tr_id in its local transaction log
5 **foreach** $A_i \in A$ **do**
6 **if** $A_i \neq agent_M$ **then**
7 $agent_M$ sends START_TRANSACTION(tr_id) to A_i
8 STEP 2: CONFIGURATION SETUP
9 Define sub-transactions $T = \{t_1, t_2, \ldots, t_k\}$
10 **foreach** $t_i \in T$ **do**
11 $t_i.incoming\_action \leftarrow$ operation to execute
12 $t_i.compensation\_action \leftarrow$ rollback operation if t_i fails
13 $t_i.next \leftarrow$ list of dependent sub-transactions (sequential or parallel)
14 $agent_M$ updates local transaction store with configuration T

Algorithm 2: Coordinator-led Execution under Eventual Consistency (no global commit)

Input : Coordinator agent $agent_M$ (initiator), participants $A = \{A_1, \ldots, A_n\}$, payload P, timeout Δ
Output: SUCCESS (notify end user) or ABORT (after compensation)

1 **Start**
2 $tr\_id \leftarrow$ generateUniqueTransactionId();
3 $log.append(tr\_id,$ started);
4 $awaiting \leftarrow A$;
5 $succeeded \leftarrow \langle\,\rangle$ (ordered);
6 **foreach** $A_i \in A$ **do**
7 **send** START_TRANSACTION(tr_id) **to** A_i
8 CONFIGURE (LOCAL)
9 Define $T = \{t_1, \ldots, t_k\}$ with t_j.incoming_action, t_j.compensation_action, t_j.next;
10 $txStore[tr\_id] \leftarrow T$
11 DISPATCH (COMMIT-THEN-REPORT)
12 **foreach** $A_i \in A$ **do**
13 **send** SUBTRANSACTION_EXEC($tr\_id, P,$ step = default) **to** A_i
14 **Collect**
15 **while** $awaiting \neq \emptyset$ **do**
16 **if** ***receive*** m *from some* A_j *within* Δ **then**
17 **if** $m.type = SUBTRANSACTION\_SUCCESS$ THEN
18 $awaiting \leftarrow awaiting \setminus \{A_j\}$;
19 **append** A_j to $succeeded$;
20 **else if** $m.type = SUBTRANSACTION\_FAILURE$ THEN
21 **goto** Compensation
22 **else**
23 **goto** Compensation

Notes on Algorithm 1. (i) **Uniqueness & durability**: per- sist tr_id and T before any dispatch. (ii) **Dissemination**: START_TRANSACTION is a notification; execution requests come later. (iii) **Determinism**: T is immutable for the life of tr_id to keep verification and recovery simple.

 ISBN: 978-9982-95-500-3

ALGORITHM 2

The Coordinator's runtime control loop dispatches work, col- lects SUCCESS/FAILURE, and on any failure/timeout triggers compensation in reverse completion order. If everyone reports success, the Coordinator records completion and *publishes* results—no global commit message is sent.

```
24 FINALIZE SUCCESS
25  log.append(tr_id, completed);
26  publish
      TRANSACTION _ SUCCEEDED(tr _ id,
      results) to end-user/monitoring;
27  return SUCCESS;
28 COMPENSATION
29  for k ← |succeeded| to 1 do
30      S ← succeeded[k];
31      send COMPENSATE(tr_id) to S;
32  foreach S ∈ succeeded do
33      wait for COMPENSATION_DONE(tr_id) from S
            up to Δ
34  log.append(tr_id, aborted);
35  publish TRANSACTION_ABORTED(tr_id);
36  return ABORT;
```

Notes on Algorithm 2. (i) **Delivery semantics**: treat messages

 ISBN: 978-9982-95-500-3

as at-least-once; handlers must be idempotent (keyed by tr_id and step). (ii) **Timeouts**: Δ models bounded waiting; on expiry, prefer compensation to avoid indefinite blocking.

(iii) **Parallelism**: if T encodes dependencies, dispatch only *ready* steps; unlock dependents when predecessors report SUCCESS.

