Distributed Simulation with Efficient Fault Tolerance

Javier Vela[1], Unai Arronategui[2], José Ángel Bañares[2(✉)], and José Manuel Colom[2]

[1] Instituto Pirenaico de Ecología, Consejo Superior de Investigaciones Científicas (IPE-CSIC), Zaragoza, Spain
jvela@ipe.csic.es
[2] Universidad de Zaragoza, Zaragoza, Spain
{unai,banares,jm}@unizar.es

Abstract. Fault tolerance is essential for the correct execution of large distributed simulations of discrete event systems, as the likelihood of faults increases with the size of the cloud infrastructure used. Achieving optimal performance and cost in a fault-tolerant distributed simulation remains a challenge. In this paper, we propose a replication-based approach in a conservative distributed simulation strategy that is specifically designed to minimize latency introduced by fault tolerance mechanisms. Unlike traditional replication methods, our method is tailored for conservative simulation, leveraging simulation messages and timing to maintain consistency while decoupling replica execution. As a result, our approach reduces the need for messaging and synchronization and maintains eventual consistency windows with low latency overhead, achieving near-nominal simulation performance in the absence of faults. If replicas have similar performance, memory usage can be lower compared to optimistic approaches, and recovery can be fast following a node failure, despite asynchronous replication. Experimental results show that without faults, the performance of a distributed simulator with fault management is similar to one without it. Recovery from a fault reveals that the main overhead is in replica provisioning, with minimal overhead for synchronization.

Keywords: Distributed Simulation · Petri Nets · Fault Tolerance · Cloud

1 Introduction

Discrete Event System (DES) simulation is a fundamental tool for analysing, predicting, and designing systems across various domains. For large and complex systems, distributed simulation becomes essential, as it allows for scalable and efficient analysis. However, as the number of computational resources in distributed simulations increases, faults become unavoidable, especially in long-running simulations. Faults can result in the loss of the simulation state, requiring fault-tolerant mechanisms to preserve data integrity and ensure successful simulation finalization.

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2025
M. Naldi et al. (Eds.): GECON 2024, LNCS 15358, pp. 261–274, 2025.
https://doi.org/10.1007/978-3-031-81226-2_23

The *Cloud* has proven to be an effective platform for distributed simulations [15]. The Cloud offers the ability to scale resources according to model size and dynamically adjust computing as needed during the simulation. Despite these advantages, the adoption of distributed simulation of DESs in industrial and commercial applications remains limited [9,10]. The Cloud's pay-as-you-go model requires efficient solutions that can guarantee the success of the simulation at a limited cost, adding a layer of complexity that hinders industrial adoption.

In distributed simulations of DESs, the success and performance of the simulation depend on each model partition executed on each node. Additionally, the overall processing speed is determined by the slowest simulator in the network. Resilience is crucial for the successful execution of large-scale distributed simulations, as faults become more common with the expansion of cloud infrastructure. Therefore, any fault tolerance mechanism must introduce minimal overhead during normal operations to ensure cost-effectiveness within the Cloud's pay-as-you-go model.

Key factors such as data storage performance, computing processing rates, network latencies, and dynamic model partitioning for load balancing are critical in distributed simulations. As pointed by Ferscha et al. [9], the complexity of these parameters makes it challenging to rely solely on analytical methods to select the optimal simulation strategy, without detailed analysis. The choice of causality consistency protocols is model-dependent and requires extensive parameter evaluation. Performance data mining and statistical analysis are recommended to determine the best approach for specific models and execution environments.

In cloud environments, fault tolerance techniques are a hot topic, focusing on proactive and reactive approaches [14] to predict failures using machine learning and artificial intelligence. A key objective of these techniques is to maintain low overhead during fault-free operations. With this requirement in mind, our work proposes a replica-based approach for large-scale simulations in the Cloud. Our approach is similar to other replica-based fault-tolerance mechanisms proposed for the Cloud [17], but it uniquely employs simulation messages and timing to maintain replica consistency and uses the model and simulation engine to recover consistency efficiently without needing to store large amounts of state data. This innovation reduces overhead and enhances fault recovery, making it a significant advancement over traditional methods.

The remainder of this paper is organized as follows: Sect. 2 summarizes the context of our previous works on distributed simulation, Sect. 3 presents the main assumptions taken into account and the replication model-based approach, Sect. 4 shows that minimum overhead is introduced by the proposed fault-tolerance mechanism in the experimental results, Sect. 5 briefly presents related work, and Sect. 6 provides some final remarks.

2 Background: Efficient Distributed Simulation of Petri Nets

The replication fault tolerance method in this study is based on the conservative strategy for distributed simulation. This section presents the fundamental concepts of the conservative approach and explains why it is central to our methodology.

Distributed simulation enhances both the execution speed and scalability when analysing complex models. However, managing causality constraints in the model poses a challenge. Our previous work focused on using *Petri Nets* as the core formalism within a *Model-Driven Engineering* approach to leverage the model at every stage of the DES lifecycle. This framework addresses challenges such as bridging the gap between model specification, code deployment in distributed simulators, and simulation load balancing [3,4].

In distributed simulation, the model is divided into partitions that are simulated on different nodes. Each partition corresponds to a *Logical Process* (LP), which performs tasks related to its assigned portion of the model and interacts with other LPs through message exchange. The model partitioning is defined at compilation time, and LPs cannot be changed dynamically. In our approach, LPs act as simulator engines executing Petri Nets, interpreting the *Linear Enabling Function* (LEF)-coded transitions specific to their partition. The LEF function defines whether transitions in the system can be triggered, based on the state of the Petri Net, reducing the time needed to determine enabled transitions and the size of data structures representing the system's subnetwork. Therefore, dynamic workload balancing between LPs can be done by redistributing their LEF-coded transitions between them [11].

We refer to the micro-kernels (LPs) in our distributed simulation as *simbots*. Each simbot operates independently, with its own clock, and is connected to others via a communication network. They execute partitions of the overall Petri Net model, which contain transitions that either originate from or are directed to other partitions. The set of all simbots in the simulator completes the Petri Net model of the DES.

To maintain causality, distributed simulation employs two primary protocols: conservative and optimistic. *Conservative protocols* use null messages to inform neighbouring model partitions about the simulation time that can advance without causing causality errors, such as receiving an event with an earlier timestamp than the current simulation time. This approach can lead to idle periods in distributed simulators waiting for events from others. *Optimistic protocols* allow the simulation to proceed, with the capability to roll back to a previous state if a causality error occurs. An *Ideal Simulation Protocol* (ISP) [12] was introduced to compare these methods. The idea is to use prior simulations to compute the *lookahead*, a lower bound for the *Local Virtual Time* (LVT) that an LP communicates to its neighbours, enabling safe advancement. While ISP serves as the optimal efficiency reference that can be achieved, it is impractical as it does not consider any overhead communications between LPs.

While our approach cannot eliminate communication overhead, exploiting the Petri Net model allows us to obtain precise lookahead and minimize waiting times. Consequently, we adopted a conservative strategy. Automating Petri Net analysis using software tools for optimal model partitioning and estimating lookahead is crucial for accelerating simulation on distributed platforms [5]. Murata and Wu [13] demonstrated that synchronic distances in a Petri Net, which measure the degree of mutual dependence between occurrences of two transitions, can be used for synchronization in distributed processing systems and can be used to compute when events will be delivered to neighbouring LPs.