ALGORITHM 3

Defines the Participant's local rule: execute and *commit* the sub-transaction against its microservice, then report the outcome. This encodes eventual consistency (local commit precedes notification).

Algorithm 3: Participant A_i on SUBTRANSACTION_EXEC(tr_id, P)

On Receive: SUBTRANSACTION_EXEC(*tr_id, P,* step)
execute local incoming_action **and COMMIT** DURABLY;
if *commit OK* **then**
  send SUBTRANSACTION_SUCCESS(*tr_id,* output) to Coordinator
else
  send SUBTRANSACTION_FAILURE(*tr_id,* reason) to Coordinator

Notes on Algorithm 3. (i) **Idempotency**: incoming_action and compensation_action should be idempotent (or guarded by a transactional outbox/inbox). (ii) **Crash safety**: persist local state before replying; after restart, resend the last outcome for tr_id.

V. PROTOCOL VERIFICATION

Protocol verification is essential for ensuring the correct- ness, completeness, and logical consistency of a commu- nication protocol. Traditionally, two primary techniques are employed: simulation and testing, and formal verification[13]. The primary drawback of simulation and testing lies in their reliance on fixed network size and topology, which limits their ability to explore all possible execution scenarios of a distributed algorithm. As a result, certain faults may remain undetected during testing and could lead to critical failures during real-world operations[13]. In contrast, formal verification techniques offer more rigorous validation and are generally categorized into:

i. **Model Checking with SPIN**: performs exhaustive behavioral validation through a formal Promela model. However, its practical application is often hindered by the state space explosion problem, where the number of system states grows exponentially with complexity[14].

ii. **Theorem Proving**: This approach relies on mathematical reasoning to verify that a system model satisfies its specified requirements. It has been successfully applied in safety- and security-critical domains such as nuclear power plants, aerospace systems, and railway infrastruc- ture [15].

\Given that the SagaMAS framework is not intended for safety or security-critical applications, its protocol is verified using the SPIN model checker. Additionally, SagaMAS adopts eventual consistency, which helps reduce the number of concurrent states that need to be explored. This design choice mitigates the state explosion issue, making SPIN a natural and effective verification method for the framework.

A. SPIN Model Checker and Its Verification Capabilitie

SPIN (Simple Promela INterpreter) is a formal verification tool specifically designed to analyze the logical consistency of concurrent systems, with a particular focus on distributed software systems and communication protocols [16], [17]. It offers a comprehensive suite of verification features, including deadlock detection, assertion checking, and the verification of Linear Temporal Logic (LTL) properties[18].

SPIN utilizes Promela (Process Meta Language) as its dedicated modeling language and supports the verification of the following key properties:

i. ***Safety Properties:*** These ensure that undesirable events ("bad things") do not occur during execution. Examples include mutual exclusion, deadlock freedom, partial correctness, and first-come-first-serve behavior[19].
ii. ***Liveness Properties:*** These guarantee that desirable events ("good things") eventually occur. Examples include:
   - Starvation freedom: Ensures that a process makes progress inifinitely often.
   - Termination: Confirms that a program does not run indefinitely and eventually completes.
   - Guaranteed Service: Verifies that every service request is eventually fulfilled[19]
iii. ***Compensation Completeness:*** SPIN can verify that all sub-transactions are either aborted or completed, but never both simultaneously, ensuring transactional in- tegrity.
iv. ***Performance Indicators:*** The tool also tracks metrics such as memory usage and the number of system states explored, providing insights into the efficiency and scalability of the model.

VI. RESULTS AND DISCUSSION

Overall, the SPIN model provides simulation-based validation of the deductive properties, strengthening confidence in the protocol's design correctness and fault resilience.

A. Safety Properties

Safety properties were captured through explicit assertions in the Promela model. These checks ensured that the system never

 ISBN: 978-9982-95-500-3

commits and aborts simultaneously, compensation is only triggered for participants that had successfully executed subtransactions, and no compensations occur in a successful commit path. Table 4 summarizes the safety verification re- sults.