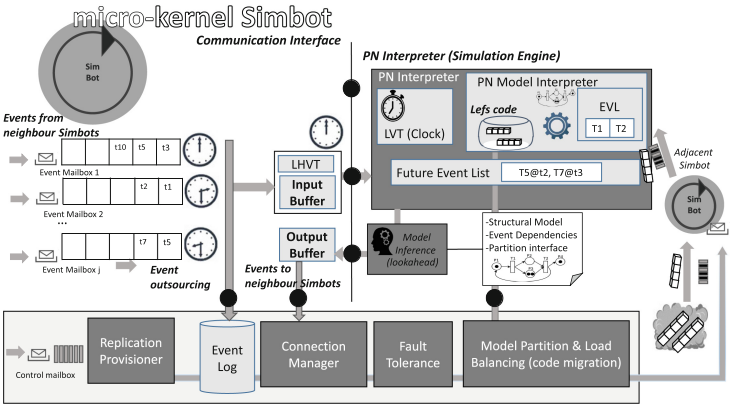

Fig. 1. Simbot Architecture

Figure 1 illustrates the architecture of a simbot as an LP using a conservative approach. Initially, the simbot calculates enabled transitions in the *Event List* (EVL). Firing these transitions produces future events stored in the *Future Event List* (FUL). Events can be internal or updates that must be communicated over the network to the affecting adjacent simbots. The simbot processes FUL events only when it is guaranteed that no events with an earlier execution time will arrive from other simbots. Each event has a timestamp indicating its occurrence time, ensuring that the simulation proceeds accurately and in order. In LEF-based transition encoding, each transition in the Petri Net has an associated value and a duration. When an event occurs, an integer updating factor is added to the LEF value. If the resulting LEF value is zero or negative, the transition becomes enabled and is added to the event list. The event's timestamp is determined by the current simulation time plus the transition's duration.

A simbot synchronizes with others via a *Communication interface*, which includes a queue of incoming messages from other simbots, ordered by timestamp. Each input queue has a timestamp field showing the timestamp of the queue's front event or the last received message if empty. The *Local Horizon Time* (LHT) is the minimum of all queue fronts, indicating the latest point to which the local clock can advance without inconsistencies. The simbot interleaves

events from its queue (FUL) with incoming message events up to the LHT, processing the earliest events first. Events are processed until the LHT is reached. Continuing the simulation beyond the LHT could result in receiving an event with an earlier timestamp, leading to inconsistencies in the simulation. If an LP's input queue is empty, the LP must wait for new messages. This mechanism can lead to deadlock, so simbots send empty messages, called null messages, with a timestamp indicating the lookahead to neighbours. The lookahead specifies the future time when it guarantees no events will be sent.

On the left, Fig. 1 shows the Communication interface responsible for maintaining message consistency by computing the LHT. On the right, there is the Petri Net interpreter used for processing events. At the bottom, the figure displays basic services, including communication, an event repository, load balancing, and fault tolerance, which will be explained in subsequent sections.

3 A Performant Simulation Approach with Fault Tolerance

The approach proposed in this paper is based on the following assumptions: 1) only crash failures are considered, meaning nodes stop working; 2) communication of events and lookahead between simbots is ordered and reliable; 3) each node runs only one simbot, so if a node fails, its corresponding simbot also fails.

A replicated state machine model, implemented within a replica group, processes all events in each simbot. Each replica of a simbot processes events in the same sequence from the same initial state, which justifies our preference for a conservative approach. Although our mechanism currently assumes a leader and a single replica per replica group, it can be easily extended by adding additional replicas to enhance robustness.

In the non-fault-tolerant operation configuration, a simbot receives event messages from its predecessors, which are the simbots simulating the subnetworks with transitions leading to it, and sends event messages to its successors, the simbots simulating the subnetworks with transitions originating from it.

In contrast, in the fault-tolerant configuration, only the leader within each simbot replica group sends event messages to its successor simbots to prevent them from receiving duplicate messages. However, all replicas of a simbot receive messages from predecessor simbots, ensuring that in the event of a crash within a replica group, one of the replicated simbots maintains the complete and correct state of the simulation. These connections between neighbouring simbots are illustrated in Fig. 2. In summary, all replicas of a simbot run asynchronous, complete simulation steps using received and local events to produce new events as nominal simbots, but only leaders are allowed to send events and lookaheads to successor simbots.

If a replica fails, the leader requests a replacement to keep the simulation's integrity. However, if the leader fails, the replica is promoted to leader and continues the simulation with a new provisioned replica. In either case, synchronization is needed to ensure all replicas remain consistent. The subsequent sections

describe the mechanisms used to maintain consistency between replicas both before and after a crash, as well as how to detect and recover from crashes.