The results indicate that the SagaMAS protocol preserves strong consistency guarantees with respect to compensation

 ISBN: 978-9982-95-500-3

TABLE 4: Safety verification results for the SagaMAS protocol

Property	Spin Check	Result
No invalid mixed commit/abort	assert(!(g_completed && g_aborted))	Satisfied
Compensation only after success	assert(!compensated[i] || succeeded[i])	Satisfied
No compensation on commit	assert(!compensated[i]) when committed	Satisfied
Saga must terminate in a decided state	assert(g_completed || g_aborted)	Satisfied

safety and transactional decision-making. No safety violations were observed across the explored state space.

B. Liveness Properties

Liveness properties were expressed in Linear Temporal Logic (LTL) and verified against the model. These prop- erties confirmed that the protocol always progresses, even- tually reaches a termination state (either commit or abort), and ensures that all participants are notified of the termina- tion through the DONE message. Additionally, per-participant progress was verified: every participant that is dispatched eventually responds to the Coordinator. Table 5 summarizes the liveness verification outcomes.

These results confirm that the SagaMAS protocol not only guarantees safety under all possible interleavings but also ensures that the system does not get stuck: every execution path eventually reaches a global decision and notifies all participants.

C. Verification Statistics

Spin also provides performance statistics during state-space exploration, which help quantify the verification efficiency. Table 6 reports the most relevant metrics, including state- vector size, maximum depth reached, number of stored states, memory footprint, and runtime.

The results show that the verification process is lightweight, completing within one second with negligible memory require- ments, further demonstrating the scalability of the protocol model.

TABLE 6: Spin verification statistics for the SagaMAS protocol

Metric	Value
State vector size	112–136 bytes
Maximum search depth	162
Stored states	60–106
Hash conflicts	0
Verification runtime	< 1 second
Memory usage	< 1 MB

The SPIN verification results highlight that the SagaMAS protocol achieves a robust balance between safety and live- ness guarantees. The safety checks confirm that the protocol

enforces consistent decision-making: no invalid mixed com- mit/abort states occur, compensations are only applied when sub-transactions succeed, and no compensation is triggered in successful commit paths. This ensures that SagaMAS adheres to its transactional integrity requirements even under complex interleaving. The liveness analysis, expressed in LTL, further demonstrates that the system cannot deadlock or stall; all transactions eventually terminate in a well-defined state and every dispatched participant reliably responds to the Coordi- nator. These properties strengthen the claim that SagaMAS is resilient against both safety violations and indefinite blocking. Finally, the verification statistics reveal that the model is efficient to analyze, with small state vectors, shallow search depths, zero hash conflicts, and runtimes under a second. This indicates that the protocol can be formally validated with modest computational resources, making it scalable for iterative refinement. Collectively, the results provide strong evidence that the SagaMAS protocol is both correct by design and pragmatic.

VII. Future Work

As future work, we plan a systematic performance study that compares against established orchestration frame- works—*Axon Framework*[2], *Eventuate Tram*[3][20], and *Net- flix Conductor*[4]—along metrics such as end-to-end latency, throughput under load, compensation overhead, recovery time under crashes/timeouts, and scalability with the number of participants. We will release workloads, scripts, and datasets to enable reproducible results, and explore deployment aspects (e.g., Kubernetes, OpenTelemetry tracing) and fault models (message loss, duplication, and partitioning) to complement the formal verification with real-world evidence.