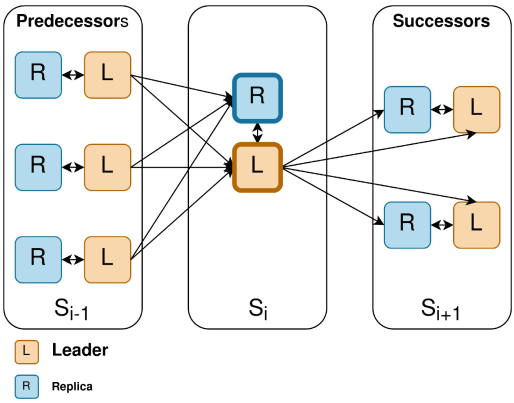

Fig. 2. Distributed Relations of Replicated Simbots

3.1 Decoupled Replication

Distributed simulations, particularly when executed across many nodes with varying capabilities, can lead to differing simulation speeds among simbots. To address this, implementing a more lenient consistency model by decoupling the execution of the leader from its replicas minimizes the risk of slowing down the entire simulation if one part lags due to delayed execution. This approach necessitates integrating techniques that ensure strict consistency convergence if a failure occurs when the leader and its replica have divergent states.

In non-fault conditions, decoupling the leader from its replicas reduces the need for constant synchronization, enhancing overall simulation efficiency. This configuration allows the leader simbot to handle the sending of external events to successor simbots, while the replica generates but does not send these events. Should the replica fail, the leader can continue the simulation unaffected, regardless of whether the replica was ahead or behind in processing. Conversely, if the leader fails, the replica can be found in one of these states at the time of the crash:

- **State D: Delayed Replica**. The replica is lagging behind the leader in processing events. When the leader crashes, the replica could resend events already dispatched by the leader when promoting as the new leader and resuming the simulation. These events should not be sent to avoid duplicates.
- **State A: Advanced Replica**. The replica has produced events, without sending them to its successors, that the leader has not yet reached. When the leader crashes, the replica could skip sending the already produced events

to its successors when promoting. These events must be stored to maintain consistency and sent when the promotion takes place.
- **State S: Synchronized Replica**. If the leader and replica are processing events at the same pace, the replica can continue seamlessly and resume event dispatch to successors.

3.2 Consistency

Tracking the state of each leader and replica is crucial for determining the correct course of action when handling events after a crash, particularly when the leader of a simbot crashes and a replica needs to be promoted. This mechanism enables the system to seamlessly restore and maintain consistency.

In this system, the *events-sent* message serves as a confirmation from the leader that an event has been sent to successor simbots. Each event produced is assigned a unique identifier, which is consistent across both the leader and the replica due to the deterministic nature of the simulation. The unique identifier ensures that every event and its confirmation carry the same serial number, thus confirming the simulation step across all replicas reliably. The reception of an events-sent message allows a replica to validate the progression of the simulation state up to the received serial number, assuming that any event not validated by a leader has not been transmitted to successor simbots.

The replication system operates within an eventual consistency window, which must converge at the point of a failure. The leader sends event confirmations to the replica but does not wait for a response, and the replica does not need to wait for these confirmations to continue the simulation. This process reduces the number of messages and the waiting time required to maintain consistency.

When a leader crashes, the events-sent message allows a replica to establish whether it is in State D (Delayed) or A (Advanced). If the replica is in an Advanced State, it must store generated event messages that the previous leader had not yet validated with an events-sent message. The stored event messages are then sent in order by the new leader before producing new ones. Conversely, if in a Delayed State, the new leader, previously a replica, might avoid sending the messages that the leader had already confirmed, as these would be duplicated to the successors.