VIII. Conclusion

This paper presented, a semi-orchestrated commit-then- report saga protocol for multi-agent microservice transactions. We specified the procedure rules formally in PROMELA and

[2]https://axoniq.io
[3]https://eventuate.io/abouteventuatetram.html
[4]https://netflixtechblog.com/netflix-conductor-a-microservices-orchestrator-2e8d4771bf40

 ISBN: 978-9982-95-500-3

TABLE 5: Liveness verification results for the SagaMAS protocol

LTL Property	Meaning	Result
<>reached progress	Protocol always makes progress	Satisfied
<>(g completed ∨ g aborted)	Saga always terminates	Satisfied
<>(reached progress ∧ (g completed ∨ g aborted))	Termination coincides with progress	Satisfied
<>(shutdown count = N)	All participants eventually receive DONE	Satisfied
□ (dispatched[i] → <>answered[i])	Every dispatched participant eventually responds	Satisfied

used SPIN to check core safety and liveness obligations, including termination, absence of conflicting concurrent ex- ecutions, and compensation correctness. The design separates agent–agent coordination from agent–microservice execution (REST+JSON), and avoids a global commit by favoring local commits with compensation, aligning with eventual consis- tency.

.

[1]

.

While SagaMAS provides a promising agent-based protocol for distributed transactions, its current evaluation is limited to formal verification. Performance benchmarks, real-world deployment tests, and comparative analysis with existing frameworks are still needed to confirm its robustness and practical value.

 ISBN: 978-9982-95-500-3

APPENDIX A
PROMELA MODEL FOR SAGAMAS

Listing A.4: Complete SagaMAS PROMELA model

```
#ifndef N
#define N 3
#endif

mtype = { START_TRANSACTION, SUBTRANSACTION_EXEC, SUCCESS, FAILURE,
          COMPENSATE, COMP_DONE, DONE, DONE_ACK };

/* channels */
chan c_to[N]   = [2] of { mtype, byte };
chan c_from[N] = [2] of { mtype, byte };

/* ---- Global propositions & bookkeeping ---------- */
bool reached_progress = false;   /* set at progress point */
bool g_completed      = false;   /* set true on commit path */
bool g_aborted        = false;   /* set true on failure/compensation path */

bool dispatched[N];              /* Coordinator sent SUBTRANSACTION_EXEC to i */
bool answered[N];                /* Coordinator received SUCCESS/FAILURE from i */
bool succeeded[N];               /* Coordinator received SUCCESS from i */
bool compensated[N];             /* Coordinator observed COMP_DONE for i during abort */
bool shutdown_seen[N];           /* Participant i observed DONE */
byte shutdown_count = 0;         /* count of participants that saw DONE */

/*                Participant                */
proctype Participant(byte id)
{
    byte tr;
end_wait:
    do
    :: c_to[id]?SUBTRANSACTION_EXEC, tr ->
        if
        :: c_from[id]!SUCCESS, tr
        :: c_from[id]!FAILURE, tr
        fi
    :: c_to[id]?COMPENSATE, tr ->
        c_from[id]!COMP_DONE, tr
    :: c_to[id]?DONE, tr ->
        if
        :: !shutdown_seen[id] ->
            shutdown_seen[id] = true;
            shutdown_count++
        :: else -> skip
        fi;
        /* Send ACK and terminate */
        c_from[id]!DONE_ACK, tr;
        break
    od
}
```



```
/*                Coordinator                */
proctype Coordinator()
```

 ISBN: 978-9982-95-500-3

```
52  {
53      byte i = 0;
54      byte tr = 1;
55      bool aborted = false;  /* local; mirrored by g_aborted when set */
                                                                    56
57      /* Initialize bookkeeping (explicit zeroing) */
58      i = 0;
59      do
60      :: i < N ->
61              dispatched[i] = false;
62              answered[i]           = false;
63              succeeded[i]  = false;
64              compensated[i]= false;
65              shutdown_seen[i] = false;
```