An essential aspect of this process is the invalidation mechanism employed by successor simbots, which might receive duplicated messages: first from the crashed leader and then from the newly promoted leader (formerly a replica). The duplication issue can arise if the leader sends a message to its successor but crashes before it can confirm this action to the replica. To address this, successors store unique identifiers of received event messages, enabling them to detect and discard any duplicates. Achieving an optimal balance between the frequency of confirmations from the leader to the replica, and the number of messages that successors need to store for detecting duplicates, is crucial for minimizing overhead and maximizing the system's efficiency.

Algorithm 1 is executed by the replica to manage the differential state between the leader and itself. It tracks the last message confirmed by the leader and the last message produced by the replica, allowing for the identification and invalidation of potential duplicate messages, and avoid sending these to the successors when promoting.

Algorithm 1. Differential State between Leader and Replica Registration

```
// Differential state between leader and replica. Init.: empty list
Q^R ← []
// Last message produced by replica and not confirmed by leader. Init: ID null message
M^R ← 0
// Last message acknowledged by leader. Init.: ID null-message
M^L ← 0
// Register reception of acknowledgement message of an event from the leader
procedure registerAcknowledgementReceived(M : Acknowledgement)
    M^L ← M_ID
    if M^R >= M^L then
        Q^R.deleteFirst()  // Replica is in advance or at the same point as the leader
    else
        no − op  // (M^R < M^L) Replica is behind
    end if
end procedure
// Event produced in replica
procedure registerProducedEvent(M : Message)
    M^R ← M_ID
    if M^R > M^L then
        Q^R.insertLast(M)  // Replica is in advance
    else
        no−op  // (M^R <= M^L) Replica is behind or at the same point as the leader
    end if
end procedure
```

3.3 Fault Detection

If a simbot crashes, the entire simulation stops as the flow of event messages ceases and simulation time cannot advance. To mitigate this, adjacent neighbouring simbots are monitored, to efficiently detect and respond to potential crashes. Additionally, within each replica group, both the leader and the replicas actively monitor each other to ensure any faults are quickly identified.

Fatal faults occur when every member of a replica group, including the leader, crashes. This scenario results in the triggering of notifications from neighbouring simbots, leading to a complete halt of the distributed simulation. Terminating the simulation is necessary because the state of the partition of the model in the crashed simbot is irretrievably lost, compromising the integrity of the entire simulation.

3.4 Fault Recovery

Fault recovery for each simbot is managed by its surviving replicas and coordinated with its adjacent neighbours.

Replica Fault. When a simbot replica crashes, the leader promptly notifies the *Replica Provisioner*, an external service that prepares the infrastructure and initiates the simbot to simulate a specific subnetwork of the model. The Replica Provisioner then dynamically provides a new replica node. Subsequently, predecessor simbots synchronize with the new replica, and the leader transfers its state to it, allowing the new replica to start connections and resume the simulation from the same point as the leader. Finally, all neighbours resume simulation with the leader and the new replica.

Leader Fault and Replica Promotion. If the leader fails, the replica is promoted to become the new leader, notifies this role change to its predecessor neighbours, and establishes new connections with the successor leaders and replicas. Depending on its state at the time of the leader's failure, the new leader will take different actions to re-establish simulation consistency:

- **State D**: The new leader is behind the old one, so it abstains from sending event messages until it produces an event with the last serial number confirmed by the old leader.
- **State A**: The new leader is more advanced than the old one, so it replays all stored messages from the last confirmation received from the old leader.
- **State S**: The new and old leader are in the same state, so no additional action is needed.

Finally, all steps of replica fault recovery are applied, involving a request for a new replica to the Replica Provisioner to ensure continuous simulation operation.

3.5 Simbot Architecture

Figure 3 illustrates the architecture and software components of a fault-tolerant simbot. The architecture is split into two primary threads: the *Simulation thread* and the *Communication thread*. The Simulation thread runs the *Simulation Engine* and the *Fault Tolerance Manager*, while the Communication thread manages the reception of messages from other simbots via the network, utilizing the *Mailbox* and the *Network Message Receiver*. The *Simbot* component is the coordinator of these threads and components. The *Connection Manager* is responsible for maintaining persistent network connections with neighbouring simbots established in Fig. 2, managing both the connections established at the beginning of the simulation and new connections created when new replicas are added after a crash. The Replica Provisioner and *Debug Server* serve as auxiliary external services.

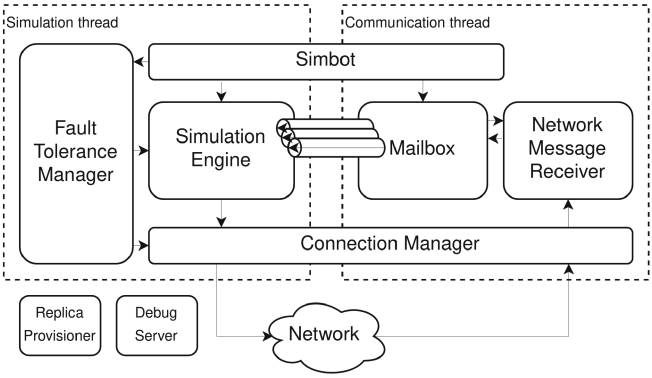

Fig. 3. Simbot Architecture for Fault Tolerant Simulation

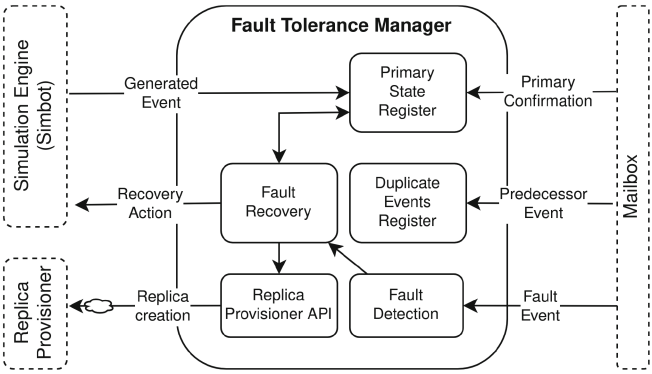

Fig. 4. Components of Fault Tolerance Manager of a Simbot

Figure 4 describes the internal elements of the Fault Tolerance Manager, which is a crucial component highlighted in Fig. 3. The Fault Tolerance Manager's functionalities include the detection and management of faults, synchronization, recovery of neighbour faults, coordination with the Replica Provisioner, and consistency management. The *Leader State Register* executes the procedures of algorithm Algorithm 1 on the replica node. The Mailbox component redirects acknowledgements from the leader and facilitates the production of local events by the Simulation Engine. The Leader State Register plays a vital role in fault recovery to ensure consistency across the simulation. The *Duplicate Event Register* manages the reception of duplicate events post-fault and ensures their dismissal. Each message is registered by the Mailbox before processing to detect any duplicates. The Network Message Receiver is tasked with the detection of faults in other simbots, with any detected faults reported to the Fault Tolerance Manager, which then manages these faults as allowed by the ongoing simulation execution. Finally, the *Fault Recovery* component initiates activities that include requesting a new replica from the Replica Provisioner.

4 Experimentation

The distributed simulator has been developed in the Rust language. The test environment consists of a cluster of 20 Raspberry Pi 4 model B with 8 GB of RAM and 1Gb Ethernet links through a 1Gb Ethernet switch. The fault-tolerant configuration includes two nodes for each simbot, a leader and one replica.

Table 1. Comparison of the Execution Time for Non-fault-tolerant vs. Fault-tolerant Distributed Simulators with Different Simulation Times and Low Simulation Load

Total Simulation time (*simseconds*)	Wall clock time (*seconds*)	
	non fault-tolerant	fault-tolerant
20	0.0067	0.0108
1000	0.162	0.253
10000	1.643	2.478
100000	17.655	22.751

Table 1 illustrates the execution time differences between the fault-tolerant simulator and the non-fault-tolerant simulator in a worst-case scenario with minimal simulation load, emphasizing the communication and coordination overhead. This minimal load represents all the operational overhead as pure communication, which is typically where fault tolerance could add significant overhead. The simulation time goes from 20 to 100000 simseconds to measure the wall clock time. However, it is evident from the results that the difference in execution times between the versions, even under these conditions, is minimal.