 ISBN: 978-9982-95-500-3

 ISBN: 978-9982-95-500-3

```
                                                                    d */
                                                                    do
i
+
+ :: (i < N && !aborted) -> dispatched[i] = true;
        c_to[i]!SUBTRANSACTION_EXEC, tr;
:                                                                     do
:
e           :: c_from[i]?SUCCESS, tr ->
l                 answered[i]  = true;
s               succeeded[i] = true; i++;
e                       break
            :: c_from[i]?FAILURE, tr ->
-               answered[i]  = true; aborted
>                                   = true;
b             g_aborted           = true; break
r                                                                     od
e
a       :: else -> break od;
k
o                                                                   if
d                                                                :: aborted ->
;                                                             /* compensate successful ones in reverse order */
i                                                                   do
=                                                                :: i > 0 ->
0                                                                     i--;
;                                                                     if
                        :: succeeded[i] ->
                      c_to[i]!COMPENSATE, tr;
/                     c_from[i]?COMP_DONE, tr;
*                       compensated[i] = true
s                   :: else -> skip fi
e           :: else -> break od
q                                                                  :: else ->
u                                                                g_completed = true
e                                                                   fi;
n
t                                                                progress:
i                                                                   atomic {
a                                                               reached_progress = true;
l                                                                /* broadcast DONE */
s                                                                   i = 0;
u                                                                   do
b                                                               :: i < N -> c_to[i]!DONE, tr; i++
t
r           :: else -> break od                                     }
a
n
s                                                             /* wait for DONE_ACK from all participants before exiting */
a                                                                   i = 0;
c                                                                   do
t                                                               :: i < N ->
i       c_from[i]?DONE_ACK, tr; i++
o   :: else -> break od;
n
s
,                                                          /* ---- Safety assertions at saga end---------------------------- */
s                                                       assert(g_completed || g_aborted);                    /* must decide */
t                                                         assert(!(g_completed && g_aborted));                /* not both */
o                                                                /* Compensation implies prior success */
p
w
h
e
n
a
b
o
r
t
e
```

 ISBN: 978-9982-95-500-3

```
                                                                    i = 0;
                                                                    do
                                                                    :: i < N ->
                    assert(!compensated[i] || succeeded[i]); i++
        :: else -> break od;
                                                /* If committed, then nobody should have been compensated */
                                                                    if
    :: g_completed -> i = 0;
                                                                        do
                                                                    :: i < N -> assert(!compensated[i]); i++
                :: else -> skip od
        :: else -> skip fi;


                                                /* Ensure all participants observed DONE */
                                                    assert(shutdown_count == N);

------------------      ------------------

}

----------------------      ----------------------
```


ISBN: 978-9982-95-500-3

```
/*                          init                          */
                          init {
byte i = 0; printf("N=%d\\n",
        N); atomic {
                                                  do
                                          :: i < N -> run Participant(i); i++
    :: else -> break od;
                                                  run Coordinator();
                                                      }
                                                  }

/*                          LTL PROPERTIES                          */
                          /* 1) Eventually reach progress (broadcast DONE) */
                                  ltl reach { <> reached_progress }

                          /* 2) Eventually terminate (commit or abort reached) */
                                  ltl termination { <> (g_completed || g_aborted) }

                          /* 3) Combined: eventually progress AND terminated */
                                ltl both { <> (reached_progress && (g_completed || g_aborted)) }

                          /* 4) Everyone eventually sees DONE */
                                  ltl all_done { <> (shutdown_count == N) }

                          /* 5) Per-participant: if dispatched then eventually answered */
                                                  #if N > 0
ltl answered_0 { [] (dispatched[0] -> <> answered[0]) } #endif
                                                  #if N > 1
ltl answered_1 { [] (dispatched[1] -> <> answered[1]) } #endif
                                                  #if N > 2
ltl answered_2 { [] (dispatched[2] -> <> answered[2]) } #endif
                                                  #if N > 3
ltl answered_3 { [] (dispatched[3] -> <> answered[3]) } #endif
                                                  #if N > 4
ltl answered_4 { [] (dispatched[4] -> <> answered[4]) } #endif
                                                  #if N > 5
ltl answered_5 { [] (dispatched[5] -> <> answered[5]) } #endif
```

 ISBN: 978-9982-95-500-3

```
#if N > 6

ltl answered_6 { [] (dispatched[6] -> <> answered[6]) }
#endif
#if N > 7

ltl answered_7 { [] (dispatched[7] -> <> answered[7]) }
#endif
```

 ISBN: 978-9982-95-500-3

 ISBN: 978-9982-95-500-3