Figure 5 displays the average execution time for each phase of the simulation (reception, simulation, and sending) for versions without fault management and with fault management. A minimal simulation load is calculated based on a model [16] where the efficiency of distributed simulation outweighs centralized approaches. A simulation load of 0.035 wall clock time seconds is used, when the load is large enough to make the distributed coupling factor $\lambda > 100$. The negligible difference of 0.269% between both simulator versions underscores that, in scenarios without faults, there is almost no overhead introduced by fault management.

Experiments in our test environment indicate that it takes approximately 0.4 s to recover from a fault, with 99% of this time spent provisioning a new node and copying all state to the new replica. For simulations involving large models, the state copy process constitutes the most significant portion of time spent in fault recovery. This insight is beneficial as most of the time added by fault tolerance is concentrated in the fault recovery process (specifically, replica provisioning and state copy) which occurs only when faults arise, not during normal operation.

Fig. 5. Execution Time of Simulation with Minimal Efficient Load (0.035 s.)

5 Related Work

Fault tolerance techniques in distributed simulations are broadly categorized into two main strategies: replication and checkpointing [6]. Replication increases the demand for computational resources and introduces synchronization overhead to ensure consistency between replicas. Checkpointing, on the other hand, involves storing recovery points and requires synchronization among processes to determine the recovery moment. Optimistic protocols maintain causality consistency through a rollback mechanism, which restores the simulation to a previously stored checkpoint when an out-of-order event is received. Consequently, the choice of fault tolerance technique often depends on the protocol used to maintain the consistency of the distributed simulation. Traditional methods generally incur significant computational costs, storage demands, or synchronization overheads.

Recent advancements in fault tolerance for distributed simulations have primarily focused on optimistic protocols, exploiting rollback mechanisms inherent to optimistic simulations to revert to states prior to faults [1]. These techniques have occasionally been adapted for conservative simulation approaches as well [8]. Fault tolerance methods have been developed for specific simulation architectures or frameworks, such as Time Warp [2], GAIA/ARTIS [7], and High-Level Architecture (HLA). These methods typically require extensive computational and network resources, which can increase simulation latency and reduce overall performance. Conversely, techniques designed to preserve performance tend to be complex and less flexible.

In contrast to the aforementioned systems, which prioritize interoperability, our approach focuses on efficiency and scalability. Our fault tolerance strategy enhances performance with minimal impact on latency compared to other methods. By tailoring our approach to conservative simulation, and leveraging simulation messages and timing, our model achieves greater resource efficiency

and adaptability, avoiding the significant computational and network overhead often associated with other fault tolerance systems.

6 Conclusion

In this paper, we have presented a replication-based approach to fault tolerance within distributed simulations, that highlights performance and resource efficiency during non-fault conditions. Our approach adopts a conservative strategy to optimize resource utilization effectively. Notably, even when fault tolerance mechanisms are active, a fault-free distributed simulation maintains performance levels comparable to those with fault tolerance disabled. Additionally, memory usage remains minimal when replicas perform similarly. Thus, this approach facilitates cost-effective deployment and execution in cloud environments, ensuring fault tolerance for extended simulations without sacrificing performance or resource efficiency.

As future work, other failure models, as network partitions and byzantine faults, could be addressed to enhance the resiliency of distributed simulations.

Acknowledgments. This work was co-financed by the Aragonese Government and the European Regional Development Fund "Construyendo Europa desde Aragón" (COSMOS research group); and by the Spanish program "Programa estatal del Generación de Conocimiento y Fortalecimiento Científico y Tecnológico del Sistema de I+D+i ", project PGC2018-099815-B-100.

References

1. Agrawal, D., Agre, J.: Recovering from multiple process failures in the time warp mechanism. IEEE Trans. Comput. **41**(12), 1504–1514 (1992)
2. Agrawal, D., Agre, J.R.: Replicated objects in time warp simulations. In: Proceedings of the 24th Conference on Winter Simulation, pp. 657–664 (1992)
3. Arronategui, U., Bañares, J.Á., Colom, J.M.: A MDE approach for modelling and distributed simulation of health systems. In: Djemame, K., Altmann, J., Bañares, J.Á., Agmon Ben-Yehuda, O., Stankovski, V., Tuffin, B. (eds.) GECON 2020. LNCS, vol. 12441, pp. 89–103. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-63058-4_9
4. Bañares, J.Á., Colom, J.M.: Model and simulation engines for distributed simulation of discrete event systems. In: GECON 2018 - International Conference on the Economics of Grids, Clouds, Systems, and Services, pp. 77–91. Springer (2018)
5. Colom, J.M.: Harnessing structure theory of petri nets in discrete event system simulation. In: Kristensen, L.M., van der Werf, J.M. (eds.) Application and Theory of Petri Nets and Concurrency, pp. 3–23. Springer, Cham (2024)
6. Damani, O.P., Garg, V.K.: Fault-tolerant distributed simulation. In: Unger, B.W., Ferscha, A. (eds.) Proceedings of the 12th Workshop on Parallel and Distributed Simulation, PADS '98, Banff, Alberta, Canada, May 26-29, 1998, pp. 38–45. IEEE Computer Society (1998)

7. D'Angelo, G., Ferretti, S., Marzolla, M., Armaroli, L.: Fault-tolerant adaptive parallel and distributed simulation. In: 2016 IEEE/ACM 20th International Symposium on Distributed Simulation and Real Time Applications (DS-RT), pp. 37–44. IEEE (2016)
8. D'Angelo, G., Ferretti, S., Marzolla, M.: Fault tolerant adaptive parallel and distributed simulation through functional replication. Simul. Modelling Practice Theory **93**, 192–207 (2019), modeling and Simulation of Cloud Computing and Big Data
9. Ferscha, A., Johnson, J., Turner, S.J.: Distributed simulation performance data mining. Future Generation Comput. Syst. **18**(1), 157–174 (2001), i. High Performance Numerical Methods and Applications. II. Performance Data Mining: Automated Diagnosis, Adaption, and Optimization
10. Fujimoto, R.M.: Research challenges in parallel and distributed simulation. ACM Trans. Model. Comput. Simul. **26**(4), 22:1–22:29 (2016). https://doi.org/10.1145/2866577
11. Hodgetts, P., et al.: Workload evaluation in distributed simulation of dess. In: GECON 2021 - International Conference on the Economics of Grids, Clouds, Systems, and Services, pp. 3–16. Springer (2021)
12. Jha, V., Bagrodia, R.: A performance evaluation methodology for parallel simulation protocols. In: Proceedings of Symposium on Parallel and Distributed Tools, pp. 180–185 (1996). https://doi.org/10.1109/PADS.1996.761576
13. Murata, T., Wu, Z.: Fair relation and modified synchronic distances in a petri net. J. Franklin Inst. **320**(2), 63–82 (1985)
14. Rehman, A.U., Aguiar, R.L., Barraca, J.P.: Fault-tolerance in the scope of cloud computing. IEEE Access **10**, 63422–63441 (2022)
15. Vanmechelen, K., De Munck, S., Broeckhove, J.: Conservative distributed discrete-event simulation on the amazon ec2 cloud: an evaluation of time synchronization protocol performance and cost efficiency. Simul. Model. Pract. Theory **34**, 126–143 (2013)
16. Varga, A., Sekercioglu, Y., Egan, G.: A practical efficiency criterion for the null message algorithm. In: Verbraeck, A., Hlupic, V. (eds.) Simulation in Industry: Proceedings of the 15th European Simulation Symposium (ESS 2003), pp. 81 – 92 (2003)
17. Zhao, W., Melliar-Smith, P., Moser, L.: Fault tolerance middleware for cloud computing. In: 2010 IEEE 3rd International Conference on Cloud Computing, pp. 67–74 (2010)